USENIX Association

Proceedings of the
28th USENIX Security Symposium

August 14–16, 2019
Santa Clara, CA, USA

Conference Organizers

Program Co-Chairs

Nadia Heninger, *University of Pennsylvania*
Patrick Traynor, *University of Florida*

Program Committee

Yasemin Acar, *Leibniz University Hannover*
Sadia Afroz, *University of California, Berkeley/ International Computer Science Institute*
Devdatta Akhawe, *Dropbox*
Johanna Amann, *International Computer Science Institute*
Adam Aviv, *United States Naval Academy*
Michael Bailey, *University of Illinois at Urbana–Champaign*
Adam Bates, *University of Illinois at Urbana–Champaign*
Vincent Bindschaedler, *University of Florida*
Joseph Bonneau, *New York University*
Nikita Borisov, *University of Illinois at Urbana–Champaign*
Sven Bugiel, *CISPA Helmholtz Center i.G.*
Kevin Butler, *University of Florida*
Joe Calandrino, *Federal Trade Commission*
Stefano Calzavara, *Università Ca' Foscari Venezia*
Yinzhi Cao, *Johns Hopkins University*
Srdjan Capkun, *ETH Zurich*
Lorenzo Cavallaro, *King's College London*
Stephen Checkoway, *Oberlin College*
Bill Cheswick, *AT&T Labs—Research*
Marshini Chetty, *Princeton University*
Mihai Christodorescu, *VISA Research*
Erinn Clark, *First Look Media*
George Danezis, *University College London*
Nathan Dautenhahn, *Rice University*
Roger Dingledine, *The Tor Project*
Adam Doupe, *Arizona State University*
Thomas Dullien, *Google*
Zakir Durumeric, *Stanford University*
Manuel Egele, *Boston University*
William Enck, *North Carolina State University*
Roya Ensafi, *University of Michigan*
David Evans, *University of Virginia*
Sascha Fahl, *Leibniz University Hannover*
Giulia Fanti, *Carnegie Mellon University*
Nick Feamster, *Princeton University*
Adrienne Porter Felt, *Google*
Earlence Fernandes, *University of Washington*
David Freeman, *Facebook*
Daniel Genkin, *University of Michigan*
Neil Gong, *Iowa State University*
Matthew Green, *Johns Hopkins Information Security Institute*
Rachel Greenstadt, *Drexel University*
Daniel Gruss, *Graz University of Technology*
Joseph Lorenzo Hall, *Center for Democracy & Technology*
Xiali (Sharon) Hei, *University of Louisiana at Lafayette*
Thorsten Holz, *Ruhr-University Bochum*
Trent Jaeger, *The Pennsylvania State University*
Rob Jansen, *U.S. Naval Research Laboratory*
Mobin Javed, *Lahore University of Management Sciences*
Chris Kanich, *University of Illinois at Chicago*
Vasileios Kemerlis, *Brown University*
Yongdae Kim, *Korea Advanced Institute of Science and Technology (KAIST)*
Lea Kissner, *Humu*
Yoshi Kohno, *University of Washington*
Farinaz Koushanfar, *University of California, San Diego*
Katharina Krombholz, *CISPA Helmholtz Center i.G.*
Ben Laurie, *Google*
Tancrède Lepoint, *Google*
Martina Lindorfer, *Technische Universität Wien*
Allison Mankin, *Salesforce*
Ivan Martinovic, *Oxford University*
Stephen McCamant, *University of Minnesota*
Jon McCune, *Google*
Patrick McDaniel, *The Pennsylvania State University*
Sarah Meiklejohn, *University College London*
Jelena Mirkovic, *USC/Information Sciences Institute*
Prateek Mittal, *Princeton University*
Veelasha Moonsamy, *Utrecht University*
Adwait Nadkarni, *College of William & Mary*
Yossi Oren, *Ben-Gurion University of the Negev*
Nicolas Papernot, *The Pennsylvania State University*
Kenny Paterson, *Royal Holloway*
Mathias Payer, *École Polytechnique Fédérale de Lausanne (EPFL)*
Giancarlo Pellegrino, *Stanford University*
Christina Pöpper, *New York University Abu Dhabi*
Brad Reaves, *North Carolina State University*
Elissa Redmiles, *University of Maryland*
Konrad Rieck, *Technische Universität Braunschweig*
Tom Ristenpart, *Cornell Tech*
Tom Ritter, *Mozilla*
Franziska Roesner, *University of Washington*
Ahmad-Reza Sadeghi, *Technische Universität Darmstadt*
Prateek Saxena, *National University of Singapore*
Nolen Scaife, *University of Florida*
Wendy Seltzer, *W3C/Massachusetts Institute of Technology*
Micah Sherr, *Georgetown University*
Deian Stefan, *University of California, San Diego*
Ben Stock, *CISPA Helmholtz Center i.G.*
Gianluca Stringhini, *Boston University*

Dave 'Jing' Tian, *University of Florida*
Luke Valenta, *University of Pennsylvania*
Ingrid Verbauwhede, *Katholieke Universiteit Leuven*
David Wagner, *University of California, Berkeley*
Byron Williams, *University of Florida*
Eric Wustrow, *University of Colorado Boulder*
Wenyuan Xu, *Zhejiang University*
Yuval Yarom, *University of Adelaide and Data61*
Tuba Yavuz, *University of Florida*
Daniel Zappala, *Brigham Young University*
Mary Ellen Zurko, *MIT Lincoln Laboratory*

Invited Talks Chair
Devdatta Akhawe, *Dropbox*

Invited Talks Committee
Alex Gantman, *Qualcomm*
Giancarlo Pellegrino, *Stanford University*
Elissa Redmiles, *University of Maryland*

Lightning Talks Chair
Christina Garman, *Purdue University*

Poster Session Chair
Brad Reaves, *North Carolina State University*

Test of Time Awards Committee
Matt Blaze, *University of Pennsylvania*
Dan Boneh, *Stanford University*
Kevin Fu, *University of Michigan*
Fabian Monrose, *The University of North Carolina at Chapel Hill*

Steering Committee
Matt Blaze, *University of Pennsylvania*
Dan Boneh, *Stanford University*
William Enck, *North Carolina State University*
Kevin Fu, *University of Michigan*
Casey Henderson, *USENIX Association*
Thorsten Holz, *Ruhr-Universität Bochum*
Jaeyeon Jung, *Samsung Electronics*
Engin Kirda, *Northeastern University*
Tadayoshi Kohno, *University of Washington*
Adrienne Porter Felt, *Google*
Thomas Ristenpart, *Cornell Tech*
David Wagner, *University of California, Berkeley*

External Reviewers

Hadi Abdullah
Bander Alsulami
Cornelius Aschermann
Teodora Baluta
Gabrielle Beck
Logan Blue
Nicole Borrelli
Sam Bretheim
Marcus Brinkmann
Claudio Canella
Benton Case
Berkay Celik
Alishah Chator
Rahul Chatterjee
Qingrong Chen
Joseph Choi
David Clayton
Shaanan Cohney
Edwin Dauber
Giulio De Pasquale
Sergi Delgado Segura
Henri Maxime Demoulin
Brian Desnoyers
Karim Eldefrawy
Evan Evtimov
Dennis Felsch
Joel Frank
Vanessa Frost
Ankit Gangwal
Peng Gao
Washington Garcia
Jordy Gennissen
Lukas Giner
Steve Gomez
Martin Grothe
Muhammad Haris Mughees
Marcella Hastings
Grant Hernandez
Grant Ho
Stefan Hoffmann
Liz Izhikevich
Sakshi Jain
Tyler Kaczmarek
George Kappos
Gabrielle Kaptchuk
Katarina Kohls
Aashish Kolluri
Georg Koppen
Ben Kreuter
Deepak Kumar
Daniele Lain
Sebastian Lauer
Kevin Liao
Moritz Lipp
Alwin Maier
Patrick McCorry
Robert Merget
Muhammad Shujaat Mirza
Rafael Misoczki
Vladislav Mladenov
Ivica Nikolic
Liang Niu
Aleatha Parker-Wood
Paul Pearce
Feargus Pendlebury
Mike Perry
Fabio Pierazzi
Ania Piotrowska
Erwin Quiring
Sanjeev Reddy
Paul Rösler
David Rupprecht
M. Sadegh Riazi
Theodor Schnitzler
Sergej Schumilo
Roei Schuster
Michael Schwarz
Will Scott
Karn Seth
Hovav Shacham
Rich Shay
Shiqi Shen
Tom Shrimpton
Camelia Simoiu
Douglas Stebila
Mohammad Taha Khan
Kejsi Take
Dennis Tatang
Aaron Tomb
Mathy Vanhoef
Luis Vargas
Liang Wang
Alexander Warnicke
Christian Wressnegger
Karl Wüst
Xiaojun Xu
Nian Xue
Haaroon Yousaf
Pinghai Yuan
Yupeng Zhang
Maximilian Zinkus

Message from the 28th USENIX Security Symposium Program Co-Chairs

Welcome to the USENIX Security Symposium in Santa Clara, CA! We hope you enjoy the outstanding technical program and invited talks. Now in its 28th year, the symposium brings together researchers and practitioners from across the field. We encourage you to engage with the community through our events, hallway track, and questions for speakers.

This was an exciting year for the USENIX Security Symposium as we transitioned to a new paper reviewing model with multiple submission deadlines. We want to use this opportunity to detail the model we instituted this year, as well as the process we used to develop it.

The USENIX board asked us in June of 2018 if we would be willing to move to multiple submission deadlines for the 2019 Symposium, and tasked us with developing a plan to do so. We studied the processes and choices made by conferences that had previously transitioned to multiple submission deadlines, including the IEEE Symposium on Security and Privacy, the Privacy-Enhancing Technologies Symposium, the ACM International Conference on Mobile Computing and Networking, and the Conference on Cryptographic Hardware and Embedded Systems, and consulted with former and current chairs of these conferences. We then developed a preliminary plan that we presented to the USENIX Security Steering Committee for feedback. After incorporating their suggested changes and receiving approval, we presented the plan to the USENIX Security community at USENIX Security 2018 in August 2018 in a community meeting to gather feedback before publishing the official Call for Papers.

We made the following choices in designing the new submission model:

- There would be four evenly-spaced submission deadlines throughout the year. We felt that this was a "sweet spot" that would allow for the two-and-a-half-month review cycle that the community was used to, while still giving authors multiple opportunities to submit their work when they felt it was ready. Since 2019 was a transitional year, we had two submission deadlines. The first deadline in the fall was November 15, 2018, and the second deadline in the winter was February 15, 2019.
- Like other conferences that have transitioned to multiple deadlines, we instituted a paper revision and resubmission process, which we describe in more detail below.
- To try to make the reviewing and revision process as constructive as possible, we introduced "journal-style" reviewing outcomes. That is, instead of rating papers "Accept", "Weak Accept", "Weak Reject", or similar, we specified that reviewers could give outcomes of "Accept", "Minor Revision", "Major Revision", "Reject and Resubmit", and "Reject".

"Accept" has the same meaning as before. "Minor Revision" replaces "Accept with Shepherding". Papers with this outcome were assigned a shepherd who articulated a specific list of textual changes, such as adding additional citations or clarifying details of experiments, that the authors were requested to make, with the specific guidance that papers in this category were not accepted until the changes were made to the reviewers' satisfaction.

"Major Revision" was the most significant change to the process. Papers receiving this outcome were returned to the authors with a list of specific changes requested by the reviewers. These included performing additional experiments, adding additional case studies or analyses, or requests for rewriting that were considered beyond the scope of what a shepherd could reasonably guide. Authors of "Major Revision" papers were invited to resubmit to either of the next two submission deadlines, with the promise that we would attempt to assign the same set of reviewers to review the resubmission, and that the resubmission would be evaluated according to the reviews and the specific changes requested by the reviewers. Papers receiving this outcome were considered to still be under submission for the next two deadlines, and we asked authors to explicitly withdraw their papers from consideration if they wished to submit the same work to another conference.

Finally, papers that were rejected could receive two possible outcomes. A "Reject and Resubmit" outcome was intended to signal to the authors that the reviewers thought the work could likely be revised to be accepted, but that the scope of the changes reviewers felt was required for acceptance was beyond what the reviewers could articulate in a specific list of "Major Revision" requests, or would likely take longer than the four months that authors would have to revise their work for a "Major Revision". Papers receiving this outcome could not submit to either of the next two submission deadlines. Papers receiving a "Reject" outcome were not allowed to resubmit for a full year after the submission date.

- There would be two in-person program committee meetings a year. While program committee meetings are expensive and time-consuming, we received feedback from many community members that they serve an important role for calibration and discussion. In the 2019 transitional year, we held only one in-person meeting associated with the winter submission deadline.
- As in previous years, for each submission deadline, we used a double-blind review process with two rounds of reviews.

We expected the total number of submissions to increase this year, in line with the experience of other conferences that have transitioned to multiple submission deadlines. Accordingly, we gathered the largest program committee ever, with 100 members and two chairs. We endeavored to assemble a diverse program committee in terms of area of expertise, seniority level, geography, gender, race, and institution type. Members of the resulting program committee were 19% from industry, government, or nonprofit, 25% female, and 27% based outside the US. The 2018 USENIX Security chairs invited members of the community to volunteer themselves and others to serve on the 2019 program committee using a web form; we found this to be an incredibly valuable resource when assembling our program committee.

A major goal of our changes to the reviewing system was to focus on returning helpful, constructive reviews to authors, and to provide as much guidance as possible in moving submitted papers towards publication. Following last year, we also assembled a Review Task Force (RTF) of five experienced program committee members to help ensure review quality and encourage positive discussion. RTF members provided feedback on reviews, helped manage online discussion, and acted as proxies for program committee members not in attendance in the in-person program committee meeting, in exchange for a reduced reviewing workload. We found significant value in the RTF, and expanded their roles this year, particularly in facilitating online discussion and helping reviewers calibrate the new review outcomes across papers.

We received 260 submissions in the fall November 15, 2018 deadline. We administratively rejected four papers for violating the call for papers, and two papers were withdrawn, leaving 254 submissions to be considered in Fall Round 1. Each paper was assigned two reviews in the first round. Following three weeks of review and a week of online discussion, 93 papers were early rejected on December 14. Of these, 41 were Rejected and 45 received a Reject and Resubmit outcome. A paper was rejected if it received only scores of Reject or Reject and Resubmit, and neither reviewer saw value in additional reviews. The outcome was agreed upon by the reviewers. Authors of rejected papers were not given the opportunity to appeal. We believed that this early rejection step was critical, because it meant that the authors could immediately begin making changes to their submission and have it evaluated by other reviewers at another venue. The authors of the remaining 161 papers were given the opportunity to respond to the reviews and specific questions from the reviewers. Each fall Round 2 paper received two or more additional reviews. After three more weeks of reviewing and 1.5 weeks of online discussion, we notified authors of the Round 2 decisions on January 18, 2019. Of these papers, 11 were Rejected, 77 received a Reject and Resubmit Outcome, 48 received a Major Revision outcome, 20 received a Minor Revision outcome, and 5 were Accepted. All 20 Minor Revision papers from the fall submission deadline were accepted by February 18, 2019.

We received 481 submissions in the winter February 15, 2019 deadline. 38 of these were resubmissions of papers that received a Major Revision decision from the fall deadline. Additionally, the authors of two Major Revision papers explicitly wrote to withdraw their paper from the USENIX Security review process. We administratively rejected 20 papers for violating the call for papers, and four were withdrawn by the authors, leaving 457 papers to be considered in the winter Round 1. We assigned the same reviewers as in the previous round to the resubmitted Major Revision papers, except in cases where additional conflicts of interest arose or were discovered between the two deadlines, and assigned two reviewers to all other papers. 211 papers were early rejected on March 21, 2019: 122 Rejected and 89 Reject and Resubmit. We also asked the reviewers on Major Revision resubmissions to make decisions in Round 1: six papers were Accepted, 24 papers received a Minor Revision, one paper received a Reject and Resubmit, and seven papers were Rejected. We chose to disallow multiple Major Revision decisions in order to make sure that paper outcomes were decided in a reasonable time frame for authors. This left 216 papers in Round 2.

The in-person PC meeting was held on April 29 and 30 at the University of Florida in Gainesville, Florida. We invited all program committee members to attend the meeting but made attendance optional; 49 program committee members attended. We were able to discuss 91 papers during the meeting, and all other decisions were made in online discussion. Among the papers in Round 2, 11 papers were Accepted, 48 received Minor Revisions, 33 received Major Revisions, 108 received a Reject and Resubmit outcome, and 16 were Rejected. Further, one Minor Revision paper was ultimately rejected by the reviewers; all the others were accepted by the camera-ready deadline of June 1.

In total, we accepted 113 of the 697 distinct, non-withdrawn submissions that we received this year, for a 16% acceptance rate overall. The acceptance rate for the resubmitted Major Revision papers was 76%. Both the number of papers accepted and the number of papers submitted are new records for the symposium, which was exceptionally competitive this year. We congratulate the authors on their excellent work and achievements!

It was an honor to be part of the large community effort that brings together the USENIX Security Symposium. The demands placed on the program committee this year were exceptionally high, both in terms of reviewing load and in calibrating a new reviewing system. Each member submitted about 22 reviews, for a total of 822 reviews in the fall and 1450 reviews in winter, or 2272 reviews total, and more than 8600 comments were left in the discussions. It is our sincere hope that the new process not only assisted in creating the strongest possible program, but also that it helped to improve the quality of reviews and mentorship provided to the community.

We would especially like to thank our Review Task Force: Kevin Butler, Srdjan Capkun, Rachel Greenstadt, Jon McCune, and Franzi Roesner. Yoshi Kohno was our steering committee liaison and was a great help. Michael Bailey also provided valuable feedback from the board. We also thank the many external reviewers who provided additional expertise. We would like to thank the invited talks committee (Devdatta Akhawe, Alex Gantmann, Giancarlo Pellegrino, Elissa Redmiles), the Test of Time award committee (Matt Blaze, Dan Boneh, Kevin Fu, Fabian Monrose), the poster session chair Brad Reaves, and the lightning talks chair Christina Garman. We are extremely grateful to the staff at USENIX who run everything behind the scenes, particularly Casey Henderson, Jasmine Murcia, and Michele Nelson. Finally, we thank all of the authors of the 703 submitted papers for participating in the 28th USENIX Security Symposium.

Nadia Heninger, *University of California San Diego*
Patrick Traynor, *University of Florida*
USENIX Security '19 Program Co-Chairs

USENIX Security '19:
28th USENIX Security Symposium

August 14–16, 2019
Santa Clara, CA, USA

Wireless Security

A Study of the Feasibility of Co-located App Attacks against BLE and a Large-Scale Analysis of the Current Application-Layer Security Landscape 1
Pallavi Sivakumaran and Jorge Blasco, *Royal Holloway University of London*

The CrossPath Attack: Disrupting the SDN Control Channel via Shared Links 19
Jiahao Cao, Qi Li, and Renjie Xie, *Tsinghua University;* Kun Sun, *George Mason University;* Guofei Gu, *Texas A&M University;* Mingwei Xu and Yuan Yang, *Tsinghua University*

A Billion Open Interfaces for Eve and Mallory: MitM, DoS, and Tracking Attacks on iOS and macOS Through Apple Wireless Direct Link 37
Milan Stute, *Technische Universität Darmstadt;* Sashank Narain, *Northeastern University;* Alex Mariotto, Alexander Heinrich, and David Kreitschmann, *Technische Universität Darmstadt;* Guevara Noubir, *Northeastern University;* Matthias Hollick, *Technische Universität Darmstadt*

Hiding in Plain Signal: Physical Signal Overshadowing Attack on LTE 55
Hojoon Yang, Sangwook Bae, Mincheol Son, Hongil Kim, Song Min Kim, and Yongdae Kim, *KAIST*

UWB-ED: Distance Enlargement Attack Detection in Ultra-Wideband 73
Mridula Singh, Patrick Leu, AbdelRahman Abdou, and Srdjan Capkun, *ETH Zurich*

Protecting Users Everywhere

Computer Security and Privacy in the Interactions Between Victim Service Providers and Human Trafficking Survivors 89
Christine Chen, *University of Washington;* Nicola Dell, *Cornell Tech;* Franziska Roesner, *University of Washington*

Clinical Computer Security for Victims of Intimate Partner Violence 105
Sam Havron, Diana Freed, and Rahul Chatterjee, *Cornell Tech;* Damon McCoy, *New York University;* Nicola Dell and Thomas Ristenpart, *Cornell Tech*

Evaluating the Contextual Integrity of Privacy Regulation: Parents' IoT Toy Privacy Norms Versus COPPA 123
Noah Apthorpe, Sarah Varghese, and Nick Feamster, *Princeton University*

Secure Multi-User Content Sharing for Augmented Reality Applications 141
Kimberly Ruth, Tadayoshi Kohno, and Franziska Roesner, *University of Washington*

Understanding and Improving Security and Privacy in Multi-User Smart Homes: A Design Exploration and In-Home User Study 159
Eric Zeng and Franziska Roesner, *University of Washington*

Hardware Security

PAC it up: Towards Pointer Integrity using ARM Pointer Authentication 177
Hans Liljestrand, *Aalto University, Huawei Technologies Oy;* Thomas Nyman, *Aalto University;* Kui Wang, *Huawei Technologies Oy, Tampere University of Technology;* Carlos Chinea Perez, *Huawei Technologies Oy;* Jan-Erik Ekberg, *Huawei Technologies Oy, Aalto University;* N. Asokan, *Aalto University*

Origin-sensitive Control Flow Integrity 195
Mustakimur Rahman Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang, *Florida State University*

(continued on next page)

HardFails: Insights into Software-Exploitable Hardware Bugs 213
Ghada Dessouky and David Gens, *Technische Universität Darmstadt;* Patrick Haney and Garrett Persyn, *Texas A&M University;* Arun Kanuparthi, Hareesh Khattri, and Jason M. Fung, *Intel Corporation;* Ahmad-Reza Sadeghi, *Technische Universität Darmstadt;* Jeyavijayan Rajendran, *Texas A&M University*

uXOM: Efficient eXecute-Only Memory on ARM Cortex-M 231
Donghyun Kwon, Jangseop Shin, and Giyeol Kim, *Seoul National University;* Byoungyoung Lee, *Seoul National University, Purdue University;* Yeongpil Cho, *Soongsil University;* Yunheung Paek, *Seoul National University*

A Systematic Evaluation of Transient Execution Attacks and Defenses 249
Claudio Canella, *Graz University of Technology;* Jo Van Bulck, *imec-DistriNet, KU Leuven;* Michael Schwarz, Moritz Lipp, Benjamin von Berg, and Philipp Ortner, *Graz University of Technology;* Frank Piessens, *imec-DistriNet, KU Leuven;* Dmitry Evtyushkin, *College of William and Mary;* Daniel Gruss, *Graz University of Technology*

Machine Learning Applications

The Secret Sharer: Evaluating and Testing Unintended Memorization in Neural Networks. 267
Nicholas Carlini, *Google Brain;* Chang Liu, *University of California, Berkeley;* Úlfar Erlingsson, *Google Brain;* Jernej Kos, *National University of Singapore;* Dawn Song, *University of California, Berkeley*

Improving Robustness of ML Classifiers against Realizable Evasion Attacks Using Conserved Features 285
Liang Tong, *Washington University in St. Louis;* Bo Li, *UIUC;* Chen Hajaj, *Ariel University;* Chaowei Xiao, *University of Michigan;* Ning Zhang and Yevgeniy Vorobeychik, *Washington University in St. Louis*

ALOHA: Auxiliary Loss Optimization for Hypothesis Augmentation 303
Ethan M. Rudd, Felipe N. Ducau, Cody Wild, Konstantin Berlin, and Richard Harang, *Sophos*

Why Do Adversarial Attacks Transfer? Explaining Transferability of Evasion and Poisoning Attacks 321
Ambra Demontis, Marco Melis, and Maura Pintor, *University of Cagliari, Italy;* Matthew Jagielski, *Northeastern University;* Battista Biggio, *University of Cagliari, Italy, and Pluribus One;* Alina Oprea and Cristina Nita-Rotaru, *Northeastern University;* Fabio Roli, *University of Cagliari, Italy, and Pluribus One*

Stack Overflow Considered Helpful! Deep Learning Security Nudges Towards Stronger Cryptography 339
Felix Fischer, *Technical University of Munich;* Huang Xiao, *Bosch Center for Artificial Intelligence;* Ching-Yu Kao, *Fraunhofer AISEC;* Yannick Stachelscheid, Benjamin Johnson, and Danial Razar, *Technical University of Munich;* Paul Fawkesley and Nat Buckley, *Projects by IF;* Konstantin Böttinger, *Fraunhofer AISEC;* Paul Muntean and Jens Grossklags, *Technical University of Munich*

Planes, Cars, and Robots

Wireless Attacks on Aircraft Instrument Landing Systems 357
Harshad Sathaye, Domien Schepers, Aanjhan Ranganathan, and Guevara Noubir, *Northeastern University*

Please Pay Inside: Evaluating Bluetooth-based Detection of Gas Pump Skimmers 373
Nishant Bhaskar and Maxwell Bland, *University of California San Diego;* Kirill Levchenko, *University of Illinois at Urbana-Champaign;* Aaron Schulman, *University of California San Diego*

CANvas: Fast and Inexpensive Automotive Network Mapping 389
Sekar Kulandaivel, Tushar Goyal, Arnav Kumar Agrawal, and Vyas Sekar, *Carnegie Mellon University*

Losing the Car Keys: Wireless PHY-Layer Insecurity in EV Charging 407
Richard Baker and Ivan Martinovic, *University of Oxford*

RVFuzzer: Finding Input Validation Bugs in Robotic Vehicles through Control-Guided Testing 425
Taegyu Kim, *Purdue University;* Chung Hwan Kim and Junghwan Rhee, *NEC Laboratories America;* Fan Fei, Zhan Tu, Gregory Walkup, Xiangyu Zhang, Xinyan Deng, and Dongyan Xu, *Purdue University*

Machine Learning, Adversarial and Otherwise

Seeing is Not Believing: Camouflage Attacks on Image Scaling Algorithms 443
Qixue Xiao, *Department of Computer Science and Technology, Tsinghua University and 360 Security Research Labs;* Yufei Chen, *School of Electronic and Information Engineering, Xi'an Jiaotong University and 360 Security Research Labs;* Chao Shen, *School of Electronic and Information Engineering, Xi'an Jiaotong University;* Yu Chen, *Department of Computer Science and Technology, Tsinghua University and Peng Cheng Laboratory;* Kang Li, *Department of Computer Science, University of Georgia*

CT-GAN: Malicious Tampering of 3D Medical Imagery using Deep Learning 461
Yisroel Mirsky and Tom Mahler, *Ben-Gurion University;* Ilan Shelef, *Soroka University Medical Center;* Yuval Elovici, *Ben-Gurion University*

Misleading Authorship Attribution of Source Code using Adversarial Learning 479
Erwin Quiring, Alwin Maier, and Konrad Rieck, *TU Braunschweig*

Terminal Brain Damage: Exposing the Graceless Degradation in Deep Neural Networks Under Hardware Fault Attacks 497
Sanghyun Hong, *University of Maryland College Park;* Pietro Frigo, *Vrije Universiteit Amsterdam;* Yiğitcan Kaya, *University of Maryland College Park;* Cristiano Giuffrida, *Vrije Universiteit Amsterdam;* Tudor Dumitraş, *University of Maryland College Park*

CSI NN: Reverse Engineering of Neural Network Architectures Through Electromagnetic Side Channel 515
Lejla Batina, *Radboud University, The Netherlands;* Shivam Bhasin and Dirmanto Jap, *Nanyang Technological University, Singapore;* Stjepan Picek, *Delft University of Technology, The Netherlands*

Mobile Security 1

simTPM: User-centric TPM for Mobile Devices 533
Dhiman Chakraborty, *CISPA Helmholtz Center for Information Security, Saarland University;* Lucjan Hanzlik, *CISPA Helmholtz Center for Information Security, Stanford University;* Sven Bugiel, *CISPA Helmholtz Center for Information Security*

The Betrayal At Cloud City: An Empirical Analysis Of Cloud-Based Mobile Backends 551
Omar Alrawi, *Georgia Institute of Technology;* Chaoshun Zuo, *Ohio State University;* Ruian Duan and Ranjita Pai Kasturi, *Georgia Institute of Technology;* Zhiqiang Lin, *Ohio State University;* Brendan Saltaformaggio, *Georgia Institute of Technology*

EnTrust: Regulating Sensor Access by Cooperating Programs via Delegation Graphs 567
Giuseppe Petracca, *Pennsylvania State University, US;* Yuqiong Sun, *Symantec Research Labs, US;* Ahmad-Atamli Reineh, *Alan Turing Institute, UK;* Patrick McDaniel, *Pennsylvania State University, US;* Jens Grossklags, *Technical University of Munich, DE;* Trent Jaeger, *Pennsylvania State University, US*

PolicyLint: Investigating Internal Privacy Policy Contradictions on Google Play 585
Benjamin Andow and Samin Yaseer Mahmud, *North Carolina State University;* Wenyu Wang, *University of Illinois at Urbana-Champaign;* Justin Whitaker, William Enck, and Bradley Reaves, *North Carolina State University;* Kapil Singh, *IBM T.J. Watson Research Center;* Tao Xie, *University of Illinois at Urbana-Champaign*

50 Ways to Leak Your Data: An Exploration of Apps' Circumvention of the Android Permissions System 603
Joel Reardon, *University of Calgary / AppCensus Inc.;* Álvaro Feal, *IMDEA Networks Institute / Universidad Carlos III Madrid;* Primal Wijesekera, *U.C. Berkeley / ICSI;* Amit Elazari Bar On, *U.C. Berkeley;* Narseo Vallina-Rodriguez, *IMDEA Networks Institute / ICSI / AppCensus Inc.;* Serge Egelman, *U.C. Berkeley / ICSI / AppCensus Inc.*

Side Channels

Spoiler: Speculative Load Hazards Boost Rowhammer and Cache Attacks 621
Saad Islam and Ahmad Moghimi, *Worcester Polytechnic Institute;* Ida Bruhns and Moritz Krebbel, *University of Luebeck;* Berk Gulmezoglu, *Worcester Polytechnic Institute;* Thomas Eisenbarth, *Worcester Polytechnic Institute and University of Luebeck;* Berk Sunar, *Worcester Polytechnic Institute*

(continued on next page)

Robust Website Fingerprinting Through the Cache Occupancy Channel 639
Anatoly Shusterman, *Ben-Gurion University of the Negev;* Lachlan Kang, *University of Adelaide;* Yarden Haskal and Yosef Meltser, *Ben-Gurion University of the Negev;* Prateek Mittal, *Princeton University;* Yossi Oren, *Ben-Gurion University of the Negev;* Yuval Yarom, *University of Adelaide and Data61*

Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation 657
Shuai Wang, *HKUST;* Yuyan Bao and Xiao Liu, *Penn State University;* Pei Wang, *Baidu X-Lab;* Danfeng Zhang and Dinghao Wu, *Penn State University*

ScatterCache: Thwarting Cache Attacks via Cache Set Randomization 675
Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard, *Graz University of Technology*

Pythia: Remote Oracles for the Masses 693
Shin-Yeh Tsai, *Purdue University;* Mathias Payer, *EPFL;* Yiying Zhang, *Purdue University*

Mobile Security 2

HideMyApp: Hiding the Presence of Sensitive Apps on Android 711
Anh Pham, *ABB Corporate Research;* Italo Dacosta, *EPFL;* Eleonora Losiouk, *University of Padova;* John Stephan, *EPFL;* Kévin Huguenin, *University of Lausanne;* Jean-Pierre Hubaux, *EPFL*

TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time 729
Feargus Pendlebury, Fabio Pierazzi, and Roberto Jordaney, *King's College London & Royal Holloway, University of London;* Johannes Kinder, *Bundeswehr University Munich;* Lorenzo Cavallaro, *King's College London*

Devils in the Guidance: Predicting Logic Vulnerabilities in Payment Syndication Services through Automated Documentation Analysis 747
Yi Chen, *Institute of Information Engineering, CAS;* Luyi Xing, Yue Qin, Xiaojing Liao, and XiaoFeng Wang, *Indiana University Bloomington;* Kai Chen and Wei Zou, *Institute of Information Engineering, CAS*

Understanding iOS-based Crowdturfing Through Hidden UI Analysis 765
Yeonjoon Lee, Xueqiang Wang, Kwangwuk Lee, Xiaojing Liao, and XiaoFeng Wang, *Indiana University;* Tongxin Li, *Peking University;* Xianghang Mi, *Indiana University*

Crypto Means Cryptocurrencies

Bite: Bitcoin Lightweight Client Privacy using Trusted Execution 783
Sinisa Matetic, Karl Wüst, Moritz Schneider, and Kari Kostiainen, *ETH Zurich;* Ghassan Karame, *NEC Labs;* Srdjan Capkun, *ETH Zurich*

FastKitten: Practical Smart Contracts on Bitcoin 801
Poulami Das, Lisa Eckey, Tommaso Frassetto, David Gens, Kristina Hostáková, Patrick Jauernig, Sebastian Faust, and Ahmad-Reza Sadeghi, *Technische Universität Darmstadt, Germany*

StrongChain: Transparent and Collaborative Proof-of-Work Consensus 819
Pawel Szalachowski, Daniël Reijsbergen, and Ivan Homoliak, *Singapore University of Technology and Design (SUTD);* Siwei Sun, *Institute of Information Engineering and DCS Center, Chinese Academy of Sciences*

Tracing Transactions Across Cryptocurrency Ledgers 837
Haaroon Yousaf, George Kappos, and Sarah Meiklejohn, *University College London*

Intelligence and Vulnerabilities

Reading the Tea leaves: A Comparative Analysis of Threat Intelligence 851
Vector Guo Li, *University of California, San Diego;* Matthew Dunn, *Northeastern University;* Paul Pearce, *Georgia Tech;* Damon McCoy, *New York University;* Geoffrey M. Voelker and Stefan Savage, *University of California, San Diego;* Kirill Levchenko, *University of Illinois Urbana-Champaign*

Towards the Detection of Inconsistencies in Public Security Vulnerability Reports 869
Ying Dong, *University of Chinese Academy of Sciences and The Pennsylvania State University;* Wenbo Guo, Yueqi Chen, and Xinyu Xing, *The Pennsylvania State University and JD Security Research Center;* Yuqing Zhang, *University of Chinese Academy of Sciences;* Gang Wang, *Virginia Tech*

Understanding and Securing Device Vulnerabilities through Automated Bug Report Analysis 887
Xuan Feng, *Beijing Key Laboratory of IOT Information Security Technology, Institute of Information Engineering, CAS, China; School of Cyber Security, University of Chinese Academy of Sciences, China;* Xiaojing Liao and XiaoFeng Wang, *Department of Computer Science, Indiana University Bloomington, USA;* Haining Wang, *Department of Electrical and Computer Engineering, University of Delaware, USA;* Qiang Li, *School of Computer and Information Technology, Beijing Jiaotong University, China;* Kai Yang, Hongsong Zhu, and Limin Sun, *Beijing Key Laboratory of IOT Information Security Technology, Institute of Information Engineering, CAS, China; School of Cyber Security, University of Chinese Academy of Sciences, China*

ATTACK2VEC: Leveraging Temporal Word Embeddings to Understand the Evolution of Cyberattacks 905
Yun Shen, *Symantec Research Labs;* Gianluca Stringhini, *Boston University*

Web Attacks

Leaky Images: Targeted Privacy Attacks in the Web 923
Cristian-Alexandru Staicu and Michael Pradel, *TU Darmstadt*

All Your Clicks Belong to Me: Investigating Click Interception on the Web 941
Mingxue Zhang and Wei Meng, *Chinese University of Hong Kong;* Sangho Lee, *Microsoft Research;* Byoungyoung Lee, *Seoul National University and Purdue University;* Xinyu Xing, *Pennsylvania State University*

What Are You Searching For? A Remote Keylogging Attack on Search Engine Autocomplete 959
John V. Monaco, *Naval Postgraduate School*

Iframes/Popups Are Dangerous in Mobile WebView: Studying and Mitigating Differential Context Vulnerabilities . . 977
GuangLiang Yang, Jeff Huang, and Guofei Gu, *Texas A&M University*

Small World with High Risks: A Study of Security Threats in the npm Ecosystem 995
Markus Zimmermann and Cristian-Alexandru Staicu, *TU Darmstadt;* Cam Tenny, *r2c;* Michael Pradel, *TU Darmstadt*

Crypto Means Cryptographic Attacks

"Johnny, you are fired!" – Spoofing OpenPGP and S/MIME Signatures in Emails. 1011
Jens Müller and Marcus Brinkmann, *Ruhr University Bochum;* Damian Poddebniak, *Münster University of Applied Sciences;* Hanno Böck, *unaffiliated;* Sebastian Schinzel, *Münster University of Applied Sciences;* Juraj Somorovsky and Jörg Schwenk, *Ruhr University Bochum*

Scalable Scanning and Automatic Classification of TLS Padding Oracle Vulnerabilities 1029
Robert Merget and Juraj Somorovsky, *Ruhr University Bochum;* Nimrod Aviram, *Tel Aviv University;* Craig Young, *Tripwire VERT;* Janis Fliegenschmidt and Jörg Schwenk, *Ruhr University Bochum;* Yuval Shavitt, *Tel Aviv University*

The KNOB is Broken: Exploiting Low Entropy in the Encryption Key Negotiation Of Bluetooth BR/EDR 1047
Daniele Antonioli, *SUTD;* Nils Ole Tippenhauer, *CISPA;* Kasper B. Rasmussen, *University of Oxford*

From IP ID to Device ID and KASLR Bypass. 1063
Amit Klein and Benny Pinkas, *Bar Ilan University*

When the Signal is in the Noise: Exploiting Diffix's Sticky Noise 1081
Andrea Gadotti and Florimond Houssiau, *Imperial College London;* Luc Rocher, *Imperial College London and Université catholique de Louvain;* Benjamin Livshits and Yves-Alexandre de Montjoye, *Imperial College London*

IoT Security

FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation 1099
Yaowen Zheng, *Beijing Key Laboratory of IOT Information Security Technology, Institute of Information Engineering, CAS, China; School of Cyber Security, University of Chinese Academy of Sciences, China;* Ali Davanian, Heng Yin, and Chengyu Song, *University of California, Riverside;* Hongsong Zhu and Limin Sun, *Beijing Key Laboratory of IOT Information Security Technology, Institute of Information Engineering, CAS, China; School of Cyber Security, University of Chinese Academy of Sciences, China*

(continued on next page)

Not Everything is Dark and Gloomy: Power Grid Protections Against IoT Demand Attacks 1115
Bing Huang, *The University of Texas at Austin;* Alvaro A. Cardenas, *University of California, Santa Cruz;* Ross Baldick, *The University of Texas at Austin*

Discovering and Understanding the Security Hazards in the Interactions between IoT Devices, Mobile Apps, and Clouds on Smart Home Platforms. 1133
Wei Zhou, *National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences;* Yan Jia, Yao Yao, and Lipeng Zhu, *School of Cyber Engineering, Xidian University; National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences;* Le Guan, *Department of Computer Science, University of Georgia;* Yuhang Mao, *School of Cyber Engineering, Xidian University; National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences;* Peng Liu, *College of Information Sciences and Technology, Pennsylvania State University;* Yuqing Zhang, *National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences; School of Cyber Engineering, Xidian University; State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences*

Looking from the Mirror: Evaluating IoT Device Security through Mobile Companion Apps. 1151
Xueqiang Wang, *Indiana University Bloomington;* Yuqiong Sun and Susanta Nanda, *Symantec Research Labs;* XiaoFeng Wang, *Indiana University Bloomington*

All Things Considered: An Analysis of IoT Devices on Home Networks 1169
Deepak Kumar, *University of Illinois at Urbana-Champaign;* Kelly Shen and Benton Case, *Stanford University;* Deepali Garg, Galina Alperovich, Dmitry Kuznetsov, and Rajarshi Gupta, *Avast Software s.r.o.;* Zakir Durumeric, *Stanford University*

OS Security

KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities 1187
Wei Wu, *Institute of Information Engineering, Chinese Academy of Sciences; Pennsylvania State University; School of Cybersecurity, University of Chinese Academy of Sciences;* Yueqi Chen and Xinyu Xing, *Pennsylvania State University;* Wei Zou, *Institute of Information Engineering, Chinese Academy of Sciences; School of Cybersecurity, University of Chinese Academy of Sciences*

PeX: A Permission Check Analysis Framework for Linux Kernel 1205
Tong Zhang, *Virginia Tech;* Wenbo Shen, *Zhejiang University;* Dongyoon Lee, *Stony Brook University;* Changhee Jung, *Purdue University;* Ahmed M. Azab and Ruowen Wang, *Samsung Research America*

ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK) 1221
Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg, *Max Planck Institute for Software Systems, Saarland Informatics Campus*

SafeHidden: An Efficient and Secure Information Hiding Technique Using Re-randomization. 1239
Zhe Wang and Chenggang Wu, *State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, University of Chinese Academy of Sciences;* Yinqian Zhang, *The Ohio State University;* Bowen Tang, *State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, University of Chinese Academy of Sciences;* Pen-Chung Yew, *University of Minnesota at Twin-Cities;* Mengyao Xie, Yuanming Lai, and Yan Kang, *State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, University of Chinese Academy of Sciences;* Yueqiang Cheng, *Baidu USA;* Zhiping Shi, *The Capital Normal University*

Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. 1257
Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin, *The Ohio State University;* Yan Solihin, *University of Central Florida*

Phishing and Scams

Detecting and Characterizing Lateral Phishing at Scale 1273
Grant Ho, *UC Berkeley and Barracuda Networks;* Asaf Cidon, *Barracuda Networks and Columbia University;* Lior Gavish and Marco Schweighauser, *Barracuda Networks;* Vern Paxson, *UC Berkeley and ICSI;* Stefan Savage and Geoffrey M. Voelker, *UC San Diego;* David Wagner, *UC Berkeley*

High Precision Detection of Business Email Compromise 1291
Asaf Cidon, *Barracuda Networks and Columbia University;* Lior Gavish, Itay Bleier, Nadia Korshun, Marco Schweighauser, and Alexey Tsitkin, *Barracuda Networks*

Cognitive Triaging of Phishing Attacks 1309
Amber van der Heijden and Luca Allodi, *Eindhoven University of Technology*

Users Really Do Answer Telephone Scams 1327
Huahong Tu, *University of Maryland;* Adam Doupé, *Arizona State University;* Ziming Zhao, *Rochester Institute of Technology;* Gail-Joon Ahn, *Arizona State University and Samsung Research*

Platforms in Everything: Analyzing Ground-Truth Data on the Anatomy and Economics of Bullet-Proof Hosting . . 1341
Arman Noroozian, *TU Delft;* Jan Koenders and Eelco van Veldhuizen, *Dutch National High-Tech Crime Unit;* Carlos H. Ganan, *TU Delft;* Sumayah Alrwais, *King Saud University and International Computer Science Institute;* Damon McCoy, *New York University;* Michel van Eeten, *TU Delft*

Distributed System Security + Verifying Hardware

Protecting Cloud Virtual Machines from Hypervisor and Host Operating System Exploits 1357
Shih-Wei Li, John S. Koh, and Jason Nieh, *Columbia University*

WAVE: A Decentralized Authorization Framework with Transitive Delegation 1375
Michael P Andersen, Sam Kumar, Moustafa AbdelBaky, Gabe Fierro, John Kolb, Hyung-Sin Kim, David E. Culler, and Raluca Ada Popa, *University of California, Berkeley*

in-toto: Providing farm-to-table guarantees for bits and bytes 1393
Santiago Torres-Arias, *New York University;* Hammad Afzali, *New Jersey Institute of Technology;* Trishank Karthik Kuppusamy, *Datadog;* Reza Curtmola, *New Jersey Institute of Technology;* Justin Cappos, *New York University*

IODINE: Verifying Constant-Time Execution of Hardware 1411
Klaus v. Gleissenthall, Rami Gökhan Kıcı, Deian Stefan, and Ranjit Jhala, *University of California, San Diego*

VRASED: A Verified Hardware/Software Co-Design for Remote Attestation 1429
Ivan De Oliveira Nunes, *University of California, Irvine;* Karim Eldefrawy, *SRI International;* Norrathep Rattanavipanon, *University of California, Irvine;* Michael Steiner, *Intel;* Gene Tsudik, *University of California, Irvine*

Crypto Means Cryptography

Mobile Private Contact Discovery at Scale. 1447
Daniel Kales and Christian Rechberger, *Graz University of Technology;* Thomas Schneider, Matthias Senker, and Christian Weinert, *TU Darmstadt*

EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats 1465
Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, and Nikhil Swamy, *Microsoft Research;* Tej Chajed, *MIT;* Nadim Kobeissi, *Inria Paris;* Jonathan Protzenko, *Microsoft Research*

Blind Bernoulli Trials: A Noninteractive Protocol For Hidden-Weight Coin Flips 1483
Emma Connor and Max Schuchard, *University of Tennessee*

XONN: XNOR-based Oblivious Deep Neural Network Inference 1501
M. Sadegh Riazi and Mohammad Samragh, *UC San Diego;* Hao Chen, Kim Laine, and Kristin Lauter, *Microsoft Research;* Farinaz Koushanfar, *UC San Diego*

JEDI: Many-to-Many End-to-End Encryption and Key Delegation for IoT 1519
Sam Kumar, Yuncong Hu, Michael P Andersen, Raluca Ada Popa, and David E. Culler, *University of California, Berkeley*

Passwords

Birthday, Name and Bifacial-security: Understanding Passwords of Chinese Web Users 1537
Ding Wang and Ping Wang, *Peking University;* Debiao He, *Wuhan University;* Yuan Tian, *University of Virginia*

Protecting accounts from credential stuffing with password breach alerting 1555
Kurt Thomas, Jennifer Pullman, Kevin Yeo, Ananth Raghunathan, Patrick Gage Kelley, Luca Invernizzi, Borbala Benko, Tadek Pietraszek, and Sarvar Patel, *Google;* Dan Boneh, *Stanford;* Elie Bursztein, *Google*

(continued on next page)

Probability Model Transforming Encoders Against Encoding Attacks 1573
Haibo Cheng, Zhixiong Zheng, Wenting Li, and Ping Wang, *Peking University;* Chao-Hsien Chu, *Pennsylvania State University*

Cryptocurrency Scams

The Art of The Scam: Demystifying Honeypots in Ethereum Smart Contracts 1591
Christof Ferreira Torres, Mathis Steichen, and Radu State, *University of Luxembourg*

The Anatomy of a Cryptocurrency Pump-and-Dump Scheme 1609
Jiahua Xu, *École polytechnique fédérale de Lausanne (EPFL);* Benjamin Livshits, *Imperial College London*

Inadvertently Making Cyber Criminals Rich: A Comprehensive Study of Cryptojacking Campaigns at Internet Scale 1627
Hugo L.J. Bijmans, Tim M. Booij, and Christian Doerr, *Delft University of Technology*

Web Defenses

Rendered Private: Making GLSL Execution Uniform to Prevent WebGL-based Browser Fingerprinting 1645
Shujiang Wu, Song Li, and Yinzhi Cao, *Johns Hopkins University;* Ningfei Wang, *Lehigh University*

Site Isolation: Process Separation for Web Sites within the Browser 1661
Charles Reis, Alexander Moshchuk, and Nasko Oskov, *Google*

Everyone is Different: Client-side Diversification for Defending Against Extension Fingerprinting 1679
Erik Trickel, *Arizona State University;* Oleksii Starov, *Stony Brook University;* Alexandros Kapravelos, *North Carolina State University;* Nick Nikiforakis, *Stony Brook University;* Adam Doupé, *Arizona State University*

Less is More: Quantifying the Security Benefits of Debloating Web Applications 1697
Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis, *Stony Brook University*

The Web's Identity Crisis: Understanding the Effectiveness of Website Identity Indicators 1715
Christopher Thompson, Martin Shelton, Emily Stark, Maximilian Walker, Emily Schechter, and Adrienne Porter Felt, *Google*

Software Security

RAZOR: A Framework for Post-deployment Software Debloating 1733
Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee, *Georgia Institute of Technology*

Back to the Whiteboard: a Principled Approach for the Assessment and Design of Memory Forensic Techniques . . 1751
Fabio Pagani and Davide Balzarotti, *EURECOM*

Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences 1769
Kangjie Lu, Aditya Pakki, and Qiushi Wu, *University of Minnesota*

DEEPVSA: Facilitating Value-set Analysis with Deep Learning for Postmortem Program Analysis. 1787
Wenbo Guo, Dongliang Mu, and Xinyu Xing, *The Pennsylvania State University;* Min Du and Dawn Song, *University of California, Berkeley*

CONFIRM: Evaluating Compatibility and Relevance of Control-flow Integrity Protections for Modern Software . 1805
Xiaoyang Xu, Masoud Ghaffarinia, Wenhao Wang, and Kevin W. Hamlen, *University of Texas at Dallas;* Zhiqiang Lin, *Ohio State University*

Privacy

Point Break: A Study of Bandwidth Denial-of-Service Attacks against Tor 1823
Rob Jansen, *U.S. Naval Research Laboratory;* Tavish Vaidya and Micah Sherr, *Georgetown University*

No Right to Remain Silent: Isolating Malicious Mixes 1841
Hemi Leibowitz, *Bar-Ilan University, IL;* Ania M. Piotrowska and George Danezis, *University College London, UK;* Amir Herzberg, *University of Connecticut, US*

On (The Lack Of) Location Privacy in Crowdsourcing Applications 1859
Spyros Boukoros, *TU-Darmstadt;* Mathias Humbert, *Swiss Data Science Center* (*ETH Zurich, EPFL*)*;* Stefan Katzenbeisser, *TU-Darmstadt, University of Passau;* Carmela Troncoso, *EPFL*

Utility-Optimized Local Differential Privacy Mechanisms for Distribution Estimation 1877
Takao Murakami and Yusuke Kawamoto, *AIST*

Evaluating Differentially Private Machine Learning in Practice 1895
Bargav Jayaraman and David Evans, *University of Virginia*

Fuzzing

FUZZIFICATION: Anti-Fuzzing Techniques 1913
Jinho Jung, Hong Hu, David Solodukhin, and Daniel Pagan, *Georgia Institute of Technology;* Kyu Hyung Lee, *University of Georgia;* Taesoo Kim, *Georgia Institute of Technology*

ANTIFUZZ: Impeding Fuzzing Audits of Binary Executables 1931
Emre Güler, Cornelius Aschermann, Ali Abbasi, and Thorsten Holz, *Ruhr-Universität Bochum*

MOPT: Optimized Mutation Scheduling for Fuzzers 1949
Chenyang Lyu, *Zhejiang University;* Shouling Ji, *Zhejiang University & Alibaba-Zhejiang University Joint Research Institute of Frontier Technologies;* Chao Zhang, *BNRist & INSC, Tsinghua University;* Yuwei Li, *Zhejiang University;* Wei-Han Lee, *IBM Research;* Yu Song, *Zhejiang University;* Raheem Beyah, *Georgia Institute of Technology*

EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers 1967
Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, and Chijin Zhou, *Tsinghua University;* Xun Jiao, *Villanova University;* Zhuo Su, *Tsinghua University*

GRIMOIRE: Synthesizing Structure while Fuzzing 1985
Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz, *Ruhr-Universität Bochum*

A Study of the Feasibility of Co-located App Attacks against BLE and a Large-Scale Analysis of the Current Application-Layer Security Landscape

Pallavi Sivakumaran
Information Security Group
Royal Holloway University of London
Email: pallavi.sivakumaran.2012@rhul.ac.uk

Jorge Blasco
Information Security Group
Royal Holloway University of London
Email: jorge.blascoalis@rhul.ac.uk

Abstract

Bluetooth Low Energy (BLE) is a fast-growing wireless technology with a large number of potential use cases, particularly in the IoT domain. Increasingly, these use cases require the storage of sensitive user data or critical device controls on the BLE device, as well as the access of this data by an augmentative mobile application. Uncontrolled access to such data could violate user privacy, cause a device to malfunction, or even endanger lives. The BLE standard provides security mechanisms such as pairing and bonding to protect sensitive data such that only authenticated devices can access it. In this paper we show how unauthorized co-located Android applications can access pairing-protected BLE data, without the user's knowledge. We discuss mitigation strategies in terms of the various stakeholders involved in this ecosystem, and argue that at present, the only possible option for securing BLE data is for BLE developers to implement remedial measures in the form of application-layer security between the BLE device and the Android application. We introduce BLECryptracer, a tool for identifying the presence of such application-layer security, and present the results of a large-scale static analysis over 18,900+ BLE-enabled Android applications. Our findings indicate that over 45% of these applications do not implement measures to protect BLE data, and that cryptography is sometimes applied incorrectly in those that do. This implies that a potentially large number of corresponding BLE peripheral devices are vulnerable to unauthorized data access.

1 Introduction

Bluetooth is a well-known technology standard for wireless data transfer, currently deployed in billions of devices worldwide [37]. A more recent addition to the Bluetooth standard is Bluetooth Low Energy (BLE), which differs from Classic Bluetooth in that it incorporates a simplified version of the Bluetooth stack and targets low-energy, low-cost devices.

Its focus on resource-constrained devices has made BLE highly suited for IoT applications [18], including personal health/fitness monitoring [22], asset tracking [8], vehicular management [13], and home automation [27]. Most of these use cases augment the functionality of the BLE device with a mobile application. This application may need to read or write sensitive or critical data on the BLE device (for example, glucose measurement values stored by a continuous glucose meter, or a field that controls a door's locking mechanism in a smart home security system). To ensure privacy and security/safety, measures should be taken to protect such data from being accessed by unauthorized entities.

The Bluetooth specification provides means for restricting access to BLE data via *pairing* and *bonding*, which are mechanisms for establishing an authenticated transport between two communicating devices. However, when multiple applications reside on a single host, as is the case with mobile devices, there is potential for a malicious application to abuse a trusted relationship between the host and the device that was initiated by an authorized application [31].

In this work, we show how a malicious application could take advantage of the BLE communication model on Android to read and write pairing-protected data on a BLE device without the user's knowledge. We also show that these unauthorized applications may be able to do so while requesting minimal permissions, thereby making them appear less invasive than even an authorized application.

We discuss various strategies, in terms of the different stakeholders involved, that can be used to secure BLE data against such unauthorized access. We argue that in the current landscape, it is up to the BLE device/application developers to implement application-layer security to protect the data on their devices. We perform a large-scale static analysis of 18,929 BLE-enabled Android applications (filtered down from an original dataset of over 4.6 million applications) to determine how many of them currently employ such protection mechanisms. While the results vary for BLE reads vs. writes, overall they show that more than 45% of the tested applications do not provide cryptography-based application-layer security for BLE data. This number rises to about 70% for those applications that are categorized under "Medical". This information,

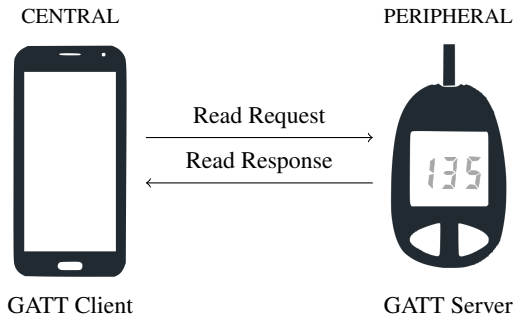


Figure 1: GATT communications between a mobile phone and a BLE-enabled glucometer.

when combined with the download counts for each application, allows us to estimate a lower bound for the number of BLE devices that may be vulnerable to unauthorized data access.

The rest of this paper is structured as follows: Section 2 provides an overview of key BLE concepts, particularly with regard to data access mechanisms and restrictions. We demonstrate unauthorized BLE data access in Section 3. This section also discusses stakeholders and possible mitigation strategies. Section 4 details our marketplace application analysis and examines the results. Related work is described in Section 5, and Section 6 provides our concluding remarks.

2 Background

Two devices that communicate using BLE will operate in an asymmetric configuration, with the more powerful device, referred to as the *central*, taking on most of the resource-intensive work. The resource-constrained device is termed the *peripheral* and performs tasks that are designed to consume fewer resources.

2.1 Data Access on BLE Devices

BLE, unlike Classic Bluetooth, can only handle discrete data known as *attributes*. Attributes are stored and accessed according to rules specified by the *Attribute Protocol* (ATT) and the *Generic Attribute Profile* (GATT), both of which are defined in the Bluetooth standard. There are different types of attributes, of which *characteristics* are the most relevant for our analysis, as they hold the actual data of interest. Related characteristics are grouped into *services*, which are exposed to connected devices [11].

When one BLE-enabled device wants to access attributes on another BLE device, the device that initiates the exchange takes on the role of GATT *client* and the other acts as the GATT *server*. In this paper, we focus on the scenario where the BLE peripheral (e.g., a glucose meter), acts as the server, and a mobile phone acts as the client, as shown in Figure 1.

2.2 BLE Attribute Permissions

Every attribute has associated with it three permissions that control how it may be accessed: (1) *Access permissions* define whether an attribute can be read and/or written. (2) *Authentication permissions* indicate the level of authentication and encryption that needs to be applied to the transport between the two devices before the attribute can be accessed. (3) *Authorization permissions* specify whether end-user authorization is required for access.

When a GATT client sends a read or write request for an attribute to a GATT server, the server will check the request against the permissions for that attribute, to determine whether the requested access mechanism is allowed and whether the client is authenticated and/or authorized, if required. An attribute is only readable or writable if its access permissions specify it to be so. In the case of authentication permissions, if the attribute requires an authenticated or encrypted link before it can be accessed (referred to as a "pairing-protecte" attribute in this paper), and if such a link is not present when the access request is made, then the server responds with an `Insufficient Authentication/Encryption` message. At this point, the client can initiate the pairing process to authenticate and encrypt the transport. If this process completes successfully, the server will fulfill subsequent requests made by the client. This procedure for handling authentication requirements is well-defined in the Bluetooth specification. Authorization requirements, on the other hand, are implementation-specific and largely left up to developers.

Once two devices complete the pairing process, they typically go through an additional *bonding* process, during which long-term keys are established. This prevents the need for going through the pairing process again if they disconnect and subsequently reconnect, provided they retain the long-term keys. Upon re-connection, the link encryption process will be initiated using the stored keys. Keys normally remain on the devices unless the devices are reset or manually unpaired by the user.

3 BLE Co-located Application Attacks

In this section, we show how *any* application on an Android device can access pairing-protected attributes from a BLE peripheral, even when the pairing process was initiated by a different application. We then explore various mitigation strategies that are available to different stakeholders in the BLE ecosystem.

These attacks were also explored by Naveed et al. in 2014, for Classic Bluetooth [31]. We show that the problem remains on newer versions of Android, and also that the situation is worse for BLE, as one of our attacks enables fewer restrictions for access and requires fewer permissions of the malicious application than even of the official application.

3.1 Attack Mechanisms

We describe two attacks: the first shows that pairing-protected data can be accessed by unauthorized applications, while the second refines the attack and reduces the number of permissions required by the unauthorized application. We use two Android applications to describe the attacks: One application that is expected to be able to connect to the BLE device and access its data ("OfficialApp"), and a different application that should *not* be able to access pairing-protected data from the device ("AttackApp"). We conducted our experiments on an Alcatel Pixi 4 mobile phone, running Android 6.0, and on a Google Pixel XL, running Android 8.1.0. Version 6.0 was the most widely-deployed release [2], while 8.1.0 was the latest stable release, as of 01 Aug 2018.

3.1.1 Attack 1: System-wide Pairing Credentials

This attack demonstrates that the BLE credentials that are stored on an Android device are implicitly available to *all* applications on the device, rather than just the application that originally triggered the pairing.

When the OfficialApp connects to the BLE device and attempts to access a pairing-protected characteristic, the resulting exchange will trigger the Android OS into initiating the pairing and bonding process (as depicted in the upper block in Figure 2). The resultant keys are associated with the link between the Android and BLE devices, rather than between the BLE device and the OfficialApp (which actually triggered the pairing). Therefore, once bonding completes, when the AttackApp scans and connects to the BLE device, the Android OS completes the connection process and automatically initiates link encryption with the keys that were generated during the previous bonding process (lower block in Figure 2). This enables the AttackApp to have the same level of access to the pairing-protected data on the device as the OfficialApp, but without the need for initiating pairing.

A key point to note here is that, not only is the unauthorized AttackApp able to access potentially sensitive information from the BLE device, but also the user is likely to be unaware of the fact that this data access is taking place, as there is no indication during link re-encryption and subsequent attribute access.

3.1.2 Attack 2: Reuse of Connection

Our second attack exploits the fact that, on Android, a BLE peripheral can be used concurrently by multiple applications [32]. In this attack, the AttackApp does not scan for BLE devices. It instead searches for connected BLE devices using the `BluetoothManager.getConnectedDevices()` API call, with `BluetoothProfile.GATT` as the argument. If the OfficialApp happens to be in communication with the BLE device at the same time, this call will return a list with a reference to the connected BLE device. The AttackApp is then able to directly connect to the GATT server and read and write to the characteristics on it (including those that are pairing-protected), without the need for creating a new connection to

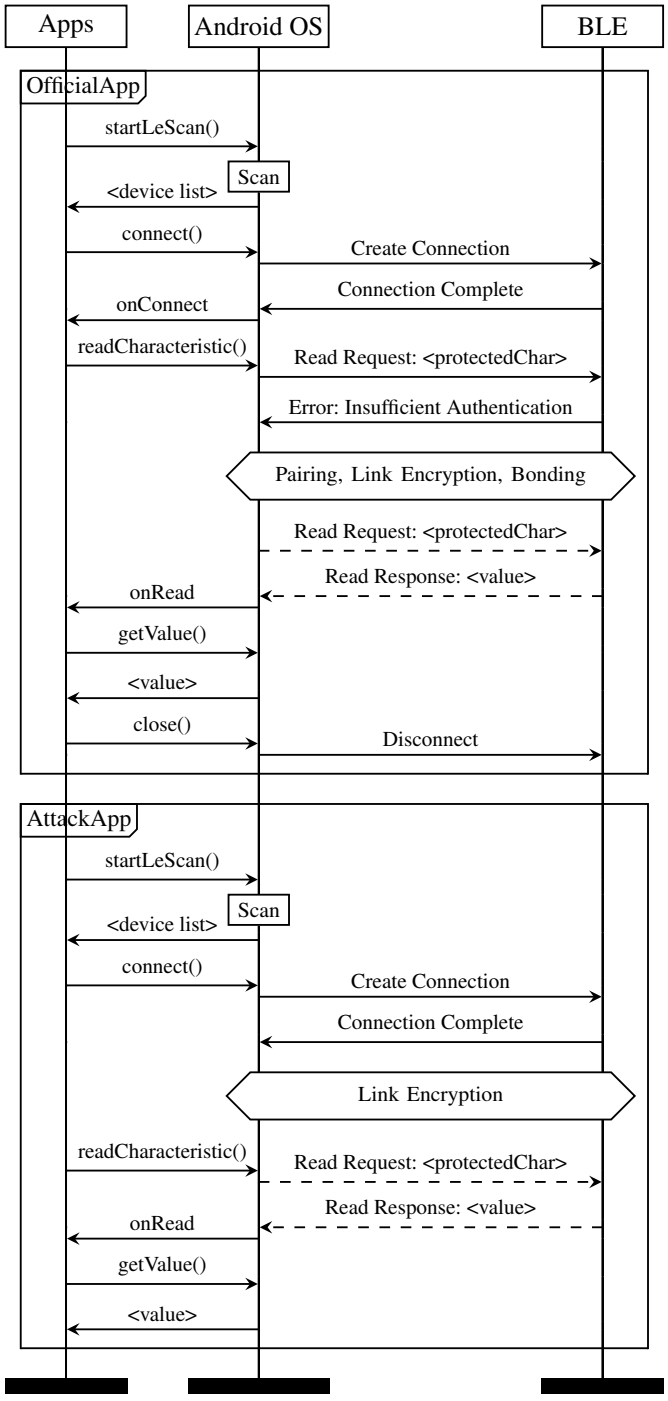


Figure 2: Attack 1 - Illustrative message exchange depicting access of pairing-protected data by unauthorized application. *Note:* Dashed lines indicate encrypted traffic.

the peripheral. This again is done surreptitiously, without the user being aware of the data access. An illustrative message flow where the AttackApp writes to a protected characteristic on the BLE device (which the OfficialApp subsequently reads) has been depicted in Figure 3.

An interesting observation from this attack is a subtle but relevant impact it has on user awareness, due to the different permissions that need to be requested by the two applications. Since both applications access data from a GATT server, they both require `BLUETOOTH` permissions. In this attack scenario, because the OfficialApp scans for the BLE device before it connects to it, it also needs to request the `BLUETOOTH_ADMIN` permission. Both `BLUETOOTH` and `BLUETOOTH_ADMIN` are "normal" permissions that are granted automatically by the Android operating system after installation, without any need for user interaction. However, due to restrictions imposed from Android version 6.0 onward, the OfficialApp also needs to request `LOCATION` permissions to invoke the BLE scanner without a filter (i.e., to scan for all nearby devices instead of a particular device). These permissions are classed as "dangerous" and will prompt the system to display a confirmation dialog box the first time they are required. Because the AttackApp merely has to query the Android OS for a list of already connected devices, it does not require these additional permissions. This makes the AttackApp appear to be less invasive in the eyes of a user, since it does not request any permission that involves user privacy. This could play a part in determining the volume of downloads for a malicious application. For example, a malicious application that masquerades as a gaming application, and which does not request any dangerous permissions, may be more likely to be downloaded by end users as opposed to one that requests location permissions.

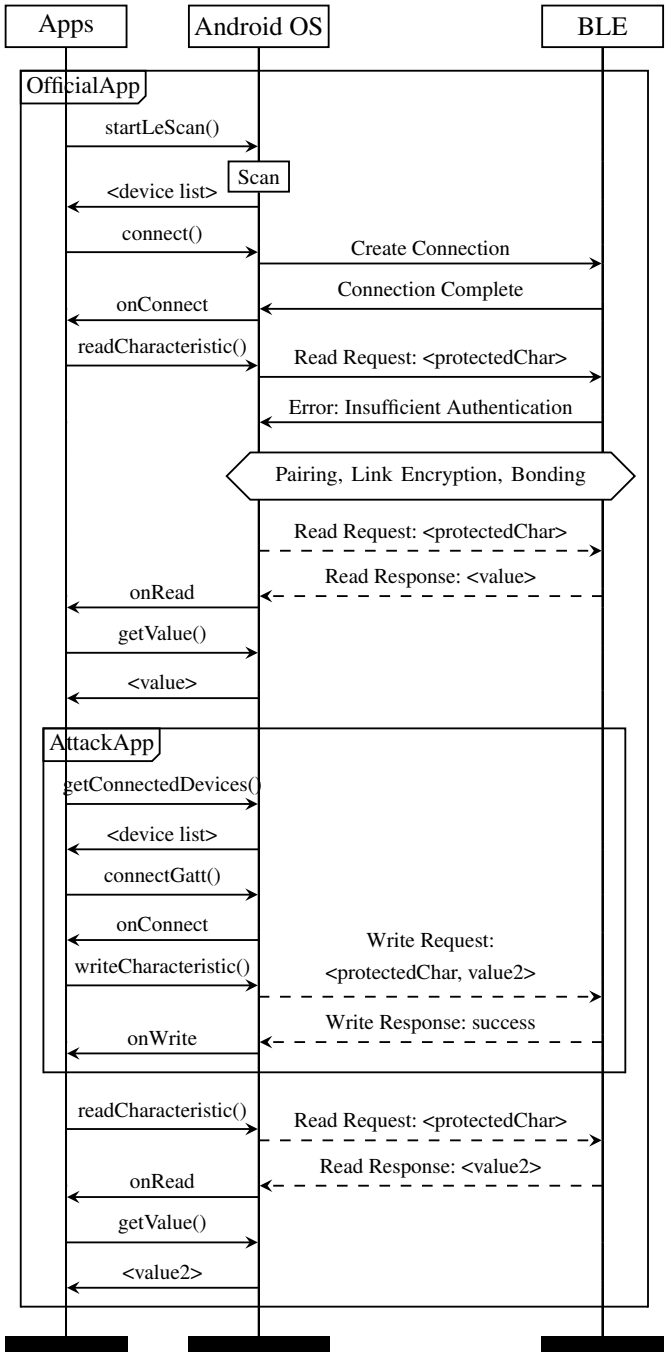


Figure 3: Attack 2 - Illustrative message exchange depicting the access of pairing-protected data by reusing an existing connection. *Note:* Dashed lines indicate encrypted traffic.

3.2 Discussion

In this section we discuss the impact of our findings, compare them with the Classic Bluetooth case, and mention some attack limitations.

3.2.1 Implications of Attack

In both of our experiments, the AttackApp was able to read and write pairing-protected data from the BLE device. The simplest form of attack would then be for a malicious application to perform unauthorized reads of personal user data (as an example) and relay this to a remote server.

We verified the practicability of this attack by testing a BLE-enabled fitness tracker that implemented the Bluetooth Heart Rate Service. According to the service specification, characteristics within this service are only supposed to be protected by pairing [9]. However, we observed that the pairing employed by the device appeared to be a non-standard implementation, and also that access to the Heart Rate Measurement characteristic was "locked" and had to be "unlocked" by first writing to certain other characteristics on the tracker. Despite this, we found that by deploying our second attack, our AttackApp was able to obtain Heart Rate Measurement readings without the need for performing any "unlocking".

This is because the AttackApp connects to the GATT server by reusing an existing connection that was initiated by the official application. The unlocking procedure would therefore already have been performed for that connection by the official application. This result shows that artificially restricting access to data using non-cryptographic means will not be effective. We notified the device developer of this issue on 01 Nov 2018, but have not yet received a response.

It should be noted that the above attack could be used by a malicious application to target other sensitive health information such as ECG, glucose or blood pressure measurements from vulnerable BLE devices, to build up a profile on a user's health. Further, Smart Home devices and BLE-enabled vehicles may hold information on a user's habits and lifestyle (e.g., time at home, alcohol consumption, driving speed), and could be exploited. It may also be possible for a malicious application to overwrite values on the BLE device, such that the written data either causes unexpected behavior on the device, or is read back by the legitimate application, thereby giving the user an incorrect view of the data on the peripheral. For example, it may be possible to update the peripheral's firmware via GATT writes. If this mechanism is not suitably protected, then a malicious application could potentially install malicious firmware onto the BLE device, as we demonstrate in Section 4.6.

3.2.2 Comparison with Classic Bluetooth

In their experiments with Classic Bluetooth, Naveed et al. found that an unauthorized Android application would not be able to obtain data from a Classic Bluetooth device if the authorized application had already established a socket connection with the device, as only one application can be in communication with the device at one time. Therefore, a malicious application would either require some side-channel information in order to determine the correct moment for data access, or would need to interfere with the existing connection, thereby potentially alerting the user [31]. This limits the attack window for the malicious application. Our experiments show that this is not the case with BLE communication channels. With BLE, there are no socket connections and if the official application has established a connection with the BLE device, then this connection can be utilized by any application that is running on the Android device. That is, a malicious application does not have to wait for the authorized application to disconnect before it can access data.

3.2.3 Attack Limitations

The main limitation for the AttackApp in the case of the first attack is that it requires the `BLUETOOTH` and `BLUETOOTH_ADMIN` permissions in its manifest, and also needs to explicitly request `LOCATION` permissions at first runtime in order to be able to invoke the BLE scanner. This enables the AttackApp to connect to the BLE device regardless of whether or not another application is also connected, but increases the risk of raising a user's suspicions.

In the second attack scenario, the obvious limitation for the AttackApp that requests only the `BLUETOOTH` permission is that the application will only be able to access data from the BLE peripheral when the peripheral is already in a connection with (another application on) the Android device. That is, data access will have to be opportunistic. This can be achieved, for example, by periodically polling for a list of connected devices.

3.3 Stakeholders, Mitigation Strategies and Awareness

In this section, we discuss potential mitigating strategies that different stakeholders within the BLE ecosystem could implement in order to prevent the attacks detailed in Section 3.1. We consider the Bluetooth Special Interest Group (SIG), Android (i.e., Google), and BLE device/application developers as stakeholders.

3.3.1 Bluetooth SIG

The Bluetooth SIG is the group that is responsible for defining and maintaining the Bluetooth standard, which provides details on pairing, bonding and BLE attribute permissions. The SIG also defines various BLE services, including some that handle user health information, e.g., the Heart Rate Service and the Continuous Glucose Monitoring Service. The Bluetooth specifications for these services require only pairing as a protection mechanism for the characteristics that hold health-related measurements [9, 10]. This protection is intended to avoid man-in-the-middle attacks and eavesdropping. However, as shown in Section 3.1, pairing will not prevent unauthorized Android applications from accessing the sensitive data held in these characteristics.

This issue could be avoided by modifying the Bluetooth specification and introducing specific security measures for protecting data at higher layers. However, this would require changes to all devices within the ecosystem, which may not be feasible due to the sheer volume of devices and applications currently available, and which could lead to fragmentation and reduced interoperability. Despite this, we believe that developers accessing Bluetooth documentation should at least be made aware of the risks involved, and have therefore notified the SIG via their Support Request Form (17 Dec 2018). We were informed (19 Dec 2018) that the case had been assigned to the appropriate team for assessment.

3.3.2 Android

Allowing all applications on an Android device to share BLE communication channels and long-term keys may well be by

design, particularly since the BLE standard does not provide explicit mechanisms for selectively allowing or denying access to data based on the source application. This model may work in some situations, for example on a platform where all applications originate from the same trusted source. However, the Android ecosystem is such that, many of the applications on a device are from different and potentially untrusted sources. In this scenario, providing all applications with access to a common BLE transport opens up possibilities for attack, as we have demonstrated.

One option to eliminate the problem is to modify how Android handles BLE communication channels. The modification would require some form of association between BLE credentials and the application that triggers the pairing/bonding process. Each data access request would then be checked against the permissions associated with the requesting application. This approach is favoured by Naveed et al., who propose a re-architected Android framework which will create a *bonding policy* when an application triggers pairing with a Bluetooth device [31]. This strategy has the advantage that Bluetooth devices will be protected by default from unauthorized access to their data. Further, assuming a suitably strong pairing mechanism is used, a minimum level of security will also be guaranteed. However, not only will the operating system(s) need to be modified, but also a mechanism will be required for ensuring that all users' mobile devices are updated. Otherwise, it is fairly likely that this measure will result in a fragmented ecosystem, with some devices running the modified operating system with protection mechanisms, and others running older versions of the OS with no protection.

Regardless of whether or not the above measure is implemented, we believe that developers should be made aware of the possibility of unauthorized applications accessing their BLE device data. At present, Android does not mention the issue in its Developer Guide [3]. In fact, to the best of the authors' knowledge, there is only a single document, from a BLE chipset manufacturer, which explicitly references the fact that multiple Android applications can simultaneously use a connection to a BLE device [32]. Apart from this, the risks of "system-wide pairing" have been mentioned in a specification issued by the Fast ID Online (FIDO) Alliance, without specific reference to mobile platforms [20].

We submitted an issue to the Android Security Team on 02 Nov 2018, focusing on the need of clear documentation so that developers are aware of the need for implementing additional protection measures if they are handling sensitive BLE data. The issue was reviewed by the security team and rated as Moderate severity (16 May 2019), based on Android's severity assessment matrix [5].

3.3.3 BLE Device/Application Developers

Despite the BLE stack containing an application layer, it could be argued that BLE is commonly viewed as a lower-layer technology for providing wireless communication capabilities, on top of which higher-layer technologies operate [12, 38]. This would result in the responsibility of securing user data being transferred from the Bluetooth SIG or Android to the BLE application/device developers. At any rate, this is the only mechanism available at present for protecting data against access by co-located applications.

That is, rather than relying solely on the pairing provided by the underlying operating system, developers can implement end-to-end security from their Android application to the BLE peripheral firmware. It may be possible to achieve such behavior via BLE authorization permissions, because even though their purpose is to specify a requirement for end user authorization, the behavior of BLE devices when encountering authorization requirements is implementation-specific. Most modern BLE chipsets implement authorization capabilities by intercepting read/write requests to the protected characteristics, and allowing for developer-specified validation.

One advantage of this method is that it gives the developer complete control over the strength of protection that is applied to BLE device data, as well as over the timings of security updates. However, leaving the implementation of security to the developer runs the risk of cryptography being applied improperly, thereby leaving the data vulnerable [17]. For existing developments, retrofitting application-layer security would mean that both an update for the Android application and a firmware update for the BLE device would be required, and there is a risk that the BLE firmware update procedure itself may not be secure [6].

Due to the lack of clear guidelines regarding attribute security in both the Android Developer Guide and the Bluetooth specification, it is also possible that developers implement no security at all, due to an assumption that protection will be handled by pairing. In the next section, we test this assertion of a lack of developer awareness by exploring the current state of application-layer security deployments via a large-scale analysis of BLE-enabled Android applications.

4 Marketplace Application Analysis

As evidenced by our experiments, it is fairly straightforward for any Android application to connect to a BLE device and read or write pairing-protected data. As discussed in Section 3.3, the only strategy available at present is for developers to implement application-layer security, typically in the form of cryptographic protection, between the Android application and the BLE peripheral.

In this section, we identify the proportion of applications that do *not* implement such security mechanisms, to demonstrate a possible lack of awareness surrounding the issue, and to be able to estimate the number of devices that are potentially vulnerable to the types of attack shown in Section 3.1.

Table 1: APKs and Downloads per Google Play Category

Category	APKs [packages]	Downloads(mn)
Health & Fitness	3012 [1263]	344.95
Lifestyle	1501 [1006]	52.60
Business	1489 [950]	39.62
Tools	1428 [891]	6308.62
Sports	1268 [685]	17.74
Travel & Local	948 [545]	31.83
Productivity	526 [305]	43.05
Entertainment	446 [284]	128.41
Music & Audio	406 [239]	51.48
Education	313 [225]	3.35
Shopping	383 [190]	144.87
Maps & Navigation	348 [181]	33.21
Medical	407 [177]	5.68
Communication	395 [146]	755.89
Finance	259 [126]	96.38
Auto & Vehicles	236 [119]	4.13
Food & Drink	146 [87]	6.23
Photography	114 [80]	45.78
Social	203 [77]	663.43
Other	746 [516]	258.41

[a] We make the assumption that all versions of an application fall under the same category.
[b] Some APKs within the dataset are no longer available on Google Play and hence, have no corresponding category. These have not been included.

To identify the presence of application-layer security, there are two possible targets for analysis: BLE peripheral firmware or Android applications. At present, there is no public repository of BLE firmware, which means that the firmware would need to be obtained from the peripherals themselves. This would necessitate the purchase of a large number of devices and would not be financially viable. Further, reverse-engineering and analyzing BLE firmware is not straightforward, as the firmware image is usually a `.hex` file, which can typically only be converted to binary or assembly. Android APKs, on the other hand, are easier to obtain, and a number of decompilers exist that allow for conversion of APKs to a human-readable format.

We therefore target Android applications for our analysis and perform the following: (1) obtain a substantial dataset of BLE-enabled Android APKs, (2) determine the BLE method calls and the cryptography libraries of interest, and (3) define a mechanism to determine whether BLE reads and writes make use of cryptographically processed data. We then apply this mechanism to our dataset and analyze the results.

4.1 APK Dataset

We obtained our dataset from the AndroZoo project [1]. This is an online repository that has been made available for research purposes and which contains APKs from several different application marketplaces. We focus on only those APKs that were retrieved from the official Google Play store, which nevertheless resulted in a sizeable dataset of over 4.6 million APKs. This dataset includes multiple versions for each application, as well as applications that are no longer available on the marketplace. We performed our analysis over the entire dataset, rather than focusing on only those APKs that are currently available on Google Play. This was in part because older versions of an application may still be residing on users' devices, and in part to be able to identify trends in application-layer security deployments over time.

As we are only interested in those applications that perform BLE attribute access, and because such access always requires communicating with the GATT server on the BLE peripheral, the APKs were filtered by the `BLUETOOTH` permission declaration and by calls to the `connectGatt` method, which is called prior to performing any data reads or writes. 18,929 APKs, comprising 11,067 unique packages[1], from the original set of 4,600,000+ APKs satisfy this criteria, and these formed our final dataset.

Application Categories

Applications are categorized in Google Play according to their primary function, such as "Productivity" or "Entertainment", and it may be possible to gauge the sensitivity of the BLE data handled by an application based on the category it falls under. For example, "Health and Fitness" applications are probably more likely to hold personal user data than "Entertainment" applications.

The number of APKs per category has been listed in Table 1 for our dataset. Approximately 23% of the APKs (18% of unique applications) fall under the categories of "Health and Fitness" and "Medical", with a cumulative download count of over 350 million. Note that the disproportionately high volume of downloads for the category "Tools" is due to the Google and Google Play applications, which include BLE capabilities and are installed on most Android devices.

4.2 Identification of BLE Methods and Crypto-Libraries

We perform our analysis against specific BLE methods and crypto-libraries. When considering BLE methods, we focus on those methods that involve data writes and

[1] An Android application may have many versions, each of which will be a separate APK file (with a unique SHA256 hash), but all of which will have the same package name. We use the terms "unique applications" or "unique packages" to denote the set of APKs that contain only the *latest* version of each application.

Table 2: BLE Data Access Methods

Access	Method Signature[a]	#APKs	% of Total Methods[b]
Read	`byte[] getValue ()`	17896	61.58%
	`Integer getIntValue (int, int)`	8051	27.70%
	`String getStringValue (int)`	2313	7.96%
	`Float getFloatValue (int, int)`	800	2.75%
Write	`boolean setValue (byte[] )`	16198	70.49%
	`boolean setValue (int, int, int)`	5542	24.11%
	`boolean setValue (String)`	627	2.73%
	`boolean setValue (int, int, int, int)`	611	2.66%

[a] All methods are from the class `android.bluetooth.BluetoothGattCharacteristic`.
[b] "% of Total Methods" refers to the percentage of occurrences of a particular method for a particular data access type (i.e., read or write), with respect to all methods that enable the same type of data access.

reads. Such methods have been listed in Table 2, and function as the starting point for our analysis. For data writes, the `BluetoothGattCharacteristic` class within the `android.bluetooth` package has `setValue` methods that set the locally-stored value of a characteristic. This is then written out to the BLE peripheral. For data reads, the same class has `getValue` methods, which return data that is read from the BLE device. In a few APKs that we analyzed, BLE data access methods were also called from within other, vendor-specific libraries. However, we do not include these in our analysis as they are now obsolete.

For cryptography, Android builds on the Java Cryptography Architecture [33] and provides a number of APIs, contained within the `java.security` and `javax.crypto` packages, for integrating security into applications. While it is possible for developers to implement their own algorithms, Android recommends against this [4]. We therefore consider only calls to these two packages as an indication of application-layer security.

4.3 BLECryptracer

Identification of cryptographically-processed BLE data is in essence a taint-analysis problem. For instance, a call to an encryption method will taint the output variable that may later be written to a BLE device. For the purpose of this paper, when analyzing data that is read from a BLE peripheral, we consider the `getValue` variants in Table 2 as sources and the cryptography API calls as sinks. For data that is written to the BLE device, we consider the cryptography API calls as sources and the `setValue` methods as sinks.

There are a number of tools available for performing taint-analysis, such as Flowdroid [7] and Amandroid [40]. However, running a subset of our dataset of APKs through Amandroid (selected because of advantages over Flowdroid and other taint-analysis tools [34]), we found that analysis of a single APK sometimes utilized over 10GB of RAM and took several hours to complete. We also found through manual analysis that many instances of cryptographically-processed data were not identified by Amandroid, especially when the BLE functions were called from third-party libraries. We therefore developed a custom Python analysis tool called BLECryptracer, to analyze *all* calls to BLE `setValue` and `getValue` methods within an APK.

BLECryptracer is developed on top of Androguard [16], an open-source reverse-engineering tool that decompiles an Android APK and enables analysis of its components. Our tool traces values to/from BLE data access functions and determines whether the data has been cryptographically processed. To achieve this, it employs a technique for tracing register values which is sometimes referred to as "slicing" and which has been utilized in several static code analyses [17, 24, 35]. It also traces fields, as well as messages passed via Intents[2] and certain threading functions, e.g., `AsyncTask`. It returns TRUE at the first instance of cryptography that it encounters and FALSE if it is unable to identify any application-layer security with BLE data.

Our tool analyzes BLE reads and writes separately, as the direction of tracing is different in the two cases. It performs three main types of tracing, in the following order:

1. Direct trace - Attempt to identify link between BLE and cryptography functions via direct register value transfers and as immediate results of method invocations.
2. Associated entity trace - If the direct trace does not identify a link between source and sink, analyze abstract/instance methods and other registers used in previously analyzed function calls.
3. "Lenient" trace - If the above methods fail to return a positive result, perform a search through all previously encountered methods (which would have originated from the BLE data access method), to determine if cryptography is used *anywhere* within them.

[2] By matching the `Extra` identifier within the calling method.

Table 3: Accuracy Statistics

Access	Tool	Confidence	App Set[a]	Detected[b]	TP	FP	TN	FN	Precision	Recall	F-measure
Read	Amandroid	N/A	92	49	44	5	10	33	90%	57%	70%
	BLECryptracer	High	92	62	58	4	11	19	94%	75%	83%
		Medium	30	11	7	4	7	12	64%	37%	47%
		Low	19	12	8	4	3	4	67%	67%	67%
Write	Amandroid	N/A	92	56	49	7	8	28	88%	64%	74%
	BLECryptracer	High	92	50	46	4	11	31	92%	60%	72%
		Medium	42	22	19	3	8	12	86%	61%	72%
		Low	20	10	5	5	3	7	50%	42%	45%

[a] Number of APKs tested. Note that, for confidence levels Medium and Low, we don't consider the APKs detected at higher confidence levels.
[b] The number of APKs that were identified as having cryptographically protected BLE data.

The first trace method will produce results that are most likely to actually have cryptographically-processed BLE data, as the coarse-grained analysis performed in the subsequent methods adds increasing amounts of uncertainty. For this reason, BLECryptracer assigns "confidence levels" of High, Medium and Low to its output, which correspond to the three trace methods above, to indicate how certain it is of the result. We evaluate these confidence levels against a modified version of the DroidBench benchmarking suite in Section 4.4.1. Note that BLECryptracer only looks for application-layer security in benign applications, and these confidence levels apply only when deliberate manipulations (i.e., malicious obfuscation techniques) are not employed to hide the data flow between source and sink.

Appendix A describes the tracing mechanism in greater detail, and also outlines how BLECryptracer combats the effects of obfuscation in benign applications.

4.4 Evaluation

We evaluated BLECryptracer, in terms of both accuracy and execution times. For comparison purposes, we have included test results from Amandroid as well.

4.4.1 Accuracy Measures

At present, there is no dataset of real-world APKs with known use of cryptographically-processed BLE data, i.e., ground truth. Therefore, in order to test our tool against different data transfer mechanisms, we re-factored the DroidBench benchmarking suite [21] for the BLE case.

Each DroidBench test application was cloned twice and modified so the data flow between the sources and sinks would be from `getValue` to a cryptography method invocation, and from the cryptography method invocation to `setValue`, to emulate cryptographically-processed reads and writes, respectively. Some DroidBench test cases were excluded as they were found to be irrelevant due to differences in the objectives of DroidBench and our test set, e.g., applications that employ emulator detection or which leak contextual information in exceptions. Further, applications where BLE data is written to or read from files, or which contain data leaks in inactive code segments were not included (as our aim is to determine whether or not BLE data is cryptographically-processed). In total, we created 184 APKs: 92 for reads and 92 for writes.

We executed BLECryptracer against our benchmarking test set, analyzed the results and obtained performance metrics in terms of the three different confidence levels. The statistics differ based on the type of access that is analyzed (i.e., reads vs. writes) due to differences in the tracing mechanisms. The same test set was also used against Amandroid for comparison. Table 3 presents the performance metrics for both tools.

In the case of BLECryptracer results, the metrics are with respect to the total analyzed APKs at each confidence level. That is, because lower confidence levels analyze only those APKs that do not get detected at higher levels, accurate metrics can only be derived by considering the set of APKs that were actually analyzed at each level. For example, when considering the analysis of BLE reads, while the entire dataset of 92 APKs is relevant for confidence level High, only the 30 APKs that do *not* result in a TRUE outcome at level High will be analyzed for confidence level Medium. This also means that, when obtaining performance metrics for confidence level High, all TRUE results obtained at levels Medium and Low are taken to be FALSE.

The DroidBench test set, and hence our benchmarking suite, is an imbalanced dataset, containing far more samples *with* leaks (77) than *without* (15). For this reason, metrics such as accuracy are not suitable for analyzing the performance of our tool when executed against this test set, as they are more susceptible to skew [23,26]. For our analysis, we compare the combined True Positive Rate (TPR) and False Positive Rate (FPR), and the combined precision-recall instead, in-line with

taint-analysis evaluations [36].

Table 3 presents the precision and recall (i.e., TPR) for both BLECryptracer and Amandroid. We further derive FPRs for both tools. With BLECryptracer, when analyzing reads, False Positive Rates steadily increase as the confidence level reduces, as expected, with values of 27% for confidence level High, 36% for Medium and 57% for Low. When analyzing writes, the values are 27%, 27% and 63%, respectively. Regardless of the data access mechanism being tested, BLECryptracer (considering only the results at High confidence, for a fairer comparison) performs better than Amandroid in terms of FPR, with 27% vs. 33% for reads and 27% vs. 47% for writes. Precision values are also better in the case of BLECryptracer for both reads and writes. In terms of the True Positive Rate, BLECryptracer performs better than Amandroid for reads at 75% vs. 57%, and slightly worse for writes at 60% vs. 64%. These results show that, overall, BLECryptracer performs better than Amandroid for analyzing the use of cryptography with BLE data.

It should be noted that three of the four False Positives obtained by BLECryptracer at the High confidence level were due to the order in which variables are assigned values (i.e., lifecycle events), which is not tested for by BLECryptracer. Other data transfer mechanisms not tested for are `Looper` and `Messenger` functions, which generate False Negatives. The remaining False Positive was due to the presence of method aliasing and was also identified as a False Positive by Amandroid. In addition, the unexpectedly low TPR (i.e., recall) at level Medium for reads is due to the relatively few cases analyzed at that level when compared to High.

4.4.2 Execution Times

We also compared BLECryptracer and Amandroid in terms of speed of execution. For this, we ran the two tools against a random subset of 2,000 APKs and compared time-to-completion in both cases. We imposed a maximum run-time of 30 minutes per APK for both tools, and only compared execution times for those cases where Amandroid did not time out (approximately 40% of the tested APKs timed out when analyzed by Amandroid. In comparison, fewer than 2% of APKs timed out when analyzed by BLECryptracer).

Figure 4 plots the time taken to analyze BLE writes using BLECryptracer vs. Amandroid. The figure shows that analysis times with BLECryptracer were, for the most part, around 3-4 minutes per application. We observed no obvious correlation between the size of the application's dex file and the execution time, for either tool. APKs that took longer to process with BLECryptracer were predominantly of confidence level "Medium", which indicates that the longer analysis times may simply have been because of having to first go through the most stringent analysis (at the highest confidence level). For Amandroid, the execution times vary to a greater extent than with BLECryptracer, due to the difference in the mechanisms employed for performing the analysis.

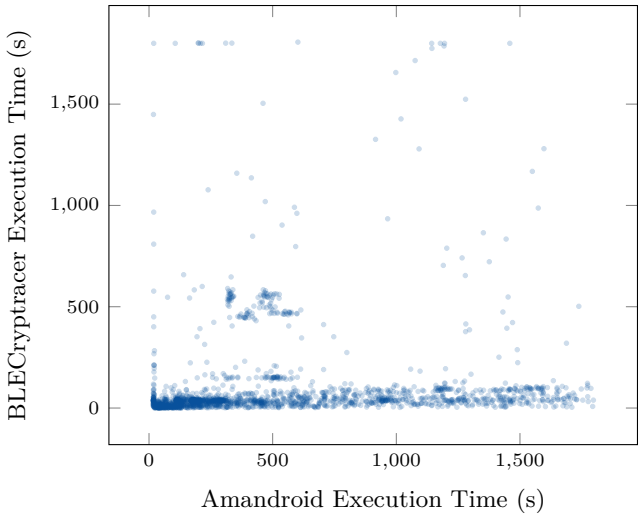


Figure 4: Comparison of time taken to execute BLECryptracer vs. Amandroid, when analyzing BLE writes.

4.5 Results from Large-Scale APK Analysis

We executed BLECryptracer against our dataset of 18,929 APKs. 192 APKs timed out when analyzing reads and 220 APKs timed out when analyzing writes, when a maximum runtime of 30 minutes was imposed. These APKs were re-tested with an increased runtime of 60 minutes. However, even with the longer analysis time, 44 and 76 APKs timed out for reads and writes, respectively, and had to be excluded from further analysis. In addition, approximately 90 APKs could not be processed via Androguard's `AnalyzeAPK` method and were excluded.

Due to the differences in performance metrics obtained for the three confidence levels during testing (as mentioned in Section 4.4), we focus on only those results that either identify cryptography at confidence level High or those where no cryptography was identified at all.

4.5.1 Presence of App-Layer Security with BLE Data

Our results show that approximately 95% of BLE-enabled APKs call the `javax.crypto` and `java.security` cryptography libraries *somewhere* within their code. While this is a large proportion of APKs, the results also indicate that a much smaller percentage of APKs use cryptographically processed data *with BLE reads and writes* (approximately 25% for both, identified with High confidence). In fact, about 46% of APKs that perform BLE reads and 54% of those that perform BLE writes (corresponding to 2,379 million and 2,075 million cumulative installations, respectively) do *not* implement security for the BLE data. Interestingly, of the 16,131 APKs that called

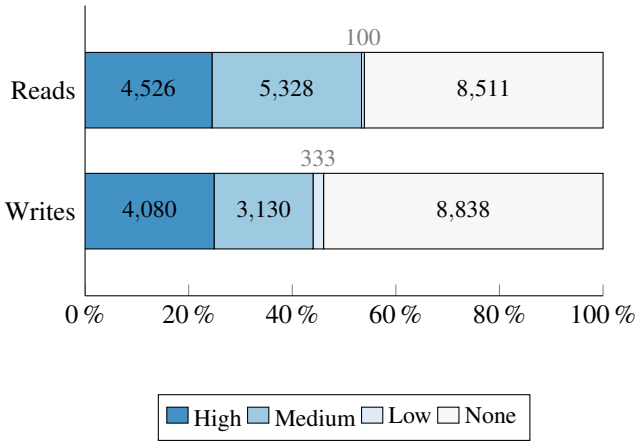


Figure 5: Analysis results depicting the presence of cryptographically-processed data with BLE writes and reads, with breakdown according to Confidence Level.

both BLE read and write functions, about 36% (i.e., more than 5,700 APKs), with a cumulative download count of 1,005 million, do not implement application-layer security for either type of data access. Figure 5 summarizes the proportion of APKs that were identified as containing cryptographically protected BLE data at the three different confidence levels.

4.5.2 Libraries vs. App-Specific Implementations

We found that many BLE-enabled APKs actually use third-party libraries for incorporating BLE functionality. To get an idea of exactly how many APKs relied on libraries, we analyze all methods within an APK that called BLE data access functions. To do this in an automated way, we compare the method class name with the application package name. If the first two components (e.g., `com.packagename`) of each match, then we take it to be a method implemented within the application. If the components do not match, we take it to be a library method. If the package name uses country-code second-level domains (e.g., uk.ac.packagename), then we compare the third components as well.

Of the APKs that called the `setValue` method, 63% used BLE functionality solely through libraries, 32% used application-specific methods only, while 4% used both. Fewer than 1% of the APKs could not be analyzed due to very short method names. Within the APKs that used both application-specific methods and libraries, around 34% used an external library to provide Device Firmware Update (DFU) capabilities, thereby enabling the BLE peripheral to be updated via the mobile application. Of the APKs that utilized only application-specific methods to incorporate BLE functionality, 67% did *not* implement application-layer security with the BLE data. This proportion was lower at 48% for applications that relied on libraries.

In the case of the APKs that called `getValue` variants, 37% used only application-specific methods, 58% used only libraries, and 5% used both. As with the `setValue` case, a higher proportion of APKs that used only app-specific BLE implementations were found to not use cryptography (60%), when compared with those that used only libraries (39%).

Table 4 presents the ten most commonly-encountered BLE libraries, their functionality, the number of APKs that use them, and the presence of cryptographically-processed BLE data within the library itself. The table shows that the most prevalent third-party packages are libraries that enable communication with BLE beacons. In fact, a single such library (Estimote) made up more than 90% of all instances of cryptographically-processed BLE writes and 85% of cryptographically-processed BLE reads (identified with High confidence). An analysis of this library suggested that cryptography is being used to authenticate requests when modifying settings on the beacon.

Apart from beacon libraries, we identified five libraries that function as wrappers for the Android BLE API. For example, Polidea wraps the API so that it adheres to the reactive programming paradigm. The libraries Randdusing, Megster and Evothings enable the use of BLE via JavaScript in Cordova-based applications. Similarly, Chromium enables websites to access BLE devices via JavaScript calls. None of these libraries handle cryptographically-processed BLE data. It is expected that developers using these libraries will implement their own application-layer security (using either JavaScript or reactive Java as appropriate).

Of the two remaining libraries, Flic, which uses cryptographically-processed data, is a library offered by a BLE device manufacturer. This library allows third-party developers to integrate their services into the Flic ecosystem, to allow them to automate certain tasks.

Finally, Nordicsemi is a library provided by a BLE chipset manufacturer to enable DFU over the BLE interface. With the newest version of the DFU mechanism, the BLE peripheral verifies that the firmware has been properly signed. Devices using the legacy DFU mechanism will not verify the firmware. However, the mobile application (and by extension, the library) does not need to handle cryptographically-processed data in either case.

4.5.3 Cryptographic Correctness

BLECryptracer identified 3,228 unique packages with cryptographically protected BLE data (with either reads or writes), with High confidence. However, the presence of crypto-libraries does not in itself indicate a secure system. We therefore further analyzed this subset of APKs to identify whether cryptography had been used correctly in them. The tool CogniCrypt [29] was utilized for this purpose. Although this tool does not formally verify the cryptographic protocol between the application and the BLE peripheral, it identifies various misuses of the Java crypto/security libraries.

Table 4: Top Ten Third-Party BLE Libraries

Library	Function	#APKs[unique]	Crypto
Estimote	Beacon	3980[2804]	Yes
Nordicsemi[a]	DFU	1238[847]	No
Kontakt	Beacon	1108[690]	No
Chromium	Web BLE	402[269]	No
Randdusing	Cordova Plugin	268[188]	No
Megster	Cordova Plugin	317[187]	No
Flic	BLE Accessory	173[164]	Yes
Polidea	BLE Wrapper	138[114]	No
Evothings	Cordova Plugin	142[84]	No
Jaalee	Beacon	102[79]	No

[a] Significant overlap present between Estimote and Nordic due to repackaging of the Nordic SDK into Estimote.

Among the 3,228 unique packages, we found that there was significant overlap between APKs in terms of the BLE libraries or functions[3] used. Removing such duplicates resulted in a set of 194 APKs. Of these, 68 were identified by CogniCrypt as having issues. However, because CogniCrypt identifies cryptography misuse within the entire APK, the results were filtered for BLE-specific functions. 24 APKs were found to have issues within or associated with the methods that cryptographically processed BLE data (as identified by BLECryptracer) and often, a single APK exhibited multiple issues. Table 5 shows the different types of misuse encountered and the number of unique packages that were identified as having such misuse. Note that because this analysis was performed over unique packages, the number of *APKs* that misuse crypto-libraries will be higher.

We manually analyzed the 24 APKs that were flagged by CogniCrypt as having BLE-relevant issues, and examined the identified instances of bad cipher modes and hardcoded keys/Initialization Vectors (IVs). With regard to insecure block cipher modes, we found that explicit use of ECB was prevalent (9 out of 10 cases), but there was also one case where `Cipher.getInstance("AES")` was used without the mode being specified, which may default to ECB depending on the cryptographic provider. When analyzing keys, we observed that several applications directly contained hardcoded keys as byte arrays or strings. Three applications retrieved keys from JSON files. In two cases, keys were generated from the `ANDROID_ID`, which is a system setting that is readable by all applications. We also observed one instance where a key was obtained from a server via HTTP (not HTTPS).

[3] There are instances where two applications may have unique package names, but which actually incorporate much the same functionality. This is often the case when the same developer produces branded variants of an application for different clients in a single industry. For example, two applications could have unique package names `com.myapp.app1` and `com.myapp.app2`, but their functionality may be derived from a common codebase `com.myapp`.

Table 5: Number of Packages with Cryptographic Misuse

Misuse Type[a]	#Unique Packages
ECB (or other bad mode)	10
Non-random key	6
Non-random IV	10
Bad IV used with Cipher	7
Bad key used with Cipher	11
Incomplete operation (dead code)	4

[a] Description of misuse based on [17, 30].

This analysis shows that several real-world applications contain basic mistakes in their use of crypto-libraries and handling of sensitive data, which means that the BLE data will not be secure despite the use of cryptography.

4.5.4 Trends over Time

Figure 6 shows the trend of application-layer security over time for applications that incorporate calls to BLE reads or writes. The graph depicts the percentage of applications that do not have cryptographic protection for either type of access. The overall downward trend suggests that there has been some improvement in application-layer security between the years 2014 and 2017 (we refrain from making observations about APKs from 2013 as they were very few in number, and about APKs from 2018 as the dataset is not yet fully populated for this year). However, it should be noted that, even in 2017, which had the smallest percentage of APKs without cryptography, these APKs corresponded to 128 million downloads, which is a significant number.

4.5.5 Application-Layer Security by Category

The percentage of applications that use cryptographically processed data from each major application category has been graphed in Figure 7. While it would be reasonable to expect that most "Medical" applications would implement some level of application-layer security, the results show that fewer than 30% of applications under this category actually have such protection mechanisms. However, it is possible that the reason for this is that the devices implement the standard Bluetooth SIG adopted profiles, which do not mandate any security apart from pairing, as mentioned in Section 3.3. In fact, of the APKs categorized under "Medical" and with no cryptographic protection for either reads or writes, we found that three of the top ten (in terms of installations) contained identifiers for the standard Bluetooth Glucose Service.

Perhaps surprisingly, APKs that are categorized under "Business", "Shopping" and "Travel & Local" appear to be the most likely to incorporate application-layer security, with around 50% of all such applications being identified as having

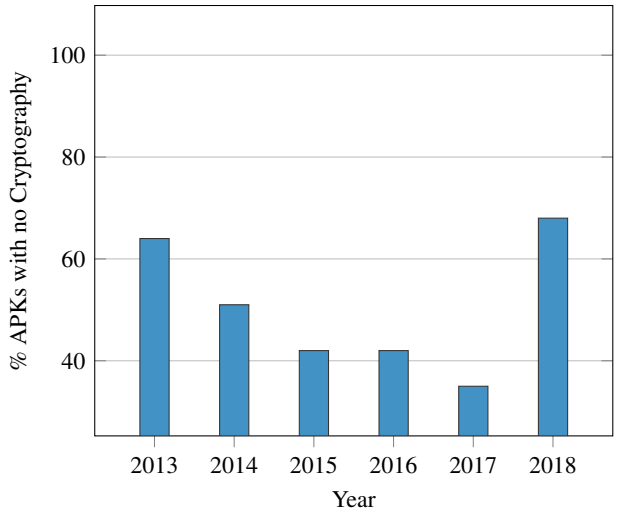


Figure 6: Application-layer security trends over time. *Notes:* Graph depicts APKs that perform BLE reads or writes, and have no cryptographic protection for either. APKs with dates that are invalid [39] or older than 2012 (when native BLE support was introduced for Android) have not been included.

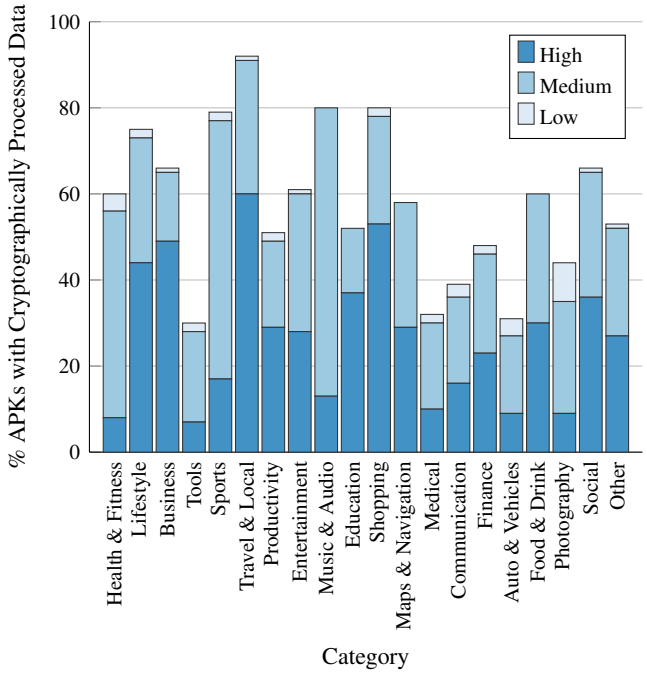


Figure 7: Presence of application-layer security in different categories of applications, averaged over BLE reads and writes, and broken down by confidence level. Only unique packages have been taken into consideration. APKs that do not currently have a presence on Google Play have been excluded, as their category cannot be identified.

cryptographically processed BLE data with High confidence. However, in over 85% of such occurrences, this was found to be due to the Estimote beacon library.

4.5.6 Impact Analysis

While 18,929 BLE-enabled applications may seem like a relatively small number of applications when compared with the initial dataset of 4.6 million+, a single application may correspond to multiple BLE devices, sometimes even millions of devices as is the case with fitness trackers [25]. For example, even if we consider the slightly restrictive case of unique applications that do not use cryptography with either reads or writes, the cumulative install count is still in excess of 1,005 million. This shows that the attack surface is much larger than may be indicated by the number of APKs.

It is of course a possibility that the data that is read from a BLE peripheral has no impact on user security or privacy (e.g., device battery levels). Understanding the data within APKs would require a more complex static analysis and is left as future work.

4.6 Case Study: Firmware Update over BLE

When analyzing our results, we found that one of the APKs that was identified as not having application-layer security was designed for use with a fitness tracker from our test device set. The tracker is a low-cost model that, based on the install count on Google Play (1,000,000+), appears to be widely used. An analysis of the APK suggested that the device used the Nordic BLE chipset, which could be put into the Legacy DFU mode, which does not require the firmware to be signed. To exploit this, we developed an APK that, in accordance with the attacks described in Section 3.1, connects to the device, sends commands to place it in DFU mode, and then writes a new modified firmware to the device without user intervention. The updated firmware in this case was a simple, innocuous modification of the original firmware. However, given that the device can be configured to receive notifications from other applications, a malicious firmware could be developed in such a way that, for example, all notifications (including second-factor authentication SMS messages or end-to-end encrypted messages) are routed to the malicious application that installed the firmware.

This attack was possible because the BLE peripheral did not verify the firmware (e.g., via digital signatures) nor the source application (via application-layer security). We have informed the application developer of the issue (02 Nov 2018), but have received no response as of the date of submission of this manuscript (18 May 2019).

While our attack was crafted for a specific device, it does demonstrate that attacks against these types of devices are relatively easy. An attacker could easily embed several firmware images within a single mobile application, to target a range of vulnerable devices.

4.7 Limitations

In this section, we outline some limitations, either in our script or due to the inherent nature of our experiments, that may have impacted our results.

4.7.1 Unhandled Data Transfer Mechanisms

As mentioned in Section 4.4, BLECryptracer does not analyze data that is written out to file (including shared preferences), or communicated out to a different application, because it is not straightforward (and many times, not possible) to determine how data will be handled once it has been transferred out of the application under analysis. It is also possible that an application obtains the data to be written to a BLE device from, or forwards the data read from a BLE device to, another entity, such as a remote server. That is, the Android application could merely act as a "shuttle" for the data, which means that an analysis of the APK would not show evidence of usage of cryptography libraries. However, the transfer of data to/from a remote server does not in itself indicate cryptographically-processed data, as plain-text values can also be transmitted in the same manner. We therefore do not analyze instances of data transfers to external entities.

BLECryptracer also does not handle data transfers between a source and sink when only one of them is processed within an `Looper` function or when the data is transmitted via messages. However, when we logged instances of where such functions were called during a trace, we found that of the APKs that utilized such data transfer mechanisms, a large percentage were identified as having cryptographic protection via other data flows. In fact, of the 8,834 APKs where cryptography was not identified with BLE writes, only 501 APKs interacted with `Looper` or `Messenger`, and an even smaller percentage of APKs were affected for BLE reads.

4.7.2 Conditional Statements with Backtracing

When backtracing a register, BLECryptracer stops when it encounters a constant value assignment. However, it is possible that this value assignment occurs within one branch of a conditional jump, which means that another possible value could be contained within another branch further up the instruction list. To identify this, the script would have to first trace forward within the instruction list, identify all possible conditional jumps, and then trace back from the register of interest for all branches. This would need to be performed for every method that is analyzed and could result in a much longer processing time per APK file, as well as potentially unnecessary overheads.

5 Related Work

User privacy has received particular attention in the BLE research community because several widely-used BLE devices, such as fitness trackers and continuous glucose monitors, are intended to always be on the user's person, thereby potentially leaking information about the user's whereabouts at all times. Some of the research has focused on the threats to privacy based on user location tracking [15, 19], while others explored the possibility of obtaining personal user data from fitness applications or devices [14, 28].

While our research *is* concerned with data access and user privacy, we focus more on the impact on privacy and security due to how the BLE standard has been implemented in mobile device architectures, as well as how it is applied by application developers in general, rather than due to individual BLE firmware design.

The work that is most closely related to ours is the research by Naveed, et al., which explored the implications of shared communication channels on Android devices [31]. In their paper, the authors discussed the issue of Classic Bluetooth and NFC channels being shared by multiple applications on the same device. They then demonstrated unauthorized data access attacks against (Classic) Bluetooth-enabled medical devices. The authors also performed an analysis of 68 Bluetooth-enabled applications that handled private user data, and concluded that the majority of them offered no protection against this attack. Finally, they proposed an operating-system level control for mitigating the attack.

Our work specifically targets pairing-protected characteristics on BLE devices, because BLE appears to slowly be replacing Classic Bluetooth in the personal health and home security domains. We demonstrate that the BLE data format and access mechanisms enable even easier attacks than in the case of Classic Bluetooth. Further, we identify the impact that the new Android permissions model (introduced in Android v6) has had on the user experience and on malicious applications' capabilities. We also perform a much larger-scale analysis over 18,900+ Android applications, to determine how prevalent application-layer security is among BLE-enabled applications.

6 Conclusions

In this paper, we analyze the risks posed to data on Bluetooth Low Energy devices from co-located Android applications. We show the conditions under which an unauthorized Android application would be able to access potentially sensitive, pairing-protected data from a BLE peripheral, once a co-located authorized application has paired and bonded with a BLE peripheral, without the user being aware of the access. We also show that, in some cases, an unauthorized application may be able to access such protected data with fewer permissions required of it than would be required of an authorized application. We then discuss mitigation strategies in terms of the different stakeholders in the BLE ecosystem.

We present BLECryptracer, an analysis tool for determining the presence of application-layer security with BLE data.

We evaluate it against the taint-analysis tool Amandroid, and present the results from executing BLECryptracer against 18,929 BLE-enabled Android APKs. Our results suggest that over 45% of all applications, and about 70% of "Medical" applications, do not implement cryptography-based application-layer security for BLE data. We also found, among the applications that *do* use cryptographically processed BLE data, several instances of cryptography misuse, such as the use of insecure cipher modes and hard-coded key values. We believe that, if this situation does not change, then as more and more sensitive use cases are proposed for BLE, the amount of private or critical data that may be vulnerable to unauthorized access can only increase. We hope that our work increases awareness of this issue and prompts changes by application developers and operating system vendors, to lead to improved protection for BLE data.

7 Availability

The code for our BLECryptracer tool is available at

https://github.com/projectbtle/BLECryptracer

This repository also contains the SHA256 hashes of the APKs in our dataset, and the source/sink files used for the Amandroid analysis. In addition, it contains source code for the benchmarking applications, as well as a comprehensive breakdown of the results per DroidBench category.

8 Acknowledgements

This research has been partially sponsored by the Engineering and Physical Sciences Research Council (EPSRC) and the UK government as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP/P009301/1).

References

[1] ALLIX, K., BISSYANDÉ, T. F., KLEIN, J., AND LE TRAON, Y. Androzoo: Collecting millions of Android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories* (2016), ACM, pp. 468–471.

[2] ANDROID. Distribution dashboard. [Online]. Available: https://developer.android.com/about/dashboards/. [Accessed: 06 Aug 2018].

[3] ANDROID. Bluetooth Low Energy overview, Apr 2018. [Online]. Available: https://developer.android.com/guide/topics/connectivity/bluetooth-le. [Accessed: 18 July 2018].

[4] ANDROID. Security tips, June 2018. [Online]. Available: https://developer.android.com/training/articles/security-tips. [Accessed: 18 July 2018].

[5] ANDROID. Security updates and resources, 2018. [Online]. Available: https://source.android.com/security/overview/updates-resources#severity. [Accessed: 18 May 2019].

[6] ARM LTD. Firmware Over the Air, 2016. [Online]. Available: https://docs.mbed.com/docs/ble-intros/en/master/Advanced/FOTA/. [Accessed: 21 July 2018].

[7] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *Acm Sigplan Notices 49*, 6 (2014), 259–269.

[8] BISIO, I., SCIARRONE, A., AND ZAPPATORE, S. A new asset tracking architecture integrating RFID, Bluetooth Low Energy tags and ad hoc smartphone applications. *Pervasive and Mobile Computing 31* (2016), 79–93.

[9] BLUETOOTH SPECIAL INTEREST GROUP. Heart Rate Profile: Bluetooth profile specification v1.0, 07 2011.

[10] BLUETOOTH SPECIAL INTEREST GROUP. Continuous Glucose Monitoring Profile: Bluetooth profile specification v1.0.1, 12 2015.

[11] BLUETOOTH SPECIAL INTEREST GROUP. Bluetooth core specification v5.0, 12 2016.

[12] BLUETOOTH SPECIAL INTEREST GROUP. Bluetooth mesh networking / an introduction for developers, 2017.

[13] BRONZI, W., FRANK, R., CASTIGNANI, G., AND ENGEL, T. Bluetooth Low Energy performance and robustness analysis for inter-vehicular communications. *Ad Hoc Netw. 37*, P1 (Feb 2016), 76–86.

[14] CYR, B., HORN, W., MIAO, D., AND SPECTER, M. Security analysis of wearable fitness devices (Fitbit). *Massachusetts Institute of Technology* (2014).

[15] DAS, A. K., PATHAK, P. H., CHUAH, C.-N., AND MOHAPATRA, P. Uncovering privacy leakage in BLE network traffic of wearable fitness trackers. In *Proceedings of the 17th International Workshop on Mobile Computing Systems and Applications* (2016), ACM, pp. 99–104.

[16] DESNOS, A., ET AL. Androguard: Reverse engineering, malware and goodware analysis of Android applications ... and more (ninja !). https://github.com/androguard/androguard.

[17] EGELE, M., BRUMLEY, D., FRATANTONIO, Y., AND KRUEGEL, C. An empirical study of cryptographic misuse in Android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 73–84.

[18] ELKHODR, M., SHAHRESTANI, S., AND CHEUNG, H. Emerging wireless technologies in the Internet of Things: A comparative study. *International Journal of Wireless & Mobile Networks (IJWMN) 8*, 5 (Oct 2016), 67–82.

[19] FAWAZ, K., KIM, K.-H., AND SHIN, K. G. Protecting privacy of BLE device users. In *USENIX Security Symposium* (2016), pp. 1205–1221.

[20] FIDO ALLIANCE. FIDO Bluetooth Specification v1.0, 2017. https://fidoalliance.org/specs/fido-u2f-bt-protocol-id-20150514.pdf.

[21] FRITZ, C., ARZT, S., AND RASTHOFER, S. Droidbench: A micro-benchmark suite to assess the stability of taint-analysis tools for Android. https://github.com/secure-software-engineering/DroidBench.

[22] GOMEZ, C., OLLER, J., AND PARADELLS, J. Overview and evaluation of Bluetooth Low Energy: An emerging low-power wireless technology. *Sensors (Basel, Switzerland) 12*, 9 (2012), 11734–11753.

[23] GUO, X., YIN, Y., DONG, C., YANG, G., AND ZHOU, G. On the class imbalance problem. In *Natural Computation, 2008. ICNC'08. Fourth International Conference on* (2008), vol. 4, IEEE, pp. 192–201.

[24] HOFFMANN, J., USSATH, M., HOLZ, T., AND SPREITZENBARTH, M. Slicing Droids: Program slicing for smali code. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing* (2013), ACM, pp. 1844–1851.

[25] IDC. Worldwide wearables market grows 7.3% in Q3 2017 as smart wearables rise and basic wearables decline, says IDC. [Online]. Available: https://github.com/secure-software-engineering/DroidBench [Accessed 16-Feb-2017].

[26] JENI, L. A., COHN, J. F., AND DE LA TORRE, F. Facing imbalanced data–recommendations for the use of performance metrics. In *Affective Computing and Intelligent Interaction (ACII), 2013 Humaine Association Conference on* (2013), IEEE, pp. 245–251.

[27] KARANI, R., DHOTE, S., KHANDURI, N., SRINIVASAN, A., SAWANT, R., GORE, G., AND JOSHI, J. Implementation and design issues for using Bluetooth Low Energy in passive keyless entry systems. In *India Conference (INDICON), 2016 IEEE Annual* (2016), IEEE, pp. 1–6.

[28] KOROLOVA, A., AND SHARMA, V. Cross-app tracking via nearby Bluetooth Low Energy devices. In *PrivacyCon 2017* (2017), Federal Trade Commission.

[29] KRÜGER, S., NADI, S., REIF, M., ALI, K., MEZINI, M., BODDEN, E., GÖPFERT, F., GÜNTHER, F., WEINERT, C., DEMMLER, D., ET AL. CogniCrypt: supporting developers in using cryptography. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (2017), IEEE Press, pp. 931–936.

[30] KRÜGER, S., SPÄTH, J., ET AL. CogniCrypt_SAST: CrySL-to-static analysis compiler. https://github.com/CROSSINGTUD/CryptoAnalysis/.

[31] NAVEED, M., ZHOU, X., DEMETRIOU, S., WANG, X., AND GUNTER, C. A. Inside job: Understanding and mitigating the threat of external device mis-binding on Android. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014* (2014).

[32] NORDIC SEMICONDUCTOR. BLE on Android v1.0.1. [Online]. Available: https://devzone.nordicsemi.com/attachment/bdd561ff56924e10ea78057b91c5c642. [Accessed: 05 Feb 2018].

[33] ORACLE. Java Cryptography Architecture (JCA) Reference Guide. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html. [Accessed: 18 July 2018].

[34] PAUCK, F., BODDEN, E., AND WEHRHEIM, H. Do Android taint analysis tools keep their promises? In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (New York, NY, USA, 2018), ESEC/FSE 2018, ACM, pp. 331–341.

[35] POEPLAU, S., FRATANTONIO, Y., BIANCHI, A., KRUEGEL, C., AND VIGNA, G. Execute this! Analyzing Unsafe and malicious dynamic code loading in Android applications. In *NDSS* (2014), vol. 14, pp. 23–26.

[36] QIU, L., WANG, Y., AND RUBIN, J. Analyzing the analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2018), ACM, pp. 176–186.

[37] RYAN, M. Bluetooth: With low energy comes low security. In *7th USENIX Workshop on Offensive Technologies, WOOT '13, Washington, D.C., USA, August 13, 2013* (2013).

[38] SILVA, B. N., KHAN, M., AND HAN, K. Internet of Things: A comprehensive review of enabling technologies, architecture, and challenges. *IETE Technical Review 35*, 2 (2018), 205–220.

[39] UNIVERSITÉ DU LUXEMBOURG. Lists of APKs. [Online]. Available: https://androzoo.uni.lu/lists. [Accessed: 12 Nov 2018].

[40] WEI, F., ROY, S., OU, X., ET AL. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 1329–1341.

```
1  .method private
     a(Landroid/bluetooth/BluetoothGatt;[B...)V
2    .locals 10
3
4    .prologue
5    const/4 v9, 0x2
6    const/4 v8, 0x3
7    const/4 v7, 0x1
8    ...
9    invoke-virtual {v0, v3},
       Landroid/bluetooth/BluetoothGattService;->
       getCharacteristic(Ljava/util/UUID;)
       Landroid/bluetooth/BluetoothGattCharacteristic;
10
11   move-result-object v3
12   ...
13   invoke-virtual {v3, p2},
       Landroid/bluetooth/BluetoothGattCharacteristic;
       ->setValue([B)Z
14   invoke-virtual {v1, v3},
       Landroid/bluetooth/BluetoothGatt;
       ->writeCharacteristic(Landroid/bluetooth/
       BluetoothGattCharacteristic;)Z
```

Figure 8: Sample smali code for BLE attribute write.

Appendix A: BLECryptracer Logic

We describe here the basic tracing mechanism employed by BLECryptracer in order to identify the presence of application-layer security for BLE data.

Backtracing BLE writes

BLE writes use one of the `setValue` methods in Table 2 to first set the value that is to be written, before calling the method for performing the actual write. BLECryptracer identifies all calls to these methods, and then traces the origins of the data held in the registers that are passed as input to the methods.

Considering the smali[4] code in Figure 8 as an example, `setValue` is invoked at Line 13 and is passed two registers as input. As `setValue` is an instance method, the first input, local register `v3`, holds the `BluetoothGattCharacteristic` object that the method is invoked on. The second input, parameter register `p2`, holds the data that is to be written to the BLE device, and is the second argument that is passed to the method `a` (Line 1). BLECryptracer identifies `p2` as the register that holds the data of interest, and traces backward to determine if this data is the result of some cryptographic processing. To achieve this, the method(s) within the APK that invoke method `a` are identified, and the second input to each such method is traced. If the BLE data had come from a local register, rather than a parameter register, BLECryptracer would trace back *within* method `a`'s instructions, to determine the origin of the data. This backtracing is performed until either a crypto-library is referenced, or a `const-<>` or `new-array` declaration is encountered (which would indicate that no cryptography is used). Note that calls to any method within the crypto-libraries mentioned in Section 4.2 are accepted as evidence of the use of cryptography with BLE data. The tool stops processing an APK at the first instance where such a method call is identified.

During execution, the BLECryptracer maintains a list of registers (set within the context of a method) to be traced, for each `setValue` method call within the application code. This initially contains a single entry, which is the input to the `setValue` method. A new register is added to the list if it appears to have tainted the value of any of the registers already in the list. This could be due to simple operations such as `aget`, `aput` or `move-<>` (apart from `move-result` variants), or it could be as a result of a comparison, arithmetic or logic operation (in which case, the register holding the operand on which the operation is performed is added to the trace list). Similarly, if a register obtains a value from an instance field (via `sget` or `iget`), then all instances where that field is assigned a value are analyzed. However, the script does not analyze the order in which the field is assigned values, as this would require activity life-cycle awareness.

Where a register is assigned a value that is output from a method invocation via `move-result`, if the method is not an external method, then the instructions within that method are analysed, beginning with the return value and tracing backwards. In some instances, the actual source of a register's value is obfuscated due to the use of intermediate formatting functions. In an attempt to overcome this, BLECryptracer traces the inputs to called methods as well. Further, if a register is used as input to a method, then all other registers that are

[4] Android applications are typically written in Java and converted into Dalvik bytecode. The smali format can be considered an "intermediate" step between the high-level Java source and the bytecode.

```
1 .method public onCharacteristicread(Landroid/bluetooth/
    BluetoothGatt;Landroid/bluetooth/
    BluetoothGattCharacteristic;I)V
2   ...
3   invoke-virtual {p2}, Landroid/bluetooth/
      BluetoothGattCharacteristic;->getValue()[B
4   move-result-object v0
5   new-instance v2, Ljava/lang/StringBuilder;
6   invoke-direct {v2},
      Ljava/lang/StringBuilder;-><init>()V
7   const-string v3, "read value: "
8   invoke-virtual {v2, v3},
      Ljava/lang/StringBuilder;->append(Ljava/lang/
      String;)Ljava/lang/StringBuilder;
9   move-result-object v2
10  invoke-static {v0},
      Ljava/util/Arrays;->toString([B)Ljava/lang/ String;
11  move-result-object v3
12  invoke-virtual {v2, v3},
      Ljava/lang/StringBuilder;->append(Ljava/lang/
      String;)Ljava/lang/StringBuilder;
13  move-result-object v2
14  ...
```

Figure 9: Sample smali code for BLE attribute read.

inputs to the method are also added to the trace list. While this captures some indirect value assignments, it runs the risk of false positives. For this reason, we have included the concept of *Confidence Levels* for the code output.

If, for an APK, the input to the `setValue` method can be backtraced to cryptography directly, via only register value transfers and as immediate results of method invocations, then a confidence level of "High" is assigned to the result. If a register cannot be traced back directly to a cryptographic output, but if an indirect trace identifies the use of a cryptography library, then a confidence level of "Medium" is assigned. Finally, in the event that no cryptography use is identified at High or Medium confidence levels, the script performs a less stringent search through all the instructions of the methods that it previously analyzed. This risks including instances of cryptography use with functions unrelated to BLE and is therefore assigned a "Low" confidence level.

Forward-tracing BLE reads

With BLE reads, a `getValue` variant is invoked and the output, i.e., the value that is read, is moved to a register. To trace this value, BLECryptracer identifies all calls to `getValue` variants, then traces the output registers and all registers they taint until either a crypto-library is referenced or the register value changes. Such value changes can occur due to `new-array`, `new-instance` and `const` declarations, as well as by being assigned the output of various operations (such as method invocations or arithmetic/logic operations).

With forward-tracing, the register holding the BLE data is considered to taint another if, for example, the source register is used in a method invocation, or comparison/arithmetic/logic operation, whose result is assigned to the destination register. The destination register is then added to the trace list. When a register is used as input to a method, then along with the output of that method, the use of the register *within* the method is also analyzed.

This method of analysis tends to result in a "tree" of traces. As an example, considering the smali code in Figure 9, the byte array output from the BLE read is stored in register `v0` (Line 4). This taints register `v3` via a format conversion function (Lines 10 and 11), which in turn taints `v2` via a `java.lang.StringBuilder` function (Lines 12 and 13). At this point, all three registers are tainted and will be traced until their values change.

The forward-tracing mode also assigns one of three confidence levels to its output. "High" is assigned when cryptographically-processed data is identified via the tracing mechanism above; "Medium" is when the use of cryptography is identified by tracing classes that implement interfaces. "Low" is assigned when a less stringent search through all encountered methods results in identification of a reference to a cryptography library (similar to the backtracing case).

Handling obfuscation

APKs sometimes employ obfuscation techniques to protect against reverse-engineering, and the question then arises as to whether these techniques may impact the results of our analysis. We therefore briefly discuss different obfuscation techniques and why they do not impact our tool.

One of the most common techniques is *identifier renaming*, where identifiers within the code are replaced with short, meaningless names. However, because Androguard operates on smali (rather than Java) code, BLECryptracer is able to overcome the challenges posed by this technique. *String encryption* is another obfuscation mechanism, but it again does not affect the output of our tool as BLECryptracer does not search for hard-coded strings. Further, our tool was verified successfully against three out of four benchmarking applications that utilized *reflection*. The most complex obfuscation techniques are *packing* and *runtime-based obfuscation*, but these are typically employed by malware. Because we are looking for vulnerable (not malicious) applications, we do not consider these techniques. Therefore, in general, we believe our analysis to be unaffected by most benign obfuscation mechanisms.

The CrossPath Attack: Disrupting the SDN Control Channel via Shared Links

Jiahao Cao[1,2], Qi Li[2,3], Renjie Xie[1,2], Kun Sun[4], Guofei Gu[5],
Mingwei Xu[1,2], and Yuan Yang[1,2]

[1]*Department of Computer Science and Technology, Tsinghua University*
[2]*Beijing National Research Center for Information Science and Technology, Tsinghua University*
[3]*Institute for Network Sciences and Cyberspace, Tsinghua University*
[4]*Department of Information Sciences and Technology, George Mason University*
[5]*SUCCESS LAB, Texas A&M University*

Abstract

Software-Defined Networking (SDN) enables network innovations with a centralized controller controlling the whole network through the control channel. Because the control channel delivers all network control traffic, its security and reliability are of great importance. For the first time in the literature, we propose the *CrossPath* attack that disrupts the SDN control channel by exploiting the shared links in paths of control traffic and data traffic. In this attack, crafted data traffic can implicitly disrupt the forwarding of control traffic in the shared links. As the data traffic does not enter the control channel, the attack is stealthy and cannot be easily perceived by the controller. In order to identify the target paths containing the shared links to attack, we develop a novel technique called *adversarial path reconnaissance*. Both theoretic analysis and experimental results demonstrate its feasibility and efficiency of identifying the target paths. We systematically study the impacts of the attack on various network applications in a real SDN testbed. Experiments show the attack significantly degrades the performance of existing network applications and causes serious network anomalies, e.g., routing blackhole, flow table resetting, and even network-wide DoS.

1 Introduction

Software-Defined Networking (SDN) becomes increasingly popular and is being widely deployed in data centers [32], cloud networks [13], and wide area networks [11]. In SDN, the control plane and data plane are decoupled. A logically centralized controller communicates with SDN switches to exchange control messages, e.g., routing decisions, via the control channel built upon a southbound protocol, e.g., OpenFlow [47]. SDN enables diversified packet processing and drives network innovation. A large number of network services and applications [26, 40, 33] benefit from it.

Unfortunately, the SDN control channel between the control plane and data plane is not well protected and can be exploited though the confidentiality and integrity of the communication over the channel are protected by the TLS/SSL protocol. We find that the control channel is under the risk of the Denial-of-Service (DoS) attack. In particular, a small portion of traffic may tear down the communication over the control channel. Existing studies focus on many security aspects of SDN, including malicious or buggy applications [63, 48], attacks on crashing controllers [60, 49, 65], attacks on disrupting switches [22, 51], and information leakage in SDN [25, 56, 19, 45] , but the security of the SDN control channel is still an open problem.

In this paper, we propose a novel attack named *CrossPath Attack*, which disrupts the SDN control channel by exploiting the shared links between paths of control traffic and data traffic. Our attack is stealthy and cannot be easily perceived by the controller since it does not directly send a large volume of control traffic to the controller. Instead, it generates well-crafted data traffic in the shared links to implicitly interfere with the delivery of the control traffic while the data traffic does not reach the controller. Thereby, real-time control messages delivered between the SDN controller and the switches are significantly delayed or dropped. In particular, since the controller performs centralized control over all network switches via the control channel, an attacker can easily break down all network functionalities enabled by various SDN applications running on the controller. The root cause of the vulnerability is the side effect incurred by shared links between paths of control traffic and data traffic in SDN. Such link sharing is a common practice in SDN with in-band control [21, 65], which can greatly reduce the cost of building a dedicated control network and simplify network maintenance, especially for large networks. However, it also opens the door for an attacker to disrupt the control channel by sending malicious data traffic to the shared links.

It is challenging to construct the attack in real networks. Unlike traditional IP networks where almost all links deliver both control traffic (e.g., OSPF or BGP updates [1, 2]) and data traffic at the same time, only a few number of links forward control traffic in SDN. For instance, an SDN network

with m switches can have $O(m^2)$ links. However, there may be m links forming a spanning tree connecting m switches with a controller to deliver the control traffic. Thus, an attacker needs to find a target path that contains the shared links between control and data traffic to send the attack traffic. However, it is difficult to know since the network topology and the routing information are invisible to end users. Moreover, none of the information can be inferred by scanning tools used in traditional IP networks due to different forwarding actions in SDN. For example, Traceroute [17] cannot work well because SDN switches usually do not decrease the time-to-live (TTL) values in packet headers.

To address the above challenge, we present a probing technique called *adversarial path reconnaissance* to find a target path of data traffic that contains the shared links. The key observation is that the delays of control messages on the SDN control channel will become higher if a short-term burst of data traffic passes through the shared links. Meanwhile, such delays that indicate the path of the current data traffic has shared links with control traffic can be measured by a host. The reason one host can measure the delays is that the first packet of a new flow will be sent to the controller to query forwarding actions, which incurs extra delays of control messages other than that of the following packets directly processed in the data plane. Thus, by crafting timing packets to measure the latency variation of the control messages with/without injecting a short-term burst of data traffic, a path containing the shared links can be correctly identified. By conducting the above reconnaissances on each possible path, a target path can finally be found.

We note the probing technique may fail to identify a target path in rare cases. We study the conditions of successful probing, and our experiments with 261 real network topologies [4] demonstrate that these conditions can be easily met in practice. Moreover, we analyze the expected number of paths that need to be explored for an attacker to find a target path. Both theoretical analysis and experimental results show the high efficiency of our probing technique. For example, it only needs to explore less than 50 paths on average if there are 1,000 paths and only 2% of them contains shared links. Experimental results in a real SDN testbed show our reconnaissances can achieve more than 90% accuracy.

In order to ensure the stealthiness of the attack, we leverage the low-rate TCP-targeted DoS [41] to generate data traffic consisting of periodic pulses in the shared links, instead of directly flooding shared links to disrupt the network. The low-rate TCP targeted DoS incurs repeated TCP retransmission timeout for TCP connections of the control channel. Compared with direct link flooding on the shared links, it significantly reduces the volume of attack traffic. Note the TCP-targeted DoS cannot effectively disrupt SDN networks without the knowledge of shared links obtained by our probing technique. Moreover, our attack is significantly different from the packet-in flooding attacks [55, 60] that trigger a huge volume of *control traffic* with bogus packets to saturate the SDN control channel. Instead, it leverages low-rate *data traffic* to disrupt the control channel and can thus succeed even in the presence of state-of-the-art SDN defenses, such as FloodGuard [60], FloodDefender [52], and SPHINX [27].

We systematically study the impacts of the attack on different SDN applications that achieve diversified network functionalities. We find that almost all SDN applications can be affected by our attack since our attack targets at disrupting the core services in SDN controllers that support these applications. In order to understand the impacts, we conduct experiments with four typical applications that have been widely deployed in SDN controllers, i.e., ARP Proxy [5], Learning Switch [6], Reactive Routing [9], and Load Balancer [7]. The results show (1) the performance of ARP Proxy can be significantly degraded, such as 10 times increase in the response delays and 95% reduction in the number of the ARP replies; (2) Learning Switch cannot successfully install forwarding decisions in the data plane and thus the throughput of the data plane is reduced to 0 Mbps; (3) Reactive Routing cannot update routing information in time and obtain incorrect topology information, which incurs various routing anomalies, e.g., routing loop, routing blackhole, routing path eviction, and flow table resetting; and (4) Load Balancer generates wrong decisions, resulting in link overloading.

In summary, our paper makes the following contributions:

- We present the CrossPath attack to significantly disrupt the SDN control channel by exploiting the shared links between paths of control traffic and data traffic.
- We develop a probing technique called adversarial path reconnaissance that can find a target path containing the shared links with a high accuracy.
- We prove the conditions of successful probing, analyze the expected number of explored paths to find a target path, and validate our analysis with experiments.
- We perform a systematical study and conduct extensive experiments on four typical SDN applications to demonstrate the impacts of the attack on various SDN network functionalities.

The rest of the paper is organized as follows. Section 2 provides background information about SDN and threat model. Section 3 presents the CrossPath attack along with an effective probing technique. Section 4 evaluates the feasibility and effectiveness of the attack both in large-scale simulations and real SDN testbeds. Section 5 further studies the impacts of the attack on different SDN applications by detailed analysis and extensive experiments. Section 6 discusses defense mechanisms that can be immediately deployed in practice to mitigate the attack. Section 7 reviews related work. Section 8 concludes the paper.

2 Background and Threat Model

2.1 Background

In this section, we briefly review the SDN architecture and a typical protocol of SDN, i.e., the OpenFlow protocol [47]. SDN enables network innovations by decoupling the control and data planes and provide programmability as well as flexibility. The control plane is logically centralized and can be deployed on commodity servers. The SDN architecture can be divided into three layers. The control layer and the application layer constitute the control plane, which runs as a network operating system, a.k.a. a controller. Various network applications can be deployed in the application layer to enable diversified network functions, such as routing, network monitoring, anomaly detection, and load balancing. The data plane layer, which consists of "dumb" SDN switches, performs low-level packet processing and forwarding based on the decisions generated by the control layer.

The dominant communication protocol between the control and data planes is OpenFlow, which has been standardized by the Open Networking Foundation (ONF) [14]. OpenFlow allows a controller to dynamically specify SDN switches' forwarding behaviors by installing flow rules. Each flow rule contains *match fields* to match against incoming packets, a set of *instructions* that describe how to process the matched packets, and *counters* that count the number and the total bytes of matched packets. OpenFlow also defines how to handle packets in a switch. When a switch receives a packet, it processes the packet based on the rule that matches the packet with the highest priority. If no rules match the packet, the switch sends the packet to the SDN controller through the control channel with a *packet_in* message. Applications running on the controller analyze the packet and make decisions. Once the decisions are made, the packet will be sent back to the switch with a *packet_out* message. The corresponding flow rules will be installed into all switches forwarding the packet with *flow_mod* messages. Such a packet processing procedure is called reactive rule installation, which has been widely used in OpenFlow networks [60, 52]. Moreover, to reduce the cost of building a dedicated control network and operating networks, in particular in large-scale networks [21, 65], OpenFlow allows the control and data traffic to share some links in the network, which is called in-band control.

2.2 Threat Model

In this paper, we consider an SDN network deployed with the OpenFlow protocol. The network uses a reactive approach to install flow rules, which is widely adopted in practice [60, 52], over an in-band control channel [21, 65]. We assume that an attacker has or compromises at least one host attached in the network, which can be easily achieved, e.g., by renting a virtual machine in an SDN-based cloud network. The goal of the attacker is to craft data traffic to disrupt the SDN control channel that delivers control traffic.

An attacker does not need to have prior knowledge on the network and any privileges of network operation. The CrossPath attack does not require the attacker to compromise the controllers, applications, and switches, or to construct man-in-the-middle attacks on the control channel to manipulate the control messages. The control channel can be protected with TLS/SSL. Furthermore, we assume that controllers, switches, and applications are well protected. For example, the network applies strict access control policies to prevent communication between controllers and attackers.

3 The CrossPath Attack

In this section, we present the CrossPath attack on disrupting the SDN control channel. Particularly, we develop a probing technique called *adversarial path reconnaissance* to accurately find a target path containing shared links.

3.1 Overview

The CrossPath attack aims to disrupt the SDN control channel by exploiting the shared links between paths of control traffic and data traffic. An attacker interferes with the transmission of control traffic by generating data traffic passing through the shared links. Thereby, the real-time control messages delivered in the control channel are delayed or dropped. As the SDN controller performs centralized control over all switches via the control channel, an attacker can almost break down all network functionalities enabled by SDN by constructing the attack. To achieve this, an attacker needs to use a host attached in the network to generate probing traffic so as to identify which path of data traffic (i.e., a target path) shares links with paths of control traffic. Then, the attacker can send attack traffic to the target path to disrupt the control channel. In order to decrease the attack rate, the attack utilizes the low-rate TCP-targeted DoS (LDoS) [41] to generate periodic on-off "square-wave" traffic, which leads to repeated TCP retransmission timeout for the TCP connections of the control channel.

Now let us use a simple example to illustrate the attack. For the ease of explanation, we use *data path* to denote the path where the data traffic is delivered and *control path* to denote the path where the control traffic is delivered. As shown in Figure 1, the network has five switches $\{s_1, s_2, s_3, s_4, s_5\}$. Host h_1 and h_3 communicate with each other via the data path $h_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow h_3$, while the control path between s_2 and the controller is $s_2 \rightarrow s_3 \rightarrow s_5 \rightarrow c$. We can observe that the link between s_2 and s_3 is shared by the control and data path. Assume host h_1 compromised by an attacker sends crafted LDoS traffic to h_3. Since the link and corresponding queues of switch ports are also used by the control

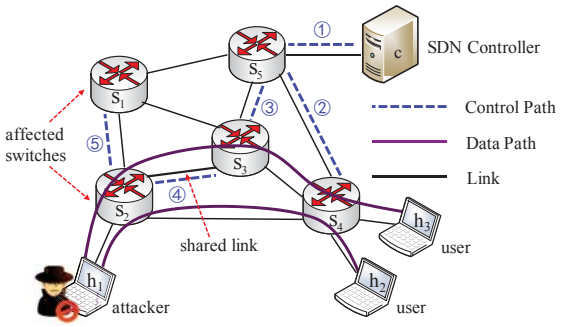


Figure 1: An example of disrupting the SDN control channel.

paths of s_2 and s_1, the control messages delivered between the switches and the SDN controller can be significantly delayed or dropped, resulting in abnormal network behaviors.

In order to successfully launch the attack, an attacker should correctly choose a target path that contains shared links. However, it is challenging to find target paths in SDN. Different from traditional IP networks that almost each link delivers data and control traffic at the same time, there are only a few number of links delivering control traffic in SDN. For instance, given an SDN network with m switches, there may be $m^2/2$ links. m links may be used to deliver control traffic so that the connectivity between the controller and all SDN switches can be ensured. Thus, only a limited number of data paths include the links shared with control paths. To identify such data paths, the attacker needs to know the network topology and routing information. Nevertheless, they are stored in the SDN controller and are invisible to the attacker. Moreover, existing scanning tools cannot be used in SDN to infer the network topology and routing information because SDN has different forwarding behaviors compared to traditional IP networks. For example, Traceroute [17] cannot infer the routing path of the packets, as SDN usually does not decrease the time-to-live (TTL) values in packet headers.

3.2 Adversarial Path Reconnaissance

To address the challenges above, we develop a probing technique called *adversarial path reconnaissance* to find target data paths that have links shared with control paths. The technique inspired by the key observation that the delay of a control path is higher if a short-term burst of the data traffic passes through the shared links. Thus, an attacker can use a host in SDN to identify the key data paths by generating data traffic and measuring the delay variations of the control paths. To achieve the goal, our adversarial path reconnaissance consists of two phases: measuring the delays of control paths and identifying a target data path.

Measuring Delays of Control Paths. In SDN, packets that cannot be matched in a switch will experience long forwarding paths and high delays, since they will be forwarded to the controller to request flow rules. We can analyze the delays of these packets to calculate the delays of control paths that share links with data paths. Assume there are two hosts h_i and h_j, and the data path between them is a sequence of consecutive links $P_d^{i,j} = < l_{h_i \to s_1}, l_{s_1 \to s_2}, \ldots, l_{s_\omega \to h_j} >$. Figure 2a shows the forwarding path and delay for a packet that is sent from h_i to h_j. The packet cannot be matched by flow rules in s_1. We can know the end-to-end delay for the packet is:

$$d_{i,j} = d_{prop}^{h_i} + \sum_{k=1}^{\omega+1} d_{trans}^{k} + \sum_{k=1}^{\omega} (d_{queue}^{k} + d_{proc}^{k}) + \delta_{i,j}, \quad (1)$$

where $d_{prop}^{h_i}$ is the propagation delay at host h_i, d_{trans}^{k} is the transmission delay at the kth link, d_{queue}^{k} is the queuing delay at the kth switch, and d_{proc}^{k} is the processing delay at the kth switch. $\delta_{i,j}$ is the delay of the control path, which is caused by querying controllers for rule installation. The delay pattern of such packet is shown in Figure 2a. However, if we send the same packet after the rule installation, the path and delay will become shorter, as shown in Figure 2b. The end-to-end delay can be expressed as follows:

$$d'_{i,j} = d_{prop}^{h_i} + \sum_{k=1}^{\omega+1} d_{trans}^{k} + \sum_{k=1}^{\omega} (\hat{d}_{queue}^{k} + d_{proc}^{k}). \quad (2)$$

Here, we change d_{queue}^{k} to $\hat{d}_{queue}^{k}$ because the queuing delay depends on the current network traffic and is time-varying. Based on equation (1) and (2), the delay of the control path is:

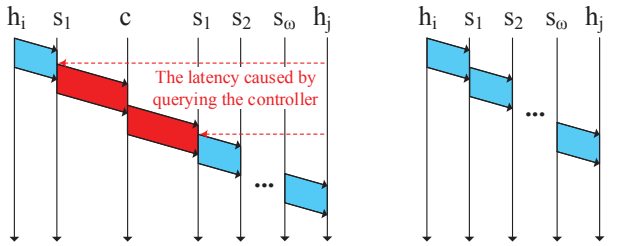
$$\delta_{i,j} = d_{i,j} - d'_{i,j} + \sum_{k=1}^{\omega} (\hat{d}_{queue}^{k} - d_{queue}^{k}) \quad (3)$$

However, if we send two packets with a short time interval, e.g., sending the same packet immediately once we receive a response to the last packet, the queuing delay d_{queue}^{k} and $\hat{d}_{queue}^{k}$ can be approximately equal. Thus, we have $\delta_{i,j} \approx d_{i,j} - d'_{i,j}$. Similarly, we have $\delta_{j,i} \approx d_{j,i} - d'_{j,i}$. We use δ to denote the sum of $\delta_{i,j}$ and $\delta_{j,i}$. We have the following equation:

$$\delta \approx (d_{i,j} + d_{j,i}) - (d'_{i,j} + d'_{j,i}). \quad (4)$$

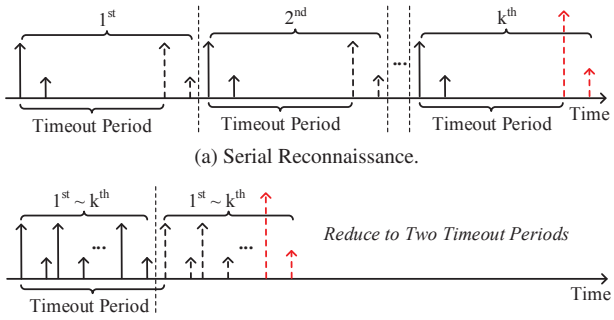
Note that, $d_{i,j} + d_{j,i}$ is the round-trip-time (RTT) of the packet that is not matched by rules, and $d'_{i,j} + d'_{j,i}$ is the RTT of the same packet matched by rules. Thus, we can infer the delay of control paths between two hosts by subtracting RTTs of these two crafted packets.

Identifying a Target Data Path. An attacker needs to send two packet streams for each possible data path in order to find a target data path crossing with some control paths, i.e., a data path containing shared links. The first packet stream is a *timing stream*, which aims to measure the delay δ shown in equation (4). The timing stream must trigger responses from the destination host in the current data path. Fortunately, many types of packets meet the requirement, such as ICMP packets, TCP SYN packets, and HTTP request packets. Moreover, each timing stream must contain a pair of



(a) A packet matches no rules in s_1. (b) A packet matches rules in all switches.

Figure 2: Different forwarding paths and delays for packets sent from h_i to h_j. c denotes the controller and s_i denotes the ith switch in the packet path.



(a) Serial Reconnaissance.

(b) Parallel Reconnaissance.

Figure 3: Two different reconnaissances of finding a target data path. Each arrow denotes a timing packet and the height of it denotes the RTT of a timing packet. A dashed arrow denotes testing packets are sent at the same time. The red arrows denote a target path is found when conducting a reconnaissance on the kth data path.

packets. The first packet must trigger new rule installation and the second packet must match the newly installed rules. This can be achieved by waiting a long enough time before sending the first timing packet to the destination, and then immediately sending another same packet after receiving a response from the first packet. The first packet can guarantee new rule installation, since old rules will be expired due to timeouts as we mentioned in Section 2. According to the previous study [44], the timeouts are usually configured as small values in order to save space of flow table and waiting for 30 seconds is enough for most cases.

The second packet stream is a *testing stream*. It contains a short-term burst of packets sent to the destination host in the current data path. These packets in the stream can be typically UDP packets. TCP packets can also be chosen if we send them with raw sockets [15] to eliminate the automatic rate control in TCP. The testing stream can be used to test whether the current data path crosses with some control paths or not in collaboration with the testing stream. An attacker can first measure the delay δ by the timing stream without transmitting the testing stream to the destination. After waiting enough time to ensure that old flow rules expire, an attacker can measure the delay again (denoted by δ') with the testing stream being transmitted at the same time. By comparing these two delays, an attacker can obtain:

(i) If δ' is significantly higher than δ, the short-term burst of packets affects the delays of some control paths. Thus, the data path currently being explored crosses with some control paths.

(ii) If δ' is similar to δ, no available evidence indicates that the data path crosses with some control paths.

Thus, we are able to find a target path by testing each path if it exists.

3.3 Improved Reconnaissance

In order to efficiently and accurately find a target data path, we apply two methods to improve our reconnaissance.

Improving Accuracy with *T-test*. Although our reconnaissance allows an attacker to know whether a data path crosses with control paths by sending only four packets, it may achieve low accuracy in practice. Various network noises can affect the reconnaissance. For example, a burst of benign traffic can also cause high latencies of control paths, which makes a non-target data path misidentified as a target data path. We find that t-test [20] can be a straightforward approach to eliminate the influences of network noise as much as possible. T-test is a statistical method that compares whether two groups of samples with random noises belong to the same distribution. It produces a p value to denote the likelihood that the two groups of samples belong to the same distribution. Typically, if p is less than a predetermined value, i.e., the significance level α [20], the two groups are considered significantly different. Thus, we can collect two groups of latencies with or without a testing stream for a data path, and apply t-test to determine whether a data path crosses with control paths according to the p value.

Improving Efficiency with Parallelization. Basically, an attacker can try to test each data path one-by-one, which is shown in Figure 3a. However, it is time-consuming. An attacker has to wait for at least a timeout value before conducting next round of testing, as obtaining the latencies of control paths with testing stream requires that the old rules have been removed. Suppose that a network has 100 data paths and the timeouts in flow rules are configured to 10s. Moreover, we assume 10 repeated reconnaissances are conducted for each path in order to apply t-test. We can calculate that finding a target path needs approximate 10,000s at the worst case, which is unbearable in practice. Fortunately, different flow rules matching specific packets make up different data paths in SDN, which means the installation and expiration of rules in two different paths are independent. Thus, the reconnaissance can be parallelized to reduce

the time. As shown in Figure 3b, an attacker can choose k pending paths. The latencies of their crossed control paths can be measured in turn by sending two timing packets for each data path. After waiting for only one timeout value, an attacker can measure the latencies again in turn while transmitting corresponding testing streams, since the old rules of each data path will expire in turn. The parallel reconnaissance allows an attacker to explore k data paths within two timeout values, which significantly improves efficiency. The maximal k depends on the maximal timeout values of flow rules and the maximal RTT of timing packets. In order to find a target data path as fast as possible, k should be subject to the inequation: $2 \cdot k \cdot RTT_{max} < timeout_{max}$. It ensures that an attacker can check whether there is a target path among k data paths within two timeout periods. If the maximal RTT of the timing packets is 20 ms in the target SDN, the parallel reconnaissance can dramatically reduce the time used by the previous example from 10,000s to less than 100s.

Based on the above designs, the algorithm of improved adversarial path reconnaissance can be easily implemented. Due to space constraints, for further details, we refer the reader to see the pseudo-code in Appendix A.

3.4 Theoretical Analysis

To understand the feasibility and efficiency of the adversarial path reconnaissance in SDN, we perform theoretical analysis to answer the following two questions:

- If there exists target data paths crossing with control paths in the network, which conditions the network must meet so that our reconnaissance can identify a target data path?
- How many data paths should be explored in order to find a target data path?

Firstly, we use an example to illustrate the network conditions that must meet for identifying a target data path before presenting the theory results. Figure 4 shows the target network where an attacker conducts reconnaissances. Each switch connects the controller through the shortest control paths. Switch s_2 and s_3 both have two different shortest control paths that can be chosen. We first consider the case where s_2 connects the controller via $< l_{s_2 \to s_5}, l_{s_5 \to s_6}, l_{s_6 \to c} >$ and s_3 connects the controller via $< l_{s_3 \to s_2}, l_{s_2 \to s_1}, l_{s_1 \to s_6}, l_{s_6 \to c} >$. Obviously, the data path from h_1 to h_2 crosses with the control path of s_3. However, an attacker cannot identify it. Measuring the delay of the crossed control paths is infeasible, since an adversary cannot trigger rule installation into s_3. If we consider another case where s_2 connects the controller via $< l_{s_2 \to s_1}, l_{s_1 \to s_6}, l_{s_6 \to c} >$ and s_3 connects the controller via $< l_{s_3 \to s_2}, l_{s_2 \to s_5}, l_{s_5 \to s_6}, l_{s_6 \to c} >$, the target data path from h_1 to h_2 crossing with the control path of s_2 can be identified. The main difference between

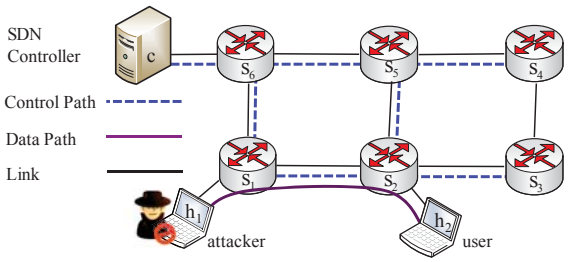


Figure 4: The target network where an attacker conducts reconnaissances.

the two cases is whether the target data path crosses with a control path of a switch belonging to the data path.

We consider a set of all the hosts in the target network $H = \{h_1, h_2, ..., h_n\}$, a set of compromised hosts $\tilde{H} = \{\tilde{h}_1, \tilde{h}_2, ..., \tilde{h}_q\}$, and a set of all the switches in the network $S = \{s_1, s_2, ..., s_m\}$. Let $p_d^{i,j}$ be the data path from host i to host j, let p_c^i be the control path of switch i, and let $S_{i,j} = \{s_1, s_2, ..., s_r\}$ be the set of switches belonging to the data path from host i to host j. Here, $\tilde{H} \subset H$ and $S_{i,j} \subset S$. $p_d^{i,j}$ and p_c^i both is a set that contains a sequence of consecutive links. In fact, we have the following theorem:

Theorem 1. *If and only if the target SDN network meets the condition:* $\exists(p_c^i \cap p_d^{j,k} \neq \emptyset)$, *where* $i \in S_{j,k}$, $j \in \tilde{H}$, $k \in H$ *and* $j \neq k$, *then there exists a target data path which can be identified by the adversarial path reconnaissance.*

Proof. We prove the theorem in two steps. We first prove the sufficient condition, i.e., if the target network meets the conditions in Theorem 1, then a target data path can be identified by the adversarial path reconnaissance. According to the conditions, we can know that a data path $p_d^{j,k}$ from a compromised host $\tilde{h}_j$ to another host h_k crosses with a control path p_c^i. The crossed control paths belong to the switches $S_{j,k}$ along the data path. An attacker can conduct the adversarial path reconnaissance on the data path. Basically, four timing packets will be sent to the data path. The first timing packet will trigger rule installation into all switches along the data path. Only after all switches finished installing rules according to the messages of the controller, the packet can reach the destination and a response packet will be sent to the compromised host. Thus, the RTT of the timing packet contains total latencies of control paths of all switches in $S_{j,k}$. The second timing packet will be sent after rule installation. The total latencies of control paths can be obtained by subtracting the RTTs of these two timing packets. After waiting at least a timeout value, another two timing packets can be sent to the data path with testing stream. The total latencies of control paths can be obtained again in a similar way; however, crossed control path p_c^i will be affected by the test stream. The reconnaissance will notice that the total latencies will be significantly higher than the previous latencies.

Thus, a target data path $p_d^{j,k}$ is identified.

We next prove the necessary condition, i.e., if a target data path can be identified by the adversarial path reconnaissance, then the target network meets the conditions in Theorem 1. We assume that a target data path $p_d^{j,k}$ is identified. Since $p_d^{j,k}$ is a target data path, it at least crosses with a control path p_c^i. Obviously, the reconnaissance can only be launched by the compromised hosts. Thus, $j \in \tilde{H}$, $k \in H$, and $j \neq k$. We only need to prove that the crossed control path belongs to a switch along with the data path $p_d^{j,k}$, i.e., $i \in S_{j,k}$. Let us consider the opposing case $i \notin S_{j,k}$. Note that the timing packets in our reconnaissance trigger rule installation into switches $S_{j,k}$ along the data path. Thus, only the latencies of control paths belonging to the switches in $S_{j,k}$ can be measured. When $i \notin S_{j,k}$, the delay variation of p_c^i cannot be noticed by our reconnaissance. Thus, there must be $i \in S_{j,k}$ if a target data path can be identified. □

Theorem 1 indicates that our reconnaissance can find a target data path only if the network meets the conditions. Fortunately, it only requires at least one data path which crosses with a control path of switches that are in the data path. Such conditions can be easily met in practice. We will show that our reconnaissance can find a target data path with various real network topologies for most cases in Section 4.1.

In order to estimate the average number of explored data paths for finding a target data path, we introduce a parameter γ denoting the total number of target data paths which can be identified in a network. In addition to the notations we used in Theorem 1, let ρ be the total number of data paths between a compromised host in $\tilde{H}$ and a host in H, and let X be a random variable denoting the number of explored data paths for finding a target data path. Obviously, if we find a target data path at the k_{th} exploration, then we have already failed to find a target data path for $k-1$ times. Thus, the probability of finding a target data path at the k_{th} exploration for the first time is:

$$P(X=k) = \frac{\gamma}{\rho-(k-1)} \prod_{j=0}^{k-2} \frac{\rho-\gamma-j}{\rho-j}, \tag{5}$$

where $1 \leq k \leq \rho-\gamma+1$. Here, we define $\prod_{j=x}^{y} a = 1$, when $x > y$. The average number of explored data paths can be calculated as:

$$\begin{aligned} E(X) &= \sum_{k=1}^{\rho-\gamma+1} k \cdot P(X=k) \\ &= \sum_{k=1}^{\rho-\gamma+1} \frac{k\gamma}{\rho-(k-1)} \prod_{j=0}^{k-2} \frac{\rho-\gamma-j}{\rho-j}. \end{aligned} \tag{6}$$

If we consider the case where there is only one compromised host in the network and each of the data paths between two hosts is different, then $\rho = n-1$. n is the number of hosts in the network. Equation (6) can be simplified as:

$$E(X) = \sum_{k=1}^{n-\gamma} \frac{k\gamma}{n-k} \prod_{j=0}^{k-2} \left(1 - \frac{\gamma}{n-1-j}\right). \tag{7}$$

Equation (7) indicates the average number of explored data paths $E(X)$ totally depends on n and γ. We will show that $E(x)$ gets small values with proper parameters and the theoretical values are consistent with our experimental values in Section 4.1. In reality, our reconnaissance can quickly find a target data path by exploring several data paths (see Figure 6 in Section 4.1).

4 Attack Evaluation

In this section, we perform large-scale simulations to demonstrate that the CrossPath attack can be launched with various network topologies. Moreover, we conduct experiments to evaluate the feasibility and effectiveness of the attack in a real SDN testbed.

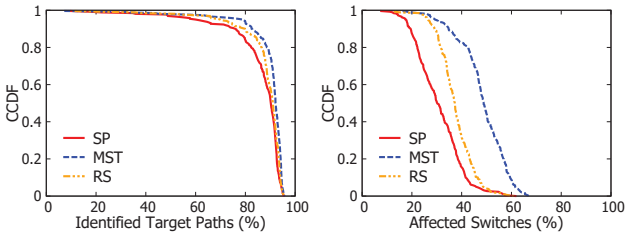
4.1 Large-Scale Simulation Experiments

Simulation Setup. We perform simulations with 261 real network topologies [4] around the world. As these network topologies do not contain hosts and routing information, we generate 100 hosts [1] in each topology and apply Dijkstra's algorithm [28] to generate the shortest data path between two hosts. Note that shortest path forwarding is commonly used in the intra-domain routing system. We add another host in each network topology as the SDN controller. The controller can connect switches via shortest paths (SP) to minimize delays, a minimum spanning tree (MST) to minimize costs, or randomly searching available paths (RS). We conduct experiments with different types of connection in turn. Moreover, for simplicity and without loss of generality, we assume that the attacker only controls one host in the network and we attach such a host to each network topology.

We note that the positions of hosts in a network will affect our experimental results. Thus, we conduct 1,000 experiments for each network topology and randomly changes the positions of all hosts in each experiment. We show the average results over 1,000 experiments for each topology.

Average Percentage of Identified Target Paths. Figure 5a shows the CCDF of the average percentage of identified target paths with 261 various network topologies. From the results, we can see all the network topologies have at least 5% identified target paths among total data paths in a network regardless of types of connections. More than 98% of the network topologies have at least 30% identified target paths. Moreover, the network tends to have more identified data paths when the controller connects switches via MST.

[1] In reality, we also conduct our experiments with 50, 500, 1000 hosts, respectively. The results are similar to those in Figure 5.



(a) (b)

Figure 5: Complementary Cumulative Distribution Function (CCDF). (a) shows the CCDF of the average percentage of identified target paths with 261 real topologies; (b) shows the CCDF of the average percentage of affected switches by attacking a target path with 261 real topologies.

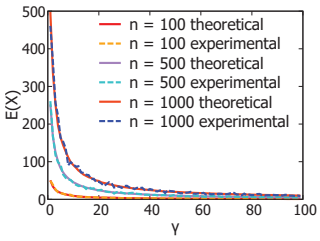


Figure 6: Comparison of theoretical values and experimental values of $E(X)$ with different n and γ.

The results demonstrate that the conditions in Theorem 1 can be easily met. An attacker can use our reconnaissance to find some target data paths to launch the CrossPath attack.

Average Percentage of Affected Switches. As attacking different target paths will affect the average percentage of switches in a network topology, we randomly attack a target path in the 1,000 experiments for a network topology and calculate the average percentage of affected switches. Figure 5b shows that more than 20% of the switches can be affected by attacking a target path for 90%, 99% and 99% of the 261 network topologies with SP, MST and RS connections, respectively. For some network topologies, attacking a target path can even affect half of the whole switches. Thus, it is possible for an attacker to attack multiple target paths to cause damages for the whole switches and incur network-wide DoS.

Average Number of Explored Data Paths. Equation (7) denotes the average number of explored data paths $E(X)$ for finding a target path totally depends on the number of data paths γ containing shared links and the number of hosts in a network n. We draw the theoretical values of $E(X)$ in Figure 6. We can see that $E(x)$ declines quickly when γ increases from 0 to 20. When there are 1,000 hosts and 40 data paths (2% of the 1,000 total data paths) containing shared links, $E(X)$ is less than 50. Moreover, $E(x)$ tends to be the same with the growth of γ. The results demonstrate that our reconnaissance can fast find a target data path and has a good scalability with a different number of hosts in the network. The experimental values of $E(x)$ are also plotted in Figure 6. Each experimental value with different n and γ is obtained by conducting 1,000 experiments to get the average number of explored data paths. The results show that the experimental values are consistent with the theoretical values.

4.2 Experiments in a Real SDN Testbed

Experiment Setup. Our testbed contains a popular SDN controller Floodlight [12], five hardware SDN switches (AS4610-54T [10]), and three physical hosts. The controller is deployed on a server with a quad-core Intel Xeon CPU E5504 and 32GB RAM. Each physical host has a quad-core Intel i3 CPU and 4GB RAM. All hosts run Ubuntu 14.04 server LTS. The network topologies, control paths and data paths are illustrated in Figure 1. An attacker first compromises host h_1 to conduct the algorithm of adversarial path reconnaissance (see Appendix A for details) for the data paths of the other hosts. The burst rate of short-term testing packets is 1 Gbps, which is the maximal rate the host can send.

The attacker then generates LDoS data traffic to disrupt the control channels of switches s_1 and s_2 by attacking the data path between h_1 and h_3. Basically, there are three parameters for the LDoS flows: burst length, inter-burst period, and peak magnitude. The previous study [42] has conducted comprehensive experiments on how different parameters determine the attack impacts of LDoS flows and how to better choose these parameters. As our paper mainly focus on studying the impacts for the SDN functionalities after the control channel is attacked by the data traffic, we apply fixed parameters in our attack. We choose the burst length as 100 ms, inter-burst period as 200 ms, and peak magnitude as the maximal speed 1 Gbps that the host can send for our all experiments in the paper. These parameters show how an attacker can affect the SDN functionalities to the maximum extent by generating data traffic to disrupt the control channel. Moreover, compared to simply flooding the target paths, which needs to send traffic with 1 Gbps all the time, the rate of our LDoS flow is only approximate 0.33 Gbps on average.

Accuracy of Reconnaissances. We first collect the delay variations in delivering control messages. The delay variation is defined as the absolute difference between the delays of control messages measured with and without testing stream. We collect 5,000 records both for two data paths in the network. We wait up to 20 seconds for each timing packet to get a response in order to obtain possible maximum delays. Figure 7 shows the distribution of the probability of the delay variation. The results demonstrate that the target data path has a significantly different probability distribution compared with the non-target data path. In particular, most delay variations with the non-target data path are less than 2 ms, while most delay variations are much larger for the target data path. These results illustrate that the discrimination

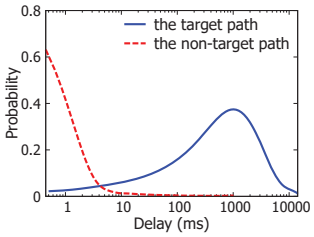


Figure 7: Probability distribution of delay variations.

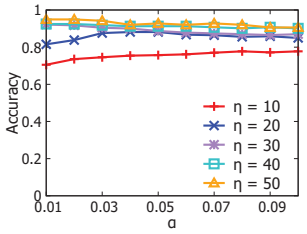


Figure 8: Accuracy of reconnaissances with different parameters.

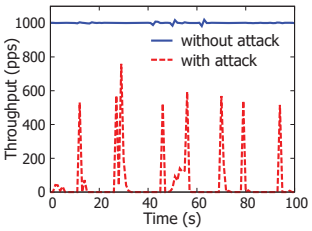


Figure 9: Throughput of control packets.

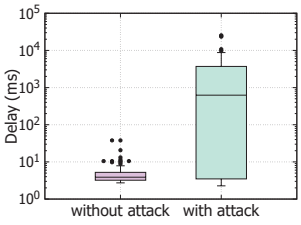


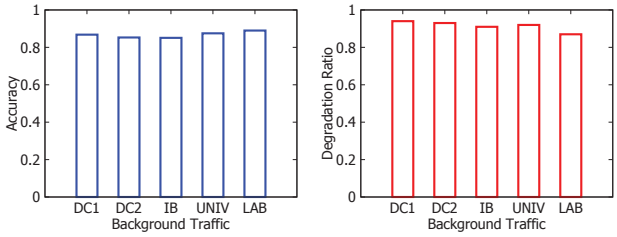
Figure 10: Delays of control packets.

between target data paths and non-target data paths can be easily identified according to the delay variations.

We then calculate the accuracy of our reconnaissance by conducting 1,000 repeated experiments with different settings of η and α. Here, η denotes the number of measured delays for each data path, which is also the size of each group in the t-test used to identify a target path. α is the significance level used in the t-test. As shown in Figure 8, the accuracy increases with the increase of η. Moreover, we can observe that the accuracy increases with the increase of α when η is smaller, e.g., 10 or 20. However, the accuracy tends to be stable when η becomes large. The reason is that two different groups will statistically different from each other and two similar groups will be statistically closer to each other with more data. It is easier to distinguish the two types of paths if we have enough data, which is not significantly impacted by the setting of α. The accuracy always reaches more than 90% with different settings of α when η is 40 or 50.

Effectiveness of the Attack. To evaluate the impact of the attack on the control packets, we configure the controller to generate 1,000 control packets per second[2] to the switch s_2. Figure 9 shows the throughput of control packets. The throughput can achieve 1,000 packets per second. However, it almost drops to 0 under the attack though there are short-term peaks of throughput. The reason is that our attack triggers TCP of control flows to periodically enter the phase of retransmission timeout. In this case, no packets will be sent within the retransmission timeout. Figure 10 shows the delay

[2]There can be thousands of control packets per second [29]. For simplicity but without loss of generality, we choose a practical value, 1,000.



(a) Accuracy of Reconnaissances. (b) Degradation Ratio of Control Traffic.

Figure 11: Robustness of the attack with different background traffic.

of control packets. The median value of delays for control packets under the attack is 687 ms, which is more than about 100 times higher than that in absence of the attack. Moreover, the delays under the attack vary within a large range from below 10 ms and to more than 10,000 ms. Note that, most delays without the attack are less than 10 ms. The results above demonstrate our attack can significantly degrade the throughput of control packets and incur high delays.

Robustness of the Attack. As background traffic may affect the reconnaissances and attack effects, we inject different background traffic into our network with TCPReplay [16] in turn. Such traffic traces comes from two Data Centers (DC1 and DC2) [3], an Internet Backbone (IB) [8], a University (UNIV) [18] and our Laboratory (LAB). Moreover, due to the limited flow table capacity in switches, we randomly choose flows from the trace to ensure that the number of rules generated by flows do not exceed the table capacity.

Figure 11a shows the accuracy of reconnaissances with different background traffic. The parameters of reconnaissances α and η are set to 0.01 and 50, respectively, which are the best parameters to get the highest accuracy (93% in Figure 8) without background traffic. When the background traffic is injected, the accuracy drops to below 90%, ranging from 85% to 89%. However, such accuracy is still satisfactory for an attacker to conduct reconnaissances. Figure 11b shows the degradation ratio of control packets. The degradation ratio is the fraction of the control packets reduced by the attack over the total control packets without the attack. We can see that the attack always causes more than 90% degradation ratio with different background traffic. Above results demonstrate that our attack achieves high robustness.

5 Attack Impacts on Network Functionalities

In this section, we perform a systematical study on the impacts of the attack on various network functionalities. We first review the common core services enabled in SDN controllers that generate different types of OpenFlow control messages and are used by various SDN applications. We then study four typical SDN applications, which use these common core services, so that we measure the impacts of

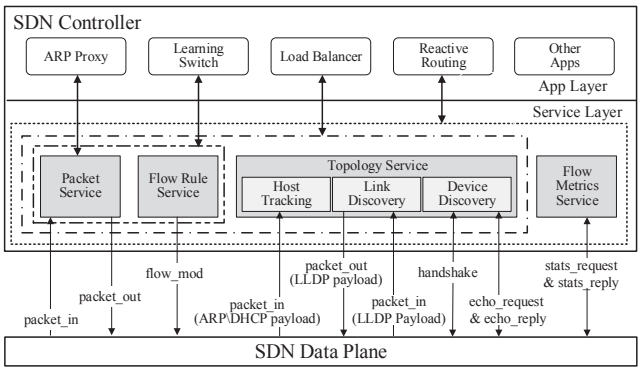
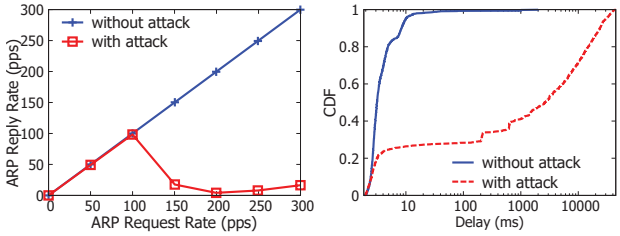


Figure 12: The core services of SDN controllers.



(a) ARP Throughput. (b) CDF of ARP Query Delay.

Figure 13: Attack impacts on ARP Proxy.

the attack on SDN functionalities.

5.1 Core Services of SDN

SDN controllers can be abstracted as a two-layer architecture though different controllers have different implementations. Applications can be deployed in the top layer to enable different network functionalities, while the low layer provides different core services that interact with switches and provide basic functionalities for the top-tier applications. As shown in Figure 12, there are four major core services:

Packet Service. The service manages packets exchanged between the control and data planes. It paraphrases *packet_in* messages containing data packets received from switches and dispatch them to applications. Meanwhile, it sends data packets back to switches via *packet_out* messages.

Flow Rule Service. The service manages flow rules. It installs or updates rules in switches via *flow_mod* messages according to the results computed by applications.

Topology Service. The service maintains the topology of end hosts, links, and switches. It discoveries new hosts and tracks their locations via *packet_in* messages embedded with an ARP or DHCP payload. It periodically sends and receives LLDP packets encapsulated in *packet_in* or *packet_out* messages to maintain link information. Besides, it establishes the control channel between switches and controllers via several *handshake* messages. The liveness of switches is periodically checked via *echo_request* and *echo_reply* messages. Applications obtain network topologies through the service.

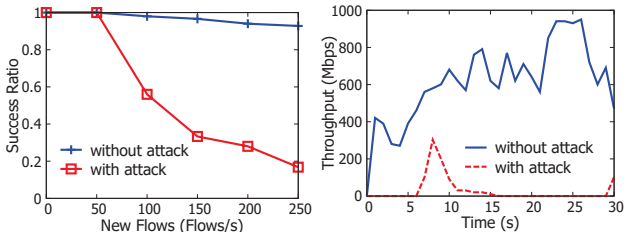
Flow Metrics Service. The subsystem is responsible for collecting flow statistics. It periodically queries the flows on network devices via *stats_request* and *stats_reply* messages, and then provides various statistics to applications.

We note that almost all applications enabling network functionalities in SDN is built on at least one of the four services. Our attack thus can affect various SDN functionalities by disrupting the transmission of control messages exchanged between these core services and switches. We will choose four typical applications that are widely deployed in SDN controllers to show the impacts of the attack on various network functionalities. The implementations of the four applications [5, 6, 9, 7] are from Floodlight [12].

5.2 ARP Proxy

SDN enables Address Resolution Protocol (ARP) similar to IP networks, which finds the association between a destination IP address and its corresponding hardware (MAC) address so that hosts can correctly send and receive IP packets. In IP networks, layer two switches flood an ARP request sent from a host to get an ARP reply. If the target IP address in the ARP request is not in the local network, a router acts as an ARP proxy to send back an ARP reply with the hardware address of its own interface. In SDN, ARP packets are handled by an *ARP proxy* application [5] in the SDN controller. When an ARP request sent by a host arrives at a switch, it will be sent to the controller via *packet_in* messages. The packet service extracts the ARP request packet from *packet_in* messages and dispatches the packet to the ARP proxy application. The application extracts the sender IP address and the source MAC address to store them into the ARP table. Meanwhile, it finds an entry that the IP address matches the target IP address in the ARP request. A corresponding ARP reply packet is created and will be sent back to the ingress switch via *packet_out* messages. Thus, the original host obtains an ARP reply.

Our attack can completely disrupt the functionality of ARP proxy by interfering with the exchange of the messages between the packet service and switches. Figure 13a shows the ARP throughput. The ARP reply rate is proportional to the ARP request rate in absence of the attack. However, under the attack, the ARP reply rate falls below 10 pps when the ARP request rate exceeds 100 pps. The reason is that the TCP flows of control traffic frequently enter the retransmission timeout phase under the attack due to the congestion. Figure 13b shows the CDF of ARP delays. More than 90% delays are less than 10 ms without the attack, while more than 70% delays are higher than 10 ms and more than 50% delays are higher than 1,000 ms with the attack. Delays under attacks significantly increase. Particularly, some delays exceed 10,000 ms, which can cause connection failures between two hosts because hosts cannot get MAC addresses.



(a) Success Ratio of Rule Installation for a Switch.

(b) Throughput for a Switch with 250 Flows/s.

Figure 14: Attack impacts on Learning Switch.

5.3 Learning Switch

The *learning switch* application [6] allows SDN switches act as normal switches in IP networks. The application examines a packet matching no rules in a switch and looks up the recorded mapping between the source MAC address and the port. If the destination MAC address has already been associated with a port, the packet will be sent to the port and corresponding rules will be installed to match subsequent packets. Otherwise, the packet will be flooded on all ports. As shown in 12, the application relies on two services. The packet service sends the packet to the controller via *packet_in* messages and back to the switch via *packet_out* messages, and the flow rule service installs rules in the switch via $flow_mod$ messages.

Our attack can effectively block installation of forwarding decisions generated by the application by disturbing the messages exchanged between the core services and switches. Figure 14 shows the impacts of the attack on the functionalities of *learning switch*. Here, we define the success ratio of rule installation as the number of successfully installed rules over the number of rule requests within a second. As shown in Figure 14a, the success ratio of rule installation in a switch always maintains over 90% with various numbers of new flows without our attack. However, it drops significantly in presence of our attack. When the rate of new flows reaches 250 flows per-second, the success ratio reduces to below 20%. Thus, *learning switch* cannot work correctly. As shown in Figure 14b, the throughput of a switch is 0 Mbps for a long time with attack when there are 250 flows/s.

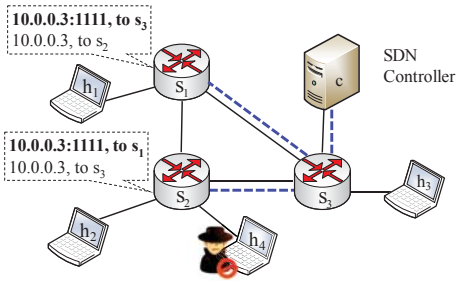
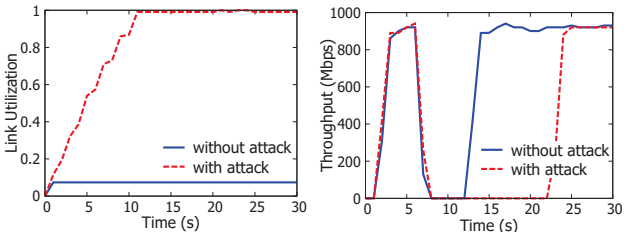


Figure 15: The network topology used in Reactive Routing.



(a) Increasing link utilization due to long-term routing rule inconsistency.

(b) Long-term routing blackhole due to delayed messages when a host is migrated.

```
17:37:46.344 INFO [n.f.t.TopologyInstance] Route [id=RouteId [src=1c:48:cc:37:ab:a0:a8:41
dst=9d:54:cc:37:ab:a0:a8:41], switchPorts=[[id=1c:48:cc:37:ab:a0:a8:41, port=37],
[id=9d:54:cc:37:ab:a0:a8:41, port=31]]]
17:38:01.62 INFO [n.f.l.i.LinkDiscoveryManager] Inter-switch link removed: Link
[src=a4:e7:cc:37:ab:a0:a8:41 outPort=38, dst=9d:54:cc:37:ab:a0:a8:41, inPort=42, latency=6]
17:38:01.95 INFO [n.f.t.TopologyManager] Recomputing topology due to: link-discovery-
updates
17:38:01.345 INFO [n.f.t.TopologyInstance] Route [id=RouteId [src=1c:48:cc:37:ab:a0:a8:41
dst=9d:54:cc:37:ab:a0:a8:41], switchPorts=[[id=1c:48:cc:37:ab:a0:a8:41, port=32],
[id=a4:e7:cc:37:ab:a0:a8:41, port=36], [id=a4:e7:cc:37:ab:a0:a8:41, port=38],
[id=9d:54:cc:37:ab:a0:a8:41, port=42]]]
```

(c) Eviction of a routing path due to a deactivated link.

```
20:55:25.510 INFO[n.f.c.i.OFChannelHandler] [[1c:48:cc:37:ab:a0:a8:41(0x0) from
192.168.100.1:59413]] Disconnected connection
20:55:26.218 INFO[n.f.c.i.OFChannelHandler] New switch connection from
/192.168.100.1:59414
20:55:27.755 INFO[n.f.c.i.OFChannelHandler] Negotiated down to switch OpenFlow version
of OF_14 for /192.168.100.1:59414 using lesser hello header algorithm.
20:55:31.698 INFO[n.f.c.i.OFSwitchHandshakeHandler] Clearing flow tables of
1c:48:cc:37:ab:a0:a8:41 on upcoming transition to MASTER.
```

(d) Cleaning of flow tables due to the reset of a switch.

Figure 16: Attack impacts on Reactive Routing.

5.4 Reactive Routing

The *reactive routing* [9] application enables flexible and fine-grained routing decisions for different flows, which is enabled in almost all controllers. When a new flow matching no rules is generated, the first packet of the flow will be sent to the *reactive routing* application. The application analyzes the packet and calculates routing paths for the new flow. Besides depending on the packet service processing data packets and flow rule service installing rules, the application also queries the topology service that provides the information of the locations of hosts, the state of switches and links.

In order to demonstrate the effectiveness of our attack, we build a network topology with four hosts and three switches, as shown in Figure 15. The IP addresses of the four hosts h_1, h_2, h_3 and h_4 are 10.0.0.1, 10.0.0.2, 10.0.0.3, and 10.0.0.4, respectively. The hosts h_1 and h_2 send packets to the host h_3. The default routing path of packets from h_1 to h_3 is $< l_{h_1 \to s_1}, l_{s_1 \to s_2}, l_{s_2 \to s_3}, l_{s_3 \to h_3} >$. The default routing path of packets from h_2 to h_3 is $< l_{h_2 \to s_2}, l_{s_2 \to s_3}, l_{s_3 \to h_3} >$. Also, a flow with TCP port 1111 from h_2 to h_3 has a different path due to a QoS requirement. Here, the compromised host h_4 sends attack (i.e., LDoS) traffic to h_3 in order to exploit the control path of switch s_2.

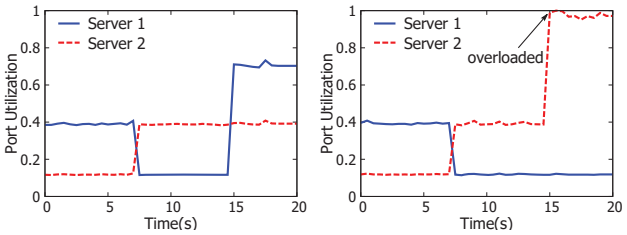
Figure 16 shows the impacts of the attack on *reactive routing*. As shown in Figure 16a, our attack incurs long-term routing rule inconsistency, which makes the link utilization reach 100%. The reason is that SDN exists transient rule inconsistency [36] which can be leveraged by our attack. In the network shown in Figure 15, packets with an IP destination address 10.0.0.3 and a destination port 1111 loop between s_1 and s_2 when the application deletes rule "10.0.0.3 : 1111, *to* s_3" while rule "10.0.0.3 : 1111, *to* s_1" remains. The rule inconsistency normally lasts for a very short period before all the commands of deleting corresponding rules of the flow are issued. However, our attack can delay the commands exchanged between the flow rule service and s_2 for tens of seconds. Thus, the packets loop between s_1 and s_2 for a long period and the link utilization between the two switches increases with more packets injected.

Figure 16b shows the long-term routing blackhole when h_3 is migrated from s_3 to s_2. The migration is finished within five seconds without the attack, as the topology service can track the new location via *packet_in* messages containing the DHCP payload when the host moves to s_2. However, the messages are significantly delayed under our attack, and thereby the routing between other hosts and h_3 cannot be updated in time, causing more than 10 seconds routing blackhole. Moreover, by blocking LLDP packets between the topology service and switches, our attack can deactivate links in the topology database and thus the corresponding routing paths will be removed. In the Floodlight controller, a link will be deactivated if no LLDP packets pass through the links within 35s. Figure 16c shows the original routing path from h_2 to h_3 is removed since our attack deactivates the link from s_2 to s_3. Moreover, our attack can reset the connections between switches and the controller by delaying control messages. Figure 16d shows the connection of switch s_2 is reset and all the flow tables are cleaned.

5.5 Load Balancer

Load balancing has been widely used to improve resource usage and throughput as well as reduce response delays, which balances the workload among multiple nodes. SDN controllers deploy the *load balancer* [7] application to achieve the goal. The application in the Floodlight controller can balance requests of clients in two way, i.e., round robin and statistics-based scheduling. Round robin scheduling randomly chooses a server from a server pool to serve a new request each time. The statistics-based scheduling chooses a server that has the lowest utilization to serve a new request, where the utilization is calculated according to the real-time statistics of the switch ports. The *load balancer* application relies on the flow metrics service to collect the statistics.

We configure the *load balancer* application in Floodlight to enable statistics-based scheduling, as it can provide better load balancing under different flow distribution of clients. In our experiments, two hosts consist of a server pool and another two hosts send flows to the servers. Figure 17a shows the utilization of switch ports connecting the two servers over time without our attack. Initially, two different elephant flows are sent to the servers, which causes the port utilization to increase to 40% and 10%, respectively. At the 7th second, the rate of the two flows exchanges. The utilization of one server reduces from 40% to 10% while another server increases from 10% to 40%. At the 14th second, a new elephant flow starts, and the application directs the flow to server #1 that has the lowest port utilization. The port utilization of server #1 reaches 70%. Unfortunately, the application will mistakenly direct the flow to server #2 under our attack. As shown in Figure 17b, the port utilization of server #2 reaches 100%. The reason is that our attack can significantly delay the *stats_request* and *stats_reply* messages exchanged between the flow metrics service and switches, and thus the applications cannot know the port utilization in time. Actually, the application considers that the port utilization of server #2 is still 10% when the new flow comes.



(a) Port Utilization of Servers without Attack.

(b) Port Utilization of Servers with Attack.

Figure 17: Attack impacts on Load Balancer for misallocating the workloads across servers.

6 Defense Schemes

In this section, we discuss possible countermeasures that network administrators can be used to mitigate the attack.

Delivering Control Traffic with High Priority. To defend against the attack, one way is to ensure forwarding control traffic with high priority, which thus can protect control traffic from being congested by malicious data traffic. According to our analysis, such a defense scheme can be enforced by carefully configuring Priority Queue (PQ) or Weighted Round Robin Queue (WRR) in switches. We note that many commercial SDN switches support at least one of the two queueing mechanisms (see Appendix C). We implement the defense scheme based on PQ and WRR in our hardware switches to deliver control traffic with high priority. The evaluation shows it can effectively protect control traffic against malicious data traffic. The detailed implementations and evaluations can be found in Appendix B.

Proactively Reserving Bandwidth for Control Traffic.

Another way to defend against the attack is to proactively reserve proprietary bandwidth for control traffic. Such a defense scheme is suitable for SDN switches that do not support PQ and WRR mechanisms. We implement the defense scheme with OpenFlow meter table in our hardware switches. We have demonstrated that control traffic can be well protected by reserving enough bandwidth. We refer the reader to Appendix B for details. The main disadvantage of the defense scheme is that the reserved bandwidth cannot be used by other traffic even there is massive free bandwidth. Our future work will focus on how to dynamically reserve the bandwidth for control traffic to make full use of it.

Disturbing Path Reconnaissances. The necessary condition to successfully launch the CrossPath attack is to find a target path containing shared links. Thus, we can prevent the attack by disturbing path reconnaissances. One way is to deliberately add random delays when installing flow rules, which may result in incorrect delay measurements of control paths when conducting path reconnaissances. Our evaluation shows that the accuracy of path reconnaissances can drops to less than 30% by adding random delays ranging from 100 ms to 1,000 ms. However, adding random delays affects the rule installation of all flows in the network. It is especially harmful to mice flows that are delay-sensitive [30]. Designing a scheme to effectively disturb path reconnaissances and reduce the impacts on network flows is worth more future research.

7 Related Work

In this section, we review related security research in SDN and legacy networks, respectively.

Reconnaissances in SDN. SDN reconnaissances has been extensively studied. Shin et al. [54] designed an SDN scanner to determine whether a network is SDN by measuring response delays of pings. Cui et al. [25] further conducted experiments in real SDN testbed to demonstrate its feasibility. Klöti et al. [39] presented a reconnaissance technique to determine if an SDN has rules for aggregated TCP flows by timing the TCP setup time. Achleitner et al. [19] designed SDNMap to reconstruct composition of flow rules by analyzing probing packets with specific protocols. Liu et al. [45] developed a Markov model to reveal rule distribution among switches. John et al. [56] presented a sophisticated inference attack to learn host communication patterns and ACL entries even if injected packets do not trigger replies. However, none of the methods can be applied to find target paths containing shared links with control paths.

Attacks on SDN and Related Defenses. SE-Floodlight [48] and SDNShield [63] are developed to provide permission control for malicious SDN applications. Some studies focus on the security of controllers, including network poisoning [31], identifier binding attacks [35], subverting SDN controllers [49], and exploiting harmful race conditions in SDN controllers [65]. Other studies focus on data plane security, including low-rate flow table overflow attacks [22], SDN teleportation, and detection on abnormal data plane [51]. Our paper focuses on the security of control channel, which is orthogonal to the existing work. Particularly, we uncover a new type of attack, which has not been discovered by existing automatic attack discovery tools [34, 43, 59] in SDN.

The packet_in flooding attack [55, 60] is mostly closest to ours. It saturates the control channel with a large amount of packet_in messages. To trigger the control messages, the attack requires generating massive bogus packets matching no rules in switches. Different from it, our attack generates low-rate data traffic to implicitly disrupt control traffic in the shared links instead of directly generating massive control traffic. Our attack can bypass the previous defenses [55, 60, 52, 27] against packet_in flooding attacks since they detect attacks by identifying and throttling malicious control traffic.

LDoS Attacks in Traditional IP Networks. Kuzmanovic et al. [41] developed low-rate TCP-targeted DoS attacks to disrupt TCP connections. Zhang et al. [66] demonstrated the attack has severe impact on the Border Gateway Protocol (BGP) by conducting real experiments. Schuchard et al. [50] extended the attack developed by Zhang et al. and designed the Coordinated Cross Plane Session Termination attack (CXPST) that allows an attacker to attack the Internet control plane by using only data traffic. Our attack differs from the previous work in three aspects. First, our attack focuses on disrupting the SDN control channel that shares a limited number of links with data paths. Second, probing techniques are required in the attack to identify target data paths containing shared links, which is necessary to ensure the effectiveness of the attack. Third, our attack in SDN has more significant impacts on diversified network functionalities including layer 2, 3 and 4 functions.

To defend against LDoS, some countermeasures have been provided in traditional IP networks, such as randomizing RTO [42] and complex signal analysis [58, 53, 23, 64, 46, 24] . However, randomizing RTO cannot fully mitigate the attack [66], and none of the methods are shown to be sufficiently accurate and scalable for deployment in real networks. Besides, they are general defenses against LDoS in traditional IP networks and are not designed to protect the SDN control channel. Defenses against LDoS attacks on BGP was described in [50], such as BGP Graceful Restart. However, it is not suitable to protect the SDN control channel with "dumb" SDN switches.

Link Flooding Attacks in Traditional IP Networks. Studer et al. [38] and Kang et al. [57] introduced link flooding attacks, which generate large-scale legitimate low-speed flows to flood and congest network critical links. They use traceroute to find critical links in traditional IP networks. Our crosspath attack also congests the critical links that deliver control traffic and data traffic in SDN at the same time. However, one major difference is that our crosspath attack

identifies the critical links with the unique SDN reconnaissance technique. Moreover, the crosspath attack can incur various damages in the whole network by disrupting the control channel due to the centralized control in SDN. Though there exist some SDN defense systems [67, 61, 62, 37] that detect link flooding attacks, they cannot defend the crosspath attack that disrupts the control channel, which these SDN defense systems depend on.

8 Conclusions

In this paper, we present a novel attack in SDN. It disrupts the control channel by crafting data traffic to implicitly interfere with control traffic in the shared links. We develop the adversarial path reconnaissance to find a target data path containing shared links for the attack. Both theoretic analysis and experimental results show that our reconnaissance works in real networks. We demonstrate that the attack can significantly disrupt various network functionalities in SDN. We hope this work attract more attention to SDN security, especially the possible attacks on the SDN control channel when deploying SDN to innovate network applications.

Acknowledgments

The research is partly supported by the National Key R&D Program of China under Grant 2017YFB0803202, the National Natural Science Foundation of China (NSFC) under Grant 61625203, 61572278, 61832013, 61872209, and U1736209, the U.S. ONR grants N00014-16-1-3214 and N00014-16-1-3216, and the National Science Foundation (NSF) under Grant 1617985, 1642129, 1700544, and 1740791. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSFC, NSF, and other sponsors. Qi Li and Mingwei Xu are the corresponding authors of the paper.

References

[1] RFC 2328. OSPF Version 2. https://tools.ietf.org/html/rfc2328/, 1998. [Online].

[2] RFC 4271. Border Gateway Protocol 4 (BGP-4). https://tools.ietf.org/html/rfc4271/, 2006. [Online].

[3] Data Set for IMC 2010 Data Center Measurement. http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html, 2010. [Online].

[4] The Internet Topology Zoo. http://www.topology-zoo.org/dataset.html, 2011. [Online].

[5] Floodlight ARP Proxy. https://github.com/mbredel/floodlight-proxyarp/, 2013. [Online].

[6] Floodlight Learning Switch. https://github.com/floodlight/floodlight/blob/master/src/main/java/net/floodlightcontroller/learningswitch/, 2014. [Online].

[7] Floodlight Load Balancer. https://github.com/floodlight/floodlight/tree/master/src/main/java/net/floodlightcontroller/loadbalancer, 2014. [Online].

[8] CAIDA Passive Monitor: Chicago B. http://www.caida.org/data/passive/trace_stats/chicago-B/2015/?monitor=20150219-130000.UTC, 2015. [Online].

[9] Floodlight Reactive Routing. https://github.com/floodlight/floodlight/tree/master/src/main/java/net/floodlightcontroller/routing/, 2016. [Online].

[10] AS4610-54T Data Center Switch. https://www.edge-core.com/productsInfo.php?cls=1&cls2=9&cls3=46&id=21, 2018. [Online].

[11] AT&T SD-WAN. https://www.business.att.com/solutions/Family/network-services/sd-wan/, 2018. [Online].

[12] Floodlight Controller. http://www.projectfloodlight.org/, 2018. [Online].

[13] Microsoft Azure and Software Defined Networking. https://docs.microsoft.com/en-us/windows-server/networking/sdn/azure_and_sdn/, 2018. [Online].

[14] Open Networking Foundation (ONF). https://www.opennetworking.org/, 2018. [Online].

[15] Raw Sockets. https://en.wikipedia.org/wiki/Network_socket#Raw_socket, 2018. [Online].

[16] TCPReplay. http://tcpreplay.synfin.net, 2018. [Online].

[17] Traceroute. https://en.wikipedia.org/wiki/Traceroute/, 2018. [Online].

[18] Traffic and Tools. http://traffic.comics.unina.it/Traces/ttraces.php, 2018. [Online].

[19] ACHLEITNER, S., LA PORTA, T., JAEGER, T., AND MCDANIEL, P. Adversarial network forensics in software defined networking. In *Proceedings of the Symposium on SDN Research* (2017), ACM, pp. 8–20.

[20] BOX, J. F., ET AL. Guinness, gosset, fisher, and small samples. *Statistical science 2*, 1 (1987), 45–52.

[21] BRAUN, W., AND MENTH, M. Software-defined networking using openflow: Protocols, applications and architectural design choices. *Future Internet 6*, 2 (2014), 302–336.

[22] CAO, J., XU, M., LI, Q., SUN, K., YANG, Y., AND ZHENG, J. Disrupting sdn via the data plane: a low-rate flow table overflow attack. In *Proceedings of International Conference on Security and Privacy in Communication Systems* (2017), Springer, pp. 356–376.

[23] CHEN, Y., HWANG, K., AND KWOK, Y.-K. Collaborative defense against periodic shrew ddos attacks in frequency domain. *ACM Transactions on Information and System Security 30* (2005).

[24] CHEN, Z., YEO, C. K., LEE, B. S., AND LAU, C. T. Power spectrum entropy based detection and mitigation of low-rate dos attacks. *Computer Networks 136* (2018), 80–94.

[25] CUI, H., KARAME, G. O., KLAEDTKE, F., AND BIFULCO, R. On the fingerprinting of software-defined networks. *IEEE Transactions on Information Forensics and Security 11*, 10 (2016), 2160–2173.

[26] DENG, J., LI, H., HU, H., WANG, K.-C., AHN, G.-J., ZHAO, Z., AND HAN, W. On the safety and efficiency of virtual firewall elasticity control. In *Proceedings of Network and Distributed System Security Symposium* (2017).

[27] DHAWAN, M., PODDAR, R., MAHAJAN, K., AND MANN, V. Sphinx: Detecting security attacks in software-defined networks. In *Proceedings of Network and Distributed System Security Symposium* (2015).

[28] DIJKSTRA, E. W. A note on two problems in connexion with graphs. *Numerische mathematik 1*, 1 (1959), 269–271.

[29] DIXIT, A., HAO, F., MUKHERJEE, S., LAKSHMAN, T., AND KOMPELLA, R. Towards an elastic distributed sdn controller. In *ACM SIGCOMM computer communication review* (2013), vol. 43, ACM, pp. 7–12.

[30] HE, K., ROZNER, E., AGARWAL, K., FELTER, W., CARTER, J., AND AKELLA, A. Presto: Edge-based load balancing for fast datacenter networks. In *ACM SIGCOMM Computer Communication Review* (2015), vol. 45, ACM, pp. 465–478.

[31] HONG, S., XU, L., WANG, H., AND GU, G. Poisoning network visibility in software-defined networks: New attacks and countermeasures. In *Proceedings of Network and Distributed System Security Symposium* (2015), vol. 15, pp. 8–11.

[32] JAIN, S., KUMAR, A., MANDAL, S., ONG, J., POUTIEVSKI, L., SINGH, A., VENKATA, S., WANDERER, J., ZHOU, J., ZHU, M., ET AL. B4: Experience with a globally-deployed software defined wan. *ACM SIGCOMM Computer Communication Review 43*, 4 (2013), 3–14.

[33] JANG, R., CHO, D., NOH, Y., AND NYANG, D. Rflow+: An sdn-based wlan monitoring and management framework. In *Proceedings of IEEE Conference on Computer Communications* (2017), IEEE, pp. 1–9.

[34] JERO, S., BU, X., NITA-ROTARU, C., OKHRAVI, H., SKOWYRA, R., AND FAHMY, S. Beads: automated attack discovery in openflow-based sdn systems. In *Proceedings of International Symposium on Research in Attacks, Intrusions, and Defenses* (2017), Springer, pp. 311–333.

[35] JERO, S., KOCH, W., SKOWYRA, R., OKHRAVI, H., NITA-ROTARU, C., AND BIGELOW, D. Identifier binding attacks and defenses in software-defined networks. In *Proceedings of USENIX Security Symposium* (2017), USENIX Association, pp. 415–432.

[36] JIN, X., LIU, H. H., GANDHI, R., KANDULA, S., MAHAJAN, R., ZHANG, M., REXFORD, J., AND WATTENHOFER, R. Dynamic scheduling of network updates. *ACM SIGCOMM Computer Communication Review 44*, 4 (2014), 539–550.

[37] KANG, M. S., GLIGOR, V. D., SEKAR, V., ET AL. Spiffy: Inducing cost-detectability tradeoffs for persistent link-flooding attacks. In *NDSS* (2016).

[38] KANG, M. S., LEE, S. B., AND GLIGOR, V. D. The crossfire attack. In *Proceedings of Symposium on Security and Privacy* (2013), IEEE, pp. 127–141.

[39] KLÖTI, R., KOTRONIS, V., AND SMITH, P. Openflow: A security analysis. In *Proceedings of International Conference on Network Protocols* (2013), IEEE, pp. 1–6.

[40] KREUTZ, D., RAMOS, F. M., VERISSIMO, P. E., ROTHENBERG, C. E., AZODOLMOLKY, S., AND UHLIG, S. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE 103*, 1 (2015), 14–76.

[41] KUZMANOVIC, A., AND KNIGHTLY, E. W. Low-rate tcp-targeted denial of service attacks: the shrew vs. the mice and elephants. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communications* (2003), ACM, pp. 75–86.

[42] KUZMANOVIC, A., AND KNIGHTLY, E. W. Low-rate tcp-targeted denial of service attacks and counter strategies. *IEEE/ACM Transactions on Networking 14*, 4 (2006), 683–696.

[43] LEE, S., YOON, C., LEE, C., SHIN, S., YEGNESWARAN, V., AND PORRAS, P. Delta: A security assessment framework for software-defined networks. In *Proceedings of Network and Distributed System Security Symposium* (2017), vol. 17.

[44] LENG, J., ZHOU, Y., ZHANG, J., AND HU, C. An inference attack model for flow table capacity and usage: Exploiting the vulnerability of flow table overflow in software-defined network. *arXiv preprint arXiv:1504.03095* (2015).

[45] LIU, S., REITER, M. K., AND SEKAR, V. Flow reconnaissance via timing attacks on sdn switches. In *Proceedings of International Conference on Distributed Computing Systems* (2017), IEEE, pp. 196–206.

[46] LUO, J., YANG, X., WANG, J., XU, J., SUN, J., AND LONG, K. On a mathematical model for low-rate shrew ddos. *IEEE Transactions on Information Forensics and Security 9*, 7 (2014), 1069–1083.

[47] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review 38*, 2 (2008), 69–74.

[48] PORRAS, P. A., CHEUNG, S., FONG, M. W., SKINNER, K., AND YEGNESWARAN, V. Securing the software defined network control layer. In *Proceedings of Network and Distributed System Security Symposium* (2015).

[49] RÖPKE, C., AND HOLZ, T. Sdn rootkits: Subverting network operating systems of software-defined networks. In *Proceedings of International Workshop on Recent Advances in Intrusion Detection* (2015), Springer, pp. 339–356.

[50] SCHUCHARD, M., MOHAISEN, A., FOO KUNE, D., HOPPER, N., KIM, Y., AND VASSERMAN, E. Y. Losing control of the internet: using the data plane to attack the control plane. In *Proceedings of the conference on Computer and communications security* (2010), ACM, pp. 726–728.

[51] SHAGHAGHI, A., KAAFAR, M. A., AND JHA, S. Wedgetail: An intrusion prevention system for the data plane of software defined networks. In *Proceedings of the Asia Conference on Computer and Communications Security* (2017), ACM, pp. 849–861.

[52] SHANG, G., ZHE, P., BIN, X., AIQUN, H., AND KUI, R. Flooddefender: protecting data and control plane resources under sdn-aimed dos attacks. In *Proceedings of IEEE Conference on Computer Communications* (2017), IEEE, pp. 1–9.

[53] SHEVTEKAR, A., ANANTHARAM, K., AND ANSARI, N. Low rate tcp denial-of-service attack detection at edge routers. *IEEE Communications Letters 9*, 4 (2005), 363–365.

[54] SHIN, S., AND GU, G. Attacking software-defined networks: A first feasibility study. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking* (2013), ACM, pp. 165–166.

[55] SHIN, S., YEGNESWARAN, V., PORRAS, P., AND GU, G. Avant-guard: Scalable and vigilant switch flow management in software-defined networks. In *Proceedings of the ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 413–424.

[56] SONCHACK, J., DUBEY, A., AVIV, A. J., SMITH, J. M., AND KELLER, E. Timing-based reconnaissance and defense in software-defined networks. In *Proceedings of Conference on Computer Security Applications* (2016), ACM, pp. 89–100.

[57] STUDER, A., AND PERRIG, A. The coremelt attack. In *European Symposium on Research in Computer Security* (2009), Springer, pp. 37–52.

[58] SUN, H., LUI, J. C., AND YAU, D. K. Defending against low-rate tcp attacks: Dynamic detection and protection. In *Proceedings of International Conference on Network Protocols* (2004), IEEE, pp. 196–205.

[59] UJCICH, B. E., THAKORE, U., AND SANDERS, W. H. Attain: An attack injection framework for software-defined networking. In *Proceedings of International Conference on Dependable Systems and Networks* (2017), IEEE, pp. 567–578.

[60] WANG, H., XU, L., AND GU, G. Floodguard: A dos attack prevention extension in software-defined networks. In *Proceedings of International Conference on Dependable Systems and Networks* (2015), IEEE, pp. 239–250.

[61] WANG, J., WEN, R., LI, J., YAN, F., ZHAO, B., AND YU, F. Detecting and mitigating target link-flooding attacks using sdn. *IEEE Transactions on Dependable and Secure Computing*, 1 (2018), 1–1.

[62] WANG, L., LI, Q., JIANG, Y., JIA, X., AND WU, J. Woodpecker: Detecting and mitigating link-flooding attacks via sdn. *Computer Networks 147* (2018), 1–13.

[63] WEN, X., YANG, B., CHEN, Y., HU, C., WANG, Y., LIU, B., AND CHEN, X. Sdnshield: Reconciliating configurable application permissions for sdn app markets. In *Proceedings of International Conference on Dependable Systems and Networks* (2016), IEEE, pp. 121–132.

[64] XIANG, Y., LI, K., AND ZHOU, W. Low-rate ddos attacks detection and traceback by using new information metrics. *IEEE Transactions on Information Forensics and Security 6*, 2 (2011), 426–437.

[65] XU, L., HUANG, J., HONG, S., ZHANG, J., AND GU, G. Attacking the brain: Races in the sdn control plane. In *USENIX Security Symposium* (2017), USENIX Association, pp. 451–468.

[66] ZHANG, Y., MAO, Z. M., AND WANG, J. Low-rate tcp-targeted dos attack disrupts internet routing. In *Proceedings of Network and Distributed System Security Symposium* (2007), Citeseer.

[67] ZHENG, J., LI, Q., GU, G., CAO, J., YAU, D. K., AND WU, J. Realtime ddos defense using cots sdn switches via adaptive correlation analysis. *IEEE Transactions on Information Forensics and Security 13*, 7 (2018), 1838–1853.

A The Algorithm of Adversarial Path Reconnaissance

Algorithm 1 shows the pseudo-code of improved adversarial path reconnaissance, which can be performed by any host in the network. The input η is the number of repeated reconnaissances for each data path and is also the number of data in each group used in the t-test. The input t_{wait} is the waiting time for rules to expire. The input t_{max} is the maximal waiting time for each timing packet to get a response in the target network, and α is the significance level used in the t-test. Here, t_{wait} must be larger than the timeouts of flow rules and t_{max} must be large enough so that most RTTs in the network do not exceed it. Step 1 gets all hosts in the network in order to explore the data paths between the compromised host and them. Step 2 initializes the maximal number of data paths that can be explored within two timeout values. The main loop is from Step 4 to Step 29. In each loop iteration, the algorithm tests k_{max} data paths. Step 5 to Step 20 collects 2η latencies of the crossed control paths for each of the k_{max} data paths. The delay of crossed control paths when the testing stream is not transmitted is obtained in Step 7 to Step 10. Step 13 to Step 18 obtain the delay while transmitting the testing stream. Step 11 and Step 19 both make the program paused for enough time so that old rules can expire before conducting the next reconnaissance. After obtaining all the latencies of possible crossed control paths for the k_{max} data paths, the t-test is applied to determine whether a data path crosses with control paths in Step 21 to Step 27. If the group of latencies with testing stream is dramatically higher than the other group, the algorithm outputs the destination host of the data path and terminates. Otherwise, the algorithm prepares for the next round of iteration in Step 28.

Algorithm 1 Adversarial Path Reconnaissance

Input: η, t_{wait}, t_{max}, α
Output: h;

1: $H \leftarrow ScanAllHosts()$
2: $k_{max} \leftarrow t_{wait}/(2 \cdot t_{max})$
3: $i \leftarrow 0$
4: **while** $i < |H|$ **do**
5: **for** $j = 0 \rightarrow \eta - 1$ **do**
6: $t_{start} \leftarrow time()$
7: **for** $k = i \rightarrow min(i + k_{max}, |H|)$ **do**
8: $d_1 \leftarrow$ sendTimingStreamTo($H[k]$)
9: $\delta_1[k].add(d_1)$
10: **end for**
11: $sleep(t_{wait} - (time() - t_{start}))$
12: $t_{start} \leftarrow time()$
13: **for** $k = i \rightarrow min(i + k_{max}, |H|)$ **do**
14: startSendTestingStreamTo($H[k]$)
15: $d_2 \leftarrow$ sendTimingStreamTo($H[k]$)
16: stopSendTestingStreamTo($H[k]$)
17: $\delta_2[k].add(d_2)$
18: **end for**
19: $sleep(t_{wait} - (time() - t_{start}))$
20: **end for**
21: **for** $k = i \rightarrow min(i + k_{max}, |H|)$ **do**
22: **if** $tTest(\delta_1[k], \delta_2[k]) < \alpha$ *and* $sum(\delta_1[k]) < sum(\delta_2[k])$ **then**
23: /* The data path from the compromised host to $H[k]$ crosses with control paths. */
24: $output(H[k])$
25: $exit()$
26: **end if**
27: **end for**
28: $i \leftarrow i + k_{max}$
29: **end while**

In our experiments on a real SDN testbed, t_{wait} is set as 30s which is larger than the default values of timeouts in the Floodlight controller in order to leave enough time for rules to be expired, and t_{max} is set to 1s for each timing packet to get a response. The settings of η and α are varied.

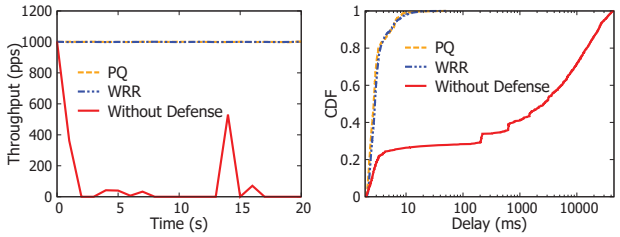
B Defense Against the CrossPath Attack

We explore two defense schemes to mitigate the CrossPath attack. The first is delivering control traffic with high priority. Hence, any malicious data traffic cannot disturb the delivery of control traffic. Such a defense scheme can be enforced with Priority Queue (PQ) or Weighted Round Robin

Table 1: The Settings for Flow Rules to Enforce Defense Strategies.

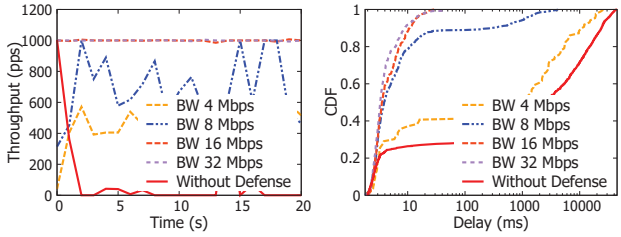
Defense Strategy	Rule	Match	Actions
Control traffic delivery with high priority [1]	#1	control flows	OutPort(x), ..., *SetQueue(ID=highPriQueue)*
	#2	data flows	OutPort(x), ..., *SetQueue(ID=lowPriQueue)*
Proactive bandwidth reservation for control traffic [2]	#1	data flows	OutPort(x), ..., *SetMeter(ID=RateLimit)*

[1] It requires SDN switches to support PQ or WRR queuing mechanism.
[2] It is used when SDN switches fail to enable PQ or WRR mechanism.



(a) Throughput of Control Packets with Queuing Policies. (b) CDF of Delays of Control Packets with Queuing Policies.

Figure 18: Evaluation on the defense scheme of delivering control traffic with high priority via PQ or WRR mechanism.



(a) Throughput of Control Packets with Bandwidth Reservation. (b) CDF of Delays of Control Packets with Bandwidth Reservation.

Figure 19: Evaluation on protecting control traffic with the defense scheme of proactive bandwidth (BW) reservation.

(WRR) [3] scheduling mechanism in SDN switches. Specifically, network administrators can inform controllers to add *SetQueue* actions to flow rules associated with switch ports in the control paths. Packets matching a flow rule with the *SetQueue* action will be directed to a queue with an *ID* set by the action. As shown in Table 1, we can set a flow rule matching control flows with a high priority queue and set a flow rule matching data flows with a low priority queue. In this way, the control traffic will always be forwarded in advance with no disturbances of other traffic.

We note that some switches in the market do not support PQ or WRR mechanisms. However, we can still mitigate the CrossPath attack by proactive bandwidth reservation for control traffic with OpenFlow meter table. A meter entry belonging to a meter table associates with various flow rules so that it can measure the total rate of packets matching the flow rules and enforce rate limiting. We can assign each flow rule matched by the data traffic a meter entry with the *SetMeter* action (see Table 1). Therefore, by limiting the maximal rate of the total data traffic, we reserve proprietary bandwidth for control traffic.

We evaluate above two defense schemes with AS4610-54T commercial hardware SDN switches in our testbed. For simplicity, we trigger 1,000 new flows per second to generate the control traffic and generate the attack traffic to disrupt the transmission of control packets. Figure 18a and 18b shows that defense schemes with PQ or WRR mechanism effectively protect the control traffic. The throughput always reaches approximate 1,000 pps over time even with the attack. The delays of more than 99% of the control packets are less than 10 ms with either of the two queuing mechanisms. Figure 19a and 19b show that proactive bandwidth reservation with meter table can also protect control traffic. The larger the reserved bandwidth is, the higher the throughput is, and also the better the delay is. In our experiments, 16 Mbps reserved bandwidth is enough to ensure forwarding control traffic. Note that compared with the queuing mechanism, it requires proactively reserving enough bandwidth for control traffic. In a large network, it may require reserving bandwidth in the order of several Gbps.

Table 2: SDN Switches with PQ or WRR Support.

Brand	Model	Queue Support	
		PQ	WRR
Pica8	All switches loaded with PicOS	√	√
Cisco	Catalyst 4500 Series Switches	√	×
Brocade	NetIron XMR Series, MLX Series, CES 2000, and CER 2000 Series	×	√
Dell	S4810, S4820T, S6000, Z9000, Z9500, and MXL switches	×	√
Huawei	CloudEngine 8800 Series	√	√

C SDN Switches with Queue Support

We investigate mainstream SDN switches and find that many switches support PQ or WRR mechanism. Table 2 shows the switches with PQ or WRR support.

[3] By configuring different weighted values to queues with WRR, similar results like PQ can be achieved.

A Billion Open Interfaces for Eve and Mallory: MitM, DoS, and Tracking Attacks on iOS and macOS Through Apple Wireless Direct Link

Milan Stute
TU Darmstadt

Sashank Narain
Northeastern University

Alex Mariotto
TU Darmstadt

Alexander Heinrich
TU Darmstadt

David Kreitschmann
TU Darmstadt

Guevara Noubir
Northeastern University

Matthias Hollick
TU Darmstadt

Abstract

Apple Wireless Direct Link (AWDL) is a key protocol in Apple's ecosystem used by over one billion iOS and macOS devices for device-to-device communications. AWDL is a proprietary extension of the IEEE 802.11 (Wi-Fi) standard and integrates with Bluetooth Low Energy (BLE) for providing services such as Apple AirDrop. We conduct the first security and privacy analysis of AWDL and its integration with BLE. We uncover several security and privacy vulnerabilities ranging from design flaws to implementation bugs leading to a man-in-the-middle (MitM) attack enabling stealthy modification of files transmitted via AirDrop, denial-of-service (DoS) attacks preventing communication, privacy leaks that enable user identification and long-term tracking undermining MAC address randomization, and DoS attacks enabling targeted or simultaneous crashing of all neighboring devices. The flaws span across AirDrop's BLE discovery mechanism, AWDL synchronization, UI design, and Wi-Fi driver implementation. Our analysis is based on a combination of reverse engineering of protocols and code supported by analyzing patents. We provide proof-of-concept implementations and demonstrate that the attacks can be mounted using a low-cost ($20) micro:bit device and an off-the-shelf Wi-Fi card. We propose practical and effective countermeasures. While Apple was able to issue a fix for a DoS attack vulnerability after our responsible disclosure, the other security and privacy vulnerabilities require the redesign of some of their services.

1 Introduction

With deployments on over one billion devices, spanning several Apple operating systems (iOS, macOS, tvOS, and watchOS) and an increasing variety of devices (Mac, iPhone, iPad, Apple Watch, Apple TV, and HomePod), Apple Wireless Direct Link (AWDL) is ubiquitous and plays a key role in enabling device-to-device communications in the Apple ecosystem. The AWDL protocol is little understood, partially due to its proprietary nature, especially when it comes to security and privacy. Considering the well-known rocky history of wireless protocols' security, with various flaws being repeatedly discovered in Bluetooth [7], WEP [74], WPA2 [88], GSM [12], UMTS [57], and LTE [51], the lack of information regarding AWDL security is a significant concern given the increasing number of services that rely on it, particularly Apple's AirDrop and AirPlay. It is also noteworthy that the design of AWDL and integration with Bluetooth Low Energy (BLE) are (1) driven by optimizing energy and bandwidth and (2) the devices do not require an existing Wi-Fi access point (AP) with secure connections but are open to communicating with arbitrary devices, thus, potentially exposing various attack vectors.

We conduct the first, to the best of our knowledge, security analysis of AWDL and its integration with BLE, starting with the reverse engineering of protocols and code supported by analyzing patents. Our analysis reveals several security and privacy vulnerabilities ranging from design flaws to implementation bugs enabling different kinds of attacks: we present a man-in-the-middle (MitM) attack enabling stealthy modification of files transmitted via AirDrop, a denial-of-service (DoS) attack preventing communication between devices, privacy leaks allowing user identification and long-term tracking undermining MAC address randomization, and targeted DoS and blackout DoS attacks (i. e., enabling simultaneous crashing of all neighboring devices). The flaws span AirDrop's BLE discovery mechanism, AWDL synchronization, UI design, and Wi-Fi driver implementation. We demonstrate that the attacks can be stealthy, low-cost, and launched by devices not connected to the target Wi-Fi network. We provide proof-of-concept (PoC) implementations and demonstrate that the attacks can be mounted using a low-cost ($20) micro:bit device and an off-the-shelf Wi-Fi card. The impact of these findings goes beyond Apple's ecosystem as the Wi-Fi Alliance adopted AWDL as the basis for Neighbor Awareness Networking (NAN) [19, 94] which, therefore, might be susceptible to similar attacks. Moreover, Google Android provides a NAN API since 2017 pending manufacturer support [38].

Specifically, our contributions are threefold. *First*, we discover security and privacy vulnerabilities in AWDL and AirDrop and present four novel network-based attacks on iOS

and macOS. These attacks are:

(1) A long-term device tracking attack which works *in spite* of MAC randomization, and may reveal personal information such as the name of the device owner (over 75% of experiment cases).
(2) A DoS attack aiming at the election mechanism of AWDL to deliberately desynchronize the targets' channel sequences effectively preventing communication.
(3) A MitM attack which intercepts and modifies files transmitted via AirDrop, effectively allowing for planting malicious files.
(4) Two DoS attack on Apple's AWDL implementations in the Wi-Fi driver. The attacks allow crashing Apple devices in proximity by injecting specially crafted frames. The attacks can be targeted to a single victim or affect all neighboring devices at the same time.

Second, we propose practical mitigations for all four attacks. *Third*, we publish open source implementations of both AWDL and AirDrop as the byproducts of our reverse engineering efforts to stimulate future research in this area.

The rest of this paper is structured as follows. Section 2 provides background information on AWDL. Section 3 shows the results of reverse engineering AirDrop. Section 4 presents an attack to activate AWDL on devices in proximity, while Section 5 shows how we leverage this activation mechanism for user tracking attacks. Sections 6 and 7 feature the desynchronization DoS attack and the MitM attack, respectively. Section 8 reports implementation security vulnerabilities and Section 9 concludes this work. We discuss mitigation techniques and related work in subsections of the respective sections describing the attacks.

2 Background on Apple Wireless Direct Link

AWDL is a proprietary wireless ad hoc protocol based on the IEEE 802.11 standard. In this section, we rely on the reverse engineering efforts of the Open Wireless Link project [81] and summarize the operation of AWDL as presented in [79, 80]. At its core, AWDL uses a channel hopping mechanism to enable "simultaneous" communication with an AP and other AWDL nodes on different channels. This channel hopping is implemented as a sequence of so-called Availability Windows (AWs). For each AW, a node indicates if it is available for direct communication and, if so, on which channel it will be. The channel value can be the channel of its AP, one of the dedicated AWDL social channels (6, 44, or 149), or zero indicating that it will not be listening on any channel. Each node announces its own sequence s consisting of 16 AWs[1] regularly in AWDL-specific IEEE 802.11 Action Frames (AFs). We call the length of such a full 16-AW sequence a *period* τ. Each AW

[1] Actually, [79] differentiates between AWs, Extension Windows (EWs), and Extended Availability Windows (EAWs) where one EAW consists of one AW and three EWs. In this work, we abstract from the smaller entities and simply use the term AW to refer to an EAW.

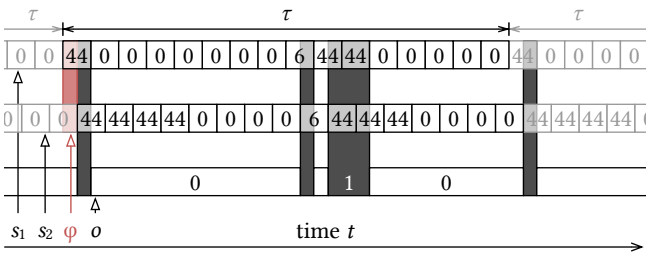


Figure 1: AWDL synchronization depicting period, phase offset, and the overlap function of two channel sequences.

has a length of 64 Time Units (TUs) where 1 TU = 1024 µs, so $\tau \approx 1$ s. Figure 1 depicts these and the following concepts.

To allow nodes to meet and exchange data on the same channel, they need to align their sequences in the time domain. AWDL nodes elect a common master and use its AFs as a time reference to achieve *synchronization*. Each master node transmits *synchronization parameters* which consist of the current AW sequence number i (0 to $2^{16}-1$) and the time until the next AW starts based on its local clock t_{AW}. When receiving an AF from its master at time T_{Rx}, a node approximates T_{AW}, the start of the next AW $i+1$ in local time as:[2]

$$T_{\text{AW}} \approx t_{\text{AW}} + T_{\text{Rx}} \tag{1}$$

and correct its clock accordingly. Eq. (1) does not achieve perfect alignment as it ignores the transmission delay as well as delays in the sender's Tx and receiver's Rx chains. The phase ϕ denotes the effective clock offset between two nodes. Typically, $\phi \leq 3$ ms in practice [79].

A node transmits frames to another AWDL node during AWs where the channels of both nodes are the same. We denote the *overlap* as the relative communication opportunities during one period: an overlap of one means that two nodes are on the same AWDL channel during all 16 AWs, while zero means that they are never on the same channel. Formally, we define the *overlap* O as the integral over the overlap function o of two sequences s_1 and s_2 taking into account the phase where $s(t)$ is the τ-periodic continuation of a sequence, i. e., $s(t+n\tau) = s(t), \forall n \in \mathbb{N}$. Then,

$$o(s_1, s_2, \phi, t) = \begin{cases} 1 & \text{if } s_1(t) = s_2(t-\phi) \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

and

$$O(s_1, s_2, \phi) = \int_{t=0}^{\tau} o(s_1, s_2, \phi, t) \quad . \tag{3}$$

3 Reverse Engineering AirDrop

AirDrop is an application that allows iOS and macOS users to exchange files between devices using AWDL as transport.

[2] Simplification of [79, Eq. (1)].

We reverse-engineered the AirDrop protocol by employing a MitM HTTPS proxy [20] and using a popular disassembler on macOS' `sharingd` daemon and `Sharing` framework where AirDrop is implemented. Based on our findings, we re-implement AirDrop in Python and make it available as open source software [78]. Next, we discuss how different discoverability settings affect the user experience. Then, we describe the technical protocol flow and, finally, explain the difference between authenticated and unauthenticated connections.

3.1 Discoverability User Setting

When opening the sharing pane (see left screenshot in Fig. 2) in AirDrop, nearby devices will appear in the user interface depending on their discoverability setting [2]. In particular, devices can be discovered (1) by *everybody* or (2) by *contacts only*. Alternatively, (3) the *receiving off* setting disables any AirDrop connection requests. AirDrop requires Wi-Fi and Bluetooth to be enabled. By default, Wi-Fi and Bluetooth are enabled, and AirDrop is set to *contacts only*. In addition, we found that devices need to be unlocked to be discovered. Based on a user study that we present in Section 5.2, we found that 80 % of the participants enable AirDrop (59.4 % in *contacts only* and 20.6 % in *everybody* mode) while the other 20 % disabled it. For the rest of the paper, we assume that a target device has AirDrop enabled and is unlocked.

3.2 Protocol and User Interaction

We describe all mechanisms involved in AirDrop including discovery, authentication, and data transfer with a visual aid in Fig. 2. The *sender* initiates the discovery procedure and transfers the data while the *receiver* responds to requests: (1a) The sender emits BLE advertisements including its hashed contact identifiers (see Section 4.1 for details), while the prospective AirDrop receiver regularly scans for BLE advertisements. (1b) The receiver compares the sender's contact hashes with contact identifiers in its own address book if set to *contacts-only* mode. If there is at least one match or if the receiver is in *everyone* mode, the receiver activates its AWDL interface. (1c) Using mDNS/DNS-SD, the sender starts to look for AirDrop service instances via the AWDL interface. (2) For each discovered service, the sender establishes an HTTPS connection with the receiver and performs a full authentication handshake (*Discover*). If authentication is successful, the receiver appears as an icon in the sender's UI. (3) When the user selects a receiver, AirDrop sends a request containing metadata and a thumbnail of the file (*Ask*). The receiver decides whether they want to accept. If so, the sender continues to transfer the actual file (*Upload*).

Next, we discuss the client and server TLS certificates and explain their usage in combination with the sender's and receiver's record data to establish an authenticated connection.

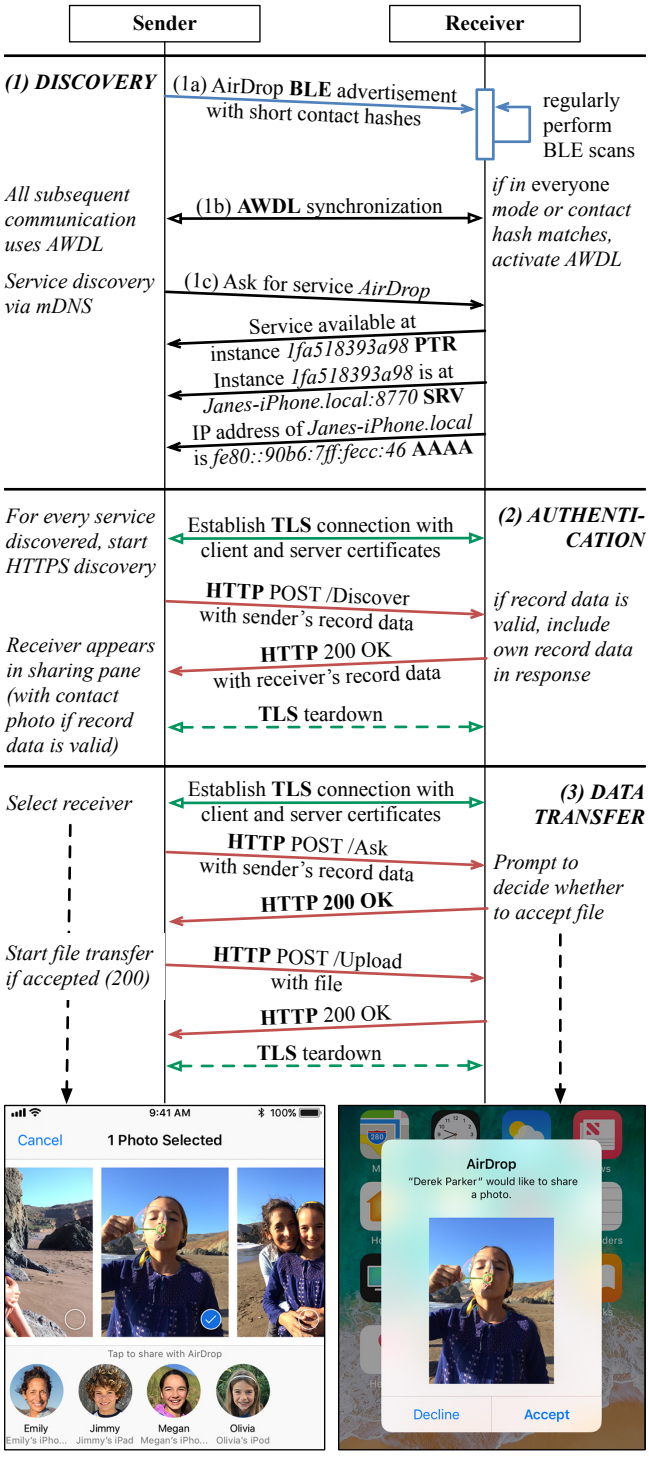


Figure 2: Typical AirDrop protocol workflow including screenshots [2] where user interaction is required.

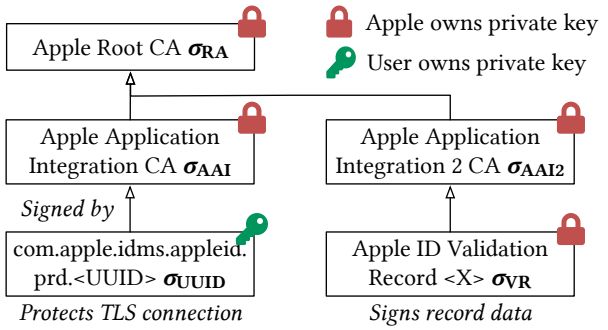


Figure 3: Certificates and CAs involved in AirDrop. Boxes contain the certificates' *common names*.

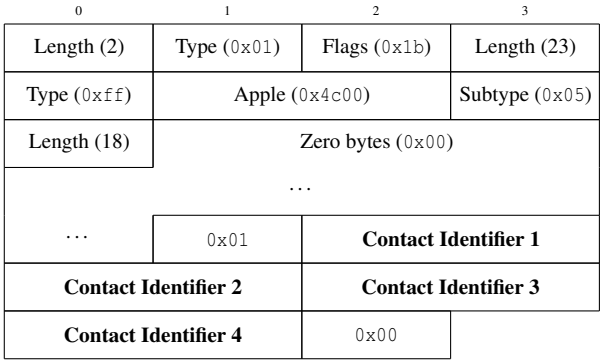


Figure 4: AirDrop BLE advertisement format showing semantics and values of individual bytes.

3.3 (Un)authenticated Connections

AirDrop will always try to set up what we call an *authenticated* connection. Such a connection can only be established between users with an Apple ID and that have each other in their address books. Authentication involves multiple certificates and CAs that we depict in Fig. 3. In order to authenticate, a device needs to prove that it "owns" a certain contact identifier c_i such as email address or phone number associated with its Apple ID, while the verifying device checks whether it has c_i in its address book. When establishing a TLS connection, AirDrop uses a device-specific Apple-signed certificate σ_{UUID} containing a *UUID*. σ_{UUID} is issued when a user logs into the device with its Apple ID. The UUID is not tied to any contact identifiers, so AirDrop uses an Apple-signed *record data* "blob" *RD* containing the UUID and all contact identifiers $c_1, \ldots, c_n$ that are registered with the user's Apple ID in a hashed form. This record data is retrieved once from Apple and then presented for any subsequent AirDrop connection. Formally, *RD* is a tuple:

$$RD = UUID, SHA2(c_1), \ldots, SHA2(c_n) \quad . \tag{4}$$

The signed record data RD_σ additionally includes a signature and a certificate chain (Fig. 3):

$$RD_\sigma = RD, sign(\sigma_{VR}, RD), \sigma_{VR}, \sigma_{AAI2} \quad , \tag{5}$$

where $sign(\sigma, X)$ is the signature of σ over X. When authenticating, a node computes SHA2 over each contact identifier in its address book and compares them with the hashes contained in the presented RD_σ and verifies that the UUID matches the certificate of the current TLS connection. The latter effectively prevents reuse of RD_σ by an attacker using a different TLS certificate.

AirDrop transparently treats a connection as *unauthenticated* if the sender or receiver fails to provide an Apple-signed TLS certificate or valid record data. This means that AirDrop will establish an unauthenticated connection with devices that use a self-signed certificate and provide no record data. While AirDrop's authentication mechanism appears to be cryptographically well-designed, we show in Section 7 how to downgrade an authenticated connection to an unauthenticated one and launch a MitM attack on the data transfer.

4 Activating AWDL on Devices in Proximity

Some of the attacks demonstrated in this work require the targets' AWDL interface to be active, which is typically not the case since an application has to request activation explicitly [79]. We have found that the BLE discovery mechanism integrated with AirDrop (see Section 3) can be exploited to activate all AWDL devices in proximity. Devices in *everyone* mode will enable AWDL immediately after receiving any AirDrop BLE advertisement. We analyze the theoretical performance of brute forcing the truncated contact hash values in AirDrop's BLE advertisements (Fig. 2) to activate the AWDL interfaces of targets in the default *contacts-only* mode. Finally, we build a PoC leveraging a low-cost ($20) BBC micro:bit device and experimentally confirm that the attack is feasible in practice with a target response time of about one second for devices that have 100 contact identifiers in their address book.

4.1 AirDrop BLE Advertisements

We show the actual BLE advertisement frames [17, Vol. 3, Sec. 11] that AirDrop uses including four contact identifier hashes in Fig. 4. They are broadcast as non-connectable undirected advertising (`ADV_NONCONN_IND`). The frames use manufacturer-specific data fields that have fixed values except for the contact hashes. In fact, we found that the contact hashes are the first two bytes of the SHA2 digest of the sender's contact identifiers that are also included in the record data (see Section 3.3). If the sender has less than four identifiers, the remaining contact hash fields are set to zero. Due to the short length, it appears feasible to use brute force to try all possible values.[3]

[3] Note that the sender still has to provide the complete hash during the HTTPS handshake before the receiver accepts the data on an *authenticated* connection.

Table 1: Symbols

SYMBOL	DESCRIPTION
$\mathbb{S}$	Contact hash search space
$\mathbb{C}$	Contacts in the target's address book
w	Target's BLE scan window
i	Target's BLE scan interval
i_{PHY}	Attacker's BLE PHY injection interval
r	Effective contact hash brute force rate
n	Tried hash values per scan window
p (p_j)	Hit probability after one (or j) scans

4.2 Brute Force Analysis

We assume that the attacker does not know the target's contacts and, thus, attempts to enable the target's AWDL interface using brute force. As the target has at least one contact identifier (the address book contains at least the user's own Apple ID), the attacker needs to try $\mathbb{S} = 2^{16} = 65\,536$ hashes in the worst case. Thus, the challenge for the attacker is to quickly send a large number of BLE advertisements *while* the target is conducting a BLE scan. In the following, we analyze *how fast* the attacker can deplete the search space and *how successful* they would be. We start investigating the results for a single BLE scan window and then extend our analysis to multiple scan intervals.

One Scan Window. Let the attacker inject BLE advertisement frames at the physical layer with an interval of i_{PHY}. Further, consider that the attacker has a single radio and that BLE uses three advertisement channels [17]. Also, recall that an AirDrop BLE frame has room for four contact hashes. Then, the attacker's effective brute force rate r can be calculated as:

$$r = \frac{4}{3 \cdot i_{\text{PHY}}} \,. \tag{6}$$

Now, we can compute the number of hash values n that the attacker can inject per scan window w [17] as:

$$n = w \cdot r \,. \tag{7}$$

Let X be a random variable, and $P(X = k)$ denote the probability that the target "sees" k known contact hashes during one scan window. Since the attacker moves through the search space sequentially, we can formulate the problem using the *urn model* in *drawing without replacement* mode which results in a hypergeometric distribution. We get:

$$P(X = k) = \frac{\binom{n}{k}\binom{\mathbb{S}-n}{\mathbb{C}-k}}{\binom{\mathbb{S}}{\mathbb{C}}} \,. \tag{8}$$

In particular, the attacker only requires one hit to activate the target's AWDL interface whose probability we call p:

$$\begin{aligned} p &= P(X \geq 1) = 1 - P(X = 0) \\ &= 1 - \frac{\binom{n}{0}\binom{\mathbb{S}-n}{\mathbb{C}-0}}{\binom{\mathbb{S}}{\mathbb{C}}} = 1 - \frac{\binom{\mathbb{S}-n}{\mathbb{C}}}{\binom{\mathbb{S}}{\mathbb{C}}} \,. \end{aligned} \tag{9}$$

Using the Stirling's approximation $\binom{n}{k} \approx \frac{n^k}{k!}$ for $k << n$, we can simplify Eq. (9) as:

$$\begin{aligned} p &\approx 1 - \frac{\frac{(\mathbb{S}-n)^{\mathbb{C}}}{\mathbb{C}!}}{\frac{\mathbb{S}^{\mathbb{C}}}{\mathbb{C}!}} = 1 - \frac{(\mathbb{S}-n)^{\mathbb{C}}}{\mathbb{S}^{\mathbb{C}}} \\ &= 1 - \left(\frac{\mathbb{S}-n}{\mathbb{S}}\right)^{\mathbb{C}} = 1 - \left(1 - \frac{n}{\mathbb{S}}\right)^{\mathbb{C}} \,. \end{aligned} \tag{10}$$

Multiple Scan Intervals. BLE devices perform scans regularly at a fixed interval i [17]. Let Y be a random variable indicating that the attacker has a hit ($Y = 1$) or not ($Y = 0$) during one scan. We assume that the attacker does not know when the target's scan window starts and, therefore, that Y is i.i.d. between scans.[4] Let j indicate the target's jth scan since the attacker started their brute force attack. Then, the probability that the attacker had k hits after j scans is given by a binomial distribution:

$$P(Y = k) = \binom{j}{k} p^k (1-p)^{j-k} \,. \tag{11}$$

Again, the attacker needs at least one hit whose probability we denote as p_j (note that $p_1 = p$):

$$p_j = P(Y \geq 1) = 1 - P(Y = 0) = 1 - (1-p)^j \,. \tag{12}$$

With Eq. (10), we get:

$$p_j \approx 1 - \left(1 - \frac{n}{\mathbb{S}}\right)^{j\mathbb{C}} \,. \tag{13}$$

We know that j depends on the time since the attack started and the target's BLE scan interval i (the target performs one BLE scan of length w per interval). Let t denote the attack duration, then $j \leq \lfloor t/i \rfloor$. Finally, we denote the success probability at time t as

$$p(t) \approx 1 - \left(1 - \frac{wr}{\mathbb{S}}\right)^{t\mathbb{C}/i} \,. \tag{14}$$

4.3 Jailbreaking BLE Advertisements

The Bluetooth standard imposes a minimum advertisement interval[5] of 100 ms for non-connectable undirected advertising [17, Vol. 6, Sec. 4.4.2.2], which we found is usually

[4] If the attacker knew the start of each scan window, they could follow a better strategy by only sending advertisements while the target is performing a scan. This way, they would deterministically succeed after they had gone through $\mathbb{S}$ once.

[5] The BLE advertisement interval accounts for a frame transmission on each of the three advertisement channels.

```
uint8_t *le_adv = airdrop_init_template()
for (uint16_t h = 0; /* loop */; h += 4) {
  airdrop_set_hashes(le_adv, h, h+1, h+2, h+3);
  for (uint16_t chan = 37; chan < 40; chan++) {
    le_adv_tx(le_adv, chan);
    sleep(0.625 /* in milliseconds */); } }
```

Figure 5: C pseudo code of our BLE brute force attack

enforced in the BLE firmware. By complying to the standard, the attacker would need at least $2^{16} = 27$ minutes to iterate through the entire search space once. If the attacker had access to the BLE physical layer to control and schedule individual transmissions, they could circumvent the standard's restrictions and, thus, iterate through the search space much faster. To this end, we extend an open source BLE firmware [65] for the Nordic nRF51822 [63] chipset to implement our brute force attack. In principle, our attack implementation is very simple and shown in Fig. 5. We use a send interval of $i_{\text{PHY}} = 0.625$ ms resulting in $r = 2133.3\,\text{s}^{-1}$ which allows the attack to iterate through $\mathbb{S}$ in only $2^{16}/f = 30.72$ s. By using three BLE radios (one for each advertisement channel), we could reduce this time to 10.24 s. However, we show that using one radio is sufficient in practice.

4.4 Target Response Times Micro Benchmark

We measure the target response time, i. e., the time it takes for a target to turn on its AWDL interface when being exposed to our attack. In particular, we measure the response time for a *contacts-only* receiver that has 10, 100, and 1000 contact identifiers in their address book. In addition, we include reference measurements for a receiver in *everyone* mode under the same attack.

Setup. For the experiment, we use a Wi-Fi sniffer (Broadcom BCM4360) to receive AWDL AFs and a $20 micro:bit device [58] to inject BLE advertisements. To get the response times, we start a brute force attack and measure the time until we receive the first AF from the target. We then stop the attack and wait until the target stops sending AFs which means that the AWDL interface has turned off. Then, we start over to collect 50 measurements per setting.

Results. We show the results for an iPhone 8 (iOS 12) in Fig. 6. The plot also includes the analytical response time distribution based on Eq. (14), assuming a BLE scan window w and interval i of 30 ms and 300 ms, respectively.[6]. We can make several observations: (1) Our analytical model does not capture our experimental results precisely but approximates them within an order of magnitude which is sufficient for our purposes. (2) The median response time of targets with only 10 contact identifiers in their address book is 10 seconds and

[6] https://lists.apple.com/archives/bluetooth-dev/2014/Sep/msg00001.html

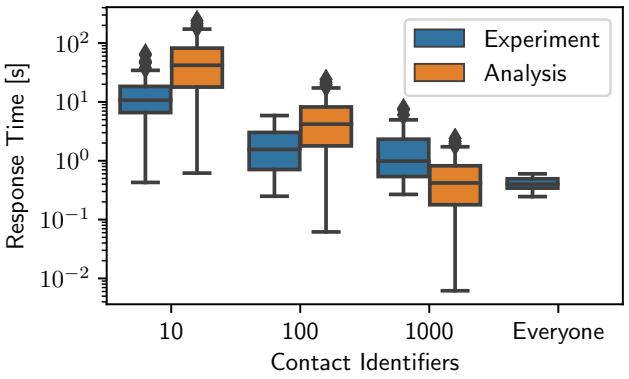


Figure 6: Time it takes until target turns on its AWDL interface after being exposed to our brute force attack.

decreases to about 1 second when more contacts are available. We found that a user has more than 136 contacts on average based on a user study that we describe in Section 5.2. (3) This means that the brute force attack is feasible for scenarios where the target will be in the attacker's communication range for a few seconds.

5 Privacy: Tracking Apple Device Users

In this section, we assess privacy issues in AWDL and find that AWDL devices are easily trackable. First, we discuss protocol fields that enable tracking. Then, we leverage the attack presented in Section 4 to perform an experimental vulnerability assessment at different locations and compare the results with a user study spanning 500 participants. Finally, we discuss possible mitigations.

5.1 Identifying Devices and Users via AWDL Protocol Fields

Even though AWDL implements MAC randomization for the IEEE 802.11 header, AWDL-specific fields contain long-term device identifiers that disclose sensitive information about the user, undermining MAC randomization. In particular, AWDL includes the following sensitive fields in the AFs which devices broadcast *in the clear* multiple times per second when the AWDL interface is active:

- The *hostname* may include parts of the user's *name*, e. g., "Janes-iPhone," which is the default when setting up a new device.
- The *real MAC address* as well as the AP the device is currently connected to.
- The *device class* differentiates between devices running macOS, iOS/watchOS, and tvOS.
- In combination with the *protocol version*, this can be used to infer the OS version, e. g., AWDL v2 is used in macOS 10.12 while AWDL v3 is used in macOS 10.13. The attacker could exploit the OS information during

reconnaissance to mount attacks on vulnerable driver implementations.

Targets need to broadcast AFs to make these vulnerabilities exploitable, which an attacker can practically enforce by mounting the attack presented in Section 4.

5.2 A Survey on the Potential of Apple Device User Tracking

The hostname set by default during Apple iOS and macOS device installation includes the user's name [3]. Due to its frame structure, the AWDL protocol aids an adversary in mapping a hostname with the MAC address of the device. This enables them to track users even if users change this hostname on their device. The combination also enables more sophisticated threats as the person's name can be combined with information from public databases (e. g., US census [86]) to infer their home and work locations, while the MAC address can be used to track them in real-time. To assess what percentage of device hostnames contain parts of the owner's name, we conducted a survey among 500 Apple device users on Amazon Mechanical Turk. This survey contained questions[7] relevant to the attacks demonstrated in this paper, and we report the statistics in the relevant sections. In particular, in the context of tracking, we asked the surveyors if it was easy for other users to find their device because their hostname contained parts of their real name. We report the results of this question along with the results of an experimental evaluation in the next section.

5.3 Experimental Vulnerability Analysis

To demonstrate the feasibility of user tracking using AWDL, we collect the number of discovered devices and check whether that device's hostname includes a person's name in four different locations within the US. We selected the locations to reflect static as well as dynamic environments. In particular, we recorded at a departure gate of an airport, in the reading section of a public library, in a moving metro train, and in the food court of a university.

Determining Whether a Hostname Contains a Person's Name. We use two databases to determine whether a hostname contains a person's name: the 2010 US Census [85] containing 162 253 family names, and the 1918–2017 baby names from the US Social Security Administration [87] containing 96 743 given names. When detecting a new AWDL node, we check string segments separated by hyphens against these two databases.[8] Note that when one segment matches the given name database, it is not matched again as a family name because it is more likely that an Apple device will include a person's given name [3].

[7] The survey questionnaire is available at https://goo.gl/forms/0okC4UphTQBnQ0FB3

[8] If a segment ends with the letter "s," we also check the segment without a trailing "s." In addition, we ignore segments containing common device names such as "iPhone," "Mac," etc. For example, for the hostname "Johns-iPhone," we try to match the strings "Johns" and "John" to our name databases.

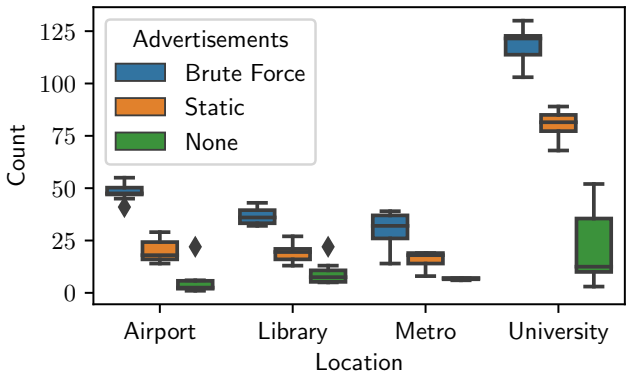


Figure 7: Discovered AWDL devices at one location during one minute.

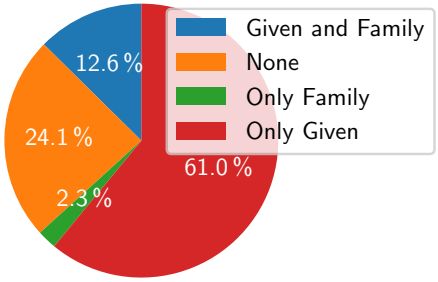


Figure 8: Persons' names distribution in hostnames.

Ethical Statement. To preserve user privacy and not having to store any sensitive user information, we fully automated the name matching procedure. In particular, we only stored salted hashes of the discovered hostnames (to differentiate between devices) together with two bits indicating whether the hostname contained a given or a family name. The salt was generated randomly, kept in memory only, and discarded after the completion of each experiment.

Setup. We do the measurements (a) without an attack (passive), (b) with static BLE advertisements containing only the "zero" contact hash, and (c) with our BLE brute force approach. With (b), only devices in the *everyone* mode should respond, with (c) we also capture those that are in *contacts-only* mode. We run each setting for 60 seconds and repeat it 10 times per location. To avoid statistical bias, we cycle through the (a) to (c) settings back to back in each iteration and use a cooldown time of 40 seconds between them. The cooldown ensures that all devices in proximity have turned off their AWDL interfaces again.

Experimental Results. Fig. 7 shows the number of discovered AWDL devices in the different locations. By using the

brute force approach, we can discover about twice as many devices compared to sending only regular advertisements. This means that in our experiments, approximately 50 % of the Apple devices are in AirDrop's *everyone* mode. Our survey complements our experimental results by indicating that 20 % of Apple device users have AirDrop turned off and, thus, are not trackable via AWDL. It is interesting to note that we are able to pick up AWDL devices even when not sending any advertisements. This can happen if a device (not controlled by us) sends out advertisements itself, for example, when a user opens the AirDrop sharing pane which apparently occurred regularly at the university location. Finally, we found that among all discovered devices, more than 75 % contain a person's name in the hostname. Most devices contain only a given name which is the default for freshly set up Apple devices [3], some contain a combination of a given and family name, and very few contain only a family name. Our survey confirms these results as 68 % answered that it was "easy" or "very easy" for others to recognize their device because it contained their name.

Outlook for Large-Scale Attack. In this analysis, we show what kind of information a motivated attacker would be able to collect. We used a single fixed physical location for each experiment and did not attempt to track any user movement. However, given that we can receive unique identifiers of Apple devices (Wi-Fi MAC address and hostname), mounting a large-scale tracking attack should be trivial for an adversary that can deploy multiple low-cost Wi-Fi and BLE nodes throughout an area.

5.4 Mitigation

We present a short-term solution and then propose two mitigation techniques that remove stable device identifiers to prevent user tracking via AWDL.

Disable AirDrop. Until Apple fixes the problem, the only way to thwart user tracking is to disable AirDrop completely. This presents a countermeasure to our attack presented in Section 4, i. e., the AWDL interface cannot be remotely activated via BLE advertisements.

Hide Real MAC Address When Not Connected to an AP. When a device connects to an AP it uses its real MAC address for communication, in which case AWDL does not disclose new information. However, we have found that the MAC address is occasionally included in AFs even when the device is not connected to an AP. This appears to be unintended behavior and should be fixed via a software update.

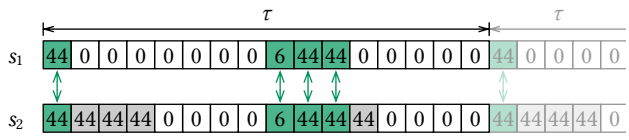
Randomize Hostname for AWDL. Apple devices transmit their hostname in AWDL AFs as well as the mDNS responses during service discovery that are used to find AirDrop instances (see Section 3). As a countermeasure, we propose to use randomized hostname with AWDL similar to MAC address randomization. If an application such as AirDrop needs the real hostname for identification, it should only be transmitted via an encrypted and authenticated channel such as TLS. In fact, AirDrop already transmits the device name in the HTTPS handshake and uses this name in the UI ignoring the hostname from mDNS responses. Therefore, hostname randomization would not require any changes to the AirDrop implementation which would retain backward compatibility.

5.5 Related Work: User Tracking

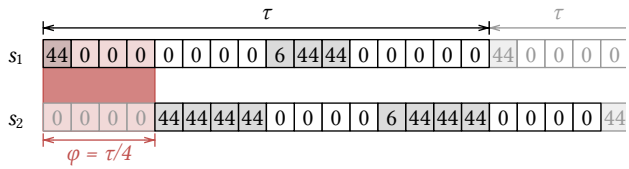
Several related works have studied the topic of user tracking from mobile devices. Some common attack vectors include using the GPS sensor [35, 48, 55, 83, 95], cellular [5, 44, 50, 67], Wi-Fi [15, 40, 69, 96, 97], radio interface fingerprinting [90], and motion sensors [25, 39, 59, 61, 62]. We believe that the above works are orthogonal to our approach, and could be used in conjunction with our approach to improve tracking performance. Many countermeasures have also been proposed to prevent tracking from the above vectors. Some of them include recommending new location frameworks and privacy metrics [9, 28, 29, 30, 49, 64], location obfuscation [1, 6, 18, 72, 92], location cloaking [42, 43], synthesizing locations [16, 46, 53, 82, 98], sensor data obfuscation [22, 23], and permission analysis [32, 45, 68]. Along with resource permissions on mobile devices, these countermeasures limit the practicality of some of the above attacks.

Some device-specific identifiers have also been used for tracking, e. g., IMEI [36, 93], BLE addresses [24, 31, 47], and MAC addresses [21, 33, 54, 60]. While IMEI-based tracking can be easily mitigated by protecting access to this information, BLE is a dominant standard for fitness trackers and smartphone communication and their addresses must be exposed. Tracking using BLE identifiers has been demonstrated to be easy. However, our approach has the added benefit for an attacker that the hostname is exposed. This allows inferring additional user information such as home and work locations, family members, or movement patterns, which are useful for more targeted tracking [34, 84]. Like BLE addresses, MAC addresses are also essential as they form the backbone of layer 2 network communication and must be exposed for networking (e. g., Wi-Fi probe requests).

MAC address randomization has been proposed to prevent device tracking through Wi-Fi probe requests [11, 26]. Today, both Apple and Google implement MAC address randomization in their mobile operating systems. Randomization does improve user privacy; however, some works have demonstrated that devices are still trackable. For example, [89] implemented an algorithm using probe request fingerprinting that has a 50 percent success rate for tracking users for 20 minutes. Another work [56] demonstrated that MAC randomization could be defeated through timing attacks, where a signature based on inter-frame arrival times of probe requests can be used to group frames coming from the same device with distinct MAC addresses. Their framework could group random MAC addresses of the same device up to 75% of cases for about 500 devices. Our work advances the scalabil-



(a) In normal operation, two channel sequences result in non-zero overlap, allowing two nodes to communicate. In this example, they can communicate during four out of 16 EAWs.



(b) A phase shift of a quarter period ($\phi = \tau/4$) results in zero overlap preventing the two nodes from communicating with each other.

Figure 9: Sketch of the desynchronization attack.

ity, tracking time and accuracy of the prior works. We show that, owing to implementation nuances in the AWDL protocol, an adversary can track millions of Apple device owners globally with 100 % accuracy.

6 DoS: Impairing Communication with Desynchronization

AWDL does not employ any security mechanisms. Instead, Apple decided to leave security mechanisms to the upper layers. Thus, while end-to-end confidentiality and integrity can be achieved using a secure transport protocol such as TLS, AWDL frames are vulnerable to forgery which renders any upper layer using AWDL susceptible to attacks on availability. In this section, we present a novel DoS attack that targets AWDL's synchronization mechanism (Section 2) to prevent two nodes from communication with each other. In the following, we describe a novel *desynchronization* attack which aims to minimize the channel sequence overlap of two targets. Next, we evaluate the attack's performance and present an effective mitigation method. Finally, we discuss related work.

6.1 Desynchronizing Two Targets

We exploit AWDL's synchronization mechanism to reduce the channel overlap by inducing an artificial *phase* offset between two targets. In order to succeed, the attacker needs to (1) get recognized as the master by both targets, (2) communicate with each target separately to (3) send different sets of synchronization parameters that result in zero (or minimal) channel overlap. Figure 9 depicts the non-zero overlap in normal operation and the zero overlap as the result of the desynchronization attack. We describe the three steps in the following.

(1) Winning the Master Election. The master election in AWDL is based on a numeric comparison of two values that are transmitted in the *election parameters*. The first value

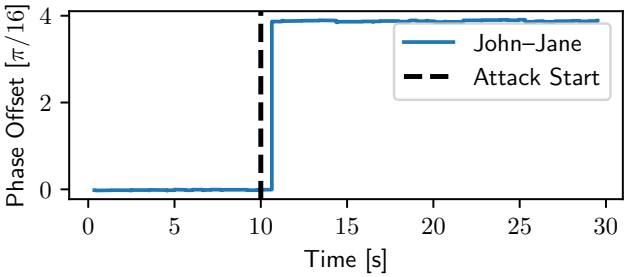


Figure 10: Phase offset between two targets before and after mounting a desynchronization attack which induces a phase shift of $\phi = \tau/4$.

is called *metric*, and each node draws one randomly upon initialization. The numeric range of the *metric* is bounded and depends on the AWDL version that runs on the node [79]. The second value is called *counter* which is initialized to a random value and increases linearly over time while the node is elected as a master. Given the metric and counter values of two nodes A and B as (m_A, c_A) and (m_B, c_B), respectively, then, A wins the master election if

$$c_A > c_B \vee (c_A = c_B \wedge m_A > m_B) \tag{15}$$

and loses otherwise. To consistently win the election, the attacker sets c and m to their maximum values.

(2) Unicasting AFs. The attacker needs to send the synchronization parameters to each target without the other one noticing. We have found that while AFs are typically sent to the broadcast MAC address `ff:ff:ff:ff:ff:ff`, AWDL nodes also accept unicast AFs. Therefore, the attacker can unicast their AFs to make sure that only the intended target receives them.

(3) Phase Shift: Different Synchronization Parameters. To desynchronize two targets, the attacker needs to send incompatible synchronization parameters that will result in a controllable offset. We explain how the attacker calculates the relevant parameters i and t_{AW} for both targets. Let us assume that the attack starts at some time T_s. An AF sent to the *first* target at some time T_{Tx} with $t = \left\lfloor \frac{T_{\text{Tx}} - T_s}{1024} \right\rfloor$ (in TU) will include the following parameters:

$$i = \left(\left\lfloor \frac{t \mod 64}{16} \right\rfloor + 4 \left\lfloor \frac{t}{64} \right\rfloor \right) \mod 2^{16} \quad \text{and,} \tag{16}$$

$$t_{\text{AW}} = 64 - t \mod 64 \quad . \tag{17}$$

For the *second* target, the attacker will calculate t as $t^{\phi} = \left\lfloor \frac{T_{\text{Tx}} - T_s - \phi}{1024} \right\rfloor$ and compute i^{ϕ}, t^{ϕ}_{AW} analogously to Eqs. (16) and (17). We verify the correctness of these calculations experimentally and show the resulting phase offset between two targets for a target phase $\phi = \tau/4$ in Fig. 10.

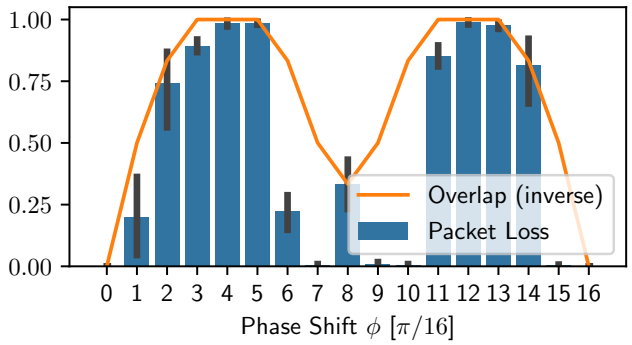


Figure 11: Packet loss for different phase shifts.

6.2 Evaluating Packet Loss

We evaluate the impact of our desynchronization attack by measuring the packet loss via the `ping` program. In particular, we use an APU board [66] equipped with a Qualcomm Atheros AR928X Wi-Fi card to act as an attacker which can inject AWDL AFs. The ICMP echo requests are sent from a MacBook Pro (Late 2015, macOS 10.13) to an iPhone 8 (iOS 12). The attacker induces different phase shifts spanning one period. The sender emits 100 ICMP echo requests per experiment which we repeat 10 times and plot the resulting packet loss in Fig. 11. The error bars indicate the standard deviation. In addition, we include the inverse channel overlap (see Section 2) which is calculated for two identical sequences which we have observed were the most common ones during our experiment: $44,44,44,0,0,0,0,0,6,44,44,0,0,0,0,0$.

At $\phi = 0$, there is no attack, while at $\phi = 16$, the targets are desynchronized by a full period which unsurprisingly does impair communication reliability. The other results indicate that the desynchronization attack significantly degrades communication between the targets, peaking at phase shifts where the targets are off by a quarter ($\phi = 4$) and three-quarter period ($\phi = 12$) which is where the channel overlap has its minima (and the inverse overlap its maxima). At these settings, the packet loss is almost 100 %. Surprisingly, some phase shifts (e. g., $\phi = 6,7,9,10,15$) result in less packet loss than the overlap predicts. We suspect the reason to be retransmissions on the MAC layer (up to 7 times in Wi-Fi [73]) which, at the cost of longer latency, increase the chance that a frame will be received in a subsequent AW.

6.3 Mitigating Desynchronization

Devices can mitigate our desynchronization attack by discarding unicast AFs. Not accepting unicast frames is an extremely effective and practical countermeasure because it will cause all nodes in range to process the same information exclusively. While this does not prevent an attacker from winning the master election and, thus, sending invalid synchronization parameters, as all nodes process the same frames, it becomes much harder to create a deterministic offset between two targets. A more sophisticated attacker could employ attacks on the PHY layer (e. g., using directional antennæ) to achieve a similar effect as that of unicasting. However, such attacks are difficult to carry out in practice.

6.4 Related Work: Reactive Jamming

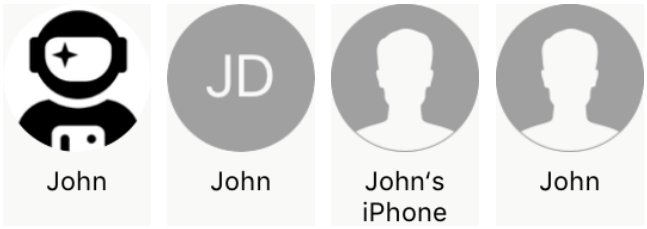
At first glance, our desynchronization attack achieves a similar effect as a *reactive jammer* [37, 52, 70, 91]. However, the desynchronization attack can be more attractive for two reasons: first, desynchronization in principle needs less energy than a jamming attack. The desynchronization attacker only needs to emit one frame every 1.5 s to maintain their position as a master node because AWDL nodes elect a new master if they have not received an AF for more than 1.5 s from their current one. In contrast, a reactive jammer needs to emit a jamming signal for every packet that the target sends. Second, it allows intercepting frames from its targets which enables to mount more sophisticated MitM attacks as presented in Section 7. In contrast, a normal jammer *kills* the frame in transit disallowing anyone (even the attacker themselves) to decode the frame [70]. There exist more sophisticated receiver designs that cancel out the jammer's own signal, but this typically requires special hardware [37]. Our desynchronization attack only requires a system with an off-the-shelf Wi-Fi chip and, thus, could even be implemented in a smartphone [71].

7 MitM: Planting Malware via AirDrop

This section describes a MitM attack on the popular AirDrop service which allows iOS and macOS devices to exchange files directly via AWDL. First, we assess the security of AirDrop and find that poor UI design choices enable an attacker to masquerade as a valid receiver. Then, we describe a complete MitM attack on AirDrop which prevents any sender to discover a valid receiver using a DoS attack and subsequently can intercept and modify any AirDrop file transmission. Finally, we discuss possible mitigations for the attack.

7.1 Ambiguous Receiver Authentication State

We have observed that AirDrop employs two different kinds of connections which we term *authenticated* and *unauthenticated* (see Section 3). Further, the user can set its device to be discoverable by *contacts only* or *everyone*. Counter-intuitively, the discoverability setting *only applies to the receiver side.* In particular, while a receiver in *contacts-only* mode will only accept files from authenticated senders, a sender will see all discoverable receivers irrespective of whether they are authentic or not. This ambiguity has profound implications for security because it is up to the user of the sending device to decide whether a connection is authenticated or not which can be non-trivial. The only visual cue to differentiate between an authenticated and unauthenticated connection is that an authenticated connection will show the receiver's name and photo from the sender's address book. Neither provides sufficient evidence to unambiguously decide whether a receiver



authentic | authentic | unauthentic | unauthentic

(a) Sender has John's contact photo (b) Sender does not have John's contact photo (c) Sender does not have John as a contact (d) Attacker spoofs John's identity

Figure 12: UI representation of a receiver.

is authentic. First, if no contact photo is available (users augment only 27 % of their contacts with a photo according to our survey), the icon contains the receiver's initials in a grey circle which is similar to that of an unauthenticated receiver (a grey circle with a head's silhouette). Second, the name that is displayed underneath unauthenticated receivers is the receiver's device name. Based on our results in Section 5, a device name contains the user's given name in the majority of the cases (more than 70 % according to our experimental evaluation in Section 5), which the attacker can exploit to create a trustworthy-looking device name. Unless users are sensitive to such subtle UI changes, an attacker can easily trick them into sending files via an unauthenticated connection. Figure 12 compares the different receiver icons including a spoofed identity by the attacker. We want to highlight the similarity between an authenticated identity (Fig. 12b) and a spoofed identity (Fig. 12d).

7.2 The Complete AirDrop MitM Attack

Our MitM attack on AirDrop is carried out in three phases. First, we break the discovery process to put ourselves in a privileged position. Second, we wait until the target receiver becomes discoverable by *everyone*, effectively forcing the user to downgrade the connection. Third, we relay and manipulate the actual data transfer to plant arbitrary files at the receiver. We illustrate the attack in Fig. 13 and explain each phase in more detail below. Also, we provide a video PoC of the attack [77].

(1) Breaking Discovery via DoS. The most crucial part of the attack is preventing the sender to discover the receiver such that it appears as an icon in the sharing pane. In particular, we need to prevent that the *Discovery* handshake via HTTPS completes successfully. In principle, such a DoS attack could be carried out via our desynchronization attack (Section 6). However, we found that it could not reliably prevent the short *Discovery* request and responses from being received. This is due to the fact that AirDrop senders increase the channel allocation when starting the discovery process, thus, increasing the overlap with the receiver even when desynchronized. As an alternative, we used the well-known TCP reset attack which sends TCP segments including an RST flag to the targets which, in turn, immediately drops the connection. For this attack, the attacker sends out an RST reply for every TCP segment that is not addressed to itself and effectively prevents any reconnection attempts from the sender to the receiver.

(2) Downgrading an Authenticated Connection. For a complete MitM attack, we need to authenticate to the receiver. Otherwise, it will deny any *Ask* or *Upload* requests. If the receiver is discoverable by *everyone*, this is trivial, since it accepts all authentication attempts, even those with a self-signed certificate which the attacker can easily generate (see Section 3.3). The receiver indicates a successful authentication attempt from a non-contact by including its device name in the *Discover* response. However, we have found that in most cases (59.4 % in our survey), users set their device to *contacts only*. In such cases, we leverage the ongoing DoS attack to force the receiver to try the *everyone* setting.

(3) Relaying and Modifying Data Transfer. Once the receiver becomes discoverable (we can check when a receiver is discoverable by everyone by periodically sending *Discovery* requests), we advertise our own AirDrop identity via mDNS and wait until the sender tries to perform the authentication handshake via HTTPS for discovery which we let succeed. We relay the sender's *Ask* request to the receiver including the original file thumbnail to make the request appear valid. After the receiver accepts the transmission request, we relay the answer back to the sender which—in turn—starts to send the actual file. We can now decide whether to relay a modified version of the file or send an entirely new one possibly containing malware to the receiver.

7.3 Implementation

Our PoC of the MitM attack consists of two components. First, we re-implement AWDL such that we were able to overhear and parse data frames not addressed to us which is required to mount a TCP reset attack. Second, we implement an AirDrop-compatible client and server which we use to probe the discoverability status of the receiver target and finally implement the MitM attack as depicted in Fig. 13. We make both projects available as open source software [76, 78].

AWDL. Our AWDL implementation [76] is written in C and runs on Linux as well as on macOS. On macOS, the implementation can be used as a drop-in replacement for Apple's own AWDL interface. We use the monitor mode and frame injection of the system's Wi-Fi card to receive and inject raw IEEE 802.11 frames. In addition, we provide a virtual network interface (via *tuntap*) to the system such that any IPv6-capable application can use AWDL. Internally, our implementation takes care of frame parsing, synchronization, election, and scheduling data frames in the correct AWs.

AirDrop. Our AirDrop implementation [78] is written in Python and implements an unauthenticated AirDrop sender

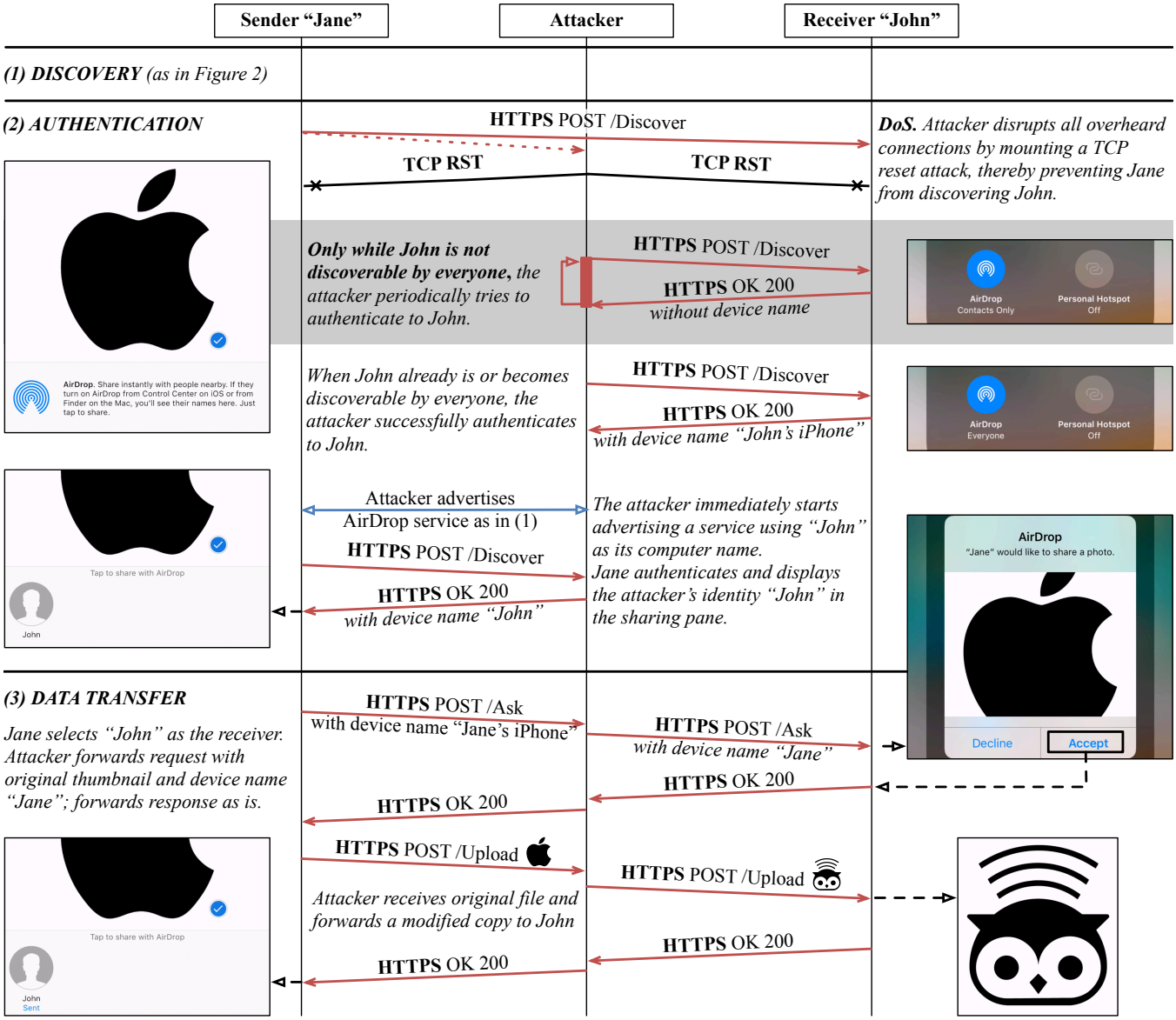


Figure 13: Protocol flow and user interaction of our MitM attack on AirDrop.

and receiver. The code exposes a command line interface which allows to *find* discoverable receivers, *send* files to them, and *receive* files from any sender.

7.4 Mitigation

We discuss possible mitigation strategies. We examine them according to complexity to implement, starting with the mitigation requiring the least number of changes to existing AirDrop implementations.

Provide Stronger Visual Cues for Authenticated Receivers. One of the core problems of the current design of AirDrop is that a user might have a hard time to differentiate between authenticated and unauthenticated receivers (see Section 7.1 and Fig. 12). Currently, the only cues to decide whether a receiver is authenticated are the display of a contact photo and contact name. We have shown that the former is not commonly available (users augment 27.4 % of their contacts with photos) and the latter can be spoofed. Therefore, we propose to provide stronger visual cues that unambiguously tell the user if a receiver is authenticated or not. This is already customary in web browsers where HTTPS-protected websites are augmented with a green (lock) symbol telling the user that the website they are visiting is authentic.

Reset *Everyone* to *Contacts Only* After a Timeout. Users might set the discoverability setting of their device to *everyone* for convenience or if they used it for one occasion and then forgot to reset it. In either case, we believe that the *everyone* setting should only be used except when it is required,

i. e., if one wants to receive a file from a non-contact. To protect negligent users, we propose to use a timeout after which the discoverability setting is reset to *contacts only*. Alternatively, one could reset the setting the next time the device is locked. This would also prevent past cases where people would receive inappropriate photographs from strangers in public places [13, 41] because, in *contacts-only* mode, devices will transparently reject all request from unauthenticated senders.

Introduce Secure AirDrop Mode for Non-Contacts. Our last proposal involves deprecating unauthenticated connections and instead establish authentication with a non-contact via an out-of-band (OOB) channel. AirDrop could transmit one-time cookies with similar functionality as the *record data* (see Section 3.3) during the initial HTTPS authentication handshake (see Section 3.2). The one-time cookies could be validated via an OOB channel such as NFC or via QR codes. After one transfer (or after a specific timeout), each device deletes its one-time cookie. By committing to the one-time cookie in the TLS handshake, a MitM attack on the OOB channel would be fruitless because the attacker could not establish a TLS connection with the same key. Unfortunately, such a mode would require one more manual step by both parties and, therefore, would impair usability.

7.5 Related Work: Attacks on AirDrop

Other attacks on AirDrop have been presented before. An impersonation attack [10] exploits mDNS/DNS-SD to redirect file transmissions to an attacker for unauthenticated connections. In particular, the attack uses forged SRV and AAAA responses to redirect an AirDrop ID to the attacker. In contrast to our work, [10] does not differentiate between *authenticated* and *unauthenticated* connections and claims that the UUID certificate (see Section 3.3) could not be bound to any contact identifiers, which we have found to be untrue. Also, the attack only works on unauthenticated connections, while our attack also targets authenticated connections via a downgrade attack and we present a complete MitM attack which allows an attacker to send malicious files to the receiver stealthily. Finally, [10] proposes a conflict detection mechanism for Multicast DNS (mDNS) to prevent their attack, which is based on the assumption that "disrupting two parties' communication through a Wi-Fi direct link or a local network is difficult for the adversary without access to the routing infrastructure of the network." In this work, we show that it is indeed practical to mount a DoS on the link layer since AWDL does not employ any security mechanisms. An earlier work [27] targeted a vulnerability in AirDrop's implementation which allowed the attacker to install files in arbitrary directories on the target's system. Apple fixed this bug in 2015.

8 Implementation Security

During our AWDL analysis and building an AWDL prototype, we found two implementation flaws in Apple's OSes that allow an attacker to crash devices in proximity.

DoS: Kernel Panic and System Crash. These flaws can be exploited by sending corrupt AFs. In particular, we can trigger kernel panics by setting invalid values in the synchronization parameters (affecting macOS 10.12) and the channel sequence (affecting macOS 10.14, iOS 12, watchOS 12, and tvOS 5), respectively. To showcase our findings, we provide a video of our PoC which exploits the second vulnerability on iOS devices [75]. The video demonstrates how an attacker mounts a targeted DoS attack that crashes a single device and a blackout DoS attack that crashes all devices in range of the attacker at the same time.

Outlook: Remote Code Execution. While not critical by themselves, the mere existence of these vulnerabilities creates a new class of threats to Wi-Fi devices as an attacker can exploit them *without any authentication towards the target*, i. e., they do not have to be on the same network. In light of past discovered remote code execution in implementations of standardized Wi-Fi procedures [8, 14], we think that a determined attacker can find similar flaws for AWDL.

9 Conclusion

The deployment of open Wi-Fi interfaces enables new types of applications for mobile devices. They allow devices in proximity to communicate with each other without being connected to the same Wi-Fi network. On the downside, this also opens new opportunities for an attacker as they no longer have to provide any kind of authentication (e. g., access to a secure Wi-Fi network). In this paper, we investigate the first protocol of this kind, i. e., Apple's proprietary AWDL. In particular, we find three distinct protocol-level vulnerabilities that allow for DoS, user tracking, and MitM attacks. In addition, we discovered two implementation bugs in Apple's OSes that cause DoS. Given the complexity of the protocol and implementations, we conjecture that more severe vulnerabilities will be found in the future. To build PoCs for these attacks, we have reverse-engineered AirDrop, a system service that runs on top of AWDL, and have implemented open versions of both AWDL and AirDrop which we make available as open source software. Finally, our findings have implications for the non-Apple world: NAN, commonly known as Wi-Fi Aware, is a new standard supported by Android which draws on AWDL's design and, thus, might be vulnerable to the similar attacks as presented in this work. This is pending further investigation.

Responsible Disclosure

We have contacted Apple about our findings on December 17, 2018. We have shared a draft of this paper as well as our PoC code and support Apple in fixing the privacy leaks (Section 5) and desynchronization issue (Section 6) in AWDL as well as the ambiguous authentication state in AirDrop (Section 7). We have reported the two implementation vulnerabilities (Section 8) earlier on August 14 and 27, 2018, respectively. Apple will not fix the first one affecting only macOS 10.12, but

has released software updates addressing the second one on October 30, 2018, for all Apple OSes [4].

Acknowledgements

This work is funded by the LOEWE initiative (Hesse, Germany) within the NICER project and by the German Federal Ministry of Education and Research (BMBF) and the State of Hesse within CRISP-DA. The work was partially supported by NSF grant 1740907. We thank the Apple Product Security team for feedback on the paper.

Availability

We release the source code of our AWDL [76] and AirDrop [78] implementations as part of the Open Wireless Link project [81] (https://owlink.org).

References

[1] Miguel E. Andrés, Nicolás E. Bordenabe, Konstantinos Chatzikokolakis, and Catuscia Palamidessi. Geo-indistinguishability: Differential Privacy for Location-based Systems. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, November 2013. doi: 10.1145/2508859.2516735.

[2] Apple Inc. Use AirDrop on your iPhone, iPad, or iPod touch, June 2018. URL https://support.apple.com/en-us/HT204144. [Accessed September 20, 2018].

[3] Apple Inc. Change the name of your iPhone, iPad, or iPod, October 2018. URL https://support.apple.com/en-us/HT201997. [Accessed November 14, 2018].

[4] Apple Inc. About the security content of iOS 12.1, 2018. URL https://support.apple.com/en-in/HT209192. [Accessed Feburary 14, 2019].

[5] Myrto Arapinis, Loretta Ilaria Mancini, Eike Ritter, and Mark Dermot Ryan. Analysis of Privacy in Mobile Telephony Systems. *International Journal of Information Security*, October 2017. doi: 10.1007/s10207-016-0338-9.

[6] Claudio A. Ardagna, Marco Cremonini, Sabrina De Capitani di Vimercati, and Pierangela Samarati. An Obfuscation-Based Approach for Protecting Location Privacy. *IEEE Transactions on Dependable and Secure Computing*, 8, January 2011. doi: 10.1109/TDSC.2009.25.

[7] Armis Inc. BlueBorne, 2018. URL https://armis.com/blueborne/. [Accessed November 14, 2018].

[8] Nitay Artenstein. Broadpwn: Remotely Compromising Android and iOS via a Bug in Broadcom's Wi-Fi Chipsets, July 2017. URL https://blog.exodusintel.com/2017/07/26/broadpwn/. [Accessed June 28, 2018].

[9] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. Boxify: Full-fledged App Sandboxing for Stock Android. In *USENIX Security Symposium*, August 2015.

[10] Xiaolong Bai, Luyi Xing, Nan Zhang, Xiaofeng Wang, Xiaojing Liao, Tongxin Li, and Shi-Min Hu. Staying Secure and Unprepared: Understanding and Mitigating the Security Risks of Apple ZeroConf. In *IEEE Symposium on Security and Privacy (S&P)*, May 2016. doi: 10.1109/SP.2016.45.

[11] Marco V. Barbera, Alessandro Epasto, Alessandro Mei, Vasile C. Perta, and Julinda Stefa. Signals from the Crowd: Uncovering Social Relationships through Smartphone Probes. In *ACM Internet Measurement Conference (IMC)*, October 2013. doi: 10.1145/2504730.2504742.

[12] Elad Barkan, Eli Biham, and Nathan Keller. Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communication. In *Advances in Cryptology (CRYPTO)*, August 2003. doi: 10.1007/978-3-540-45146-4_35.

[13] Sarah Bell. Police investigate 'first cyber-flashing' case, 2015. URL https://www.bbc.com/news/technology-33889225. [Accessed September 25, 2018].

[14] Gal Beniamini. Over The Air: Exploiting Broadcom's Wi-Fi Stack (Part 2), April 2017. URL https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_11.html. [Accessed June 28, 2018].

[15] Laurent Bindschaedler, Murtuza Jadliwala, Igor Bilogrevic, Imad Aad, Philip Ginzboorg, Valtteri Niemi, and Jean-Pierre Hubaux. Track Me If You Can: On the Effectiveness of Context-based Identifier Changes in Deployed Mobile Networks. In *Network & Distributed System Security Symposium (NDSS)*, 2012.

[16] Vincent Bindschaedler and Reza Shokri. Synthesizing Plausible Privacy-Preserving Location Traces. In *IEEE Symposium on Security and Privacy (S&P)*, May 2016. doi: 10.1109/SP.2016.39.

[17] Bluetooth Special Interest Group (SIG). Bluetooth Specification Version 4.1. Technical report, December 2013.

[18] Nicolás E. Bordenabe, Konstantinos Chatzikokolakis, and Catuscia Palamidessi. Optimal Geo-Indistinguishable Mechanisms for Location Privacy. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, November 2014. doi: 10.1145/2660267.2660345.

[19] Stuart D. Cheshire. Proximity Wi-Fi. *U.S. Patent Application*, (US 2018/0083858 A1), March 2018. URL https://patents.google.com/patent/US20180083858A1.

[20] Aldo Cortesi, Maximilian Hils, Thomas Kriechbaumer, and contributors. mitmproxy: A free and open source interactive HTTPS proxy, 2010–. URL https://mitmproxy.org. [Version 3].

[21] Mathieu Cunche. I Know Your MAC Address: Targeted Tracking of Individual Using Wi-Fi. *Journal of Computer Virology and Hacking Techniques*, 10, November 2013.

[22] Anupam Das, Nikita Borisov, and Matthew Caesar. Tracking Mobile Web Users Through Motion Sensors: Attacks and Defenses. In *Network and Distributed System Security Symposium (NDSS)*, February 2016.

[23] Anupam Das, Nikita Borisov, and Edward Chou. Every Move You Make: Exploring Practical Issues in Smartphone Motion Sensor Fingerprinting and Countermeasures. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2018, January 2018. doi: 10.1515/popets-2018-0005.

[24] Aveek K. Das, Parth H. Pathak, Chen-Nee Chuah, and Prasant Mohapatra. Uncovering Privacy Leakage in BLE Network Traffic of Wearable Fitness Trackers. In *ACM Workshop on Mobile Computing Systems and Applications (HotMobile)*, February 2016. doi: 10.1145/2873587.2873594.

[25] Sanorita Dey, Nirupam Roy, Wenyuan Xu, Romit Roy Choudhury, and Srihari Nelakuditi. AccelPrint: Imperfections of Accelerometers Make Smartphones Trackable. In *Network and Distributed System Security Symposium (NDSS)*, February 2014.

[26] Adriano Di Luzio, Alessandro Mei, and Julinda Stefa. Mind Your Probes: De-Anonymization of Large Crowds Through Smartphone WiFi Probe Requests. In *IEEE International Conference on Computer Communications (INFOCOM)*, April 2016. doi: 10.1109/INFOCOM.2016.7524459.

[27] Mark Dowd. MalwAirDrop: Compromising iDevices via AirDrop. In *Ruxcon*, October 2015. URL http://2015.ruxcon.org.au/slides/.

[28] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, October 2010.

[29] Kassem Fawaz and Kang G. Shin. Location Privacy Protection for Smartphone Users. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, November 2014. doi: 10.1145/2660267.2660270.

[30] Kassem Fawaz, Huan Feng, and Kang G. Shin. Anatomization and Protection of Mobile Apps' Location Privacy Threats. In *USENIX Security Symposium*, August 2015.

[31] Kassem Fawaz, Kyu-Han Kim, and Kang G. Shin. Protecting Privacy of BLE Device Users. In *USENIX Security Symposium*, August 2016.

[32] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android Permissions Demystified. In *ACM Conference on Computer and Communications Security (CCS)*, October 2011. doi: 10.1145/2046707.2046779.

[33] Julien Freudiger. How Talkative is Your Mobile Device?: An Experimental Study of Wi-Fi Probe Requests. In *ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)*, June 2015. doi: 10.1145/2766498.2766517.

[34] Sébastien Gambs, Marc-Olivier Killijian, and Miguel Núñez del Prado Cortez. Show Me How You Move and I Will Tell You Who You Are. In *ACM SIGSPATIAL International Workshop on Security and Privacy in GIS and LBS (SPRINGL)*, 2010. doi: 10.1145/1868470.1868479.

[35] Sébastien Gambs, Marc-Olivier Killijian, and Miguel Núñez Del Prado Cortez. De-anonymization Attack on Geolocated Data. *Journal of Computer and System Sciences*, 80, December 2014. doi: 10.1016/j.jcss.2014.04.024.

[36] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *International Conference on Trust and Trustworthy Computing (TRUST)*. Springer, June 2012. doi: 10.1007/978-3-642-30921-2_17.

[37] Shyamnath Gollakota, Haitham Hassanieh, Benjamin Ransford, Dina Katabi, and Kevin Fu. They Can Hear Your Heartbeats: Non-Invasive Security for Implantable Medical Devices. In *ACM SIGCOMM Computer Communication Review*, October 2011. doi: 10.1145/2043164.2018438.

[38] Google. Wi-Fi Aware, 2017. URL https://developer.android.com/guide/topics/connectivity/wifi-aware. [Accessed June 28, 2018].

[39] Jun Han, Emmanuel Owusu, Le T. Nguyen, Adrian Perrig, and Joy Zhang. ACComplice: Location Inference Using Accelerometers on Smartphones. In *IEEE International Conference on Communication Systems and Networks (COMSNETS)*, January 2012. doi: 10.1109/COMSNETS.2012.6151305.

[40] Xiuping Han, Zhi Wang, and Dan Pei. Preventing Wi-Fi Privacy Leakage: A User Behavioral Similarity Approach. In *IEEE International Conference on Communications (ICC)*, May 2018. doi: 10.1109/ICC.2018.8422764.

[41] Harry Harris. Oakland-Maui flight: Pepper spray emergency follows disturbing photo, 2018. URL https://www.eastbaytimes.com/2018/09/01/oakland-maui-pepper-spray-disturbing-photo-delay/. [Accessed September 25, 2018].

[42] Benjamin Henne, Christian Kater, Matthew Smith, and Michael Brenner. Selective Cloaking: Need-to-Know for Location-based Apps. In *IEEE Conference on Privacy, Security and Trust*, July 2013. doi: 10.1109/PST.2013.6596032.

[43] Baik Hoh and Marco Gruteser. Preserving Privacy in GPS Traces via Uncertainty-aware Path Cloaking. In *ACM Conference on Computer and Communications Security (CCS)*, October 2007. doi: 10.1145/1315245.1315266.

[44] Byeongdo Hong, Sangwook Bae, and Yongdae Kim. GUTI Reallocation Demystified: Cellular Location Tracking with Changing Temporary Identifier. In *Network and Distributed System Security Symposium (NDSS)*, February 2018. doi: 10.14722/ndss.2018.23349.

[45] Jinseong Jeon, Kristopher K. Micinski, Jeffrey A. Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S. Foster, and Todd Millstein. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications. In *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, October 2012. doi: 10.1145/2381934.2381938.

[46] Ryo Kato, Mayu Iwata, Takahiro Hara, Akiyoshi Suzuki, Xing Xie, Yuki Arase, and Shojiro Nishio. A Dummy-based Anonymization Method Based on User Trajectory with Pauses. In *ACM Conference on Advances in Geographic Information Systems (SIGSPATIAL)*, November 2012. doi: 10.1145/2424321.2424354.

[47] Constantinos Kolias, Lucas Copi, Fengwei Zhang, and Angelos Stavrou. Breaking BLE Beacons For Fun But Mostly Profit. In *ACM European Workshop on Systems Security (EuroSec)*, April 2017. doi: 10.1145/3065913.3065923.

[48] John Krumm. Inference Attacks on Location Tracks. In *International Conference on Pervasive Computing (PERVASIVE)*. Springer, 2007.

[49] B. Krupp, N. Sridhar, and W. Zhao. SPE: Security and Privacy Enhancement Framework for Mobile Devices. *IEEE Transactions on Dependable and Secure Computing*, 14, July 2015. doi: 10.1109/TDSC.2015.2465965.

[50] Denis F. Kune, John Koelndorfer, Nicholas Hopper, and Yongdae Kim. Location Leaks Over the GSM Air Interface. In *Network & Distributed System Security Symposium (NDSS)*, February 2012.

[51] Chi-Yu Li, Guan-Hua Tu, Chunyi Peng, Zengwen Yuan, Yuanjie Li, Songwu Lu, and Xinbing Wang. Insecurity of Voice Solution VoLTE in LTE Mobile Networks. In *ACM Conference on Computer and Communications Security (CCS)*, October 2015. doi: 10.1145/2810103.2813618.

[52] Guolong Lin and Guevara Noubir. On Link Layer Denial of Service in Data Wireless LANs. *Wiley Journal on Wireless Communications and Mobile Computing*, 5, May 2005.

[53] Hua Lu, Christian S. Jensen, and Man Lung Yiu. PAD: Privacy-area Aware, Dummy-based Location Privacy in Mobile Services. In *ACM International Workshop on Data Engineering for Wireless and Mobile Access (MobiDE)*, June 2008. doi: 10.1145/1626536.1626540.

[54] Adriano Di Luzio, Alessandro Mei, and Julinda Stefa. Mind Your Probes: De-anonymization of Large Crowds Through Smartphone WiFi Probe Requests. In *IEEE INFOCOM*, April 2016. doi: 10.1109/INFOCOM.2016.7524459.

[55] Sathiamoorthy Manoharan. On GPS Tracking of Mobile Devices. In *IEEE International Conference on Networking and Services (ICNS)*, April 2009. doi: 10.1109/ICNS.2009.103.

[56] Célestin Matte, Mathieu Cunche, Franck Rousseau, and Mathy Vanhoef. Defeating MAC Address Randomization Through Timing Attacks. In *ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)*, July 2016. doi: 10.1145/2939918.2939930.

[57] Ulrike Meyer and Susanne Wetzel. A Man-in-the-Middle Attack on UMTS. In *ACM Workshop on Wireless Security (WiSe)*, October 2004. doi: 10.1145/1023646.1023662.

[58] Micro:bit Educational Foundation. Micro:bit website, 2018. URL https://microbit.org. [Accessed September 20, 2018].

[59] Arsalan Mosenia, Xiaoliang Dai, Prateek Mittal, and Niraj K. Jha. PinMe: Tracking a Smartphone User Around the World. *IEEE Transactions on Multi-Scale Computing Systems*, 3, 2017. doi: 10.1109/TMSCS.2017.2751462.

[60] A. B. M. Musa and Jakob Eriksson. Tracking Unmodified Smartphones Using Wi-fi Monitors. In *ACM Conference on Embedded Network Sensor Systems (SenSys)*, November 2012. doi: 10.1145/2426656.2426685.

[61] Sashank Narain, Triet D. Vo-Huu, Kenneth Block, and Guevara Noubir. Inferring User Routes and Locations Using Zero-Permission Mobile Sensors. In *IEEE Symposium on Security and Privacy (S&P)*, May 2016. doi: 10.1109/SP.2016.31.

[62] Sarfraz Nawaz and Cecilia Mascolo. Mining Users' Significant Driving Routes with Low-power Sensors. In *ACM Conference on Embedded Network Sensor Systems (SenSys)*, November 2014. doi: 10.1145/2668332.2668348.

[63] Nordic Semiconductor. nRF51822, 2018. URL https://www.nordicsemi.com/eng/Products/Bluetooth-low-energy/nRF51822. [Accessed September 20, 2018].

[64] Simon Oya, Carmela Troncoso, and Fernando Pérez-González. Back to the Drawing Board: Revisiting the Design of Optimal Location Privacy-preserving Mechanisms. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, October 2017. doi: 10.1145/3133956.3134004.

[65] Paulo Borges. BLESSED, 2014. URL https://github.com/pauloborges/blessed. [Accessed September 20, 2018].

[66] PC Engines. APU2 platform, 2018. URL https://www.pcengines.ch/apu2.htm. [Accessed November 14, 2018].

[67] Zhiyun Qian, Zhaoguang Wang, Qiang Xu, Z. Morley Mao, Ming Zhang, and Yi-Min Wang. You Can Run, but You Can't Hide: Exposing Network Location for Targeted DoS Attacks in Cellular Networks. In *Network & Distributed System Security Symposium (NDSS)*, February 2012.

[68] Franziska Roesner. Designing Application Permission Models that Meet User Expectations. *IEEE Security & Privacy*, 15, February 2017. doi: 10.1109/MSP.2017.3.

[69] Piotr Sapiezynski, Arkadiusz Stopczynski, David Kofoed Wind, Jure Leskovec, and Sune Lehmann. Inferring Person-to-Person Proximity Using WiFi Signals. *ACM Interactive, Mobile, Wearable and Ubiquitous Technologies*, 1, June 2017. doi: 10.1145/3090089.

[70] Matthias Schulz, Francesco Gringoli, Daniel Steinmetzer, Michael Koch, and Matthias Hollick. Massive Reactive Smartphone-based Jamming Using Arbitrary Waveforms and Adaptive Power Control. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, July 2017. doi: 10.1145/3098243.3098253.

[71] Matthias Schulz, Daniel Wegemer, and Matthias Hollick. The Nexmon Firmware Analysis and Modification Framework: Empowering Researchers to Enhance Wi-Fi Devices. *Computer Communications*, 2018. doi: 10.1016/j.comcom.2018.05.015.

[72] Reza Shokri, George Theodorakopoulos, Carmela Troncoso, Jean-Pierre Hubaux, and Jean-Yves Le Boudec. Protecting Location Privacy: Optimal Strategy Against Localization Attacks. In *ACM Conference on Computer and Communications Security (CCS)*, October 2012. doi: 10.1145/2382196.2382261.

[73] IEEE Computer Society. Wireless LAN medium access control (MAC) and physical layer (PHY) specification, December 2016.

[74] Adam Stubblefield, John Ioannidis, and Aviel D. Rubin. Using the Fluhrer, Mantin, and Shamir Attack to Break WEP. In *Network and Distributed System Security Symposium (NDSS)*, February 2002.

[75] Milan Stute. Video of Proof-of-Concept Denial-of-Service Attack Crashing iOS Devices, 2018. URL https://youtu.be/M5D9NeKapUo.

[76] Milan Stute. Open Apple Wireless Direct Link Implementation in C, 2019. URL https://seemoo.de/owl.

[77] Milan Stute. Video of Proof-of-Concept Man-in-the-Middle Attack on AirDrop, 2019. URL https://youtu.be/5T7Qatoh0Vo.

[78] Milan Stute and Alexander Heinrich. Open AirDrop Implementation in Python, 2019. URL https://seemoo.de/opendrop.

[79] Milan Stute, David Kreitschmann, and Matthias Hollick. One Billion Apples' Secret Sauce: Recipe for the Apple Wireless Direct Link Ad hoc Protocol. In *ACM Conference on Mobile Computing and Networking (MobiCom)*, October 2018. doi: 10.1145/3241539.3241566.

[80] Milan Stute, David Kreitschmann, and Matthias Hollick. Demo: Linux Goes Apple Picking: Cross-Platform Ad hoc Communication with Apple Wireless Direct Link. In *ACM Conference on Mobile Computing and Networking (MobiCom)*, October 2018. doi: 10.1145/3241539.3267716.

[81] Milan Stute, David Kreitschmann, and Matthias Hollick. The Open Wireless Link Project, 2018. URL https://owlink.org.

[82] Akiyoshi Suzuki, Mayu Iwata, Yuki Arase, Takahiro Hara, Xing Xie, and Shojiro Nishio. A User Location Anonymization Method for Location Based Services in a Real Environment. In *ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS)*, November 2010. doi: 10.1145/1869790.1869846.

[83] Galini Tsoukaneri, George Theodorakopoulos, Hugh Leather, and Mahesh K. Marina. On the Inference of User Paths from Anonymized Mobility Data. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, March 2016. doi: 10.1109/EuroSP.2016.25.

[84] Jayakrishnan Unnikrishnan and Farid Movahedi Naini. De-anonymizing Private Data by Matching Statistics. In *IEEE Allerton Conference on Communication, Control, and Computing (Allerton)*, October 2013. doi: 10.1109/Allerton.2013.6736722.

[85] US Census Bureau. Frequently Occurring Surnames from the 2010 Census. URL https://www.census.gov/topics/population/genealogy/data/2010_surnames.html. [Accessed September 25, 2018].

[86] US Department of Commerce. US Census Bureau. URL https://www.census.gov. [Accessed September 25, 2018].

[87] US Social Security Administration. Popular Baby Names: Beyond the Top 1000 Names. URL https://www.ssa.gov/oact/babynames/index.html. [Accessed September 25, 2018].

[88] Mathy Vanhoef and Frank Piessens. Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2. In *ACM Conference on Computer and Communications Security (CCS)*, October 2017. doi: 10.1145/3133956.3134027.

[89] Mathy Vanhoef, Célestin Matte, Mathieu Cunche, Leonardo S. Cardoso, and Frank Piessens. Why MAC Address Randomization is not Enough: An Analysis of Wi-Fi Network Discovery Mechanisms. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS)*, May 2016. doi: 10.1145/2897845.2897883.

[90] Tien Dang Vo-Huu, Triet Dang Vo-Huu, and Guevara Noubir. Fingerprinting Wi-Fi Devices Using Software Defined Radios. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, July 2016. doi: 10.1145/2939918.2939936.

[91] Triet Dang Vo-Huu, Tien Dang Vo-Huu, and Guevara Noubir. Interleaving Jamming in Wi-Fi Networks. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, July 2016. doi: 10.1145/2939918.2939935.

[92] Yu Wang, Dingbang Xu, Xiao He, Chao Zhang, Fan Li, and Bin Xu. L2P2: Location-aware Location Privacy Protection for Location-based Services. In *IEEE INFOCOM*, March 2012. doi: 10.1109/INFCOM.2012.6195577.

[93] Te-En Wei, Albert B. Jeng, Hahn-Ming Lee, Chih-How Chen, and Chin-Wei Tien. Android Privacy. In *IEEE Conference on Machine Learning and Cybernetics*, July 2012. doi: 10.1109/ICMLC.2012.6359654.

[94] Wi-Fi Alliance. Neighbor Awareness Networking Technical Specification Version 2.0. Technical report, 2017.

[95] Hao Wu, Weiwei Sun, and Baihua Zheng. Is Only One Gps Position Sufficient to Locate You to the Road Network Accurately? In *ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*, 2016. doi: 10.1145/2971648.2971702.

[96] Yunze Zeng, Parth H. Pathak, and Prasant Mohapatra. WiWho: Wifi-based Person Identification in Smart Spaces. In *ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, April 2016. doi: 10.1109/IPSN.2016.7460727.

[97] Jin Zhang, Bo Wei, Wen Hu, and Salil S. Kanhere. WiFi-ID: Human Identification Using WiFi Signal. In *IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS)*, May 2016. doi: 10.1109/DCOSS.2016.30.

[98] Lichen Zhang, Zhipeng Cai, and Xiaoming Wang. FakeMask: A Novel Privacy Preserving Approach for Smartphones. *IEEE Transactions on Network and Service Management*, 13, June 2016. doi: 10.1109/TNSM.2016.2559448.

Hiding in Plain Signal: Physical Signal Overshadowing Attack on LTE

Hojoon Yang, Sangwook Bae, Mincheol Son, Hongil Kim, Song Min Kim, and Yongdae Kim
Korea Advanced Institute of Science and Technology (KAIST)
{omnibusor, hoops, mcson, hongilk, songmin, yongdaek}@kaist.ac.kr

Abstract

Long-Term Evolution (LTE) communication is based on an open medium; thus, a legitimate signal can potentially be counterfeited by a malicious signal. Although most LTE signaling messages are protected from modification using cryptographic primitives, broadcast messages in LTE have never been integrity protected. In this paper, for the first time, we present a signal injection attack that exploits the fundamental weaknesses of broadcast messages in LTE and modifies a transmitted signal over the air. This attack, which is referred to as signal overshadowing (named `SigOver`) has several advantages and differences when compared with existing attacks using a fake base station. For example, with a 3 dB power difference from a legitimate signal, the `SigOver` attack demonstrated a 98% success rate when compared with the 80% success rate of attacks achieved using a fake base station, even with a 35 dB power difference. Given that the `SigOver` attack is a novel primitive attack, it yields five new attack scenarios and implications. Finally, a discussion on two potential countermeasures leaves practical and robust defense mechanism as a future work.

1 Introduction

Long-Term Evolution (LTE) technology utilizes broadcast signals to transmit essential information from a cellular network to user devices. At minimum, the information broadcasted by an LTE base station, which is referred to as an evolved NodeB (eNB), includes the synchronization information and radio resource configurations required for a User Equipment (UE) to access the cellular network. Based on the received broadcast signals, a UE registers with the network by performing an Authentication and Key Agreement (AKA) procedure. After registration, the UE monitors the broadcast signals for various objectives. For example, when the UE does not have a connection with an eNB due to its inactivity, it needs to listen to paging messages regularly to check the messages transmitted to it.Even when the UE has an active connection with an eNB, the UE keeps listening broadcast signals to determine potential changes in system-wide radio configurations which are required to be updated, and to identify the arrival of messages intended to multiple UEs.

Despite its various practical applications, the broadcast signal is not security-protected at all. In LTE, communication between a UE and network is secured only after successful authentication and security handshake procedures, namely Non-Access Stratum (NAS) and Access Stratum (AS) security mode procedures for the protection of *unicast* messages. Unprotected broadcast signals may be unavoidable to a certain extent in wireless communication; however, they subject the system and UEs to various vulnerabilities that can be exploited.

Previous studies [21, 26, 36, 39, 40] reported on several attacks that exploit *unprotected* broadcast signals. In general, such attacks employ a fake base station (FBS) that attracts UEs to be connected to itself by transmitting a signal stronger than those of the legitimate base stations. The attacks mainly exploit the paging messages, resulting in undesirable effects on the UE, e.g., out-of-service and battery drains. Notably, such FBS-based attacks entail noticeable characteristics (e.g., high signal power) and/or outcomes (e.g., service denial) that enable the victim UEs to identify the presence of the FBS (see Section 3.5 for details).

In this paper, we propose a new approach referred to as the `SigOver` attack, which injects a manipulated broadcast signal into UEs without employing an FBS. The `SigOver` attack overwrites a portion of the legitimate signal using the manipulated attack signal. The `SigOver` attack is based on the fact that the UE decodes a stronger signal when it concurrently receives multiple overlapping signals, which is referred to as the *capture effect* [51]. The main technical component of the attack is to synchronize the timing of the attack signal with that of the targeted legitimate signal so that the UE only decodes the attack signal (see Section 3). This attack is both stealthy and far-reaching. It is stealthy because the attack signal, which is transmitted at a significantly low power level, only overshadows the targeted signal; whereas the other signals/messages between the victim UEs and network remain intact. It is far-

reaching because the attack signal can simultaneously affect a large number of nearby UEs with low signaling and a low computational cost. Note that the SigOver attack does not require any active communication with the UEs, and it does not relay messages between UEs and an eNB.

The *SigOver attack* is the first practical realization of *the signal overshadowing attack on the LTE broadcast signals* using a low-cost Software Defined Radio (SDR) platform and open source LTE library [43]. The SigOver attack was made practical by addressing the following challenge: *time and frequency synchronization*. To overshadow the legitimate signal using the malicious signal, the SigOver attack needs to be tightly time-synchronized with the eNB's downlink physical channel to which the victim UE is listening. To achieve time synchronization, we leverage the synchronization signals of the eNB that are transmitted periodically with a fixed time gap. For accurate frequency synchronization, we employ a Global Positioning System (GPS) disciplined oscillator.

The feasibility of the SigOver attack was verified by testing it against 10 smartphones (listed in Section 5) connected to an operational network[1]. For the experiments, we introduced five new attack scenarios, which included the signaling storm, denial of service (DoS) against UEs, network downgrade, and UE location tracking (Section 5). The experimental results reveal that the SigOver attack overshadows the target signal and causes the victim device to decode it with a 98% success rate and a power difference of only 3 dB from a legitimate signal. On the other hand, attacks utilizing an FBS have only 80% success rate even with a 35 dB power difference. This implies that the SigOver attack is significantly more efficient than the attacks using the FBS.

Finally, two potential countermeasures against the SigOver attack are discussed in Section 6: (1) digital signature based solution and (2) channel estimation based detection. Moreover, a practical and robust solution to the SigOver attack is left as a future work.

Our contribution are summarized as follows:

- **First signal overshadowing attack on LTE:** To the best of our knowledge, the SigOver attack is the first realization of a signal overshadowing attack on LTE broadcast signals.
- **Implementation and evaluation:** We demonstrate the practicality and stealthiness of the SigOver attack via extensive real world experiments with high attack success rate.
- **Novel attack scenarios and implications:** We present novel attack scenarios and analyze their implications in detail based on the experiments.
- **Countermeasures:** We investigate prevention and detection strategies against the SigOver attack, e.g., the digitally signing on broadcast signals for prevention, and leveraging the changing nature of the physical signal for detection.

[1] All the experiments were conducted based on the permission of the operators.

2 Background

In this section, we present a brief description of the LTE network architecture and the essential procedures of radio connection establishment, mobility management, and security setup between a device and an LTE network. (See the table in Appendix B for the acronyms used in this paper.)

2.1 LTE Network Architecture

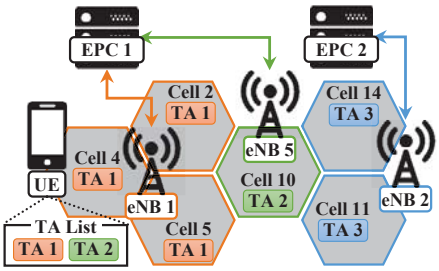


Figure 1: LTE network architecture

An LTE network consists of a UE, eNB, and Evolved Packet Core (EPC) components as illustrated in Figure 1.

A **UE** is an end device that provides various LTE services (i.e., voice and data services) to a subscribed user. It includes a smart card referred to as the Universal Subscriber Identity Module (USIM), which stores a permanent identity (International Mobile Subscriber Identity, IMSI) or a temporary identity (Globally Unique Temporary Identity, GUTI) for user identification, and a cryptographic key for encryption and integrity protection.

An **eNB** is an LTE base station, which provides a wireless connections for UEs to receive services enabled at the LTE network. A single eNB covers multiple sites (referred to as cells in LTE), which are identified by a Physical layer Cell Identity (PCI).

An **EPC network** is responsible for control functions such as authentication, mobility and session management, and user plane services. For mobility management, a Mobility Management Entity (MME) in the EPC network manages a set of Tracking Areas (TAs), each of which contains several eNBs.

2.2 LTE Physical Layer Initial Access

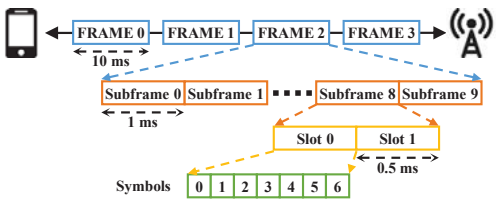


Figure 2: LTE frame structure type 1 [2]

LTE frame. The UE and eNB communicate with each other based on the radio frame structure, as shown in Figure 2[2].

[2] The LTE-Frequency Division Duplex (FDD) mode was employed in this study, as used by the majority of operators in the world [18].

Each frame has a duration of 10ms and comprises 10 subframes, each of which has a duration of 1ms. A single subframe is further divided into two slots of equal duration and each slot comprises seven Orthogonal Frequency Division Multiplexing (OFDM) symbols.

Downlink Scheduling. In LTE, radio resources are allocated in the unit of the Physical Resource Block (PRB) [2] that contains 12 subcarriers (each with a bandwidth of 15 KHz) and consumes one slot in time (0.5ms). The number of available PRBs in a frequency band is determined by the system bandwidth. Depending on the size of the data, an eNB allocates PRBs within a subframe (1ms), which is the smallest scheduling time interval.

Channel estimation. When a signal travels through a wireless channel, the signal gets distorted due to several factors, e.g., attenuation, phase-shift, and noise. To accommodate those factors, wireless devices estimate the channel using the following equation: $Y(k) = H(k)X(k)$, where $Y(k)$, $H(k)$ and $X(k)$ represent a signal received by a UE, the channel coefficient, and the signal transmitted by an eNB, respectively. In LTE, a UE performs channel estimation based on the Reference Signal (RS) transmitted by the eNB. The UE calculates $H(k)$ from $H(k) = \frac{Y(k)}{X(k)}$ as it already knows $X(k)$ and $Y(k)$ value of RS. To minimize the effects of noise in the channel estimation, $H(k)$ of RS is averaged using an averaging window.

Cell search. When a UE is turned on, it has to find a suitable cell to establish radio connections. To this end, it first attempts to measure the Received Signal Strength Indication (RSSI) of the candidate frequency channels. The UE selects the channel with the highest RSSI based on the measurement. Thereafter, the UE obtains time synchronization on a subframe basis and the PCI of the cell by listening to a Primary Synchronization Signal (PSS) and a Secondary Synchronization Signal (SSS). The UE then decodes the Master Information Block (MIB) to acquire the System Frame Number (SFN) and other physical channels.

System information acquisition. After completing the cell search procedure, the UE decodes a Physical Control Format Indicator CHannel (PCFICH) and a Physical Downlink Control CHannel (PDCCH) to decode downlink data. The UE knows the number of OFDM symbols used to carry the PDCCH at each subframe through the PCFICH. The UE then decodes the PDCCH that contain the information on the resource blocks that the data and the demodulation scheme required by the UE. After decoding the two channels, the UE decodes the other system information broadcasted through a Physical Downlink Shared CHannel (PDSCH). There are 22 System Information Blocks (SIBs), each of which contains different cell-related system information [3]. Among them, SIB1 and SIB2 are essential for a UE to connect to a cell. The availability of other SIBs is indicated in SIB1.

Random access. A UE performs a Random Access CHannel (RACH) procedure to establish a radio connection with the eNB. To this end, the UE randomly chooses a Random Access (RA) preamble sequence and transmits it to the eNB. Unless the same preamble sequence is simultaneously transmitted from a different UE, the UE successfully completes the RA procedure.

2.3 Mobility Management

Radio Resource Control (RRC). When all the steps above have been completed, the UE carries out a connection establishment procedure with the eNB (called RRC connection establishment procedure). Upon the completion of the procedure, the UE enters the *RRC Connected* state in which it can communicate with the eNB. When there are no incoming and outgoing data for a certain time period, the radio connection between the UE and eNB is released, and the UE enters the *RRC Idle* state, to reduce battery consumption.

Non-Access Stratum (NAS). NAS is a network layer protocol between the UE and MME for mobility and session management. To register with the LTE network, the UE carries out an ATTACH procedure. After the UE is successfully registered with the LTE network, the MME knows the TA to which the UE belongs and provides the UE of a list of TA identifiers (TAIs). This TAI list is used by the UE to report its location to the MME.

Idle state behavior. In the *RRC Idle* state, the UE periodically wakes up to read paging messages and SIB 1. When there is incoming message to the UE, the MME that tracks the UE sends a paging to all eNBs in the entire TAs assigned to the UE, and those eNBs broadcast a paging message to inform the UE of the arrival message. The paging message contains either the temporary or permanent identity of the UE. If the UE receives the paging message, it sends a `RRC connection request` and a `Service request` message to the LTE network. Paging is also used to notify the system information change or provide emergency alerts such as the Earthquake and Tsunami Warning System (ETWS) and Commercial Mobile Alert System (CMAS). The UE also reads the SIB1 to identify the current TA. If the UE enters into a new TA that is not in the TAI list, the UE sends a `Tracking Area Update (TAU) request` to the MME to report its location. In addition, the UE periodically measures the power and quality of the serving cell and neighboring cells by calculating the Reference Signal Received Power (RSRP) and Reference Signal Received Quality (RSRQ). When the RSRP of a neighboring cell is higher than that of the serving cell by a certain threshold, the UE selects new cell and camps on it (i.e., cell re-selection).

2.4 Establishing Security Context

When a UE establishes a wireless connection with an eNB, it registers with the LTE network to achieve a full connection with the network (this behavior is called ATTACH) by providing its permanent identity, IMSI. Then, the MME and the UE mutually authenticate each other and carry out a key

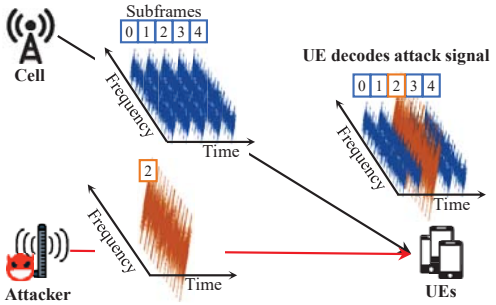


Figure 3: Overshadowing attack at a glimpse: By exploiting the fixed transmission timings of LTE subframes, the attacker injects a crafted subframe (in brown) that precisely *overshadows* the legitimate subframe (in blue) without errors.

agreement procedure to create a *security context* (i.e., NAS security context) for encryption and integrity protection. After the Authentication and Key Agreement (AKA) procedure, most messages between the UE and the MME are encrypted and integrity protected with cryptographic primitives. On the other hand, all initial procedures before establishing a security context in the AKA procedure are not encrypted and integrity protected by design. Those unprotected messages include paging, SIBs and several network layer initial messages specified in the LTE standard [5].

3 Overshadowing LTE Broadcast Message

In this section, we present the attack model, followed by a description of the `SigOver` attack. The `SigOver` attack is demonstrated by using an SDR that is widely used today (i.e., Universal Software Radio Peripheral (USRP) [16]). Lastly, we compare the `SigOver` attack with typical FBS attacks to show the effectiveness of the former.

3.1 Attack Model

We assume an active adversary with minimum privilege. The proposed attack model can be described as follows: (i) The adversary does not know the LTE key of the victim UE. (ii) The adversary is able to eavesdrop on the downlink broadcast messages transmitted from the legitimate LTE cell to the victim UE(s). However, as the victim key is unavailable, the encrypted messages cannot be decrypted. Note that (ii) is trivially achievable because messages are transmitted through the open medium. Under the above assumptions, we show that an active adversary can inject malicious messages into the victim UE(s) by overwriting the legitimate messages. This is achieved by carefully crafting a message that overlaps a legitimate message with respect to time and frequency. In Section 3.5, we discuss the fundamental differences between the proposed attack model and typical FBS attacks [21, 22, 36, 37, 39].

3.2 `SigOver` Attack Overview

This section briefly outlines the design of the `SigOver` attack. As discussed in Section 2, the LTE downlink is scheduled in a subframe granularity with a duration of 1ms. Each subframe is encoded separately by the base station, and is therefore decoded accordingly by the UE. Under this frame structure, Figure 3 conceptually illustrates the `SigOver` attack, where the attacker injects a crafted subframe (brown color) that precisely *overshadows* the legitimate subframe (blue color). Since the subframes are decoded independently from one another, the legitimate (non-overshadowed) subframes are generally not affected. At the same time, the injected subframe is crafted such that the UEs that have received and decoded the subframe behave based on the included information, which typically yields an abnormal or malicious behavior - an intended behavior by the attacker. The inherent vulnerability of LTE broadcast messages enables an attacker to launch various types of attacks using legitimately-looking messages (i.e., insidiously).

In principle, the `SigOver` attack leverages the capture effect [51], wherein the stronger signal is decoded when multiple simultaneous wireless signals (i.e., legitimate and crafted subframes) collide in the air. This is true for signals with a slight power difference of 3 dB [29]. Two technical challenges to launch the `SigOver` attack are (i) carefully crafting the overshadowing message to be decoded by the victim UEs (Section 3.3), and (ii) the stringent requirement of the transmission timing and frequency for precise overshadowing (Section 3.4).

3.3 Crafting a Malicious Subframe

Here we illustrate how to craft a subframe that can be successfully decoded at the victim UE for a successful attack.

Communication configuration matching. For the `SigOver` attack, the attacker must first identify the physical configuration of the legitimate cell on which the victim UEs are camping, to determine the structure of the attack subframe. The necessary physical configuration information for valid subframe construction includes the PCI, channel bandwidth, PHICH configuration, and transmission scheme (or the number of antenna ports); all of which are available to the attacker once the attacker camps on the same legitimate cell. In particular, PCI is calculated from the PSS/SSS, and the remaining information is obtained from the MIB. Furthermore, the attacker must synchronize with the SFN of the legitimate cell, which is also available in the MIB, to determine the injection time of the attack subframe.

Subframe structuring and injection. In LTE, when a UE reads a broadcast message, it decodes the following information from a subframe: i) a Control Format Indicator (CFI) that contains the control channel structure, ii) Downlink Control Information (DCI) that contains the allocated resource (i.e., resource blocks) for the message, and iii) the resource blocks (RBs) that contain the message itself. The CFI and DCI are

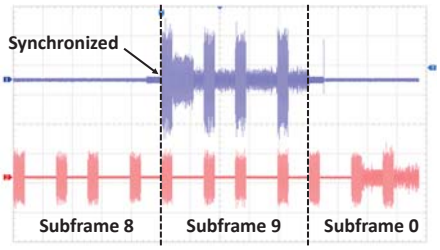


Figure 4: Oscilloscope snapshot showing precise time synchronization between a legitimate (in red) and a crafted signal (in blue).

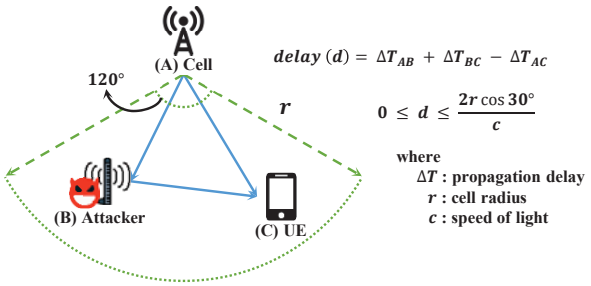


Figure 5: Propagation delay in the 3-sector cell configuration according to the location of the victim UE and the attacker. The attacker and victim UE are assumed to be within a cell coverage (the green sector form)

transmitted over the PCFICH and PDCCH respectively; and the message is transmitted over the PDSCH. Therefore, to inject a subframe, the attacker needs to craft a subframe that contains the PCFICH, PDSCH and PDSCH. However, the injected subframe containing those values may not be decoded correctly at the UE due to a channel estimation error. Note that the UE estimates the channel from the RS transmitted by the legitimate eNB, yet the estimation result may be inappropriate to decode the injected subframe correctly. To address this issue, the RS is included in the subframe for the `SigOver` attack, which significantly increases the robustness of the `SigOver` attack.

The last technical challenge related to the decoding of the crafted subframe is with respect to wireless channel estimation and equalization, for recovery from the signal distortion due to the channel. In the `SigOver` attack, the channel is estimated either dominantly (even solely depending on the paging occasion) from the crafted subframe (*RRC Idle*), or it is averaged from consecutive subframes (*RRC Connected*) along with multiple legitimate subframes. In the former, a single injection is sufficient for the attack (i.e., decoding of the crafted subframe). In the latter, repeated injections are needed to effectively reflect the wireless channel between the attacker and the victim UE. According to our measurement (Section 4) which injected one subframe for every SFN, `SigOver` attack reaches over 98% success rate in less than a second while maintaining reliable communication for legitimate subframes. In Appendix A, we present empirical results showing that legitimate communication is minimally affected by `SigOver` attack using several services including web browsing and streaming.

3.4 Accurate Overshadowing

Overshadowing requires the crafted subframe to overlap the legitimate signal precisely in both the time and frequency domains. This subsection discusses how this is achieved.

Time synchronization. To precisely overshadow legitimate subframes, an attacker needs to know the subframe timing (to determine when a subframe starts) and SFN (to determine when to inject the subframe with respect to the frame number) from the legitimate cell. The attacker obtains the subframe timing from the synchronization signals (i.e., PSS/SSS) and the SFN from the MIB of the legitimate cell. The attacker continuously obtains the subframe timing and the updated SFN, as the values vary over time depending on the channel condition. With the knowledge of the subframe timing and the SFN, the `SigOver` attack precisely synchronizes the transmission time of the crafted subframe with that of the target broadcast message (see Figure 4).

As shown in Figure 5, however, the crafted subframe transmitted at the acquired subframe timing may still arrive at the UE with a slight timing offset (with reference to the legitimate subframe) due to the propagation delay. Although the delay (d) is unavoidable (as the propagation delay is immeasurable by the attacker), its impact is minimal. This is because the baseband processor in the UE is designed to compensate the delay due to mobility and environmental effects [48]. Since the maximum delay that can be compensated is dependent on the baseband processor of the UE, we perform the following experiments to measure the delay. We assumed the typical three-sector cell configuration wherein the transmission angle of the cell is 120° [10]. The delay (d) is maximized when the attacker and the victim UE are located at both ends of the arc. This translates to $d = 8.66\mu$s under a typical cell radius of approximately 1.5km in urban environments. We measured the offset tolerance on two devices with different basebands (Qualcomm and Exynos), and the tolerance was larger than the maximum delay (i.e., 8.66μs) (see Section 4 for detailed experimental results).

Frequency synchronization. The operating frequency of a radio device is determined by the oscillator, where it inevitably suffers from a device-specific offset that is randomly imposed during manufacturing and generated during operation due to environmental effects (e.g., temperature). Such an imperfection in the oscillator is reflected in the radio signal as carrier frequency offset. In LTE, there are a number of readily available techniques [27, 50] to compensate for offsets up to a certain level (e.g., Up to ±7.5KHz for PSS based compensation in the LTE 15KHz subcarrier spacing [38]). Therefore, for the reliable implementation of the `SigOver` attack, the offset should be maintained below that level in the UE, at all times.

The LTE standard defines the minimum frequency accu-

Table 1: Comparison of the SigOver, FBS and MitM Attacker

	Stealthiness	Power efficiency	Attack sustainability
FBS	Low	Low	Low
MitM	Limited*	Low	Limited*
SigOver	High	High	High

* "Limited" means that the attack works in an limited environment

racy of a base station of ±50 ppb [1] for macro base stations. To satisfy the requirement, eNBs are equipped with highly accurate oscillators and further augmentation techniques such as a precision time protocol and GPS. In contrast, the SigOver attack was run on a typical, inexpensive SDR with an inaccurate oscillator (±2500 ppb for the USRP X310 [16]). A GPS disciplined oscillator (GPSDO) was employed, namely, an oven-controlled crystal oscillator (OCXO) to reduce the frequency offset to an appropriate level. The GPSDO provides a sufficient accuracy of ±25 ppb [14, 32] and is highly stable (±1 ppb when the GPS locked). This indicates a frequency offset of up to ±270Hz (at 3.6GHz with an offset of 75 ppb), within the LTE FDD frequency range of 460MHz-3.6GHz [6]. Our experiment confirms that all 10 devices (listed in Section 5) can compensate such small frequency offset to enable a reliable SigOver attack.

3.5 Comparison of SigOver, FBS and MitM Attacks

Although the FBS and Man-in-the-Middle (MitM) attacks can be used for broadcast message manipulation in LTE, only the former has appeared in the literature. In this section, we extensively analyze the FBS and MitM attacks in comparison with the SigOver attack with respect to stealthiness, power efficiency, and sustainability (see Table 1).

3.5.1 Attacks using FBS

The FBS attack is one of the most commonly used attacks against cellular networks [21, 22, 26, 36, 39, 40]. In an FBS attack, the attacker (i.e., the FBS) attracts victim UEs to camp on itself by transmitting a stronger signal than legitimate cells. The attacker then injects unprotected yet legitimate-looking messages to the victim UE. The FBS attack has the following limitations when compared with the SigOver attack.

Power Efficiency. In general, a UE selects the cell that transmits signals with the highest power. However, it has not been well investigated how much stronger signal the FBS should transmit than a legitimate base station to attract nearby victim UEs. This is an important question to be answered for the attacker, as higher power increases the chance of attracting the UE at a greater risk of being detected through power measurements (e.g., RSRP, RSRQ). According to the experimental measurements, the FBS attack reaches 100% success at 40 dB, whereas the SigOver attack was 98% successful at 3 dB (see Table 2). In particular, the FBS requires a power consumption greater than that of the SigOver attack by a factor of 5000, to achieve a comparable attack success rate.



(a) FBS attack (b) SigOver attack

Figure 6: Signaling messages during FBS and SigOver attack

Stealthiness. In general, an FBS is configured to masquerade as a legitimate base station as presented in previous studies [21, 22, 39, 40]. For example, the FBS broadcasts the same MIB and SIB1/2 messages as those of the legitimate cell and may use the same PCI to make itself indistinguishable from the legitimate ones. Nevertheless, the FBS inevitably carries several unique and clear signatures for detection [25, 30, 33, 49, 53]. First, as discussed earlier, an FBS attack uses ×10,000 stronger power (cf. ×2 for SigOver attack) than a legitimate cell, which is a clear indicator of the FBS. Second, when an FBS attracts a victim UE camping on a legitimate cell, the victim must undergo a cell re-selection process, where the victim UE reads the MIB and SIB1/2 messages from the FBS (Figure 6a). Third, the operation of the FBS may be very different from the legitimate ones due to its limited physical capabilities compared to the real base station[3]. Such operational characteristics include a relatively low paging rate, in addition to different RF properties such as a large frequency offset due to the low cost hardware. Finally, the FBS cannot establish a secure connection with the UE or transport protected NAS messages between the UE and the network (i.e., MME), which results in a denial of service to the UE. Hence, it is highly probable that a UE is able to detect an FBS. On the other hand, the mechanism of the SigOver attack is to precisely overshadow a specific broadcast message without interfering the synchronization between the victim UE and current cell. Therefore, as shown in Figure 6b, the UE does not carry out cell re-selection or reconfigure any cell-specific parameters. The UE subject to the SigOver attack maintains secure signaling connections with the legitimate eNB and MME.

Sustainability. If a victim UE camps on an FBS, it cannot receive the service via the FBS. This can be used by the UE as a potential FBS detection mechanism as mentioned above. To avoid such detection, the FBS may take the following strategy: it injects a malicious message to the UE, and releases a connection (e.g., by causing a radio link failure or triggering cell re-selection at the UE) so that the victim UE returns to a legitimate cell. Under this scenario, however, the injected

[3] A state-sponsored attack with unlimited resource and capabilities was not considered.

message should be selected such that the attack sustains even if the UE makes a cell change (e.g., `TAU Reject` [39]) or has an immediate impact on the UE (e.g., emergency warning message [21]). Thus, it is not an appropriate attack vector to exploit broadcast messages (e.g., SIB messages) that are refreshed when the serving cell changes. This makes the FBS attack either limited in terms of attack scope (as exploitable messages are very limited) or less sustainable in its duration.

3.5.2 MitM attacks

Recently, a new type of FBS attack referred to as the aLTEr [37] attack was discovered. This is an MitM attack that employs an FBS with eNB and UE capabilities. The eNB component of the FBS impersonates a legitimate eNB by relaying the messages from the eNB to a victim UE. In addition, the UE component of the FBS impersonates the victim UE by relaying the messages from the UE to the eNB. By sitting between the victim UE and the eNB, the MitM attacker manipulates user plane messages since the messages are not integrity-protected in LTE. The MitM attack inherits two aforementioned limitations of the FBS attack, namely, a high power consumption and low stealthiness, since the MitM attacker should attract the victim UEs in the same manner. Meanwhile, in principle, the MitM attack does not affect the connection between the victim UE and the eNB, thereby making the attack sustainable. However, we noticed that it is non-trivial to implement an MitM attacker for various reasons. First, to maintain the connection with a victim UE, the MitM attacker should relay all uplink and downlink messages exchanged between the victim UE and the eNB. To this end, the attacker must know the UE's radio resource settings configured by the eNB and configure the radio resource for the UE accordingly. Otherwise, the radio connection between the UE and the eNB may become unstable or fail. However, since the message that contains the radio resource setting (i.e., `RRC reconfiguration`) is encrypted, the attacker cannot properly configure the UE's radio resource. We note that the `RRC reconfiguration` contains a large number of PHY, MAC, RLC, and PDCP configurations for the UE.

To address this issue, the aLTEr attack used the radio configuration in a heuristic manner under the following conditions: (i) a victim UE receives the service using the *default radio configuration*, and (ii) the default radio configuration of an operator is stable. That is, only a few parameters (e.g., scheduling request (SR) and channel quality indicator (CQI) configuration) are changed for each radio configuration; whereas the others are the same. Thus, the attacker only needs to guess the CQI and SR configurations. However, in the real world, the eNB frequently changes the UE's radio configuration depending on the service that the UE is using and/or the current channel condition (e.g., initiating carrier aggregation, starting a Voice/Video call service, service priority, or channel quality change due to mobility). We observed that when a UE watched a YouTube video for 2 minutes under a bad channel condition, it received 9 `RRC reconfiguration` messages from the eNB, where the length of each message varied from 18 bytes to 109 bytes. Note that, as the attacker is only able to know the message length and the sequence of message delivery, it may not correctly guess the configuration. We also observed that 8 out of 9 messages have different CQI configurations which also need to be guessed.

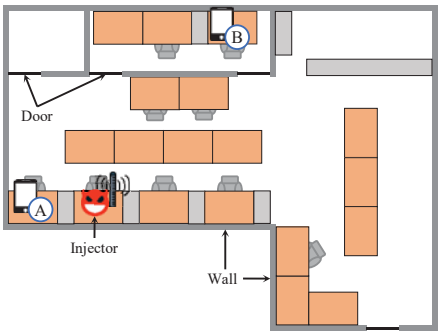


Figure 7: Experiments are conducted at two UE locations, A and B: A is 2m away from the attacker with line of sight. B is 10m away from the attacker, separated by a wall (i.e., non-line of sight). We refer to the former and the latter as LOS and NLOS, respectively.

These limitations apply to all MitM attacks, even when the attacker attempts to manipulate the broadcast message. However, the `SigOver` attack does not suffer from such limitations, as it only utilizes a persistent radio configuration acquired from the MIB of the legitimate cell (see Section 3.3).

4 Real World Experiment

In this section, we perform `SigOver` attack in the wild, and analyze the reliability of the attack.

4.1 Experimental Setup

We implement the `SigOver` attack based on the *pdsch_enodeb*, which contains a basic transmission function as part of srsLTE [43]. We add a custom-built receive function for time synchronization with the legitimate cell. The subframes were crafted using the srsLTE library. Moreover, an USRP X310 [16] equipped with a UBX [15] daughter board and GPSDO [14] was employed, which was connected to an Intel Core i5-3570 machine with an Ubuntu 14.04. To overshadow the signal from a legitimate eNB, the USRP was augmented with ZVE-2W-272 amplifier [28], if needed. Victim UEs are commercial smartphones that camp on a legitimate LTE cell with a 20MHz bandwidth. In addition, the diagnostic monitor tools (e.g., SCAT and XCAL [8, 42]) were used for the analysis of the transmitted and received messages at the UE.

Figure 7 illustrates the two locations within a university office, where two sets of experiments were conducted, as follows: (LOS) The victim UE and the attacker were in the same room, separated by a distance of 2m. (NLOS) The victim UE and the attacker were in different rooms separated by a

wall and distance of 10m. These two environments were used for experiments throughout the study.

Implementation details. An attacker acquires the information of the target benign cell (PCI, MIB) using *pdsch_ue* or diagnostic tools [8, 42]. She acquires time synchronization with the target cell (mimicking the procedure for a benign UE to camp on a cell by getting the PSS/SSS and MIB). After she obtains the arrival timing and SFN information of the LTE frame transmitted by the benign cell, she transmits the malicious message to the target SFN. Thereafter, she continuously receives the PSS/SSS (every 5ms) and MIB (every 10ms) transmitted by the benign cell and updates the synchronization information. Self-interference may cause synchronization problems, because Rx and Tx are in the same frequency. However, due to the precise overshadowing, the `SigOver` attack can minimize the effects on the legitimate PSS/SSS and MIB (there was no case of losing synchronization due to the self-interference).

As a minor issue, the USRP X310 generated an unintended high peak signal at the beginning and end of the signal when carrying out a burst transmission which `SigOver` attack does. This is due to the state change of the front-end components of the SDR. When there was no transmission, it was in the idle state. When transmission occurred, the transition to the transmitting state caused unwanted noise. We resolve this problem by simply padding zero to the front and back ends of the signal to separate the unwanted noise from the original signal, and by compensating the delay due to the zero padding during transmission.

Ethical considerations. As the attacker, we use a downward-facing dome-shaped antenna to minimize upward interference. In addition, we perform the experiments on the first basement level, which is the lowermost floor of the building. The basement floor was restricted during the experiments to prevent normal users from receiving the crafted signal. The experimental results with respect to the impact of the crafted signal revealed that the users upstairs and outside the building normally communicate with the legitimate base station without being affected by the signal. The signaling storm attack explained in Section 5.1.1 was run in a carrier's shielded testbed network, since the attack may cause a DoS on an operational network.

4.2 Practicality

In this section, we evaluate the practicality and robustness of the `SigOver` attack in the LOS/NLOS environment. We use an LG G7 ThinQ smartphone with SnapDragon845, which is the latest Qualcomm LTE chipset. We inject a paging message with the S-TMSI[4] intentionally set as an invalid value of `0xAAAAAAAA`, to differentiate the injected subframe from the legitimate subframes.

[4] S-TMSI is the shortened form of GUTI.

Table 2: Success rate of `SigOver` and FBS* attack

Relative Power (dB)	1	3	5	7	9
`SigOver`	38%	98%	100%	100%	98%
Relative Power (dB)	25	30	35	40	45
FBS attack	0%	0%	80%	100%	100%

* The FBS sets the same freq. band, PCI, MIB and SIB1 to the legitimate cell. If the victim UE camped on the FBS within 10s after it operates, the attack was considered a success. The FBS experiment was run 10 times for each power level. The `SigOver` experiment was performed with 100 paging messages for each power level.

Table 3: Success rate of `SigOver` attack in various conditions.

	LOS	NLOS
RRC Connected	97%	98%
RRC Idle	100%	98%

Power cost. The `SigOver` attack exploits the capture effect, where it injects a stronger signal to overshadow the legitimate signal, which is at a lower power level. Moreover, we inject 100 paging messages into a victim UE in the *RRC Idle* state, and measure the success rate of the attack depending on the relative power between the injected and legitimate signals in the LOS environment. Table 2 shows that the `SigOver` attack achieves the success rate of 98% at 3 dB.

Attack robustness. Table 3 summarizes the success rates of the `SigOver` attack for different combinations of experimental settings (LOS/NLOS) and RRC states (Idle/Connected). Each measurement was an average of 120 injected paging messages. In the *RRC Idle* state, we inject a paging message at the exact paging occasion (e.g., Subframe 9) and paging frame (e.g., SFN%256 = 144) of the victim UE. As discussed in Section 3.3, in the *RRC Idle* state, the channel estimation is carried out solely on the injected signal; whereas in the *RRC Connected* state, the average of the channel estimated from a set of the injected and legitimate signals is considered. In other words, in the *RRC Idle* state, injected signals are individually decoded without the impact of the legitimate signals; thus successful attacks (i.e., correct decoding) can be achieved with a single injection. However, in the *RRC Connected* state, repeated injection is required to overcome the influence of the legitimate signals. To achieve this, we inject a paging message at the exact paging occasion/frame of the victim UE. Simultaneously, we also inject a subframe with RS at every SFN, to reflect the channel of the injected signal and enable a successful attack. As shown in Table 3, the `SigOver` attack maintained a success rate greater than 97% in different RRC states and the LOS and NLOS setups, thus validating the robustness of the `SigOver` attack with respect to operating modes and environmental factors (e.g., multi-path). Finally, during the experiments the victim UE neither reported any radio link failures nor initiated radio connection re-establishment (i.e., `RRC Reestablishment request`). This implies that the `SigOver` attack is non-disruptive to the victim UE and its service. Furthermore, we verify that the `SigOver` attack maintains 100% success rate for over 100 SIB 1 and SIB 2 messages in the *RRC Idle* state and LOS setup.

Table 4: Time tolerance of two smartphones.

Time (μs)	LG G7 (Qualcomm)	Galaxy S9 (Exynos)
Min.	-2.93	-2.60
Max.	9.77	8.46
Max. tolerance*	12.7	11.06

* Note that the `SigOver` attack succeeds if $d <$ Max. tolerance, regardless of the cell radius; where d is defined in Section 3.4

Attack coverage. As described in Section 3.4, a crafted subframe may arrive at victim UE with a slight timing offset due to the propagation delay of the injected signal from the attacker to the victim UE. The decoding of the crafted subframe requires the offset to be bounded within the tolerance range of the UE LTE chipset. Hence, the largest tolerable offset determines the maximum propagation delay; or equivalently, the maximum distance between the attacker and the UE (i.e., the attack coverage). The attack coverage was experimentally evaluated, wherein the propagation delay between the attacker and the UE was emulated by time-shifting the transmission timings of the crafted subframes. We gradually changed the shift in the unit of 10 samples (=0.33μs at 30.72Msps), until the crafted subframes were not decoded; which indicates the maximum delay tolerance. Table 4 presents the tolerance measured from two smartphones with different basebands – LG G7 (Qualcomm), and Galaxy S9 (Exynos). The tolerance offset was consistently higher than 8.66μs across all the devices. With reference to the tolerance-distance relationship discussed in Section 3.4, the results indicate that the `SigOver` attack can cover the entire urban cell (typical radius of 1.5 km) at all times, irrespective of the relative positions of the UE and attacker.

5 Attack Scenarios and Implications

This section presents several attack scenarios using the `SigOver` attack, in addition to their practical implications. The `SigOver` attack can be used to exploit two broadcast messages; SIB and paging. All the attacks were run in the LOS setup presented in Section 4, with the exception of the signaling storm attack. To validate the proposed attacks on the various baseband chipset types, ten LTE capable smartphones were employed: one Intel (iPhone XS), six Qualcomm (Galaxy S4/S8/S9, LG G2/G6/G7), and three Exynos (Galaxy S6/S8/S9) chipset equipped smartphones.

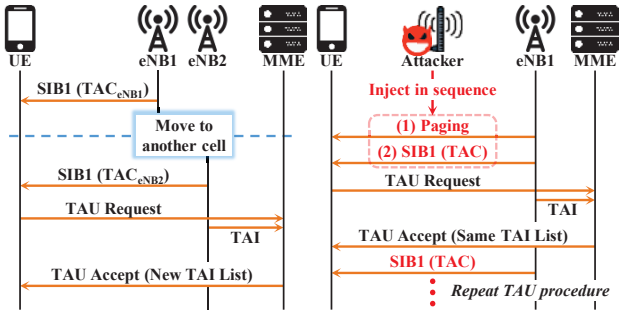
5.1 Attacks Exploiting SIB

In this section, a discussion on two types of attacks via SIB injection, namely, signaling storm and selective DoS, is presented.

5.1.1 Signaling Storm

Attack mechanism. When a UE moves to a new cell, the UE retrieves the Tracking Area Code (TAC[5]) contained in the SIB1 from the new cell and validates it using the TAI list in the

[5]TAC is the shortened form of TAI.



(a) Normal TAU (b) SigOver TAU

Figure 8: Normal and attack case for TAU procedures

UE. If the TAC is not included in the list of TACs on the UE, the UE initiates a TAU procedure to notify the LTE network of the updated TAC. The `SigOver` attack incurs the signaling storm by repeatedly triggering invalid TAU. Figure 8 illustrates the attack process when compared with the normal (i.e., without attack) operation. The attacker first overshadows a paging message with the `system_Info_Modification` field set as true, thus forcing the UEs to read SIB1. The SIB1 is then overshadowed using a spoofed TAC, thus leading to the TAU. It should be noted that the TAU request messages are directed to the legitimate eNB, because the `SigOver` attack preserves the radio connection between the victim and the legitimate eNB. Repeating this procedure results in the signaling storm on the LTE network. On the contrary, under normal circumstances, the TAU is performed only once each time the UE moves to another TA not included in the TAI list.

Validation. This attack was validated using a carrier's testbed LTE network with nine LTE devices[6] registered to the testbed network. Each device was running the diagnostic monitor tools (e.g., SCAT, XCAL [8, 42]) for the analysis of the UE-side signaling messages throughout the attack. Figure 9 reveals that a single UE carries out an average of seven TAU procedures per second, which is unlikely under the normal conditions without the attack. Moreover, the UE-side signaling messages were analyzed to better understand the behavior of the network under the attack. When the victim UE carries out the TAU with the spoofed TAC (irrespective of the validity of the TAC value), the network returns the same list of TACs previously provided during the legitimate registration. This is because the serving cell is unchanged. That is, the list of TACs still does not include the victim UE's spoofed TAC. Hence, the victim UE repetitively carries out the TAU upon receiving the SIB1 message from the attacker. Nokia reports [31] that a UE generates approximately 45 service requests[7] during a peak busy hour. However, the signaling storm via the `SigOver` attack induces a more significant network traffic, e.g., an attacker is able to trigger an average of 25,200 TAUs per UE per

[6]The iPhone was excluded because our monitoring tool does not support it

[7]UE sends a `Service request` during the connection initiation to the LTE network.

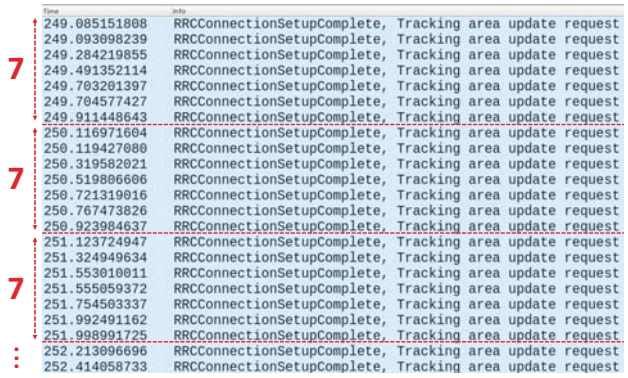


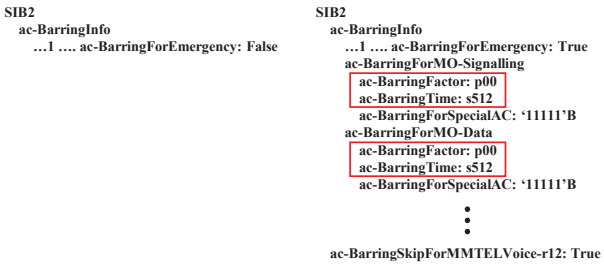
Figure 9: Wireshark snapshot of `TAU Request` messages generated by the SIB1 spoofing.

hour. Given that the number of signaling messages generated through the TAU and service request is similar, the attacker can generate more traffic than that generated during a peak hour by a factor of 560. This clearly demonstrates the significant impact of the signaling storm attack, which imposes a heavy signaling load on the network and causes severe battery drainage for the UE.

Boosted impact of Qualcomm chipset. A sustained signaling storm attack requires the attacker to continually inject SIB1 messages. However, the smartphones equipped with the Qualcomm baseband (e.g., Galaxy S4/S8/S9, LG G2/G6/G7) malfunctioned, thus generating TAUs indefinitely after a single SIB1 injection. In particular, the UE continued to perform the TAU procedure, even after the attacker stopped injecting SIB1[8]. The malfunctioning UE exhibits a normal behavior to the user, which indicates that the data/call service can be used without disrupting the user. Although the malfunction can be fixed by setting the UE in airplane mode, the user is unlikely to do so without noticing any problems. This indicates that the attack is sustained, even with the low-cost efforts to further strengthen its impact.

Infeasibility of the FBS or Rogue UEs. The signal storm attack seems to be achievable with an FBS. However, the injection of malicious SIB1 (containing the spoofed TAC) via the FBS does not lead to the signaling storm attack. This is because under the FBS, the `TAU request` from the victim UE is directed to the FBS, instead of the legitimate LTE network. In other words, the signals do not reach the LTE network; thus, the signaling storm attack is inherently unachievable for the FBS. Moreover, exploiting a number of rogue UEs may induce the signaling storm on the network. However, this approach is limited with respect to its scalability, wherein it requires multiple radio devices and SIM-cards for each device, to induce the same effect as the `SigOver` attack. On the other hands, the `SigOver` attack uses a single radio device that covers an entire cell and forces several authentic users camping on the cell to initiate the TAU procedure.

[8]The root cause of this malfunctioning is the implementation logic of the Qualcomm LTE chipset, which did not read the SIB1 after completing the TAU. As a result, they could not recognize the legitimate SIB1 that contained correct TAC, and the TAU was carried out until the legitimate SIB1 was re-read.



(a) Original SIB2 (b) Malicious SIB2

Figure 10: Access control feature in SIB2 message

5.1.2 Selective DoS through Access Barring

Attack mechanism. The cellular network has control over the number of UEs that can access the network. This feature is to manage the amount of traffic and maintain the stability of the network under specific conditions, e.g., a disaster. The control is realized using the *BarringFactor* parameter in SIB2, which is exploited by the `SigOver` attack to block the victim UE. By setting *BarringFactor* as 0 (via overshadowing), an attacker can restrict all data traffic and signaling from the UE (i.e., mobile originating)[9], which leads to DoS.

Figure 10 presents the configuration of the malicious SIB2 in the crafted subframe in comparison with the original SIB2 in a legitimate subframe. To maximize the impact of the attack, the `SigOver` attack sets the *BarringTime* to 512s, which is the maximum value as per the standard. Note that *BarringTime* can be refreshed if the attacker repeats the attack within the remaining *BarringTime*; thus, a persistent DoS can be achieved. To properly inject the crafted subframe (similarly to the signaling storm), the attacker first overshadows a paging message with `system_Info_Modification`. Thereafter, she overhears the legitimate SIB1 to extract the SFN, from which the attacker can obtain the schedule of the next SIB2 for overshadowing. A potential extension of this attack is service-specific DoS to *selectively* block only the targeted services (e.g., voice call, video conference, and SMS). This leverages a new service-specific barring feature introduced in 3GPP specifications [7].

Validation. This attack was validated using 10 different smartphone models. Upon the successful `SigOver` attack (i.e., injected paging and SIB2 are received); entire data services, which include web browsing and video streaming were blocked on all 10 devices. From the analysis of the device logs, it was found that all the devices failed to initiate any connection when applications made multiple connection requests. This confirms the feasibility of the barring via the `SigOver` attack. Moreover, the service-specific DoS was validated using

[9]The attacker can also block the mobile terminating traffic by overshadowing the paging channel of the victim UE.

the Samsung Galaxy S9 based on the Exynos chipset.

Comparison with the FBS. An FBS can also inject malicious SIB2. However, the attack is only valid when the FBS is turned on, and immediately stops when the FBS is turned off. This is because the victim UE connects to the legitimate cell shortly after disconnection from the FBS. During the connection to the legitimate cell, the victim UE reads the legitimate SIB2, which recovers UE services. Conversely, the services of the victim UE remain blocked after `SigOver` attack stops, as this does not incur cell reselection. Furthermore, the FBS cannot achieve the service-selective DoS, as it cannot provide the LTE service.

5.2 Attacks Exploiting Paging

In this section, we present three attacks through the `SigOver` attack on the paging message: DoS attack, network downgrading, and location tracking.

5.2.1 DoS Attack by Overshadowing Paging with IMSI

Attack mechanism. When the GUTI of the UE is unavailable, the network sends paging message with IMSI as an identifier of UE. As defined in the 3GPP standards, upon receiving the paging that contains the IMSI, the UE terminates all service sessions and initiates the registration procedure using the IMSI as the identifier [5]. This implies that the DoS attack can be realized by injecting the paging message with IMSI[10]. Specifically, the attacker injects a paging message that contains the IMSI of the victim UE at the paging occasion/frame of the victim UE. This attack detaches a UE from the cellular network services, which include voice call and data services, thus indicating a DoS at the UE. As the registration procedure (which follows the service termination) automatically recovers the services, the attack is sustained by the repeated injection of the paging message.

Validation. This attack was validated using 10 different smartphone models in two different operation states (*RRC Idle* and *RRC Connected*). Specifically, in the *RRC Idle* state, we confirmed that the UEs successfully received the overshadowed paging message. Furthermore, the internal logs in the UEs confirmed the expected impact of the attack, i.e., detachment from the network followed by the registration procedure, thus leading to DoS.

For following experiment, we launched the attack on the UE in the *RRC Connected* state. Note that the `SigOver` attack enables the attacker to convey the crafted message to the UE on the existing radio connection between the UE and the eNB. We first make a voice call on the victim UE to force the UE to enter the *RRC Connected* state. We then transmitted the paging message with IMSI to the UE. Interestingly, we observed that not all UEs handled the paging messages in the *RRC Connected* state. In particular, the Samsung Galaxy S8/S9, LG G6/G7 (Qualcomm), Samsung Galaxy S8/S9 (Exynos), and Apple iPhone XS (Intel) properly handled the paging message with IMSI, after which the call was immediately aborted (service termination). Meanwhile, the Samsung Galaxy S6 (Exynos), and Galaxy S4, LG G2 (Qualcomm) did not respond to the attack in the *RRC Connected* state.

The inconsistencies between the devices stem from the ambiguity of the 3GPP standards. The mechanism used to handle paging in the *RRC Connected* state is loosely defined, without specific direction on paging with IMSI, e.g., only information on paging with the system information notification or CMAS/ETWS [3] is provided. In summary, by injecting the paging message with IMSI , the `SigOver` attack can realize a DoS on the victim UE in *RRC Idle* and *RRC Connected* states, depending on the device.

Comparison with the FBS. This attack scenario was extensively discussed in the previous work [21, 35] leveraging the FBS. Although the impact and the attack vectors are equivalent, the applicability of the existing attacks is limited when compared with the `SigOver` attack. This is because the `SigOver` attack uniquely enables the attacker to deliver the paging message to the UE which has an active radio connection with the network, whereas other works are only applicable to UEs that use no services; thus indicating the wider applicability of the `SigOver` attack.

5.2.2 Network Downgrading Attack via CS Paging

Attack mechanism. In this attack, an attacker injects a paging message with a Circuit Switched (CS) notification (with the S-TMSI of the victim UE) to intentionally downgrade victim UEs to the 3G network. Upon the reception of the CS paging, the UE initiates the Circuit Switched Fall-Back process and transits to the 3G network. That is, the `SigOver` attack enables the attacker to force the UE to a slower connection.

Validation. We experimentally confirmed that the victim UE in the *RRC Idle* state immediately switched to the 3G network when the attacker's CS paging was received, after which it soon reverted back to the LTE network because there was no actual service in the 3G network. The attack was effective for the state-of-the-art smartphones, e.g., the Samsung Galaxy S8/S9, LG G6/G7 (Qualcomm), and Samsung Galaxy S8/S9 (Exynos), as they were able to respond the CS paging message the both *RRC Idle* and *RRC Connected* states. However, similar to the paging attack with IMSI, some smartphones did not respond to the CS paging in the *RRC Connected* state, and were therefore immune to the attack. Interestingly, when the Samsung Galaxy S8 (Qualcomm) dropped to the 3G network due to the attack, the LTE connection was never restored while using data service.

Comparison with the existing attack. Tu *et al.* demonstrated the throughput degradation attack against an victim UE by invoking the CS paging, which is similar to our attack [47]. However, in this study, the network was driven to send the paging message on behalf of the attacker, by establishing a

[10] Acquiring IMSI is extensively discussed in the previous work [11, 44]

call with the UE in the 3G network. It should be noted that, in the SigOver attack, the paging message is directly transmitted by the attacker. This attack inherently exposes the attacker's phone number, thus making the attack easily detectable by the operator. In comparison, the SigOver attack silently transmits the CS paging to the victim UE. Furthermore, the existing work cannot downgrade the victim UEs in the *RRC Connected* state to the 3G network, since the network does not send a paging message to the victim UE in the *RRC Connected* state; whereas the SigOver attack can deliver the paging message.

5.2.3 Coarse-grained Tracking of a UE

Attack mechanism. As explained in Section 2, following the completion of the RA procedure, the UE attempts to establish an RRC connection by sending a Connection request (containing UE identity) to the cell. If the UE holds the previously assigned temporary identity (i.e., S-TMSI), this identity is included in the Connection request as well. Otherwise, a random value is selected. Upon the receipt of the UE's request, the cell replies with the Connection setup that contains the UE's identity (the S-TMSI or the random value). By checking this identity, each UE is able to recognize if its RA procedure was successful. If the procedure fails, the UE retries the RA procedure. The abovementioned procedure used to resolve connection conflicts is referred to as a *contention resolution*.

In this attack, an attacker exploits the contention resolution technique to perform coarse-grained location tracking of the target victim. First, the attacker with the knowledge of the S-TMSI of the victim UE injects a paging message with the S-TMSI[11]. The attacker then eavesdrops on the Connection setup messages transmitted from the legitimate cell[12]. When the Connection setup message that contains the S-TMSI of the victim UE is received, the attacker confirms that the victim UE resides within the coverage of the cell by sniffing the downlink messages.

Validation. We validated this attack using all the smartphone models in this work. We confirmed that the attacker is able to identify the presence of the victim UE by injecting a single paging message and eavesdropping on the Connection setup message sent to the victim UE.

Comparison with the FBS. An FBS can achieve the same results by monitoring the IMSI in Identity Response message. However, the FBS requires an active connection to the target victim to transmit the message. Therefore, the attack is limited by the FBS with respect to its stealthiness and power efficiency. In a previous study, it was reported that RNTI-TMSI mapping can be applied to passively monitor the victim's TMSI [37]; however, the SigOver attack provides an active method by which the victim can be located.

[11] Due to the space limit, a detailed discussion on how an attacker acquires the S-TMSI of the target UE was omitted. However, this has been extensively investigated in previous studies [19, 22, 23, 37].

[12] Since the RRC connection procedure is not encrypted, the attacker can eavesdrop on any downlink messages during the connection procedure of the UEs.

6 Defending Against SigOver Attack

In this section, we present an outline of two possible defense strategies against the SigOver attack. We start the feasibility of the fundamental solution as a prevention measure, in which all the broadcast signals were digitally signed by adopting the Public Key Infrastructure (PKI). We then discuss a short-term solution for the detecting SigOver attack, which leverages the changing nature of the physical signal during the processing of the overshadowing signal.

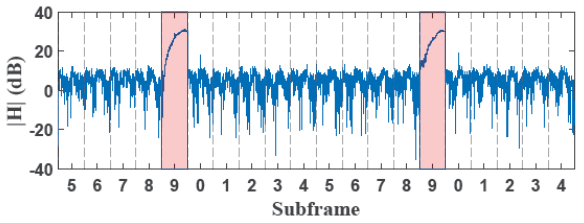
6.1 Digitally Signing Broadcast Messages

As the SigOver attack exploits the lack of integrity protection in broadcast messages, one natural defense against SigOver attack is to employ integrity protection in the messages using a digital signature scheme. For this, each base station needs to have a certificate issued by its operator and a UE needs to be provisioned with a root certificate (e.g., self-signed one by the operator) to verify the certificate of the base station. However, this natural defense has at least several deployment and technical challenges.

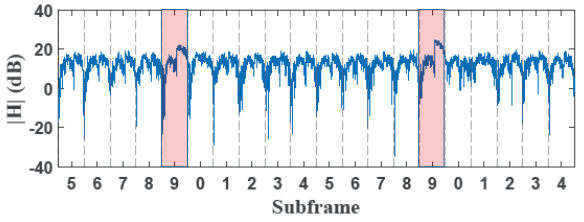
Deployment challenges: In 5G, the 3GPP introduced a public key encryption for IMSI in the initial registration, to provide privacy protection for the permanent identifier. For this, each UE is provisioned with its home operator public key, thereby it was assumed that a public key provisioning mechanism to the UE is in place. This provisioning mechanism could also be used to provision a public key (or a signing certificate) for base station certificate verification. However, in roaming scenarios, the UE need to acquire the public key of the visited network operator, which is trusted by the home operator. This essentially requires a PKI for the global cellular networks that span the world and non-trivial trust relationships among multiple operators in different jurisdictions. Furthermore, managing certificate revocation lists are another obvious burden.

Technical challenges: Signing every single broadcast message may incur a substantial computational overhead at the base station, considering the low periodicity of essential broadcast messages such as MIB (40ms) and SIB1/2 (80ms). Furthermore, message size increases due to the signature and certificate broadcasting (e.g., using a new SIB) would result in a higher power consumption at the base station. Similarly, from UE's perspective, verifying certificate and signature would require additional power consumption, resulting in a faster battery drain. Such a power consumption may be prohibitive to low-power Internet of Things devices that need to survive many years without battery replacements.

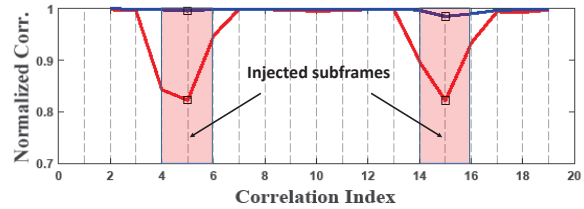
An ID-Based Signature scheme (IBS) [9, 41] can be considered as a cost-effective alternative, as it has substantially low key management overhead and eliminates the certificate broadcast and verification overhead. However, the IBS requires UEs to get synchronized with the public parameters from KMS [17]. This is problematic to UEs that do not have a



(a) LOS setup



(b) NLOS setup



(c) Normalized cross-correlation on LOS (Red line) and NLOS (Blue line)

Figure 11: Fluctuation of the channel estimation magnitude after the `SigOver` attack: Sudden magnitude changes could be used for detection metric.

subscription as they may not be able to get the public parameters from the network. Note that the unsubscribed devices are also supposed to receive the ETWS or CMAS messages as long as they have cellular capabilities.

6.2 Leveraging the Channel Diversity

According to communication theory, a wireless channel varies significantly with a displacement of only a quarter of the wavelength, which is 3.57cm for 2.1GHz LTE [46]. This is referred to as the channel diversity, and it is highly applicable to the attacker and victim UE, which are expected to be at different locations – i.e., the wireless channel between the attacker and UE is likely to be disparate from that between the eNB and UE. Therefore, the injection of the attack signal, which reflects the channel between the attacker and UE, naturally forces the channel information recovered at the UE to deviate from when only the legitimate subframes are present (without attack). In other words, the detection of such a variance in the channel could serve as a defense technique.

The wireless channel can be conventionally represented as H [46] in complex representation. The magnitude $|H|$ uniquely defines different wireless channels depending on how efficiently the signal power is delivered. Hence, an abrupt change in $|H|$ is an effective metric to detect `SigOver` attack.

Figure 11a presents $|H|$ of the injected (Subframe 9) and legitimate signals measured during the experiment in LOS setup, where the attacker is located 2m away from the victim UE. This clearly demonstrates the severe fluctuation of $|H|$ when the attack occurs, indicating effortless detection.

Despite its effectiveness, the robustness of leveraging the channel is problematic. In particular, the general application of the technique to various scenarios is not trivial, due to the various factors that have a potential influence on H. Figure 11b shows a detection failure example in NLOS setup, when the power of the injected signal was low. That is, the impact of the attack signal to H gradually fades out as the energy decreases, down to the point where it is difficult to detect. Figure 11c clearly demonstrates this challenge, wherein the drop in the correlation was fuzzy in NLOS setup, unlike in the LOS setup (strong injection signal). In summary, leveraging the channel is a potential solution, where we leave the design of robust techniques as future work.

6.3 Discussion on Potential Solutions

Both approaches discussed in the previous sections present challenges to be addressed and/or limitations. However, we note that the exploits demonstrated in Section 5 are only a few examples rather than an exhaustive list. The effects of the `SigOver` attack would be broader and more damaging if the cellular network is utilized for critical domains, e.g., vehicular networks and industrial systems. Therefore, in principle, the intrinsic broadcast vulnerabilities of the cellular system should be addressed. Meanwhile, it is recommended that critical services should have their own security protection instead of relying on those of other protocol layers. For example, the issue of the ETWS or CMAS may be better addressed at the application level[13], instead of being based on SIB protection, since the SIB is only a transport mechanism for those critical application messages.

7 Related Work

In this section, we describe previous work that exploits the signal overshadowing concept. We then present the signaling storm, in addition to attacks that exploit the non-integrity protection.

Signal overshadowing in wireless channel. The signal overshadowing attack, which exploits the use of an open medium and the capture effect, has been widely conducted in the wireless systems such as GPS [20, 45] and Low-Rate Wireless Personal Area Networks (LR-WPANs) [52]. Pöpper *et al.* presented a symbol flipping attack on the Additive White Gaussian Noise (AWGN) channel [34], with a fine-grained overshadowing of the signal at the symbol level. However, it requires exact information with respect to the timing, amplitude, and phase, which is difficult to achieve in the real

[13] The 3GPP already conducted a study on the security aspects of Public Warning System (PWS) [4].

world. Similar to this study, Wilhelm et al. demonstrated the possibility of signal overshadowing and its impact on IEEE 802.15.4 [52]. In comparison, the SigOver attack is the first comprehensive study in which the signal overshadowing attack on the LTE was realized, in addition to the validation of its practicability. Moreover, we present the novel attack scenarios by leveraging the SigOver attack.

Message manipulation in LTE. The LTEInspector [21] conducted a paging channel hijacking attack and paging message injection attack, which seems similar to this study. However, this study has two key differences: 1) the definition of the injection attack and 2) its realization method. First, the SigOver attack silently injects the victim with malicious messages while making the victim keep being synchronized with the legitimated eNB. As a result, during the SigOver attack, the uplink response message of the victim naturally goes to the legitimated eNB. However, the victim UE in LTEInspector transmitted its uplink response message to the malicious eNB after receiving the manipulated message, which is the general response action for the attack with the FBS. Thus, it is more similar to existing attacks using FBS. Second, the SigOver attack overwrites the target signal with malicious signals without requiring a connection to the malicious base station. To this end, we investigate various requirements of the SigOver attack as described in Section 3. Despite other requirements, LTEInspector only considered the paging cycle and its occasion, which are the part of timing synchronization requirements.

Attacks exploiting non-integrity protection. Extensive research has been conducted on the manipulation of messages with no/weak-integrity protection [21, 22, 26, 36, 39, 40]. As discussed in Section 3.5, such attacks mainly leverage an FBS. Although they exploited the broadcast messages in LTE, but the attacks have limited implications. This is because their operational logic inevitably produces limitations with respect to stealthiness, power efficiency, and attack sustainability.

8 Concluding Remarks and Future Work

Signal overshadowing is an intuitive method for the manipulation of LTE broadcast messages with no integrity protection, which was not addressed in previous studies. In this paper, we present the SigOver attack, which outlines the first realization of a signal overshadowing attack on the LTE network. We implement the SigOver attack using a low-cost SDR and open source LTE library, while resolving the challenges in satisfying the stringent transmission requirements and crafting a malicious frame. The feasibility and effectiveness of SigOver attack was demonstrated in five novel attacks, and an extensive analysis of the relative advantages of the SigOver attack over those of the FBS and MitM attacks was carried out. The key features of the SigOver attack are stealthiness, power-efficiency, and sustainability, which have not been achieved simultaneously by previous attacks. The evaluation revealed that the SigOver attack achieves a 98% success rate with low power cost.

Finally, two potential approaches to defending against the SigOver attack were proposed, which leveraged the digital signature and channel diversity. As acknowledged, both approaches have challenges and limitations to be addressed; however, they can be used as a basis for the development of a reliable and robust solution.

The cellular industry is rapidly transitioning to the 5G network of cellular systems equipped with advanced radio technologies and enhanced security features. However, the fundamental broadcast security issues discussed in this paper have not addressed in design. Considering significant changes made in the 5G New Radio (NR), it is left as a future study to evaluate 5G NR against the SigOver attack. As this paper turns on the spotlight on the security of broadcast messages, we believe that 3GPP standard body and cellular network community need to consider the design of broadcast messages seriously.

Acknowledgments

We sincerely thank Dr. Soo Bum Lee for his detailed and valuable comments on the earlier version of the draft. In addition, we would like to thank the anonymous reviewers for their insightful comments. This work was supported by Institute for Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (2018-0-00831, A Study on Physical Layer Security for Heterogeneous Wireless Network).

References

[1] 3GPP. ETSI TS 36.104. Base Station (BS) radio transmission and reception, 2017.

[2] 3GPP. ETSI TS 36.211. Physical channels and modulation, 2011.

[3] 3GPP. ETSI TS 36.331. RRC Protocol specification, 2017.

[4] 3GPP. TR 33.969. Study on Security aspects of Public Warning System (PWS), 2014.

[5] 3GPP. TS 24.301. Non-Access-Stratum (NAS) protocol for Evolved Packet System (EPS); Stage 3, 2017.

[6] 3GPP. TS 36.101. User Equipment (UE) radio transmission and reception, 2017.

[7] 3GPP. TS 36.331. Evolved Universal Terrestrial Radio Access (E-UTRA); Radio Resource Control (RRC); Protocol specification, 2017.

[8] ACCUVER. XCAL. http://accuver.com/acv_products/xcal/.

[9] Dan Boneh and Matt Franklin. Identity-based encryption from the weil pairing. In *Annual international cryptology conference*, pages 213–229. Springer, 2001.

[10] Bruno Clerckx and Claude Oestges. *MIMO wireless networks: channels, techniques and standards for multi-antenna, multi-user and multi-cell systems*. Academic Press, 2013.

[11] Adrian Dabrowski, Nicola Pianta, Thomas Klepp, Martin Mulazzani, and Edgar Weippl. IMSI-catch me if you can: IMSI-catcher-catchers. In *Proceedings of the 30th annual computer security applications Conference*, pages 246–255. ACM, 2014.

[12] Sebastian Egger, Tobias Hossfeld, Raimund Schatz, and Markus Fiedler. Waiting times in quality of experience for web based services. In *Quality of Multimedia Experience (QoMEX), 2012 Fourth International Workshop on*, pages 86–96. IEEE, 2012.

[13] Juanita Ellis, Charles Pursell, and Joy Rahman. *Voice, video, and data network convergence: architecture and design, from VoIP to wireless*. Elsevier, 2003.

[14] Ettus. GPSDO OCXO. https://www.ettus.com/product/details/GPSDO-MINI.

[15] Ettus. UBX 160MHz Board. https://www.ettus.com/product/details/UBX160.

[16] Ettus. USRP X300/X310 Spec Sheet. https://www.ettus.com/content/files/X300_X310_Spec_Sheet.pdf.

[17] Michael Groves. Elliptic Curve-Based Certificateless Signatures for Identity-Based Encryption (ECCSI). *RFC6507*, 2012.

[18] GSA. Evolution from LTE to 5G: Global Market Status. Aug. 2018.

[19] Byeongdo Hong, Sangwook Bae, and Yongdae Kim. GUTI Reallocation Demystified: Cellular Location Tracking with Changing Temporary Identifier. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[20] Todd E Humphreys, Brent M Ledvina, Mark L Psiaki, Brady W O'Hanlon, and Paul M Kintner. Assessing the spoofing threat: Development of a portable GPS civilian spoofer. In *Radionavigation Laboratory Conference Proceedings*, 2008.

[21] Syed Rafiul Hussain, Omar Chowdhury, Shagufta Mehnaz, and Elisa Bertino. LTEInspector: A Systematic Approach for Adversarial Testing of 4G LTE. In *Proceedings of the Network and Distributed Systems Security (NDSS)*, 2018.

[22] Hongil Kim, Jiho Lee, Eunkyu Lee, and Yongdae Kim. Touching the Untouchables: Dynamic Security Analysis of the LTE Control Plane. In *IEEE Symposium on Security & Privacy (SP)*. IEEE, 2019.

[23] Denis Foo Kune, John Koelndorfer, Nicholas Hopper, and Yongdae Kim. Location leaks on the GSM Air Interface. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2012.

[24] Younes Labyad, Mohammed MOUGHIT, Abderrahim Marzouk, and Abdelkrim HAQIQ. Impact of Using G. 729 on the Voice over LTE Performance. *International Journal of Innovative Research in Computer and Communication Engineering*, 2(10), 2014.

[25] Zhenhua Li, Weiwei Wang, Christo Wilson, Jian Chen, Chen Qian, Taeho Jung, Lan Zhang, Kebin Liu, Xiangyang Li, and Yunhao Liu. FBS-Radar: Uncovering Fake Base Stations at Scale in the Wild. In *NDSS*, 2017.

[26] Huang Lin. LTE REDIRECTION: Forcing Targeted LTE Cellphone into Unsafe Network. In *Hack In The Box Security Conference (HITBSecConf)*, 2016.

[27] Konstantinos Manolakis, David Manuel Gutiérrez Estévez, Volker Jungnickel, Wen Xu, and Christian Drewes. A Closed Concept for Synchronization and Cell Search in 3GPP LTE Systems. In *2009 IEEE Wireless Communications and Networking Conference*, pages 1–6, April 2009.

[28] minicircuit. ZVE-2W-272. https://www.minicircuits.com/WebStore/dashboard.html?model=ZVE-2W-272%2B.

[29] Johan Moberg, Mattias Löfgren, and Robert S Karlsson. Throughput of the WCDMA Random Access Channel. In *IST Mobile Communication Summit*, 2000.

[30] Peter Ney, Ian Smith, Gabriel Cadamuro, and Tadayoshi Kohno. SeaGlass: enabling city-wide IMSI-catcher detection. *Proceedings on Privacy Enhancing Technologies*, 2017(3):39–56, 2017.

[31] David Nowoswiat. Managing LTE Core Network Signaling Traffic. https://www.nokia.com/blog/managing-lte-core-network-signaling-traffic/.

[32] OPENBTS. Ettus Research USRP. http://openbts.org/w/index.php?title=Ettus_Research_USRP.

[33] Shinjo Park, Altaf Shaik, Ravishankar Borgaonkar, Andrew Martin, and Jean-Pierre Seifert. White-Stingray: Evaluating IMSI Catchers Detection Applications. In *USENIX Workshop on Offensive Technologies (WOOT). USENIX Association*, 2017.

[34] Christina Pöpper, Nils Ole Tippenhauer, Boris Danev, and Srdjan Capkun. Investigation of Signal and Message Manipulations on the Wireless Channel. In *Proceeding of the European Symposium on Research in Computer Security (ESORICS)*, 2011.

[35] Muhammad Taqi Raza, Fatima Muhammad Anwar, and Songwu Lu. Exposing LTE Security Weaknesses at Protocol Inter-Layer, and Inter-Radio Interactions. In *International Conference on Security and Privacy in Communication Systems*, pages 312–338. Springer, 2017.

[36] David Rupprecht, Kai Jansen, and Christina Pöpper. Putting LTE Security Functions to the Test: A Framework to Evaluate Implementation Correctness. In *10th USENIX Workshop on Offensive Technologies (WOOT)*, 2016.

[37] David Rupprecht, Katharina Kohls, Thorsten Holz, and Christina Pöpper. Breaking LTE on Layer Two. In *IEEE Symposium on Security & Privacy (SP)*. IEEE, 2019.

[38] Stefania Sesia, Matthew Baker, and Issam Toufik. *LTE-the UMTS long term evolution: from theory to practice*. John Wiley & Sons, 2011.

[39] Altaf Shaik, Ravishankar Borgaonkar, N Asokan, Valtteri Niemi, and Jean-Pierre Seifert. Practical Attacks Against Privacy and Availability in 4G/LTE Mobile Communication Systems. *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[40] Altaf Shaik, Ravishankar Borgaonkar, Shinjo Park, and Jean-Pierre Seifert. On the Impact of Rogue Base Stations in 4G/LTE Self Organizing Networks. In *WISEC*, pages 75–86, 2018.

[41] Adi Shamir. Identity-based cryptosystems and signature schemes. In *Workshop on the theory and application of cryptographic techniques*, pages 47–53. Springer, 1984.

[42] Signaling Collection and Analysis Tool (SCAT). https://github.com/fgsect/scat.

[43] srsLTE. https://github.com/srsLTE/srsLTE.

[44] Daehyun Strobel. IMSI catcher. *Chair for Communication Security, Ruhr-Universität Bochum*, 14, 2007.

[45] Nils Ole Tippenhauer, Christina Pöpper, Kasper Bonne Rasmussen, and Srdjan Capkun. On the requirements for successful GPS spoofing attacks. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 75–86. ACM, 2011.

[46] David Tse and Pramod Viswanath. *Fundamentals of wireless communication*. Cambridge university press, 2005.

[47] Guan-Hua Tu, Chi-Yu Li, Chunyi Peng, and Songwu Lu. How voice call technology poses security threats in 4g lte networks. In *Communications and Network Security (CNS), 2015 IEEE Conference on*, pages 442–450. IEEE, 2015.

[48] Jan-Jaap Van de Beek, Magnus Sandell, and Per Ola Borjesson. ML estimation of time and frequency offset in OFDM systems. *IEEE transactions on signal processing*, 45(7):1800–1805, 1997.

[49] Thanh van Do, Hai Thanh Nguyen, Nikolov Momchil, et al. Detecting IMSI-catcher using soft computing. In *International Conference on Soft Computing in Data Science*, pages 129–140. Springer, 2015.

[50] Qi Wang, Christian Mehlfuhrer, Christian Mehlführer, and Markus Rupp. Carrier frequency synchronization in the downlink of 3GPP LTE. In *21st Annual IEEE International Symposium on Personal, Indoor and Mobile Radio Communications*, pages 939–944, Sep. 2010.

[51] Kamin Whitehouse, Alec Woo, Fred Jiang, Joseph Polastre, and David Culler. Exploiting the capture effect for collision detection and recovery. In *Embedded Networked Sensors, 2005. EmNetS-II. The Second IEEE Workshop on*, pages 45–52. IEEE, 2005.

[52] Matthias Wilhelm, Jens B Schmitt, and Vincent Lenders. Practical message manipulation attacks in IEEE 802.15. 4 wireless networks. *Proceedings of MMB & DFT*, 2012.

[53] Zhou Zhuang, Xiaoyu Ji, Taimin Zhang, Juchuan Zhang, Wenyuan Xu, Zhenhua Li, and Yunhao Liu. Fbsleuth: Fake base station forensics via radio frequency fingerprinting. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 261–272. ACM, 2018.

Appendix

A Impact on Quality of Services

We measure the impact of the quality of services under the `SigOver` attack, where the malicious paging messages are transmitted at the every subframe 9. This implies that legitimate subframes at subframe 9 are overshadowed and lost, whereas non-overshadowed legitimate subframes may also be affected by crafted subframes. Specifically, the RS of the crafted subframes perturbs the channel estimation averaged among crafted and non-overshadowed legitimate subframes (in *RRC Connected* state), which may disturb the equalization and incur errors. Despite such factors, the impact of `SigOver` attack is validated to kept minimal, as demonstrated in this section under a range of common, but distinct services of voice call, web surfing, FTP download, and live streaming. We note that measurements were carried out under a reliable `SigOver` attack (>97% success rate) for the UE in the *RRC Connected* state.

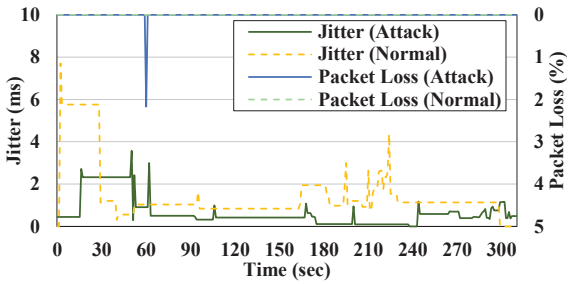


Figure 12: Call jitter and packet loss

Voice call. The UEs camping on LTE network use Voice over LTE (VoLTE) as a call service. We evaluate the impact of the `SigOver` attack on the key factors with respect to the VoLTE performance [13]; or equivalently, the call quality, e.g., data rate, jitter, and packet loss. Such metrics were measured before and after the attack for comparison. The data rate was kept stable after the attack, and omitted for brevity. Figure 12 illustrates the jitter and the packet loss. The jitter was consistently less than 10ms, and the packet loss is mostly kept as zero. Moreover, both were sufficient to support high quality call services [24]. This keeps the `SigOver` attack stealthy without degrading user experience.

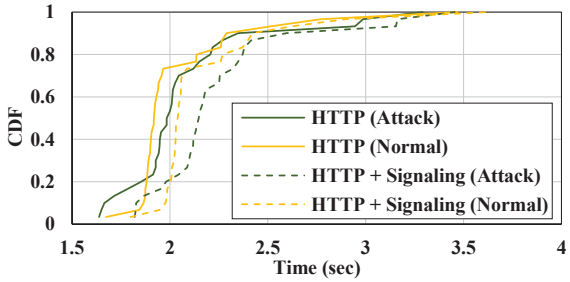


Figure 13: Webpage loading time

Web-browsing. We extend the measurements to web browsing, which is one of the most frequently used services. Specifically, the time required to load multiple identical web pages with and without the attack. Figure 13 presents the results, with 'HTTP' representing the total duration of HTTP data exchange for page loading. 'Signaling' is the time required for RRC connection establishment. Under the `SigOver` attack, the time from the RRC connection initiation to the web page downloading is delayed by an average of only 80ms when compared with the case without the attack. Previous studies [12] have shown that the impact of such lag on the quality of the experience is negligible.

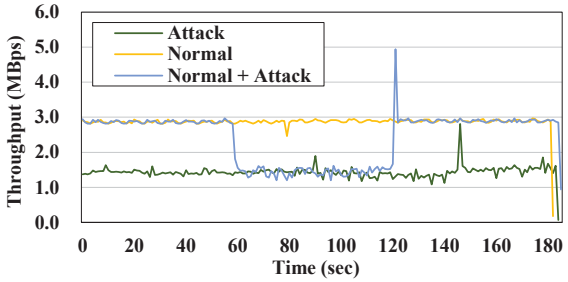


Figure 14: FTP throughput

FTP downloading. Figure 14 reveals that the FTP exhibited a significantly different performance under the `SigOver` attack. This is due to dynamically controlled modulation, to overcome the bit error in the communication. The `SigOver` attack incurs bit errors, which force the UE to use a robust modulation of QPSK, which has a limited throughput. Conversely, without the attack, the bit error is kept low. In this case, the UE used 64QAM which is less robust but supports higher throughput than QPSK. However, this impact is less likely to be experienced by the users and FTP rarely used on smartphones.

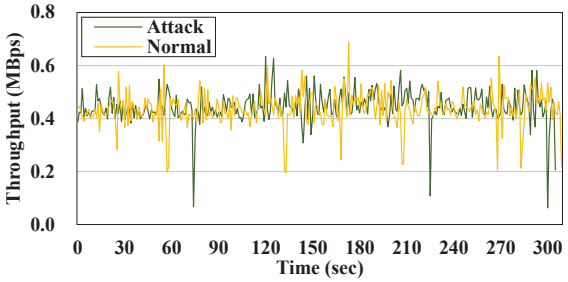


Figure 15: YouTube Live throughput: The average was 0.445 and 0.436MBps for attack and normal case, respectively.

Live streaming. Figure 15 shows the throughput of the YouTube live streaming at a resolution of 1080p. In summary, neither buffering nor interruption occurred under the `SigOver` attack during a 5-min video clip. The result of the live streaming differs from that of the FTP downloads, as streaming throughput was not as high as that of the FTP.

B ACRONYMS

3GPP Third Generation Partnership Project
AKA Authentication and Key Agreement
AS Access Stratum
CFI Control Format Indicator
CMAS Commercial Mobile Alert System
CQI Channel quality indicator
CS Circuit Switched
DCI Downlink Control Information
eNB Evolved Node B
EPC Evolved Packet Core
ETWS Earthquake and Tsunami Warning System
FBS Fake Base Station
FDD Frequency Division Duplex
GPSDO GPS disciplined oscillator
GUTI Globally Unique Temporary Identity
IMSI International Mobile Subscriber Identity
LOS Line of sight
LTE Long Term Evolution
MIB Master Information Block
MME Mobility Management Entity
NAS Non Access Stratum
NLOS Non-line of sight
OFDM Orthogonal Frequency Division Multiplexing
PCFICH Physical Control Format Indicator CHannel
PCI Physical layer Cell Identity
PDCCH Physical Downlink Control CHannel
PDSCH Physical Downlink Shared CHannel
PHICH Physical HybridARQ Indicator CHannel
PRB Physical Resource Block
PSS Primary Synchronization Signal
RA Random Access
RACH Random Access CHannel
RB Resource Block
RRC Radio Resource Control
RS Reference Signal
RSRP Reference Signal Received Power
RSRQ Reference Signal Received Quality
SAE System Architecture Evolution
SDR Software Defined Radio
SFN System Frame Number
SIB System Information Block
SSS Secondary Synchronization Signal
S-TMSI SAE Temporary Mobile Subscriber Identity
TA Tracking Area
TAI TA identity
TAU Tracking Area Update
UE User Equipment

UWB-ED: Distance Enlargement Attack Detection in Ultra-Wideband

Mridula Singh, Patrick Leu, AbdelRahman Abdou, Srdjan Capkun
Dept. of Computer Science
ETH Zurich
{firstname.lastname}@inf.ethz.ch

Abstract

Mobile autonomous systems, robots, and cyber-physical systems rely on accurate positioning information. To conduct distance-measurement, two devices exchange signals and, knowing these signals propagate at the speed of light, the time of arrival is used for distance estimations. Existing distance-measurement techniques are incapable of protecting against adversarial distance enlargement—a highly devastating tactic in which the adversary reissues a delayed version of the signals transmitted between devices, after distorting the authentic signal to prevent the receiver from identifying it. The adversary need not break crypto, nor compromise any upper-layer security protocols for mounting this attack. No known solution currently exists to protect against distance enlargement. We present *Ultra-Wideband Enlargement Detection* (UWB-ED), a new modulation technique to detect distance enlargement attacks, and securely verify distances between two mutually trusted devices. We analyze UWB-ED under an adversary that injects signals to block/modify authentic signals. We show how UWB-ED is a good candidate for 802.15.4z Low Rate Pulse and the 5G standard.

1 Introduction

Ranging and positioning information is often necessary for mobile autonomous systems, robots and cyber-physical systems to operate successfully. These systems are used in security and safety critical applications. Drones are becoming more popular for transportation and rescue [24], and autonomous systems are being increasingly tested and integrated as part of the ecosystem. The 5G community emphasizes the importance of designing the wireless protocols for the safety of the autonomous vehicles [33]. A stringent requirement for these systems is to avoid crashing into, *e.g.,* buildings, pedestrians, properties, or each other [25]. For example, keeping drones and autonomous vehicles on their intended paths and preventing their collision can be achieved only if they are able to calculate their relative positions accurately and securely. Figure 1 shows that an adversary can manipulate the perceived distance between two mutually trusted devices by the distance reduction and enlargement attacks.

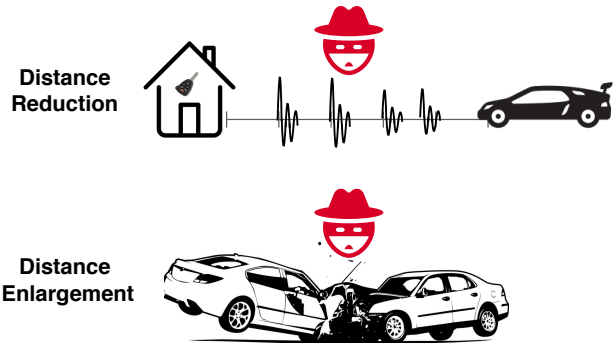


Figure 1: Ranging systems are vulnerable to distance reduction and enlargement attacks.

Conventional ranging systems, such as GPS and WiFi Positioning Systems (WPS) [34], are useful for benign environments and coarse-granular geolocation. However, they provide insufficient precision for accurate distance estimations (*e.g., cm*-level granularity), suffer availability constraints (*e.g.,* indoors, outdoors), and are relatively slow to calculate locations for fast and mobile autonomous systems. More importantly, the aforementioned ranging systems are susceptible to various spoofing attacks [4, 14, 28].

Two-way time-of-flight (ToF)-based ranging systems (which map ToF to distance as signals propagate at the speed of light) have the potential to conduct accurate, fast, and secure distance measurements. Examples include high precision Ultra-wide Band (UWB) ranging systems, some of which are now available off-the-shelf [1, 9, 13, 35]. Numerous previous efforts were directed towards protecting these systems from distance-reduction attacks, *e.g.,* for access control. These mainly rely on the principle that propagation speeds

Version: February 18, 2019.

are bounded by the physical characteristics of the media, and cannot be sped-up. For example, distance bounding protocols return an upper bound on the measured distance, armed by the fact that an adversary would not succeed in guessing (secret) bit level information [5, 6]. Other techniques are based on tailoring modulations to prevent distance-reduction attacks at the physical layer [26]. None of these approaches prevent distance enlargement attacks.

Distance enlargement attacks can deviate vehicles from their intended paths, or cause physical collisions. Existing protection approaches rely on dense, and often fixed, verification infrastructures, *e.g.,* towers. These may not exist, and often do not; installing them in outdoor settings is a costly affair, and not necessarily feasible (*e.g.,* in drone-based military missions behind enemy lines). Distance enlargement is a more devastating attack than distance shortening because an adversary in the communication range only needs to annihilate (cancel) [23] or distort the authentic signals to prevent the receiver from identifying them and using their time-of-arrival (ToA) for ranging. The adversary then simply replays a delayed version of the authentic signals, which it has already received by positioning itself in the vicinity of the sender or the receiver. The adversary need not guess these signals, nor compromise any upper-layer protocols to do that. The amount of delay corresponds to the adversary-intended distance to enlarge. In a collision-avoidance system of automobiles or self-driving cars for example, a few meters ($\sim$ a few nanoseconds) could be catastrophic.

We present *Ultra-Wideband Enlargement Detection* (UWB-ED)—the first known modulation technique to detect distance enlargement attacks against UWB ranging based on ToF. UWB-ED relies on the interleaving of pulses of different phases and empty pulse slots (*i.e.,* on-off keying). Unable to perfectly guess the phase, this leaves the adversary with a 50% chance of annihilating pulses (similarly for amplification). As a result, some of the affected (authentic) pulses will be amplified, while others will be annihilated. Unaffected pulses will remain intact, while positions that originally had no pulses may now have adversary-injected ones. The technique presented herein gets the receiver to seek evidence indicating whether such a deformed trail of pulses in the transmission was indeed authentic, albeit corrupt.

Similar to Singh *et al.* [26] (which addresses distance-reduction attacks), we leverage a randomized permutation of pulses. However, unlike [26], we cannot simply look for whether these are out of order, and ignore them if so because that is precisely the adversary's objective in distance-enlargement: misleading the receiver to ignore the authentic signals. Instead, UWB-ED checks the *energy distribution* of pulses: comparing the aggregate energies of a subset of pulses at the positions where high energy was expected (as per the sender-receiver secret pulse-permutation agreement), with others where low energy was expected. To subvert this, the adversary would be forced to inject excessive energy throughout the whole transmission, which could then be detected using standard DoS/jamming-detection techniques.

We derive the probability that an adversary succeeds in a distance-enlargement attack against UWB-ED. This is also useful in setting input parameters, *e.g.,* balancing an application's security requirements and ranging rate, while accounting for channel conditions. For example, we show how proper parameterization of UWB-ED limits an adversary's success probability in enlarging distances to $< 0.16 \times 10^{-3}$.

In summary, the paper's contributions are twofold.

- UWB-ED—a novel, readily-deployable modulation technique for detecting distance enlargement attacks against UWB ToF ranging systems, requiring absolutely no verification infrastructure, and making no impractical assumptions limiting adversarial capabilities.

- Analytical evaluation to UWB-ED, where the probability of adversarial success is derived as a function of input parameters and channel conditions. This evaluation is also validated using simulations.

The sequel is organized as follows. Sections 2 and 3 provide background and detail the threat model. The new distance enlargement detection technique is explained in Section 4, and evaluated in 5. Section 6 complements with a related discussion, and 7 is related work. Section 8 concludes.

2 Background and Motivation

A device's position can be estimated using the distances between itself and other landmarks with known locations; or it could be expressed using a coordinate system, *e.g.,* in a Cartesian plane. The distance between two devices can be measured using radio signal properties, such as received signal strength [3], phase [30], or the signal's propagation time including ToF and ToA [15]. Reduction or enlargement of the calculated distances can lead to wrong positioning.

Adversarial distance reduction has been analyzed in previous literature [31], but limited work was performed on enlargement attacks. Preventing enlargement is achieved when a node is inside a polygon determined by an infrastructure of devices/towers, where verifiable multilateration [31] is applied. Enlargement attacks are harder to detect without an infrastructure. Signal strength-based systems do not provide strong security guarantees during high variations of signal strengths in some channel conditions. For distance reduction attacks, the adversary can amplify a degraded signal but for enlargement, degradation is in the adversary's favor.

One-way ToF systems, such as GPS, can be spoofed to reduce/enlarge distances [4, 14]. Two-way ToF, such as UWB, provides secure upper bound by using distance bounding along with secure modulation techniques [5, 6, 26]. This provides strong guarantees against reduction attacks, but is still susceptible to enlargement attacks.

2.1 UWB

IEEE 802.15.4a and IEEE 802.15.4f have standardized impulse radio UWB as the most prominent technique for precision ranging. IEEE 802.15.4z [2] is in the process of standardizing UWB to prevent attacks on the ranging systems. Off-the-shelf UWB ranging systems were recently developed [1, 9, 13, 35], and the research community/industry has expressed tremendous interest in these systems (*e.g.,* for autonomous vehicles). Because current standards do not prevent enlargement attacks, it is important to mitigate them before standards are deployed in practice.

Symbol Structure. UWB systems operate over wide segments of licensed spectrum. They have to be compliant with stringent regulatory constraints. Firstly, the power spectral density should not exceed -41.3dBm/MHz, averaged over a time interval of 1ms. Secondly, the power measured in a 50MHz-bandwidth around the peak frequency is limited to 0dBm. Due to these constraints, the power per pulse is limited. To support longer distances, the energy of multiple pulses is aggregated to construct meaningful information. Figure 3 shows On-Off-keying (OOK) modulation, as used in IEEE 802.15.4f-based UWB ranging systems. Each symbol has two pulses and two empty slots. The symbol length is represented as T_b and the spacing between consecutive pulses is T_s. Information bits are encoded in the position of the pulse.

Symbol Detection. Figure 2 shows a conventional non-coherent energy detector (ED) receiver [32]. The energy detector receiver is consist of square-law device to compute instantaneous received signal power and an energy integrator. For the received signal $r(t)$, the output of the receiver can be expressed as:

$$E(k) = \int_{T_s*k}^{T_s*k+T_I} [r(t)]^2 dt \tag{1}$$

where $T_s * k$ is the integration start time, T_I the integration window size, and T_s the spacing between consecutive pulses.

These receivers perform squaring and integration, making phase information irrelevant for pulse detection. In the case of multi-pulse per symbol, the energies of multiple pulses are aggregated. For the orthogonal hypothesis tests H_1 and H_0 for bit 1 and 0 respectively, the decision of the ED receiver is made in favor of the positions with higher energy.

$$b(i) = \begin{cases} 0 & E_{H_0}(i) \geq E_{H_1}(i) \\ 1 & E_{H_0}(i) < E_{H_1}(i) \end{cases} \tag{2}$$

2.2 Distance-Enlargement Attack

In contrast to reduction attacks, to enlarge the distance, the adversary need not predict the authentic signal. Instead, it replays the authentic signal by replaying an amplified version of

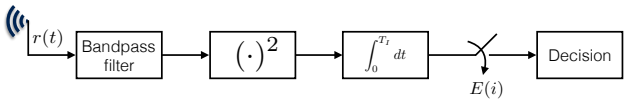


Figure 2: Non-coherent energy detector receiver.

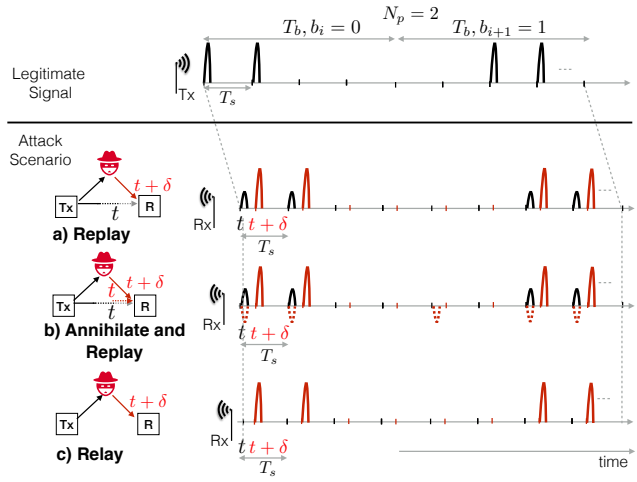


Figure 3: Various attack scenarios on UWB.Black and red colors represent authentic and adversary signals respectively. Dotted red represent adversarial signal-annihilation attempts.

it after some delay. The receiver gets both, authentic and adversary's signal superimposed. Because these authentic signals also reach the receiver, the adversary cannot control how the receiver processes them. None of the existing ranging systems is secure against enlargement attack- be it UWB -802.15.4z, WiFi- 802.11, or GPS. Signal replay is a typical strategy to mount distance enlargement attacks. Other enlargement attacks, such as jamming, alters the output of the receiver's automatic gain control (AGC), and are likely to expose the adversary [22, 27]. Complementing signal replay by signal annihilation prevents the receiver from detecting the authentic signal. Annihilation is possible due to the predictable symbol structure.

In Fig. 3, the devices know each other's communication range, and could verify that they are within that range, *e.g.,* using secure ranging (see Fig. 4). For short LoS distances, a symbol length of $N_p = 1$ (*i.e.,* one pulse-per-symbol) could suffice. Longer distances are attained by longer symbols ($N_p = 2$ in Fig. 3). Pulses are separated by time T_s, which should be more than the channel's delay spread. The length of the symbol (T_b) is determined by the number of pulses per symbol, and the interval between two consecutive pulses $(N_p \cdot T_s)$. Figure 3 also shows instances of replay attacks on these symbols. When an adversary replays authentic signals after some delay (δ), both authentic and replayed signals are received. To deceive the receiver, the adversary needs to annihilate authentic signals.

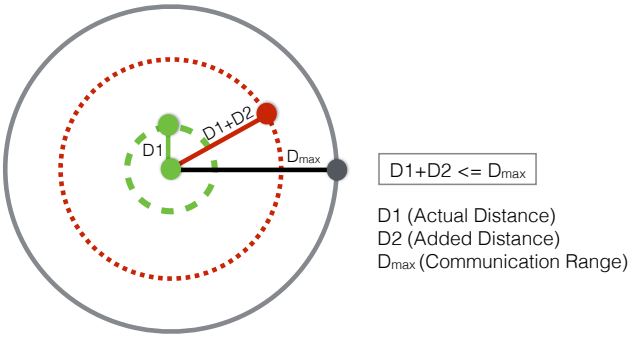


Figure 4: If $D1 + D2 > D_{\max}$, the devices realize they are outside each other's communication range without the need to run distance-enlargement detection protocol.

In Fig. 3a, an authentic signal reaches the receiver at time t, and the adversary's signal at $t + \delta$. If the receiver backtracks in time (searching for earlier-received signals), the authentic signal will be encountered. Figure 3b shows how the predictability of the symbol structure enables an adversary to annihilate its pulses (by emitting a reciprocal pulse phase), preventing the receives from detecting it. Figure 3c shows the case when nodes are not in the communication range (or signal is attenuated by channel condition); the receiver does not get authentic signals, just adversary-relayed (and delayed) signals.

3 Threat Model

We focus on the scenario where there are two devices in a wireless network that are interested to securely measure the physical distance between them, and protect the measurements from a third-party adversary. The devices know their maximum communication range. The adversary's objective is to enlarge the distance that the devices measure. The adversary cannot directly block or modify messages on the channel (*cf.* Dolev-Yao's adversary [10]); it can rather inject signals, and through such injection it can block/modify the authentic signals. If successful, this injection can lead to jamming, signal annihilation, and/or content modification. This model captures the capabilities of man-in-the-middle (MITM) attacks in wireless settings, and is typical in previous literature [7, 12]. The model also fits well with our target application scenario: the communicating devices are typically mobile and move (drive or fly) in formation. In such scenarios, it is unlikely that an adversary prevents the signals of one device from reaching the other by physical obstacles, and is thus limited to injecting signals.

We assume the adversary is able to communicate and listen on any channel the devices use. However, because the devices are communicating over UWB, the adversary is unable to deterministically annihilate pulses without knowing their phase (positive or negative). Existing hardware is not fast enough to enable the adversary to sample a pulse's phase and react by injecting the reciprocal pulse promptly due to the very narrow UWB pulse width of ≈ 2 *ns*. We therefore assume that the adversary will not be able to deterministically annihilate pulses from the channel, only with some probability < 1. It succeeds in annihilating pulses if it guesses the phase of the pulse correctly. We over-approximate the adversary by providing the capability to synchronize attack signal with the authentic transmission. Signal synchronization is a hard problem, but an adversary can achieve it by using stable clock and distance information.

We assume the adversary knows the actual physical distance between the two devices at any point in time. The adversary can calculate this using several means, *e.g.,* by eavesdropping on unencrypted position announcements the devices make. The adversary can also position itself along the direct path between the two devices, measure the distance between itself and each from that position, and add both distances. To measure these distances, the adversary's device can perform two-way ranging with each device independently, pretending to be the other device; or even without such impersonation, it could perform one-way ranging after synchronizing its clock with each device separately.

We assume the devices themselves are not compromised; the adversary cannot attach a physical cable to their interfaces, nor hijack their firmware. However, the adversary can have multiple network cards and antennas, and is not energy-bounded. It can be stationary or mobile.

UWB-ED (Section 4) involves transmitting, between the victim devices, a code of n pulses, α of which are data-representing, and the remaining β are absent of energy, where $n = \alpha + \beta$. We assume the adversary knows the values of α and β, but not the positions of these pulses in the transmission. (Their positions are determined by both devices pseudo-randomly in each transmission.) The adversary can learn these parameters by remaining passive in the vicinity of the victim devices, silently observing their transmissions.

Finally, we assume that it is not in the adversary's interest to prevent the devices from communicating, *e.g.,* by shielding them, or jamming the channel.

4 UWB-ED Design

UWB-ED consists of two phases conducted between both devices: Distance Commitment and Distance Verification. Figure 5 shows a timing diagram of both phases. In the first, the devices measure the distance between them using a two-way ranging protocol. The distance measured in this phase (t^c_{tof}) should not exceed the supported communication range (t^{max}_{tof}). In the distance verification phase, the devices measure their distance by exchanging verification codes (generated using a special UWB-ED modulation). To detect enlargement attacks, devices look for distorted traces of that code. The

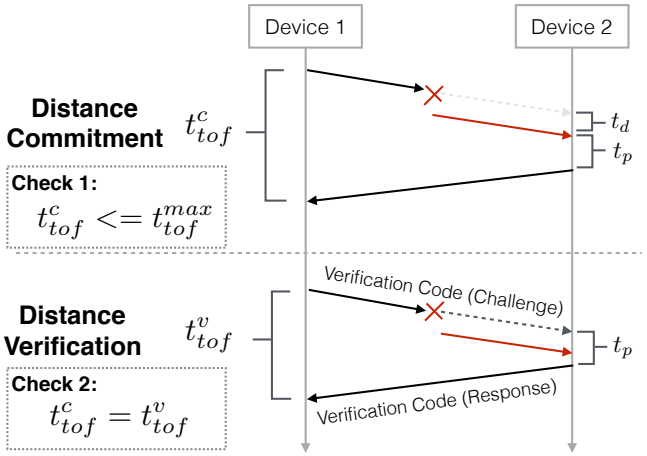


Figure 5: Timing diagram of UWB-ED operation. See inline (Section 4) for notation.

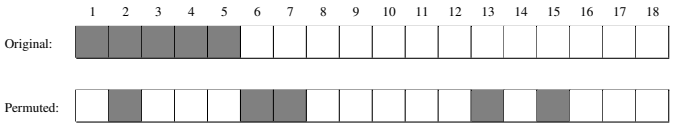


Figure 6: An example verification code with a randomly-looking pulse reordering, where $\alpha = 5$, $\beta = 13$, and the code contains $n = \alpha + \beta = 18$ pulses. Upon receiving the permuted code pulses as per the secret agreement between the sender and receiver, the receiver knows that Bin_α will contain the received energies at the positions (gray) {2, 6, 7, 13, 15}, which are the expected high-energy pulses. Bin_β will contain the rest: {1, 3, 4, 5, 8, 9, 10, 11, 12, 14, 16, 17, 18}.

attack is detected when such traces are found, $t^c_{tof} > t^{max}_{tof}$, or when $t^c_{tof} \neq t^v_{tof}$ (Fig. 5). By enlarging distance in the commitment phase, the adversary increases t^c_{tof} by t_d, but fails to enlarge the distance in the verification phase. Annihilation attempts on the challenge frame are shown, but the adversary can also attack responses from both devices.

Distance Commitment Phase. The devices measure secure upper bound by using distance bounding along with secure modulation techniques [5, 6, 26]. This provides strong guarantees against reduction attacks but is susceptible to enlargement attacks. The distance committed in this phase should not exceed the communication range (*i.e.,* an enlargement attack is detected when $t^c_{tof} > t^{max}_{tof}$). This check ensures that the nodes can communicate without a relay. An adversary enlarging distance by more than the communication range is also exposed using this check.

Distance Verification Phase. In this phase, the committed distance is verified, *i.e.,* an enlargement attack is detected when $t^c_{tof} \neq t^v_{tof}$. To achieve this, the devices measure their distance using round-trip time-of-flight, with both challenge and response messages protected using specially crafted *verification codes* (*i.e.,* special UWB-ED modulation). In this exchange, the sender initiates the distance verification phase by transmitting a verification code; the receiver tries to detect the presence of that code, or traces thereof, in the transmission, despite the adversary's efforts to *trail-hide* its existence from the channel (Section 2.2). The verification code and its check is applied to both time-of-flight messages. Both devices first agree on the code's structure as follows.

4.1 Modulation/Verification Code Structure

Code length. The code consists of n positions, α of which have energy, and the remaining $\beta = n - \alpha$ are empty, *i.e.,* absent of pulses (conceptually similar to OOK modulation, where $\alpha = \beta$). The code length affects the performance and security of the presented modulation technique. Larger α and β values improve the security by reducing the probability of adversarial success in mounting undetectable distance-enlargement attack. However, increasing the code length reduces the frequency of conducting two-way ranging. Additionally, the Federal Communications Commission (FCC) imposes restrictions on the number of pulses with energy, effectively limiting α per unit of time. As such, β could be independently increased to compensate for the loss of code length. Setting these parameters is discussed in Section 5.

Pulse phase. The sender uses a random-phase for the α pulses it transmits. Each phase is equally likely. The phase will be irrelevant for the receiver because ED receivers are agnostic to the phase, as explained in Section 2.1. The sender need not share this information with the receiver since the receiver measures the energy, not the polarity of the pulse.

Pulse permutation. The sender and receiver secretly agree on a random permutation of the n positions, obtained from a uniform distribution. Figure 6 shows an example before and after the permutation. The verification code can thus be considered a sequence of $\{-1, 0, 1\}$ pulses, where $\{-1, 1\}$ represent the phase, and $\{0\}$ pulse absence.

Spacing between pulses. The time between two consecutive pulses, T_s, is normally lower bounded by the delay spread of the channel. We submit that T_s should be such that $T_s > 2d/\mathbf{c}$, where d is the distance between the two devices. If the adversary replays the authentic signal delayed by more than the equivalent RTT, the attack will be detected by the mismatch between the measured RTT and the one equivalent to the committed distance. To avoid being detected, the adversary would thus replay its delayed version of a pulse within the T_s time window. As such, authentic pulse i will not overlap with the adversary's delayed version of pulse $i-1$, or any further adversary pulses $i-2$, $i-3$, *etc.*

An example code structure, and adversarial attempts to corrupt and replay it, is shown in Fig. 7.

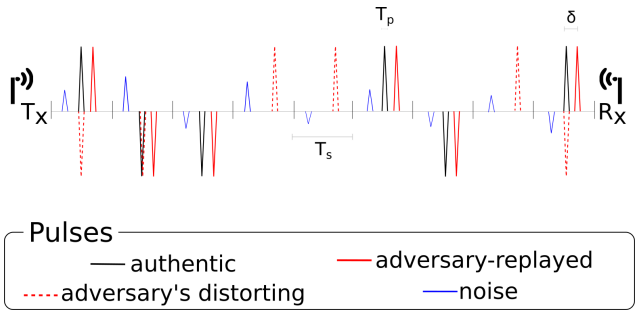


Figure 7: An example verification code of n slots (9 of which are shown), the spacing T_s between consecutive pulses is $1\mu s$ and pulse width T_p is $2ns$. An adversary transmits a pulse to distort the legitimate pulse (dashed red). The adversary also replays the authentic signal with the delay δ (solid red). Best viewed in color.

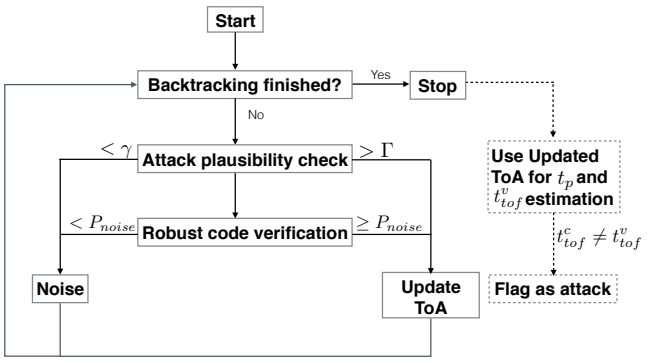


Figure 8: The receiver backtracks to detect enlargement attacks. An event is flagged as an attack when the aggregate energy is higher than Γ (*e.g.,* DoS, jamming), *i.e.,* the data looks more similar to a verification code than noise. The last flagged position is used for the ToF estimation.

4.2 Verification Code Identification

Upon receiving a transmission, the receiver starts processing the code associated with the highest preamble's peak. The code associated with a peak is the train of T_s-spaced pulses that start at a fixed time interval (*e.g.,* agreed upon between the sender and receiver) after the peak. This peak however may not be authentic, and could be the adversary's replayed version. The receiver thus backtracks at fixed time steps corresponding to the pulse width T_p (*e.g.,* 2 *ns*), trying to identify if another version of the code (or a possible distorted imprint of it) was present in the transmission at an earlier time. The receiver does not need to backtrack further beyond some time T_0, knowing the maximum communication range. If the last distance verification occurred recently, the verified range could be used (in combination with the devices' upper bound motion speeds) to reduce the backtracking time.

Backtracking requires the receiver to record transmissions. If an earlier version of the code is found (and their difference exceeds the receiver's standard precision, *e.g.,* ± 10 *cm* for DecaWave [9]), it is used for ToF estimation.

As shown in Fig. 8, the receiver performs Attack Plausibility check and Robust Code Verification to detect attacks until the maximum backtracking time is reached. For each code, the receiver does not look for an exact match of the transmitted pulses in their positions simply because that could be easily bypassed with minimal adversarial efforts (as explained in Section 2.2). Instead, the receiver proceeds as follows. Knowing the mapping of the pulse positions, the receiver distributes the received powers of each pulse among two bins, Bin_α and Bin_β. The former will have the values of the received power (*e.g.,* in Watts) of the energy-present pulse positions, the latter energy-absent positions (Fig. 6).

Attack Plausibility check. For each candidate verification code obtained during backtracking, the overall received signal power (the aggregate of Bin_α and Bin_β) is measured, and compared to a predefined threshold, γ. This threshold is based on the receiver's noise figure. If the aggregate exceeds γ, a potential verification code has been found. Otherwise it gets discarded as noise. The aggregate energy is then compared to another threshold, Γ. This is calculated based on the overall aggregate energy the receiver expects to receive based on the measured distance in the commitment phase, following the path loss model. Artificial distance enlargement caused by the adversary in the commitment phase lowers the receiver's calculated Γ (because of the higher path loss), thus increases the likelihood of the actual received aggregate to exceed Γ. If the aggregate exceeds Γ, an adversary may possibly be injecting energy into the channel to distort the authentic code. If the verification code is neither discarded as noise ($< \gamma$) nor exceeds Γ, the receiver proceeds to the Robust Code Verification check.

Robust Code Verification. Now the receiver checks the verification code content. If the receiver simply flags the presence of one or more pulses (above noise) in Bin_β as an attack, false positives increase because such pulses could occur for many legitimate reasons (*e.g.,* noise spikes, reflections, interfering transmissions, antenna orientation, or multipath).[1] Instead, the receiver performs a sequence of binary hypothesis tests on random pulse samples. It tests if the candidate code is more similar to an authentic code than noise. It chooses $r \leq \alpha$ random pulses from the α in Bin_α (where r is the number of pulses per symbol), aggregates their received powers and compares that to the aggregate of another r pulses randomly chosen from the β in Bin_β. If the aggregate of those selected from Bin_α is larger, the receiver identifies this as a candidate authentic code, and records its ToA. Finally, the distance is calculated based on the recorded ToA of the most recently

[1] If the receiver instead interprets a pulse in Bin_β as an indication that the code is not authentic and continues backtracking, it may very well skip the authentic code thus helping the adversary.

received code, and a mismatch with the committed distance is flagged as an attack.

A candidate verification code could be again noise, which has slipped the Attack Plausibility check perhaps due to some sporadic noise spikes in the transmission. Noise has a probability of $\leq P_{\text{noise}}$ to satisfy the Robust Code Verification check, where P_{noise} is derived as (32) in Section 5.1.4. As such, the receiver estimates the probability that the above condition is satisfied. This is done by repeating the random sampling υ times, and checking if the ratio of the number of times the condition is satisfied to υ exceeds P_{noise}. This would indicate the code is not noise, and is either authentic or adversary-replayed. Regardless, the receiver uses the ToA of the most recent code found.

4.3 Setting the Energy Thresholds.

Setting the upper-bound threshold, Γ. To set Γ, the receiver relies on the committed (unverified) distance between itself and the sender. This dictates the path loss—the amount of power loss per pulse as pulses propagate the medium. Larger committed distance causes the receiver to expect less power, thus setting a lower Γ. Thus, by increasing the committed distance, the adversary helps divulge its malice.

The path loss function $f()$ for outdoor UWB LoS is [20]:

$$f(d) = PL_0 + 10 \cdot n \cdot log\left(\frac{d}{d_0}\right) \tag{3}$$

where d is the distance in meters, and PL_0 is a constant representing the path loss at the reference distance d_0. For UWB LoS channel model, these constants are set to [20]:

$$f(d) = -46.3 - 20\ log(d) - log\left(\frac{6.5}{5}\right) \tag{4}$$

This is calculated in the standard signal ratio unit, dB, where:

$$\text{Power ratio (in } dB) = 10\ log\ (\text{ratio}) \tag{5}$$

The path loss function thus expresses the power loss as

$$f(d) = 10\ log\left(\frac{(\lambda_b)^2}{(\lambda_{\text{sent}})^2}\right) \tag{6}$$

or

$$\frac{(\lambda_b)^2}{(\lambda_{\text{sent}})^2} = 10^{f(x)/10} \tag{7}$$

where $(\lambda_b)^2$ is the pulse instantaneous power the receiver *expects*, and $(\lambda_{\text{sent}})^2$ is that the sender has actually sent, *e.g.,* both in Watt. Knowing the constant pulse power of the sender, then the pulse power is expected to be received as:

$$(\lambda_b)^2 = (\lambda_{\text{sent}})^2\ 10^{f(x)/10} \tag{8}$$

The receiver then calculates Γ as follows:

$$\Gamma = \alpha\ (\lambda_b + N)^2 + \beta\ (N)^2 \tag{9}$$

where d is the (unverified) distance in meters between the sender and receiver obtained at commit stage, either true or artificially enlarged in case of an attack. N is an instantiation of zero-mean Gaussian noise at the receiver, *i.e.,* the noise present in the receiver's channel and cannot be removed [19].

There are other factors that contribute to the degradation of power. These factors could cause further power loss E, typically up to $E = -8\ dB$ more [17, 21]. If the receiver sets Γ as that after the expected further degradation (*i.e.,* too small Γ value), false positives may increase because such additional signal-degradation factors may or may not occur—if they do not, the receiver would then falsely assume such relatively "too high" aggregate energy is due to an attempted attack. Accordingly, the receiver sets Γ based only on the (almost certain) path loss deterioration. Any further power loss would then be added benefit to the adversary, as it allows the adversary to inject more pulses into the channel to corrupt the authentic code without exceeding Γ.

Setting the lower-bound threshold, γ. If the aggregate energy is $< \gamma$, it would be either due to noise or a substantial deterioration of the authentic signal where no meaningful information could be recovered during the Robust Code Verification. Too high γ leads to false negatives; too low triggers Robust Code Verification even for noise. For critical applications seeking to prevent false negatives, γ could be set conservatively based on the receiver's noise variance σ_N^2:

$$\gamma = (\alpha + \beta) \cdot \sigma_N^2 \tag{10}$$

4.4 Attack Resilience

Here we explain how UWB-ED resists standard enlargement attacks. More complex attacks are discussed in Section 6.

4.4.1 Detecting Signal Replay

An adversary that simply replays authentic pulses does not win because the receiver backtracks to detect earlier copies of the code. UWB-ED provides resilience to benign signal distortion, *e.g.,* due to channel conditions or antenna orientation, because the receiver looks for similarities between the code and the received signal (versus exact data match), allowing for a higher bit error rate. In general, poor channel conditions (low SNR) can be compensated for by increasing the symbol length, r, minimizing the bit error rate.

4.4.2 Complicating Signal Annihilation

The unpredictability of the pulse phase means an adversary must either wait to detect it and immediately inject the reciprocal pulse for annihilation, or inject a random-phased pulse hoping it is the reciprocal. The former is infeasible in practice for UWB (see Section 3). The latter results in amplifying or annihilating the authentic pulse, each with a 50% chance. Amplification is unfortunate to the adversary, as the adversary

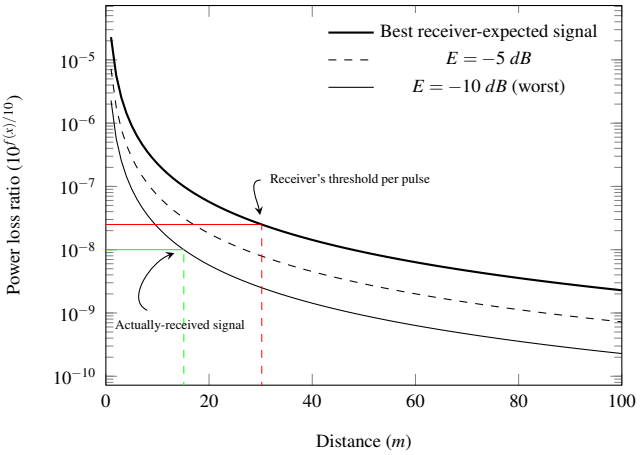


Figure 9: The best expected signal power as calculated by the receiver using the path loss function in (4), the signal at $E = -5\ db$ of further power loss, and at $E = -10\ db$ (worst expected). If the distance is $D_1 = 15.11\ m$ (green line), and the adversary doubles it, *i.e.,* by adding $D_2 = 15.11\ m$ to make it $D_1 + D_2 = 30.22\ m$ (red line), the receiver will set the threshold following the fake distance, at $10^{f(D_1+D_2)/10} = 10^{-7.6}$. The adversary's room is the difference between the red and green lines on the y-axis. At $D_2 = 32.68\ m$, the adversary has no room. Best viewed in color.

now needs to compensate with an equivalent amplitude, A. Amplification doubles the amplitude. The estimated energy of the pulses will thus amount to $\sim A^2$, and the adversary-contributed amplification to $\sim (2A)^2$.

Since the result is indeterministic for the adversary, it leads us to the next discussion: how successful would the adversary be in "contaminating the evidence" that an authentic verification code existed, and how much energy room does the adversary have to do that before exceeding Γ?

4.4.3 Mitigating Evidence Contamination

To hide the authentic code, the adversary tries to inject energy into the channel, hoping it annihilates as many of Bin_α pulses as possible. We thus calculate the room available to the adversary here, and use that to derive the probability of adversarial success in distance enlargement in Section 5.

Figure 9 shows the path loss function in (7) as used by the receiver to detect the threshold Γ, as well as the worst receiver-expected signal after additional deterioration. The receiver sets the threshold based on the best expected signal. The room available for the adversary to add energy depends on the actual signal received. The most favorable situation to the adversary is when the received signal power is the worst (lowest E), which allows the adversary to inject pulses without exceeding Γ. For example, in Fig. 9, if the actual distance between the sender and receiver is $D_1 = 15.11\ m$ (green line), and the

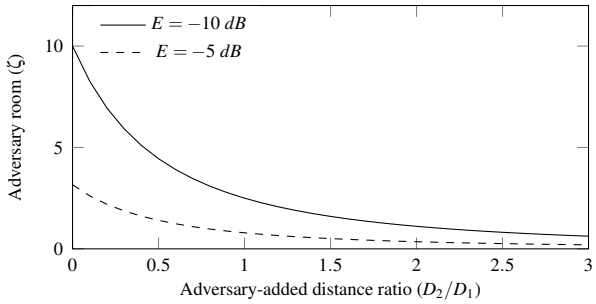


Figure 10: Adversary's room to add energy, ζ in (12), against the ratio of the adversary-added to true distance (D_2/D_1); E represents additional signal degradation beyond path loss.

adversary is trying to add $D_2 = 32.68\ m$ to make the distance $D_1 + D_2 = 47.79\ m$ (red line), the receiver will set Γ using the fake distance, $D_1 + D_2$. At such a relatively large added distance, D_2, the received pulse power is unlikely to fall below $f(D_1) + E = 10^{-8}(\lambda_{\text{sent}})^2$ at, *e.g.,* $E = -10\ dB$. The room available to the adversary to inject energy becomes too small, significantly reducing its chances of success.

The room-per-pulse, R, available to the adversary to enlarge the distance thus lies in-between the received signal and Γ, and is calculated in dB as:

$$R = f(D_1 + D_2) - (f(D_1) + E) \tag{11}$$

where E represents other channel degrading factors, and the distances D_1 and D_2 (in meters) are respectively the true distance between both devices, and the extra distance the adversary intends to add. This room is thus expressed as:

$$\zeta = 10^{R/10} \tag{12}$$

Figure 10 plots ζ at various distance ratios D_2/D_1.

Recall that the adversary may succeed to annihilate some of the pulses falling in Bin_α. But since Bin_β in the authentic code have nothing but noise, adding pulses into those will result in an increase in the overall aggregate energy. As such, this available energy room in (11) by itself does not give a perfect indication to the adversary's chances of success.

4.5 A Numerical Example

Figure 11 shows an example verification code, expanded from Fig. 6, where the adversary injects $k = 10$ random-phased pulses. For simplicity, the figure assumes $N = 0$. If the distance between the sender and receiver is $D_1 = 4\ m$, and the adversary is trying to enlarge it by $D_2 = 4.5\ m$ to make it $D_1 + D_2 = 8.5\ m$, and assuming $(\lambda_{\text{sent}})^2 = 7.6\ \mu W$, then the receiver expects a best case received power of:

$$\begin{aligned}(\lambda_b)^2 &= (\lambda_{\text{sent}})^2\ 10^{f(D_1+D_2)/10} \\ &= 7.67 \times 10^{f(8.5)/10} = 2.4\ \mu W\end{aligned} \tag{13}$$

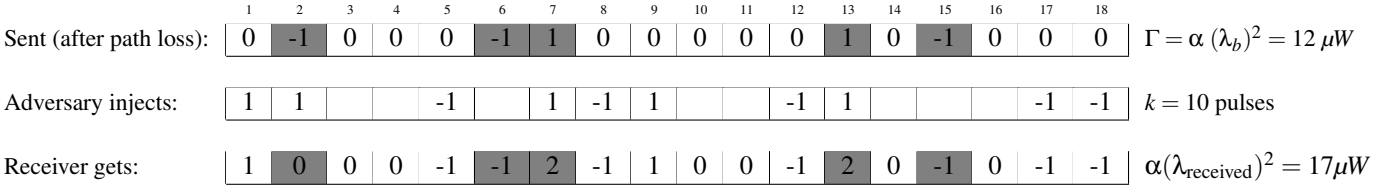


Figure 11: An example of the random-phased Bin_α pulses (dark gray) reordered following the permutation in Fig. 6. After the adversary injects $k = 10$ random-phased pulses at random positions, the receiver will get the summation at each pulse position.

From (10) at $N = 0$ and $\alpha = 5$ (as in Fig. 11), it then calculates the threshold as:

$$\Gamma = \alpha\ (\lambda_b)^2 = 12\ \mu W \tag{14}$$

At $E = -10\ dB$, the actual signals are received as:

$$(\lambda_w)^2 = (\lambda_{\text{sent}})^2\ 10^{(f(D_1)+E)/10} \approx 1\ \mu W \tag{15}$$

Now assuming the adversary is $D_3 = 6\ m$ away from the receiver, and uses a random-phased pulse with transmission power of $(\lambda_{\text{sent}}^{\text{adversary}})^2 = 15.77\ \mu W$. At $E = -10\ dB$, the receiver would receive the adversary's signals as:

$$(\lambda')^2 = (\lambda_{\text{sent}}^{\text{adversary}})^2\ 10^{(f(D_3)+E)/10} \approx 1\ \mu W \tag{16}$$

So in the best case for the adversary, where the signal is highly deteriorated, the adversary would then have a per-pulse room of $R = 3.45\ dB$ to add energy, which amounts to $7\ \mu W$ more, *i.e.,* up to $\Gamma = 12\mu W$. In Fig. 11, after the adversary injects its $k = 10$ pulses at the example random positions and with the random phases shown, it results in annihilating a single pulse (at position 2), amplifying two pulses (at positions 7 and 13), and adding seven more 1 μW pulses for an increase of the overall aggregate to be 17 μW. This exceeds $\Gamma = 12\ \mu W$, and this attack would thus be detected.

5 Evaluation

We evaluate UWB-ED by deriving the probability of success for an adversary enlarging the distance. We also validate that model using simulations in Section 5.2.

5.1 Probability of a Successful Attack

The adversary hides the authentic code by having the aggregate of the r pulses that the receiver chooses from Bin_β exceed Bin_α. The adversary must also avoid injecting too much energy to not exceed Γ. Not knowing which pulse belongs to which bin, the adversary injects k pulses at random positions thus affecting k of the n pulses in the code.

To that end, the probability of mounting a successful attack, P_{sa}, is the intersection of the probability of two events (the checks in Fig. 8): the aggregate of the energy pulses chosen from Bin_β ($b\beta$) exceeds that of Bin_α ($b\alpha$), and the added energy is $\leq \Gamma$:

$$P_{\text{sa}}(\alpha,\beta,r,\Gamma,k) = P_{b\beta>b\alpha}(\alpha,\beta,r,k) \cap P_{\leq\Gamma}(\alpha,\beta,k) \tag{17}$$

5.1.1 Probability of successfully evading the Robust Code Verification check ($P_{b\beta>b\alpha}$)

To evade this, the adversary must have an energy aggregated from Bin_β exceed Bin_α. When the adversary injects k pulses into the channel, x will fall into Bin_α, and the remaining $k-x$ into Bin_β. $P_{b\beta>b\alpha}$ is then the probability of this distribution occurring multiplied by the probability of the attack succeeding under this distribution, for all possible such distributions $0 \leq x \leq \alpha$ and $0 \leq k-x \leq \beta$. To calculate the probability of the distribution occurring, consider the general case of a bucket containing two types of objects (*e.g.,* colored pearls): I of the first type, and J of the second. If ψ objects are selected at random, the probability that i and j of the ψ are respectively of the first and second type ($i + j = \psi$) is:

$$\frac{\binom{I}{i}\ \binom{J}{j}}{\binom{I+J}{i+j}} \tag{18}$$

where $\binom{n}{r}$ denotes n choose r and is given by:

$$\binom{n}{r} = \begin{cases} \dfrac{n!}{r!(n-r)!}, & 0 \leq r \leq n \\ 0, & \text{otherwise} \end{cases}$$

Similarly, the probability that x and $k-x$ of the adversary's k pulses respectively affect the α in Bin_α and β in Bin_β is:

$$\frac{\binom{\alpha}{x}\ \binom{\beta}{k-x}}{\binom{\alpha+\beta}{k}}$$

For all possible such distributions, we have:

$$P_{b\beta>b\alpha}(\alpha,\beta,r,k) = \sum_{x=0}^{\alpha} \left(p_{\alpha,\beta,r,k}(x) \cdot \frac{\binom{\alpha}{x}\ \binom{\beta}{k-x}}{\binom{\alpha+\beta}{k}} \right) \tag{19}$$

where $p_{\alpha,\beta,r,k}(x)$ is the probability $b\beta > b\alpha$ given the adversary affected x and $k-x$ pulses in Bin_α and Bin_β respectively.

To derive $p_{\alpha,\beta,r,k}(x)$, we assume for simplicity a unity power-per pulse, *i.e.,* the sender's and the adversary's pulses

reach the receiver after path loss and other factors at a constant energy of $\pm 1\mu W$.[2] This is similar to the example given in Fig. 11. Every adversary-added pulse in Bin_β will result in a 1 μW of added energy from the receiver's point of view since the receiver's aggregation is agnostic to a pulse's phase. For Bin_α, after the adversary affects x pulses, some will be annihilated while others will be amplified. From the receiver's point of view, after the adversary's pulses are injected, Bin_α will have a mix of $2^2 = 4\mu W$ and 0 μW (adversary-affected) pulses, as well as the original 1 μW unaffected pulses.

More 0 μW (annihilated) pulses in Bin_α raises the chances that $b\beta > b\alpha$, which is in the adversary's favor. Since every affected pulse in Bin_α will either result in a 0 μW or a 4 μW pulse, there are 2^x possible outcomes. Of those, there are $\binom{x}{g}$ ways that g 0 μW pulses will occur. The probability that the x adversary-injected pulses that fell in Bin_α result in a annihilation of g pulses is thus $\binom{x}{g}/(2^x)$. For all possible numbers of annihilated pulses $0 \le g \le x$, the adversarial success probability in the event that x fell in Bin_α is:

$$p_{\alpha,\beta,r,k}(x) = \sum_{g=0}^{x} \left(p_{\alpha,\beta,r,k,x}(g) \cdot \frac{\binom{x}{g}}{2^x} \right) \tag{20}$$

where $p_{\alpha,\beta,r,k,x}(g)$ is the probability $b\beta > b\alpha$ given g annihilated pulses in Bin_α.

When Bin_α has g annihilated (0 μW), $x-g$ amplified (4 μW), and $\alpha - x$ unaffected pulses (1 μW), the probability of $b\beta > b\alpha$ in the event x fell in Bin_α, and g of the x pulses were annihilated is the probability that an aggregate of $m-1$ is chosen from Bin_α and an aggregate of $\ge m$ is chosen from Bin_β. For each possible $0 \le y_1, y_2 \le r$, we have:

$$p_{\alpha,\beta,r,k,x}(g) = \sum_{y_1=0}^{r} \sum_{y_2=0}^{r} \left(\frac{\binom{g}{y_1}\binom{x-g}{y_2}\binom{\alpha-x}{r-y_1-y_2}}{\binom{\alpha}{r}} \cdot \sum_{i=m}^{r} \frac{\binom{k-x}{i}\binom{\beta-(k-x)}{r-i}}{\binom{\beta}{r}} \right) \tag{21}$$

where m is:

$$\begin{aligned} m &= 0^2 \times y_1 + 2^2 \times y_2 + 1^2 \times (r - (y_1 + y_2)) + 1 \\ &= r - y_1 + 3y_2 + 1 \end{aligned} \tag{22}$$

At $r = \alpha$ (*i.e.,* selecting all Bin_α pulses) and $\alpha \le \beta$, we get:

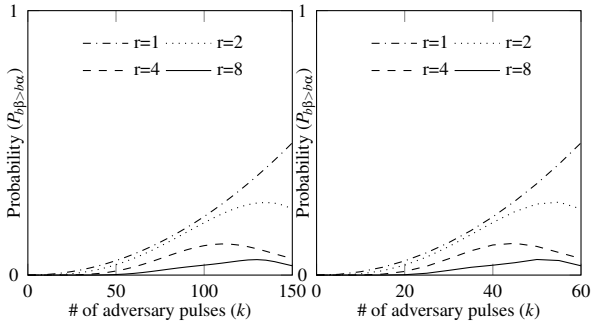
$$p_{\alpha,\beta,r,k,x}(g) = \sum_{i=m'}^{r} \frac{\binom{k-x}{i}\binom{\beta-(k-x)}{r-i}}{\binom{\beta}{r}} \tag{23}$$

where m' is:

$$\begin{aligned} m' &= 2^2 \times (x-g) + 1^2 \times (\alpha - x) + 1 \\ &= 4(x-g) + (\alpha - x) + 1 \end{aligned} \tag{24}$$

Figure 12 plots $P_{b\beta>b\alpha}$, where $\alpha = 50$. From these results, increasing β is not necessarily effective for the Robust Code Verification check to detect attacks, since the adversary maintains its success probability by increasing k proportionally; there is a visually similar pattern of adversarial success probability in both Fig. 12a and 12b. As such, the advantage of the empty pulses in Bin_β does not quite manifest in the Robust Code Verification check, rather the Attack Plausibility check.

[2] Analogous analysis applies for non-constant energy.



(a) $\beta = 100$ (b) $\beta = 10$

Figure 12: Probability that the Robust Code Verification check fails to detect the adversary's attack, plotted using (19) in Section 5.1.1, at $\alpha = 50$ and $0 \le k \le \alpha + \beta$.

Another observation is that higher r lowers the adversary's success probability. For example at $\beta = 100$ (Fig. 12a), the adversary has a 27% chance at $r = 2$ (which occurs at $k = 135$), versus 5.85% at $r = 8$ (at $k = 130$). In Section 5.1.3, we show that at $r = \alpha$, we get the optimal security results.

5.1.2 Final Probability of Adversary's Success

In (17), the event that the aggregate energy after the adversary's pulses is $\le \Gamma$ and the event that $b\beta > b\alpha$ are dependent, and thus their intersection is not their product. Recall that in (20), g is the number of annihilated pulses, $x-g$ is the number of amplified pulses in Bin_α, and $k-x$ is the number of added pulses in Bin_β. The aggregate-energy does not exceed Γ when the adversary's pulses satisfy the inequality:

$$\begin{aligned} &(k-x)\ (\lambda' + N)^2 + (x-g)\ (\lambda' + \lambda_w + N)^2 + \\ &(\alpha - x)\ (\lambda_w + N)^2 + (\beta - (k-x) + g)\ (N)^2 \le \Gamma \end{aligned} \tag{25}$$

where λ' is defined as in (16), and Γ in (10).

If the adversary uses a variable pulse power randomly chosen from a distribution with a mean much different from λ_w, authentic pulses colliding with their reciprocal will not be fully annihilated. The adversary thus sets its power such that its mean at the receiver matches the sender, *i.e.,* $\overline{(\lambda')^2} = (\lambda_w)^2$. Assuming $(\lambda_w)^2 = (\lambda')^2$ in (25), we get:

$$k + 2x - 4d + \alpha \le \frac{\alpha\, \lambda_b^2 - \varepsilon}{\lambda_w^2} \tag{26}$$

where ε is a representation of noise, and evaluates to:

$$\varepsilon = N\ (\lambda_w\ (2k + 2\alpha - 4g) - \lambda_b(2\alpha))$$

As $\varepsilon \to 0$, (26) becomes:

$$k + 2x - 4d \leq \alpha \left(\frac{\lambda_b^2}{\lambda_w^2} - 1 \right) \tag{27}$$

From (13) and (15), we have:

$$\begin{aligned} \frac{\lambda_b^2}{\lambda_w^2} &= \frac{(\lambda_{\text{sent}})^2 \ 10^{f(D_1+D_2)/10}}{(\lambda_{\text{sent}})^2 \ 10^{(f(D_1)+E))/10}} \\ &= 10^{(f(D_1+D_2)-(f(D_1)+E))/10} \\ &= \zeta \end{aligned} \tag{28}$$

where ζ, from (12), represents the room-per-pulse available to the adversary to add energy into the channel.

We now calculate $p_{\alpha,\beta,r,k}(x,\Gamma)$, similar to (20) as:

$$p_{\alpha,\beta,r,k}(x,\Gamma) = \sum_{g=0}^{x} \left(p_{\alpha,\beta,r,k,x,\Gamma}(g) \cdot \frac{\binom{x}{g}}{2^x} \right) \tag{29}$$

such that

$$p_{\alpha,\beta,r,k,x,\Gamma}(g) = \begin{cases} p_{\alpha,\beta,r,k,x}(g), & k + 2x - 4d \leq \alpha(\zeta - 1) \\ 0, & \text{otherwise} \end{cases} \tag{30}$$

Using (29), the final adversarial success probability is:

$$P_{\text{sa}}(\alpha,\beta,r,\Gamma,k) = \sum_{x=0}^{\alpha} \left(p_{\alpha,\beta,r,k}(x,\Gamma) \cdot \frac{\binom{\alpha}{x}\binom{\beta}{k-x}}{\binom{\alpha+\beta}{k}} \right) \tag{31}$$

Figures 13a and 13b plot P_{sa} in (31). At $\zeta = 20$, Γ is too high to reduce P_{sa}, but the Robust Code Verification check enables the receiver to limit it to $P_{\text{sa}} < 0.16 \times 10^{-3}$. At $\zeta = 10$, P_{sa} stops growing beyond 0.73×10^{-4}, which limits the adversary's pulses to $k = 495$ for its highest success chance.

Figure 13c shows the effect of β on P_{sa}; P_{sa} is almost constant with β, at around 0.2×10^{-3}, and only starts dropping when β is sufficiently large so that the aggregate energy after the adversary's pulses exceeds Γ. At a certain point, increasing β no longer helps. For example, at $\zeta = 5$ and $\beta \geq 400$, $P_{\text{sa}} \approx 0$. β should thus be set wisely, reflecting the application's sensitivity to distance increases and channel conditions, to avoid increasing transmission lengths unnecessarily.

5.1.3 Symbol length (r)

Figures 13d and 13e plot P_{sa} against the ratio of $r : \alpha$. As shown, longer symbol length (larger r) is better for security; the best results are achieved when the ratio is 1 ($r = \alpha$).

5.1.4 False positives: noise passing Robust Code Verification

Higher-than-usual noise in the channel might satisfy the Robust Code Verification check. Since the receiver backtracks, it is imperative to calculate the probability, P_{noise}, that noise in the channel satisfies that check. Unlike the adversary's pulses targeted to alter the authentic code, such a candidate trail of noise pulses does not get added to the sender's code because they are at different positions. Without loss of generality, we can separate the noise-intervals in low-energy and high-energy, *e.g.,* across the median of the distribution of N^2. We refer to the number of high-energy intervals as κ. The probability that noise satisfies the Robust Code Verification check is the probability that x of κ pulses fell into Bin_α, by the probability of satisfying the test in that event, $p'_{\alpha,r}(x)$:

$$P_{\text{noise}}(\alpha,\beta,r,\kappa) = \sum_{x=0}^{\alpha} \left(p'_{\alpha,r}(x) \cdot \frac{\binom{\alpha}{x}\binom{\beta}{\kappa-x}}{\binom{\alpha+\beta}{\kappa}} \right) \tag{32}$$

where,

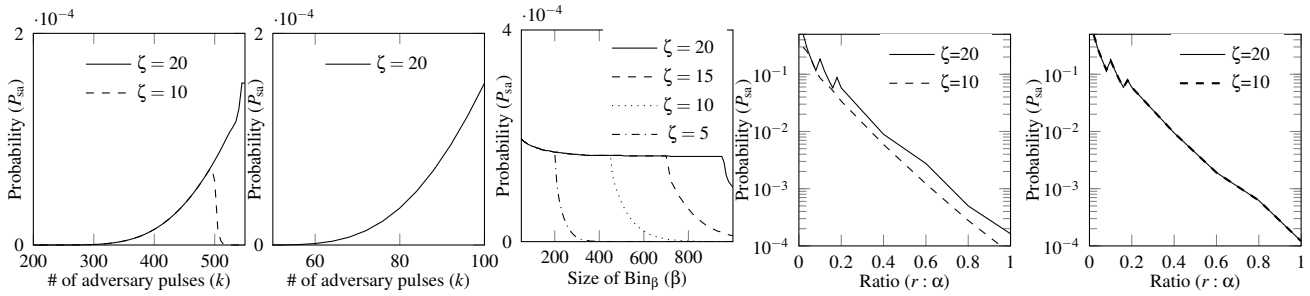
$$p'_{\alpha,r}(x) = \sum_{y=0}^{r} \left(\frac{\binom{\alpha-x}{r-y}\binom{x}{y}}{\binom{\alpha}{r}} \cdot \sum_{i=0}^{y} \frac{\binom{\beta-(\kappa-x)}{r-i}\binom{\kappa-x}{i}}{\binom{\alpha}{r}} \right) \tag{33}$$

This is the probability that an aggregate of y is chosen from Bin_α, and of $\leq y$ from Bin_β. Since we separate along the median, the expected κ is $(\alpha+\beta)/2$. Figure 14 plots P_{noise} against α using (32) at $\kappa = (\alpha+\beta)/2$ and $\beta = 100$. Intuitively (and as the chart confirms), $P_{\text{noise}} \longrightarrow 0.5$ as $\alpha \longrightarrow \infty$.

Since a candidate verification code is discarded as noise if the Robust Code Verification check is satisfied with a probability $< P_{\text{noise}}$ (recall: Fig. 8), the adversary must have a success probability of at least $1 - P_{\text{noise}}$ to hide the authentic code from the receiver. At $r = \alpha$, $P_{\text{noise}}(80,100,80,40) = 0.53$, and the adversary must thus have a success probability of at least 0.47. As this is much higher than the calculated probabilities in Section 5.1.2, the adversary will not be able to disguise authentic code as noise. The value 0.53 is a lower-bound; in practice P_{noise} should be set ≥ 0.53 depending on applications' requirements and channel conditions.

5.2 Validating the Probabilistic Model

The use of prototype implementation using Software Defined Radios (SDRs) and simulations are well-established methods for evaluating wireless systems. Existing SDRs do not support UWB. Therefore, we validate the probabilistic model above with simulations. The channel condition such as noise, multipath effect, and path loss are important factors to consider while designing a wireless system. The IEEE 802.14.4a [18] channel model for different environments is purposefully provided for UWB. The preamble and the verification code are converted into physical layer signals using this model for the outdoor LoS conditions. The model generates the pulse and multipath components to resemble the real world effect of the channel condition. We assume that upper layers, *e.g.,* Medium Access Control (MAC) layer, could decide on when to perform enlargement detection so that it doesn't interfere with



(a) $\beta = 500$; $r = \alpha = 50$. (b) $\beta = 50$; $r = \alpha = 50$. (c) $r = \alpha = 50$. (d) $\alpha = 50$ and $\beta = 500$ (e) $\alpha = 50$ and $\beta = 50$

Figure 13: Adversarial success probability in (31).

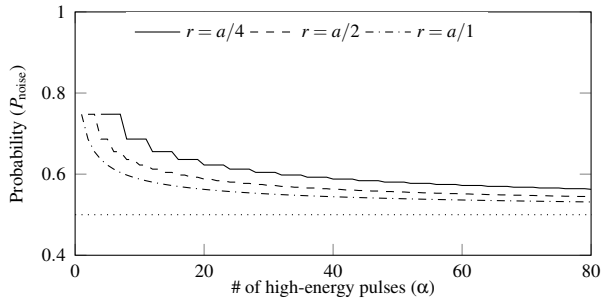


Figure 14: Probability that noise passes the Robust Code Verification check, calculated using (32); $\kappa = \alpha/2$, $\beta = 100$.

other ranging applications. The simulations account for the noise and interference due to the noise figure of the receiver and multipath components. To verify the simulation setup, we performed a thorough evaluation to cross-check simulation metrics with previous proof-of-concept implementation [26]. Each pulse uses 500 MHz bandwidth, and the sampling time between consecutive pulses is 1 μs. Transmission power is limited to -35 dBm/MHz, well under the limits applied by the FCC/ETSI regulations [11]. The energy is further reduced to adapt to path loss model and extra losses (E; *cf.* Fig. 9).

An adversary is simulated to inject k signals to annihilate or distort the authentic code, and to replay a delayed and amplified versions of the authentic signals. Similar to our assumptions, the adversary in the simulator is capable of annihilating the pulse and its multipath if the phase is guessed correctly; it doubles the amplitude of the pulse otherwise. The time difference between authentic and delayed signals is $\delta = 200ns$ in the simulations (see Fig. 7).

Before demodulation, additive white Gaussian noise (AWGN) is added to the signal. The receiver in Section 2.1 is implemented for code verification; it always locks on to the highest peak, *i.e.,* the peak generated by the adversary due to its replay attack. The communication range is considered $100m$, and the backtracking restricted to $660ns$.

The goal of our validation is to (1) confirm the probabilistic model's correctness, and (2) analyze the effect of the parameters abstracted from the model, namely noise and the receiver's ability to reconstruct the signal after long distance propagation. In practice, the latter point can be accounted for by increasing the number of pulses ($n = \alpha + \beta$)—see below.

Validating $P_{b\beta>b\alpha}$. Figure 15 shows the validation for $P_{b\beta>b\alpha}$, at a simulated distance between both devices of $d = 10m$. A boxplot is drawn at distinct k, where each scenario is run 10^6 times. The results confirm that abstracting noise from the model does not largely affect its accuracy. Next we show the effect of longer distances on the model.

Validating P_{sa}. Figure 16 shows the validation for P_{sa}, at $r = \alpha$ and $P_{noise} = 0.8$. Results are shown for different k, at distances of $10m$ and $100m$. Each scenario is run 10^6 times, and P_{sa} is calculated as the proportion of these where the adversary succeeded to hide the authentic code. Again the results show comparable patterns between the model and simulations. There is a slight horizontal shift at k due to the abstracted noise. In the simulator, Γ is set as in (9), which may be a bit too high or low depending on actual noise patterns. In Fig. 16a, Γ was relatively low, causing a drop in the simulated P_{sa} at smaller k compared to the model. In Fig. 16b, Γ was relatively high, replicating P_{sa} at higher k.

Another difference between simulations and the model manifests with increasing the distance d between both devices. In practice, in UWB, receivers increase their ability to reconstruct the signals (hence, the SNR) by aggregating over more pulses. We noticed that the model provides such comparable probability patters when we decrease α and β in the model proportionally with increasing d in simulations. For example in Fig. 16b where $d = 100m$, α and β in the simulator had to be increased from 15 and 158 to 50 and 500 respectively ($\sim$ tripled) to account for the increased distance.

Validating the false positives. We also used simulations to confirm that noise would not be falsely mistaken for authentic code upon proper selection of P_{noise} and Γ. For various distances between $10m$ and $100m$, the probability of a false positive was $\sim 1 \times 10^{-6}$, confirming the noise analysis in

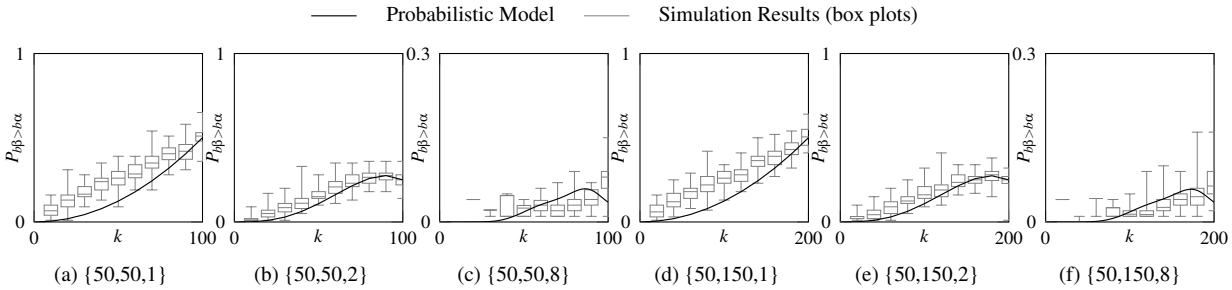


Figure 15: Probability of adversary's failure calculated using (19), and simulations results validating the probabilistic derivations. Each scenario is run with the $\{\alpha, \beta, r\}$ parameters shown in the charts' individual captions.

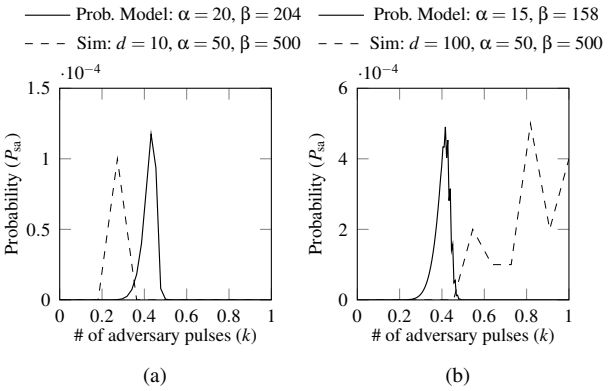


Figure 16: The attack is detected when the aggregate energy is between γ and Γ, but $P_{b\beta>b\alpha}$ is more than P_{noise}. The attack is also detected when energy aggregate is more than Γ; $\zeta = 5$.

Section 5.1.4.

In conclusion, the simulated probabilities follow comparable patterns with the model, and are in the same range. The model derived herein thus serves as a formal means for evaluating the efficacy and suitability of UWB-ED in practice. The results also show that the channel condition, such as path loss, noise, and interference due to multipath components, does not affect the performance and security of the system. An adversary can increase the noise level, which can increase false positives. High false positives may eventually cause DoS (which the adversary can mount anyway by jamming the channel), but the adversary remains unable to enlarge distances.

6 Discussion

Adaptive attacks. An adversary can notice the effect of each of its added pulses on the resultant energy, whether annihilated or amplified. It can then adapt its attack strategy by dynamically deciding k based on the number of pulses it has added/annihilated so far during the transmission. The adversary can then utilize its knowledge of n, α and β in order to, not only decide the optimal value of k statically before the transmission begins, but also adjust their distribution in realtime. This attack does not succeed because the adversary cannot control the resultant pulse phase. Injecting excessive energy in Bin_β exceeds Γ; injecting in Bin_α does not guarantee annihilation because of the unpredictable phase.

Varying energy levels. To achieve perfect signal annihilation, an adversary uses the same amplitude expected at the receiver. Instead of injecting k pulses each with a constant energy of, *e.g.,* $2\mu W$, the adversary can inject one pulse with an energy of, *e.g.,* $2k\mu W$. If all k pulses fell in Bin_β, the aggregate energy would be the same as when that single high-energy pulse also falls in Bin_β. However, intuitively, the adversary is better off injecting multiple pulses with constant energies for two reasons. First, multiple pulses in Bin_β have higher chances of being selected than a single pulse, thus evading the Robust Code Verification check. Second, for those that fall in Bin_α, any leftover energy after annihilating a pulse, regardless of the phase, will be counted towards the overall aggregate, thus hurts the adversary's cause.

Influencing Γ through distance shortening. Instead of enlarging distances directly, the adversary can first mount a distance-reduction attack to trick the devices into using higher Γ (recall: smaller signal attenuation due to shorter path loss leads to higher Γ calibration). It is thus imperative to complement UWB-ED with a distance-reduction detection [5, 6, 26]. Devices should alternate between both techniques; *e.g.,* if distances of d_1 and d_2 are verified using respectively UWB-ED and a distance-reduction detection technique, it should be concluded that the actual distance, d, is in the range $d_1 \leq d \leq d_2$ (d_1 is a lower bound, d_2 an upper).

Influencing the number of pulses, n. An adversary can inject a low stream of noise-like energy, not too high to be detected as jamming. However because Γ is set beforehand, it is not influenced by the adversary. By injecting noise, the adversary actually hurts its own cause as it reduces the amount of energy it can use strategically to prevent code detection.

Integrating UWB-ED with 802.15.4z and 5G. The 802.15.4z enhanced impulse radio task group is defining a

series of physical layer improvements to provide secure and precise ranging [2]. Those include additional coding, preambles, and improvement to existing modulations to increase ranging integrity and accuracy. UWB-ED is a potential candidate for enlargement detection in 802.15.4z. It adheres to the low pulse repetition (LRF) mode frequency (1-2 MHz), works with non-coherent receivers, and supports up to $100m$.

The 3GPP technical specifications groups are designing the 5G-new radio technology, and it aims to include secure and precise ranging based on wireless signals [16,33]. Properties such as high carrier frequencies, large bandwidths, large antenna arrays, device-to-device communication, and ultra-dense networking will help attain this objective. It is early to say the exact modulation techniques 5G will use for distance measurement, but it is safe to assume that wideband will be used to attain position accuracy; beamforming techniques will achieve long distances. This system is equivalent to setting $r = 1$ herein without restrictions on α, as transmission power restrictions imposed on UWB do not apply to 5G. However, the security of 5G can be increased further, as it allows for the use of beamforming and coherent receivers.

7 Related Work

Detecting enlargement attacks has lately been a prominent research area. Previous literature explored timing acquisition at the preamble, and data ambiguity at payload. Taponecco *et al.* [27] show that the success of enlargement attacks using replay (or overshadowing) depends on the amount of delay the adversary introduces. Such success is harder for controllable attacks, where the adversary is required to position nodes at specific locations. Compagno *et al.* [8] provide a probabilistic model for the success of overshadowing attacks, which captures different channel conditions and leading edge detection techniques for ToA estimation. None of the above efforts considered adversarial signal annihilation.

Tippenhauer *et al.* [29] explored a theoretical approach to detect adversarial signal annihilation for distance enlargement: using a single pulse-per-symbol (consecutive integration windows represent a symbol). They found that modulation with a 2ns slot size, *i.e.,* mostly equivalent to a pulse width, might help detect signal annihilation. This, however limits the ranging technique to short distances. The effect of multipath on that scheme in practice is also unclear, since reflected signals would directly interfere with authentic ones causing distortion (no empty gaps between authentic pulses). In contrast, UWB-ED allows for increased distances by increasing the symbol length, and the sampling time between consecutive pulses is sufficient to handle the multipath effect.

8 Conclusion

We present UWB-ED—the first known technique to detect distance-enlargement attacks against standard UWB ranging systems. UWB-ED is readily deployable for current off-the-shelf receivers, requiring no additional infrastructure. Evaluation is performed by deriving the probability of adversarial success in mounting distance enlargement attacks. Results show that the verification code structure herein prevents signal annihilation. The code also allows the use of longer symbol length at the receiver, which is essential to achieve longer distance in the energy constrained UWB system. UWB-ED is thus a good candidate for enlargement detection in practice (*e.g.,* for 802.15.4z and 5G).

References

[1] 3db. 3db Access AG - 3DB6830 ("proximity based access control"). https://www.3db-access.com/Product.3.html. [Online; Accessed 22. October 2018].

[2] Task Group 4z. IEEE 802.15 WPAN "enhanced impulse radio". http://www.ieee802.org/15/pub/TG4z.html. [Online; Accessed 22. October 2018].

[3] P. Bahl and V. N. Padmanabhan. RADAR: an in-building RF-based user location and tracking system. In *IEEE INFOCOM*, volume 2, pages 775–784, 2000.

[4] K. Bauer, D. McCoy, E. Anderson, M. Breitenbach, G. Grudic, D. Grunwald, and D. Sicker. The Directional Attack on Wireless Localization -or- How to Spoof Your Location with a Tin Can. In *IEEE GLOBECOM*, pages 1–6, 2009.

[5] Ioana Boureanu, Aikaterini Mitrokotsa, and Serge Vaudenay. Towards Secure Distance Bounding. Cryptology ePrint Archive, Report 2015/208, 2015. https://eprint.iacr.org/2015/208.

[6] Stefan Brands and David Chaum. Distance-bounding protocols. In *EUROCRYPT*, pages 344–359. Springer, 1994.

[7] M. Cagalj, S. Čapkun, R. Rengaswamy, I. Tsigkogiannis, M. Srivastava, and J. Hubaux. Integrity (I) codes: message integrity protection and authentication over insecure channels. In *IEEE Symposium on Security and Privacy (S&P)*, pages 15 pp.–294, 2006.

[8] A. Compagno, M. Conti, A. A. D'Amico, G. Dini, P. Perazzo, and L. Taponecco. Modeling Enlargement Attacks Against UWB Distance Bounding Protocols. *IEEE Transactions on Information Forensics and Security*, 11(7):1565–1577, 2016.

[9] DecaWave. DecaWave "dw1000 product description and applications". https://www.decawave.com/products/dw1000. [Online; Accessed 22. October 2018].

[10] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[11] Robert J Fontana and Edward A Richley. Observations on low data rate, short pulse uwb systems. In *IEEE International Conference on Ultra-Wideband (ICUWB)*, pages 334–338, 2007.

[12] Shyamnath Gollakota, Nabeel Ahmed, Nickolai Zeldovich, and Dina Katabi. Secure in-band wireless pairing. In *USENIX Security Symposium*, 2011.

[13] Humatics. Time Domain's PulsON ("p440"). http://www.timedomain.com/products/pulson-440/. [Online; Accessed 23. October 2017].

[14] Todd E. Humphreys. Assessing the spoofing threat: Development of a portable gps civilian spoofer. In *Institute of Navigation GNSS (ION GNSS)*, 2008.

[15] Benjamin Kempke, Pat Pannuto, and Prabal Dutta. SurePoint: Exploiting Ultra Wideband Flooding and Diversity to Provide Robust, Scalable, High-Fidelity Indoor Localization. In *ACM SenSys*, pages 318–319, 2016.

[16] Xingqin Lin, Jingya Li, Robert Baldemair, Thomas Cheng, Stefan Parkvall, Daniel Larsson, Havish Koorapaty, Mattias Frenne, Sorour Falahati, Asbjörn Grövlen, and Karl Werner. 5G New Radio: Unveiling the Essentials of the Next Generation Wireless Access Technology, 2018.

[17] A. F. Molisch. Ultrawideband propagation channels-theory, measurement, and modeling. *IEEE Transactions on Vehicular Technology*, 54(5):1528–1545, 2005.

[18] A. F. Molisch, D. Cassioli, C. Chong, S. Emami, A. Fort, B. Kannan, J. Karedal, J. Kunisch, H. G. Schantz, K. Siwiak, and M. Z. Win. A Comprehensive Standardized Model for Ultrawideband Propagation Channels. *IEEE Transactions on Antennas and Propagation*, 54(11):3151–3166, 2006.

[19] Andreas F. Molisch. *Wireless Communications*. Wiley Publishing, 2nd edition, 2011.

[20] Andreas F. Molisch, Kannan Balakrishnan, Chia chin Chong, Shahriar Emami, Andrew Fort, Johan Karedal, Juergen Kunisch, Hans Schantz, Ulrich Schuster, and Kai Siwiak. IEEE 802.15.4a channel model - final report. In *Converging: Technology, work and learning. Australian Government Printing Service*. [Online; Accessed 4. November 2018], 2004.

[21] A. Muqaibel, A. Safaai-Jazi, A. Bayram, and S. M. Riad. Ultra wideband material characterization for indoor propagation. In *IEEE Antennas and Propagation Society International Symposium*, volume 4, pages 623–626, 2003.

[22] Pericle Perazzo, Lorenzo Taponecco, Antonio A. D'amico, and Gianluca Dini. Secure Positioning in Wireless Sensor Networks Through Enlargement Miscontrol Detection. *ACM Transactions on Sensor Networks*, 12(4):27:1–27:32, 2016.

[23] Christina Pöpper, Nils Ole Tippenhauer, Boris Danev, and Srdjan Čapkun. Investigation of Signal and Message Manipulations on the Wireless Channel. In Vijay Atluri and Claudia Diaz, editors, *Computer Security – ESORICS 2011*, pages 40–59. Springer, 2011.

[24] Swiss Post. Drones as transportation vehicle. https://www.post.ch/en/about-us/company/media/press-releases/2017/swiss-post-drone-to-fly-laboratory-samples-for-ticino-hospitals, May 2018.

[25] Mary-Ann Russon. Drones to the rescue! http://www.bbc.com/news/business-43906846, May 2018.

[26] Mridula Singh, Patrick Leu, and Srdjan Čapkun. UWB with Pulse Reordering: Securing Ranging against Relay and Physical Layer Attacks. In *NDSS*, 2019.

[27] L. Taponecco, P. Perazzo, A. A. D'Amico, and G. Dini. On the Feasibility of Overshadow Enlargement Attack on IEEE 802.15.4a Distance Bounding. *IEEE Communications Letters*, 18(2):257–260, 2014.

[28] Nils Ole Tippenhauer, Kasper Bonne Rasmussen, Christina Pöpper, and Srdjan Čapkun. Attacks on Public WLAN-based Positioning. In *ACM/Usenix MobiSys*, 2009.

[29] Nils Ole Tippenhauer, Kasper Bonne Rasmussen, and Srdjan Čapkun. Physical-layer Integrity for Wireless Messages. *Computer Networks*, 109(P1):31–38, 2016.

[30] Deepak Vasisht, Swarun Kumar, and Dina Katabi. Decimeter-level localization with a single wifi access point. In *USENIX NSDI*, pages 165–178, 2016.

[31] S. Čapkun and J. Hubaux. Secure positioning of wireless devices with application to sensor networks. In *IEEE Computer and Communications Societies.*, volume 3, pages 1917–1928, 2005.

[32] K. Witrisal, G. Leus, G. J. M. Janssen, M. Pausini, F. Troesch, T. Zasowski, and J. Romme. Noncoherent ultra-wideband systems. *IEEE Signal Processing Magazine*, 26(4):48–66, 2009.

[33] Henk Wymeersch, Gonzalo Seco-Granados, Giuseppe Destino, Davide Dardari, and Fredrik Tufvesson. 5G mmWave Positioning for Vehicular Networks. *Wireless Communications*, 24(6):80–86, 2017.

[34] Paul A Zandbergen. Accuracy of iPhone locations: A Comparison of Assisted GPS, WiFi and Cellular Positioning. *Blackwell Transactions in GIS*, 13(s1), 2009.

[35] Zebra Technologies. "sapphire dart ultra wideband (uwb) real time locating system 2010.". https://www.zebra.com/us/en/solutions/location-solutions/enabling-technologies/dart-uwb.html. [Online; Accessed 22. October 2018].

Computer Security and Privacy in the Interactions Between Victim Service Providers and Human Trafficking Survivors

Christine Chen
Paul G. Allen School of Computer Science & Engineering
University of Washington

Nicola Dell
The Jacobs Institute
Cornell Tech

Franziska Roesner
Paul G. Allen School of Computer Science & Engineering
University of Washington

Abstract

A victim service provider, or VSP, is a crucial partner in a human trafficking survivor's recovery. VSPs provide or connect survivors to resources such as medical care, legal services, employment opportunities, etc. In this work, we study VSP-survivor interactions from a computer security and privacy perspective. Through 17 semi-structured interviews with staff members at VSPs and survivors of trafficking, we surface the role technology plays in VSP-survivor interactions as well as related computer security and privacy concerns and mitigations. Our results highlight various tensions that VSPs must balance, including building trust with their clients (often by giving them as much autonomy as possible) while attempting to guide their use of technology to mitigate risks around revictimization. We conclude with concrete recommendations for computer security and privacy technologists who wish to partner with VSPs to support and empower trafficking survivors.

1 Introduction

Human trafficking is a crime in which a perpetrator, or "trafficker", preys on vulnerable individuals through atrocities such as sexual exploitation, forced labor, or the removal of organs [30]. As a conservative estimate, around 24.9 million individuals worldwide are being exploited in this manner [16]. Technology is playing an increasing role in this ecosystem, from enabling trafficking via online platforms (e.g., [2, 18]) to aiding in the detection and halting of trafficking (e.g., [6, 25]).

In this work, we focus on a previously understudied role that technology plays in the human trafficking ecosystem: technology in the interactions between trafficking survivors and organizations known as victim service providers, or VSPs. VSPs exist to support their clients by providing resources such as temporary shelter, help with employment and legal issues, and mental health support. In this work, we focus on VSPs providing resources to individuals who are exiting or recovering from a trafficking situation. These resources are critical in protecting these individuals from former or future exploiters ("revictimization").

Our research is driven by the following questions: How do VSPs communicate and interact with their clients (trafficking survivors), and, particularly, what role does technology play in that interaction? What are VSPs' computer security concerns and threat models, both for themselves and on behalf of their clients? What technical (or non-technical) strategies do they use to mitigate these concerns? And, ultimately, what opportunities exist to better safeguard VSPs and their clients from a computer security perspective?

To investigate these questions, we conducted a qualitative interview study with 17 participants, including staff members at VSPs and several trafficking survivors. We analyzed these interviews using thematic analysis common in qualitative research. Our findings shed light on the general role of technology in VSP-survivor interactions (Section 4.1), the computer security concerns and threat models of VSPs and their clients (Section 4.2), and the corresponding defenses, where present (Section 4.3). We identify fundamental tensions and challenges that must be taken into account by technologists who wish to improve VSP and client security and privacy (Section 5).

At a high level, we find that VSPs make technology-related choices with the goals of protecting their clients from revictimization and other harm. Specific instances of how VSPs protect clients include helping clients lock down social media accounts and enforcing shelter rules restricting photos or social media posts (that may reveal the shelter's location). We also find that, sometimes, the most effective means for VSPs and their clients to interact are not the most conducive to client safety. For example, despite the potential risk of trafficker-compromised accounts, some VSPs use Facebook to communicate with clients because it provides a reliable way to reach them even in the absence of cellular service. More generally, we find that our participants must balance building client trust and maintaining contact with imposing technology-related client safety rules.

From findings such as these, we distill concrete recom-

mendations for those in the computer security and privacy community, and for technologists at large, wishing to help support survivor-VSP relationships. For example, we provide guidelines on securing communications in situations when the client's device is compromised by an adversary with physical access and raise awareness around the threat posed to survivors by publicly available information (e.g., public records) online. In investigating the interactions between survivors of human trafficking and VSPs from a computer security and privacy perspective, our work contributes to the larger push to leverage technology for good in the fight against human trafficking.

2 Background and Related Work

There is a growing body of research examining the role of technology in both facilitating and fighting human trafficking (e.g., [2, 12, 17–19, 24]). In the computer science community in particular, prior work has developed technology to aid investigators in examining online sex ads and online forums for trafficking activity [6, 25].

Focusing on the victim service provider ecosystem, there has been research that explores the ways anti-trafficking organizations utilize technology to collaborate with each other [29] as well as efforts within the VSP community to leverage technology in providing help to trafficking victims [20]. Work outside of the technical realm has examined how survivors of trafficking [4] or domestic violence [13] experience and react to the assistance provided by VSPs.

Beyond human trafficking, the computer security and privacy community has studied other specific (often at-risk) populations, including journalists [22], refugees [28], and undocumented immigrants [14]. Most relevant to our work is research studying computer security and privacy for survivors of intimate partner violence [5, 10, 11, 21]. Where relevant, we highlight similarities between our findings with these prior studies on related populations.

In this work, we focus on an aspect of the human trafficking ecosystem that has not been rigorously studied from an academic, technical perspective: the interactions *between* VSPs and trafficking survivors. We ask, from a computer security and privacy perspective: how does technology enable or hinder these relationships, and how do VSPs and their clients consider and mitigate the potential technology-related risks that may undermine the survivor's path to recovery?

3 Methodology

Between March and July 2018, we conducted 17 semi-structured interviews with staff members at victim service provider organizations and several trafficking survivors.

Recruitment. Recruitment took place through several primary methods: introductions facilitated by community members and anti-trafficking leaders, the authors' personal connections, and snowball sampling. Our recruitment advertisements specified that we were looking for advocates who work with labor trafficking and/or sex trafficking survivors to speak about how they use technology in their work. We also specified that participants would be compensated $30.

Participants. Table 1 provides an overview of the 17 study participants. The 17 participants represented 11 different organizations; survivors P14 and P17 were not affiliated with a specific organization at the time of the study. 16 participants were based in the U.S. and one participant was based in a Southeast Asian country. Most participants were based in urban areas.

As Table 1 shows, most participants currently focus on serving survivors of sex trafficking (though some participants may have previously helped labor trafficking survivors as well). To avoid confusion, we do our best throughout this paper to call out results that are specific to interactions with labor trafficking survivors or sex trafficking survivors. Finally, to be clear, note that some of the participants who focus on sex trafficking survivors naturally also serve individuals in the sex trade who may not technically fall within the parameters of sex trafficking (e.g. individuals who claim to be in the sex trade voluntarily).

Study Protocol. The interviews ran between 60-90 minutes. We began with groundwork questions to understand the participant's role in supporting clients and general thoughts on technology's influence on the trafficking ecosystem. We then asked questions that would help surface how VSPs use technology in their interactions with clients and what, if any, concerns exist around this technology usage. We asked about participants' experiences with technology with regards to first contact with clients, client intake, organization and client safety, and day-to-day interactions. To avoid priming participants to overemphasize their computer security and privacy concerns, most questions focused generally on technology in client-VSP interactions and related concerns but did not mention computer security and privacy in particular.

Finally, we showed participants two prototypes for secure communication (created by others): single-use URLs (a URL that leads to sensitive content, which gets changed to innocuous content when the same URL is accessed again) and disappearing messages [1, 8, 9]. Our goal was to elicit reactions and threat models using these concrete examples, not to propose these particular technologies as perfect solutions. To avoid participants giving inflated positive responses towards the tools (participant response bias), we stated these goals clearly for participants and also stated that we did not make the tools. We asked questions like: When, if at all, might you use this? How could it be helpful? How could it introduce more risk? The full interview protocol can be found in Appendix A.

ID	Job Title	Focus	Client Nationality	Client Age
P1	Advocate, Survivor Leader	Sex Trafficking	Domestic	Adult
P2	Advocate	Sex Trafficking	Domestic	Adult
P3	Director	Sex Trafficking	Domestic	All
P4	Director	Sex Trafficking	Domestic	Youth, TAY
P5	Advocate	Labor Trafficking	International	Adult
P6	Director	Sex Trafficking	Domestic	All
P7	Advocate	Sex Trafficking	Domestic	Youth, TAY
P8	Advocate	Labor and Sex Trafficking	Domestic, International	All
P9	Advocate	Sex Trafficking	Domestic	Youth, TAY
P10	Advocate, Survivor Leader	Sex Trafficking	Domestic	Youth, TAY (to 25)
P11	Advocate	Sex Trafficking	Domestic	Youth, TAY (to 30)
P12	Advocate	Labor and Sex Trafficking	International	Adult
P13	Advocate	Sex Trafficking	Domestic, International	TAY
P14	Survivor Leader	Sex Trafficking	N/A	N/A
P15	Advocate	Sex Trafficking	Domestic, International	did not disclose
P16	Director	Labor and Sex Trafficking	Domestic, International	Adult
P17	Survivor Leader	Sex Trafficking	N/A	N/A

Table 1: Summary of Participants. *Advocates* support clients one-on-one, *Directors* oversee the VSP's human trafficking services (managing advocates as well as interacting with clients), and *Survivor Leaders* are survivors of trafficking (in this case, sex trafficking) who are raising awareness and leading trainings on the issue. Transition age youth (TAY) are individuals between the ages of 16 and 24 [32]; where specified, participants also worked with clients slightly outside of this range.

Ethical Considerations. Our study was declared exempt by the University of Washington human subjects review board (IRB). We obtained informed consent from participants to conduct and (optionally) to audio record the interview. As the interviews could touch on highly sensitive topics (especially for survivors), we ensured that participants knew that they could skip questions and request a break at any time. We also emphasized that participants should provide only as much detail in their answers as they felt comfortable with. All electronic files were password protected, and physical consent forms and notes were stored in a secure location.

Data Analysis. We continued conducting interviews until no new themes emerged (saturation). We analyzed the data thematically using a common methodology for qualitative data [3]. We conducted multiple passes through the data in which we iteratively identified and clustered themes, or codes, present in the data. Two researchers independently read through transcripts of several interviews, generated an initial set of codes, met in person to develop an initial codebook, and iteratively refined this codebook by applying it to additional interviews. Once the codebook was finalized, two researchers divided up the remaining interviews and coded them. We emphasize that the nature of our data is qualitative, not quantitative, so we do not report on raw numbers of participants who made certain statements in the results.

4 Results

We now turn to our results. After providing an overview of the general practices our participants and their organizations use in interacting with trafficking survivors, we will present the security and privacy concerns and mitigation strategies — and tensions and challenges — that arise in these interactions. We use the terms "survivor" and "client" interchangeably, depending on the context and following the norms described by our participants during the interviews. At times, we also use the term "victim" and note that VSP clients may not be fully removed from a trafficking situation when they are receiving services.

4.1 Client-VSP Interactions

This section provides background and context for the more in-depth security and privacy discussions in later sections.

4.1.1 Role of VSPs

Though VSPs may help trafficking victims escape their situations, their primary role is to help clients with the many challenges they face on the path to stability, including looking for employment, applying for housing, dealing with legal matters, and coping with severe trauma. Importantly, as we investigate in this paper, VSPs protect clients and train

clients to protect themselves from revictimization into a trafficking situation.

Some of our participants work at VSPs that provide shelter for clients. These arrangements range from emergency shelters (with very low barrier to entry — e.g., a client can stay even if he or she is on drugs) to long-term homes (where the client must be committed to actively working towards goals and self-sustainability). As we discuss in Section 4.3, shelter locations are sometimes confidential to help protect clients.

As an overarching challenge in providing services to clients, participants described the delicate balance they must walk between building trust with their clients — so that they can best advise and maintain contact with them — and doing what they believe is best for the client. As clients have left (or as they are in the process of deciding whether to leave) a situation where they have had little control over their lives, participants often talked about how crucial it is to give clients as much autonomy as possible. For example, P13 talked about working with clients who want to find a job. While she would like her clients to go to school, she does not force her idea of what would be best on the client. Throughout out results, we will see this tension recur in the context of technology-related guidelines and choices that VSPs are hesitant to push on their clients.

4.1.2 First Contact

Clients typically make their first contact with VSPs through referrals — e.g., from law enforcement, schools, or other VSPs — or via a phone hotline. Hotlines may receive emergency calls, playing a similar role to 9-1-1 for clients and trafficking victims. For example, P7 described answering hotline calls from individuals who are running for their lives at the moment, and P8 talked about how they will dispatch a Lyft or Uber to a caller who has just escaped.

Dispersal and discovery of hotline numbers happens in a variety of ways. Beyond relying on word of mouth (a common method), participants talked about posters with the hotline number placed in public locations such as hospitals, train stations, and rest stops. One organization has their hotline number on a local Spanish TV channel. Another mode of dispersal is through personal items (e.g., soap, essential oil, hats, etc.) handed out to at-risk individuals (e.g. farm workers) with the hotline number hidden discreetly on the object.

For individuals still in trafficking situations, calling the hotline can be dangerous (if the individual is constantly being monitored by their trafficker) or even impossible (if the individual does not have a device). In these situations, participants described the ingenuity of their clients in finding ways to access technology to get help. For example, one of P16's labor trafficking clients saved up enough money from tips to buy a burner phone from a gas station. While the burner phone did not have the capability to connect to the Internet, he had seen a hotline number earlier and committed it to memory. As another example:

> P8: I've had a few clients who, in escaping... [were] able to get access to a hidden phone or discretely (on an app that their trafficker isn't aware...is a messaging app)...send messages to a friend who helps them get help...

From advocates who work with sex trafficking survivors, we heard how clients will search the web for help:

> P2: We've had a couple people. I'm like, "How did you learn about us?" She goes, "I googled prostitutes [city]."

At the same time, participants worried that lack of technical expertise could make it challenging for clients to find help online. For example:

> P1: I think what people have a hard time with is search words. I think people don't understand how Google works, and how to search for things.

Mention of direct outreach by VSPs to potential clients was rare, but one participant uses the phone numbers in online sex ads to conduct text message campaigns to contact individuals who might want help leaving. Another participant, P3, said that her organization reaches out to people who like the organization's Facebook page to see if they need help.

4.1.3 Continued Communication

Our participants typically communicate with clients via phone calls, SMS, social media (e.g., Facebook), email, and in person. Participants generally talked of using the communication method that their clients feel most comfortable with. P16 described how digital communication can help put clients at ease.

> P16: I find that many of our clients are more comfortable engaging through technology because it's less raw. It's a step removed in some ways...

Communication methods that work over WiFi were often mentioned as important, as clients may not be able to afford reliable cellular service or even a reliable device:

> P2: A client right now has a phone. It's not connected to any service, but she can connect to WiFi, so she and I can use Facebook Messenger instead of texting. That's true for a lot of our clients, because phones get turned off and on all the time, numbers change all the time. I can still reach them on Facebook, on Facebook Messenger. You can log in to any computer or any phone to access it.

As we will discuss further in Section 4.3, participants' and their clients' threat models also influence their choice of communication method.

4.2 Threat Models and Security Concerns

We now turn specifically to the threat models and computer security related concerns voiced by our participants, both for themselves as individuals and representatives of their organizations, and on behalf of their clients. We found that many of the security concerns or goals that our participants voiced ultimately revolved around preventing revictimization and protecting the physical safety of clients and VSPs. In this work, we focus primarily on technology-related issues, but highlight other concerns as well where necessary for context.

4.2.1 Trafficker as Primary Adversary

The most common adversary for VSPs and clients were the clients' former trafficker(s) or potential future trafficker(s).

Compromising Online Accounts and Communications. VSP clients' communications may be compromised by traffickers, either digitally or via physical access. In many cases, traffickers have access to account credentials directly. P5, who works with labor trafficking survivors, described one tactic traffickers use to gain such access and alludes to the way low digital literacy can harm international and/or labor trafficking victims:

> P5: What if their trafficker has access to their email or helped them set up the email account. Just the client never knew that and now I'm communicating with the client and [the trafficker] is reading our information?...I feel with our clients, they're just so vulnerable and a lot of them were brainwashed...using a cell phone or using Facebook, a lot of them, their traffickers opened the account for them and they think, "Oh he was just being helpful. He wanted me to communicate with my family."

Traffickers may also compromise or intercept communications via physical access to clients' devices. For example, in the sex trafficking ecosystem:

> P7: I've had different guys that'll pick up [my clients'] phone and pretend to be them, go through their messages.

Despite the risk that a trafficker might physically see or digitally intercept communication intended for a victim, P1 weighed such risks against the benefits of reaching trafficking victims in her text outreach work. Note that the term "pimp" is another way of referring to the trafficker.

> P1: And I don't think it's at the expense of the victim, okay? I think, people ask this question because they're like, "Well don't you think that their pimp is gonna beat them up because they got this message?" Potentially. 100% yes...It's either, I get information out there that will potentially give them an out, or they just don't get anything.

Tracking Location. Another concern was traffickers tracking down former victims after their escape. P16 has had clients who found GPS trackers on their cars; P7 described the use of tracking apps on phones:

> P7: It's usually...through [the victims'] device because most of the pimps get [the device], so they have the family tracking, different apps and stuff like that...One of my girls has shown me that they can pull it up on their computer and you can see where all of [the victims] are at one time.

Using Online Information to Track Down Survivors. Even if a survivor's devices or accounts are not directly compromised, participants worried about the use of online public information to track down survivors. This fear is exacerbated by the fact that the trafficker often knows key information (like birth date, social security number, etc.) that allows access even to protected information.

For example, P14 is herself a survivor, and she generalized from her own experiences the ways traffickers can utilize public information to relocate survivors. Specifically, she explained that traffickers can find where survivors have moved by searching publicly available Department of Motor Vehicles (DMV) records; they can use survivors' addresses and social security numbers to access and potentially lock them out of their own bank accounts; and they can then track survivors' activities by observing the details of bank transactions.

As another example, P17 described being found via medical records and an old Facebook page she had thought was gone. P16 talked about how shared rewards systems (like grocery rewards cards) can reveal to a trafficker where a survivor is shopping and what they are purchasing.

Undermining VSP Operations. Participants also discussed the ways that traffickers seek to undermine the efforts of VSPs. P7 talked about traffickers hanging out near VSPs to recruit, and P16 talked about a trafficker sending a victim into a survivor program to recruit others directly. P2 mentioned that traffickers have called her organization's hotline looking for survivors.

For shelters where the location is confidential, participants described various ways in which this confidentiality could be compromised. A common concern, for example, was that people living in the shelter might accidentally reveal its location (or the location of shelter guests) to traffickers via pictures or other posts on social media. P14 felt that location confidentiality was a challenging, if not impossible, goal:

> P14: ...how confidential really is any kind of building? I mean, you're gonna see it on Google Maps eventually. Whether or not you see it this year or three years from now when they do their next picture, you're gonna see it. So it's not gonna be necessarily confidential for long.

On a related note, P14 talked about an organization she knows that did drone footage of their safe house as a cautionary tale to VSPs of how easily location confidentiality can be breached.

> P14: Which to me, anybody who is logical, you're doing drone footage of a supposed safe house. Well, any good hacker is gonna be able to pinpoint on a map exactly where that safe house is. Now, you're no longer safe.

4.2.2 Other Threats and Concerns

Beyond traffickers as direct adversaries, our interviews surfaced several other threats and concerns that VSPs and their clients may contend with.

Availability of VSP Resources. VSPs have limited resources which can come under intentional or unintentional contention. For example, several participants discussed ways in which the availability of the emergency hotline can be impacted, either by suspicious callers or by callers who misinterpret the function of the hotline. For example, P12 talked about how the hotline gets calls from people who need help with their subway cards because there are posters with the hotline number in the city's subway system.

Post-Trafficking Limbo. Several participants (P11, P13, P15) discussed how sex trafficking survivors can be at greater risk for abusive or unhealthy relationships:

> P13: They'll minimize the [domestic violence] because "at least he's not selling me"...They'll minimize the psychological violence that they're causing them or the emotional.

P13 gave an example of the potential consequences, describing a client who has an abusive, controlling boyfriend:

> P13: Like, within the last two months she's gotten like four new numbers, four new phones or five new phones. So I'm starting to think that this guy is breaking all her phones so that she doesn't have communication with anyone.

P13 described how this constantly changing communication environment severely limits the amount of help she can offer the client, e.g., prolonging the process of helping her find employment. Prior work [27] describes how domestic violence can serve as a "push factor" into sex trafficking. Our findings suggest that the push factor can also work in the opposite direction from sex trafficking to domestic violence. Computer security in the context of intimate partner violence has also been covered extensively in prior work [5, 10, 11, 21].

Labor trafficking survivors are especially desperate to get a job, and P5 described how they will sometimes even consider asking people in their community back home (who got them into labor trafficking in the first place) about jobs for undocumented individuals. Furthermore, the internet is not always a safe place to look for re-employment. In describing ways that technology facilitates trafficking, P8 talked about online frauds that can lead to labor trafficking:

> P8: We've had clients who respond to certain Craigslist ads for either a place to live or a job and then once they get into this suspect situation end up...getting trafficked.

Online Triggers. Several participants mentioned the risk of online triggers that may push a sex trafficking survivor towards revictimization. For example, survivors who are friends with individuals still in sex work (whether voluntarily or not) are constantly bombarded on social media with reminders of their past (e.g., a friend might post about the amount of money she made in a night):

> P2: We talk about "environmental triggers," and you can avoid an area of town as part of a safety plan around relapse prevention, but do you also have to delete your Snapchat and maybe your Instagram? And maybe get a new Facebook? If you're still "friends," on any social media platform, with anyone from that life, you're gonna be seeing triggers constantly.

In a similar vein, P4 worried about all the things her (underage) clients might stumble across on the Internet. She told the story of a time one of her clients was doing homework and clicked on a Youtube video. Youtube's content suggestions led the viewer to increasingly explicit content. P4 was also fearful that images of past (digitized) exploitation might surface on the Internet and haunt her clients later in life.

Concerns around Law Enforcement and Legal Systems. Both VSP staff members and survivor leaders voiced concerns related to the interactions between victims/survivors and law enforcement. On one extreme, a survivor leader explained that local law enforcement was complicit in her trafficking. This is a real concern — e.g., police officers in New York City were recently charged for involvement in a prostitution enterprise [31]. Thus, this survivor worried that, if law enforcement ever came to the shelter (to help with some emergency), this could be triggering for shelter residents who may have had negative experiences with law enforcement. Participants also voiced concern over victims of trafficking being charged with crimes. For example, prostitution is largely illegal in the United States, and sex trafficking survivors may be charged under prostitution laws or with other crimes committed during the time they were being trafficked [26, 27].

Participants also expressed frustration over regulations that make it difficult to establish trust with clients, such as legislation in some states requiring advocates to report runaways who come to their shelter to parents and law enforcement. This disincentivizes minors to come to the shelter:

> P10: If they're listed as missing or as a runaway, I'm obligated to let law enforcement know where they are...I do have a lot of professional protocols I have to follow as well. I adhere to those, as much as they suck. Our laws just don't always enable us to do what actually needs to be done.

As we discuss in Section 4.3, these concerns can drive the communication choices of clients and VSPs. We also want to note that some participants described strong partnerships with law enforcement, and one survivor leader described the immense support she received from law enforcement in her recovery.

Authenticity of First Contact. Finally, for VSPs who do direct outreach to potential clients, it is difficult to convince the recipient of their good intent. P1 reaches out to potential sex trafficking victims via text message, and those receiving her text messages are sometimes fearful that she is the police. P5 has heard about this kind of outreach, but has not started using it because she knows the individuals she would be reaching, potential labor trafficking victims, would be highly supicious:

> P5: It's like, "Should I trust this?"... "Should I respond to that or is it just another trap for me? Am I going to get in trouble?"

Victims' concern about both traffickers and potential legal ramifications, discussed above, may lead to this skepticism.

4.3 Technical Defenses and Mitigations

We now consider the concrete steps that VSPs take to mitigate their own and their clients' computer security and privacy related concerns (with the ultimate goals of avoiding revictimization and other harm, as discussed above). Overall, we observed four categories of technology-related defensive strategies: (1) guiding or explicitly restricting how clients use technology, (2) protecting communications with clients, (3) protecting data about clients, and (4) protecting the VSP's resources and employees. These strategies are in addition to physical and non-technological defenses, e.g., security cameras and bullet-proof windows in shelters, and heuristics that VSPs use to identify suspicious hotline callers.

4.3.1 Guiding Client Technology Use

In order to protect clients from revictimization or other threats, VSPs guide—or in some cases, explicitly restrict—their use of technology. This guidance is typically done in the form of safety planning with a client, or through rules and guidelines about the use of technology in a shelter.

Technology Safety Planning. VSPs work closely with their clients to help them be safe and feel safe. Given our focus on computer security, we focus primarily on technology safety planning here, but note that other forms of safety planning (e.g., for physical and emotional safety) are related.

One key goal of technology safety planning is to keep a client's whereabouts hidden from a former trafficker. Participants talked about a variety of strategies, including avoiding different parts of town, overhauling communication methods and social media accounts, and turning off location services on devices. For example:

> P10: Some people, they need to create a whole new social media everything, change their phone number and email address, and they need to just like literally disappear... That means that your accounts are completely, 100% locked down and you have pseudonyms, and you don't use your face in any pictures that you post. You never post your location. You don't have location turned on on your phone. You get a brand new phone because you don't know what kind of app trackers there are.

Some participants simply give guidance to clients, while others, like P10, actively help clients configure their devices:

> P10: I tell them . . . "Give me everything. Give me all the stuff." I sit down and lock everything down.

Our survivor leader participants described the precautions they take themselves and would recommend to other survivors. For example, P14 only uses location services when she needs help with navigation, changes her passwords regularly, and avoids making location-tagged posts on social media. Likewise, survivor leader P17 uses pseudonyms on her profiles and deactivates her Facebook regularly. In addition:

> P17: When I'm at the store and they're like, "Can we please have . . . your zip code or your phone number," I always say no, my phone number is very private. When I check in at hotels, I always have a process so my name doesn't associate with the hotel. My medical information is protected like everybody else's but I always have to have a conversation, like look, there are people that have every piece of information about me and they will call to get my medical information so I need a code word or something associated with my account.

While a common sentiment was that location services on phones should be turned off, on the flip side, clients sometimes want location services *on* as part of the safety plan:

> P2: We've had some clients...who want their location services on, so that I can use my iPhone and find them if something goes south, or to get an Uber sent to them in a crisis, or whatever the case may be. They want us to be able to track them.

Participants also helped their clients with general online safety best practices, For example, some participants (e.g., P11, P15) talked about helping their clients understand that folks on the Internet cannot always be trusted:

> P15 (who works with sex trafficking survivors):

> We've had some residents, when they're here, they start dating again. So they're on Facebook, they're on different websites, so I have candid conversations, "Okay, are you telling someone where you're gonna go? Are you telling another friend where you're meeting this new person online? What time can we expect you back?" Things like that. Trying to put that idea into their head that you should not trust people on the Internet.

Overall, there was the sense that safety planning is individualized rather than one-size-fits-all. For example, P14 mentioned that factors like how long someone was trafficked and where and whether or not the individual has made changes to personal appearance (e.g., dyeing hair) all impact safety planning. P15 also mentioned how age plays into technology use and, subsequently, technology safety planning: clients in their 40s and 50s tend to not have a large social media presence in the first place, while younger clients find it much more difficult to take a social media hiatus.

Stepping back, we observe a tension between granting autonomy to clients and providing maximum security:

> P15: Because I always bring up to people, you probably should turn off location on your phone. But I do kind of leave it up to them. It's not mandatory that they turn it off. 'Cause we come from an empowering place. We don't want to be telling them what to do.

In addition, working through potential threats and how to defend against them can be highly stress-inducing — one of the challenges mentioned by our participants revolved around balancing the client's safety with their mental health. P8 talked about being careful in safety planning to not trigger the client and cause the client to become more fearful. P14 talked about the fine balance between making someone untrackable and keeping them sane. P15 pointed out that the survivor most likely is already safety planning and is in the best position to look out for himself or herself.

Shelter Technology Rules. Participants described sometimes enforcing technology-related rules in shelters. A number of rules or guidelines centered around attempting to protect the identity of people in the shelter and the confidentiality of its physical location. For example, some shelters disallow photos or videos:

> P1: We have to think about all our clients and their privacy, especially we say, "If you're taking a photo, somebody could be walking behind you and you've breached their privacy because somebody could know who they are."

Similarly, P16's organization asks that shelter residents refrain from posting on social media and turn off location services on their devices. These rules aim to prevent accidental disclosure of the shelter location through, say, a Facebook post containing the resident's location. At P4's organization, these rules hold for visitors as well: P4 mentioned that visitors have posted photos of general surroundings and staff asked them to take them down. P16 said that they also put up a sign notifying clients that if they use the VSP main office WiFi they are vulnerable to being tracked to that location — but overall considered providing WiFi to clients more important than mitigating this risk by not providing WiFi.

Additionally, some participants mentioned restricting or monitoring how clients use the computers provided in shelters, with the goal of shielding clients from content or activities that might put them at risk of revictimization. For example, at P3's organization, advocates ensure that clients are not using computers to post sex ads. In addition, clients can set up email accounts for employment, and they are told outright that the accounts are not private (i.e., the advocate may inspect the client's emails if there is cause to believe that the client is using the email account for other purposes).

In general, we found that organizations who work with younger clients (under 18) have more regulations around technology usage. For example, P4 checks the browsing history of the shelter's computers weekly. P6 was in the process of purchasing software to help manage computer usage. Her organization has an adult sit with clients who need to use a computer. At the under-18 shelter run by P9's organization, clients physically check in their phones when they enter.

The main challenge here lies in balancing technology rules with the fact that technology use is an increasingly crucial part of life for many people, helping them feel "normal." And, as P14 explains, strict restrictions on technology use can actually undermine security goals:

> P14: If we deprive them of that technology, they're gonna do it behind our back. They're gonna find a way to get a burner phone, they're gonna find a way, and a friend, and whatever to get a phone ... whether we endorse it and give them a safe way to do it, or whether we let them do it behind our backs and potentially risk all of us ...

4.3.2 Protecting Communications with Clients

Given the potential for compromise—by traffickers or others—of communications with clients, VSPs carefully consider their choice and use of communication methods.

Choice of Communication Medium. Participants discussed two main goals when selecting communication methods: (1) protecting the content of communications from potential adversaries and (2) maintaining contact with clients. We summarize the pros and cons of communication methods from the perspectives of advocates and clients, with respect to both convenience and security, as reported by participants.

SMS. Participants described SMS, or texting, as common and useful. It gives survivors space to choose whether to respond to VSPs or not, allows for urgent communication and quick

access to resources, and is more discreet than phone calls.

P7 suggested that individuals in the sex trade (whether voluntarily or not) are moving away from using text due to concerns about law enforcement confiscating phones and searching text messages. From the advocate perspective, some participants expressed unease that, with texting, there is no authentication ensuring that you are communicating with the person you think you are communicating with.

Phone. Talking on the phone, by contrast, allows participants to authenticate communication partners via voice:

> P15: Texting can be tricky. Even if I've worked with the client for a while and I'm still texting them. Sometimes it's like I'd rather just call and talk to them so that I know it's them talking to me and not someone else on their phone.

Phone calls were also generally considered a sufficiently secure communication method—as long as the line is not being tapped. However, clients were sometimes not comfortable talking on the phone when they were still in physically dangerous situations.

Social Media. Social media — especially Facebook — emerged as a very popular way for VSPs to communicate with clients. (Indeed, related work on technology in the trafficking ecosystem [2] has found that victims of domestic minor sex trafficking utilize Facebook in app and website form more frequently than any other online service.) We observe that this prevalence may cause challenges, as social media can blur the personal/professional boundary between VSPs and clients, and it may not be secure: our own findings as well as prior work on intimate partner violence [10, 11] suggest that abusers commonly access victims' Facebook accounts.

In our interviews, however, the benefits of using social media to communicate with clients seemed to outweigh these risks. A key benefit of Facebook was that clients could retain access to it when they switch devices or phone numbers, and that using it does not require a cell phone plan:

> P7: Facebook. All the time. 'Cause they can always go to the library and get on it. They can always go somewhere and get on their Facebook.

Some clients consider Facebook to be a more discreet communication method compared to phone calls:

> P7: And more of them use Facebook because they don't want somebody calling their phone or having access to their phone. I've had a couple people that are like, "Do not call this number ever. I will call you. Don't ever leave a message on this number, don't ever call it, don't ever do anything, 'cause I have this one chance to call you and if you call back, it's going to be bad for me."

Snapchat can play a similar role, with the additional benefit of supporting disappearing messages by default. P6 recounted the case of a client who ran away from foster care but remained in touch with — and eventually asked to be rescued by — someone at P6's organization via Snapchat. Here, the client's use of Snapchat may have helped protect the communication from being discovered by the trafficker.

Email. By contrast, email was not mentioned as a common or convenient communication method, largely because of a lack of access by clients. For example:

> P12: ...a lot of them don't have e-mail. If they do, they don't know how to access their e-mail. Somebody else helped them set up their e-mail and they forgot the password and their username...

Though (according to P4) email can be helpful as a last-ditch attempt to establish communication when a client's phone number has changed, participants generally described preferring Facebook Messenger in this situation.

Secure Communication. Despite the serious security and privacy concerns faced by our participants and their clients, we heard very few cases of VSPs using existing secure communication technologies with their clients. P16 was an exception:

> P16: ...we like WhatsApp because it's encrypted and because it's a safe storage for the conversations. We cannot use Facebook or Instagram or Snapchat because it's not secure. I know a bunch of programs that will respond through Facebook Messenger and we're not going to do that especially with all the privacy concerns.

We hypothesize that the limited use of WhatsApp (and no mention of other secure communication tools, such as Signal) reflects the tendency of VSPs to choose communication methods that are already familiar to clients (e.g., Facebook or Snapchat in some cases, or texting and phone calls in the cases where professional boundaries preclude social media use).

In Person. Participants also used a non-technological strategy for mitigating digital security concerns: meeting their clients in person. Meeting in person has the benefits of avoiding communicating through any potentially compromised digital medium and allowing for the authentication of the client to the VSP (and vice versa). For example:

> P15: I think in terms of texting with clients and what not, I really prefer to meet someone in person. Especially if I'm meeting someone new. If my first contact with someone is through text message, I don't know if it's that person talking to me. I don't know, it could be their trafficker, could be someone lying to me, making up a name to try to figure out where the house is.

We note that meeting in person cannot defend against the case where a client's phone has been compromised by an adversary who uses it as a remote microphone to eavesdrop on conversations. One participant took steps to mitigate this

risk: P8 obtained Faraday bags for the organization to use when there is concern that a client's device might be compromised (it blocks incoming and outgoing signals from the device). P8 talked about using it with a client and how it allowed them to talk more freely without fear of being monitored. The downside of meeting in person is that it can be challenging for survivors to get to the VSP. P16 stressed the importance of technology in this context:

> P16: They are working three jobs, the kids are home alone, they want to be home with the kids so I think that's actually a nice example of how technology is so helpful for us. They like physically, economically, emotionally cannot get to an appointment but...if we can communicate through a text that could be the lifeline. Or email.

Meeting a client in person also means that the VSP knows, by definition, where the client is physically located. This can be at odds with a VSP's desire to avoid turning a client in to law enforcement, e.g., as would be required if the client is listed as missing or as a runaway. P7 discussed how communicating digitally can provide a loophole for this case:

> P7: You can hit me up for services and tell me, "I need this, this, and that,"... But if I don't know your location, I can't report you... I don't know where you are. It kind of covers our back because we can still serve them without having the legality to report them.

Message Content and Authentication. No matter the chosen communication method, there is always the risk of the trafficker or another adversary monitoring communications in real-time or reading them later, e.g., by leveraging access to a client's account, or by simply overhearing a phone call. To mitigate such a threat, participants reported using ad-hoc techniques to obfuscate the content of their communications with clients. For example, P1 is very brief in her phone communications with clients and checks in beforehand to make sure it is a good time to talk. P10, P3, and P7 use predetermined codewords or code phrases when communicating with clients via SMS or social media. P7 described an authentication strategy in which her client asks or answers a specific question that they established previously to start off communication.

New Devices. P8 reported clients getting new devices as a safety precaution. Limited financial resources on the part of VSPs constrain how much VSPs can help clients acquire new devices. Typically the phones provided by VSPs come from sources such as government programs or Verizon's HopeLine program (which provided recycled phones to domestic violence survivors but is now phasing out). P16's organization receives donated phones but does not have funds to purchase data plans. They still give the phones to clients as a way to call 9-1-1 in the case of an emergency.

Challenges and Tensions in Protecting Communication. In addition to learning about existing ways participants work to protect client communications, we also explored their reactions to other technologies: single-use URLs and disappearing messages [1, 8, 9]. These explorations surfaced several challenges and tensions.

First, a risk with securing a communication channel is that it may make it more difficult for a client to access information. For example, several participants pointed out that although single-use URLs may prevent an adversary from accessing sensitive information via an already-used link, they also prevent the client from *re*-accessing that information.

Second, participants explained that appearing to hide something can put a client in danger (e.g., causing the trafficker/exploiter to become violent):

> P7: Any way that somebody can open the thing and tell that you're being secretive is a scary thing. 'Cause then you're hiding something from me. "What are you doing behind my back? Who are you telling?" There's a lot of paranoia...

The above quote came up in the context of disappearing messages that require a password to view the message, but we observe that this tension may arise for any communication tool that clearly has security as a goal. On a related note, participants described the strong psychological power traffickers have over their victims. P6 discussed how traditional security mechanisms (such as passwords) may fail as the trafficker has such power over the victim that he or she can easily compel the victim to reveal secrets.

> P6: Because they have been so conditioned, so coerced, that it's [the victims] telling anything that they're asked...They're the ones that have a problem keeping a secret.

4.3.3 Protecting Data About Clients

Participants talked about the strategies they use to secure the data that VSPs collect and store about clients.

Access Control for Internal Databases. Most participants talked about using databases to store client information. A few participants explicitly mentioned how each staff member at the organization has their own login credentials and also talked about access controls on the data.

> P15: Someone working in our admin department, like a secretary, they shouldn't be able to open our case notes. There's that kind of protection.

P15 was concerned about the cloud-based nature of the case notes software her organization uses. She was worried that if staff members logged in at home, client information could be seen by the staff member's family or roommates.

Interactions with External Organizations. P7 described how technology aids the referral process and how protocol

dictates that sometimes meetings must take place in person:

> P7: We have a secured email that people can send referrals to...so we get a lot of our referrals from social workers and different people like that. Or I'll have a teacher or a counselor be like, "I can't give you much information, but I'd like you to come in and meet with ..." 'Cause they can't send it over social media or emails, anything.

Several participants described strict protocols for sharing client information with external organizations:

> P15: We have a very specific release form. So if someone wants me to connect with their substance abuse worker that they're working with, I need that form filled out with that substance abuse worker's name, my name, the client's name, the client's signature saying you can tell this substance abuse worker ... You can tell them my name and my date of birth. We ask them to be very specific.

Securing Internal Communications. P15's organization uses encrypted email internally. Other participants mentioned strategies for protecting client identities in internal communications more informally, such as using client initials or first names only in communications and files:

> P8: We're also very careful about using client names. Even in inner work emails or text messages or anything like that. I have all my clients saved in my work phone just as initials. So even if someone is reading a text conversation, they wouldn't know who that was with. Within our database system, everyone is assigned a client number, so we do often use that when we're emailing.

Minimizing Data Collection. Finally, participants described a general principle of storing the least possible amount of client data:

> P16: We intentionally don't write...detailed notes and don't jot down information that could potentially harm them. Not even casually because even if we make a note on the intake form or we write down on a post-it that's technically a part of the case now. ...Keep it brief, keep it vague, keep it objective because anything can be used against the client in court.

In some cases, though, collecting sensitive data is necessary or useful. For example, P13 described how other staff members at her organization make copies of a client's important documents during intake (e.g., birth certificate, Medicaid card, ID, etc.). P13 does not do so, because she considers this information to be sensitive — but she pointed out that the copies can be critical when a client loses the original document.

4.3.4 Protecting VSP Resources and Employees

Finally, we turn to the strategies that VSPs use to protect their own resources and staff.

VSP Location Confidentiality. Participants described various strategies for keeping the location of the VSP's office or shelter confidential. In addition to the shelter technology rules described above, some participants described protecting the address of the shelter by not mentioning it in digital communications with clients. For example, P8 specified that the office address is never texted or emailed out. Likewise, P16 does not share the shelter address. Instead, clients are given the address of a neutral location several blocks away and staff meet them there and bring them to the shelter. Other non-technical strategies for protecting the VSP's location include requiring clients and visitors to sign a confidentiality form, asking clients arriving in a Lyft or Uber to be dropped off several blocks away, only meeting clients at the VSP office if absolutely necessary, and making the shelter look physically inconspicuous (e.g., like a vacant office building).

Personal/Professional Boundaries. Participants described attempting to separate their personal and professional lives, to protect their own physical and emotional well-being.

The primary technical strategies participants described involve separating personal and professional communications. Almost all participants mentioned having separate work emails and work cellphones. With respect to social media, some participants had a strict policy of not interacting with clients on social media (e.g., finding it unprofessional), while others found it invaluable for reaching and maintaining contact with clients (as discussed above).

Participants who do use social media to interact with clients often use separate personal and professional social media accounts. For example, P7 talked about how her personal Facebook account has strict privacy settings, and how she made her friends list inaccessible on her work account to protect those friends. P2, who generally works with individuals who (voluntarily or not) are in the sex trade, talked about carefully regulating when she looks at her work Facebook account because she does not know what she might be exposed to. Another possible concern is that Facebook may unexpectedly reveal to traffickers or others the connection between survivors and VSP staff members (e.g., by suggesting a VSP staff member as a friend to a trafficker through the "People You May Know" feature [15]), but this concern was not mentioned by our participants.

Participants also mentioned a variety of non-technical strategies for protecting themselves, including meeting clients in public locations, letting others know where the participant is going to meet a client, being vigilant of physical surroundings, and the importance of personal self-care and therapy. Ultimately, however, participants accepted the inherent risk in the work that they do:

> P6: I don't make any claims that we're gonna protect [volunteers] from something bad happening. But then, it happens with these girls all the time. And if we're not willing to walk into that garbage in danger with them, to me it's kind of the same as throwing them out to the wolves. 'Cause they can't get out. They can't choose to not be at risk.

5 Discussion

We now take a step back from our findings, surfacing broader lessons and making concrete recommendations for technologists wishing to support VSP-client interactions.

5.1 Broader Lessons for Technologists

Tensions and Challenges. Our findings surface a number of tensions and challenges that influence how VSPs and their clients use technology. These must be understood and considered by technologists wishing to work in this space.

Limited resources on the part of both clients and VSPs. For example, clients may not have access to cell phone plans, limiting their communication technology choices to those that support WiFi (e.g., Facebook Messenger). They also may have limited memory on their devices, or may frequently change devices and phone numbers. Technology solutions must take into account these potential limitations.

Limited and varied technology expertise. We found that computer security and privacy literacy and practices varied widely among VSP staff and clients. Participants' defensive strategies ranged from technologically advanced (e.g., using Faraday bags) to abstaining from technology. Participants described knowledge gaps among clients (e.g., clients not realizing that their Facebook profiles can be found via a web search), but we also spoke with survivors who are going to extensive lengths to protect themselves digitally. Technology must be designed for this range of knowledge and expertise, and there may be a role for computer security education and training specifically designed for VSPs and their clients.

Balancing client trust and technology access with safety. VSPs must balance building client trust with enforcing rules intended to protect clients and the VSP. As one of our participants put it, a VSP that it is too strict in terms of rule enforcement risks becoming, in some ways, like a trafficker to its clients. Even well-intentioned rules and guidelines around technology use can ultimately reduce safety as clients figure out ways to circumvent the rules. Thus, our participants commonly gave clients space to make their own technology-related choices (echoing prior findings about journalists deferring to the choices of their sources [22]).

Double-edged sword of technology. Access to technology can be a critical part of recovery — survivors can connect with new or former support networks, communicate with VSPs, and use technology for job searches and other critical tasks. However, this same access opens survivors up to potential risks, including being tracked down or monitored by former/future traffickers and being exposed to content that may make recovery harder. This tension echoes findings in the domestic violence context [21].

Balancing safety planning with client mental health. Finally, solutions must take into account the trauma and psychological challenges that survivors face — and avoid "triggering" survivors or causing them to be unnecessarily fearful.

Lack of Systematization and Need for Personalization. Our findings suggest that there is little systematization among VSPs around technology in VSP-client interactions and safety planning. For example, one organization uses WhatsApp because they perceive it to be more secure than Facebook, while many other participants discussed commonly using Facebook to communicate with clients. These differences may stem from factors such as: differences in the technology expertise and experiences of the VSPs and their clients; the fact that, in some organizations, VSP staff work relatively independently without top-down restrictions; and the fact that different clients face different risks and thus different mitigation strategies are necessary or effective.

These observations lead us to two conclusions: First, there may be benefit in systematizing computer security related guidelines and trainings for VSP staff, to help inform them about potential risks and benefits with different technology choices. Second, technology-based interventions cannot be "one size fits all" but must enable personalized approaches for survivors and VSPs in different situations.

5.2 Directions for Technologists

Authentication and First Contact. Technology can help VSPs reach out to potential clients, e.g., through the text messaging program discussed by some of our participants. However, a challenge with directly contacting trafficking victims or survivors is how to authenticate the first contact and build trust. There may be opportunities for technology designers to help address this challenge, e.g., through the (re)design of messaging tools for this population or through rigorous A/B testing of different message content for direct outreach to potential clients.

In the other direction, some participants discussed how difficult it can be for clients to find or contact VSPs when they are looking for help. Possible technology-based improvements here include real-time chat systems to replace phone hotlines (as "sometimes picking up the phone...is not an option", P7) and proactive help by search engines that suspect a user is attempting to find trafficking-related resources.

Designing for VSP-Client Communications. Based on our

findings, we recommend those designing for secure (i.e., hidden from the trafficker) client-VSP communications take into account the following lessons:

- Raising the trafficker's suspicions can be dangerous, so sensitive messages should look innocuous or be easily hidden to provide plausible deniability around the content, intent, and/or recipient of the message.
- It is also important to account for the complex psychology of the victim-trafficker relationship, and how this makes it challenging for the victim to keep secrets (e.g., passwords) from the trafficker.
- Solutions must work with devices with varying levels of functionality (e.g., phones with limited memory) and in the face of changing phone numbers and devices.
- It is important to plan for adversaries (traffickers) with physical access to a client's device.
- Solutions that can fit into existing popular communication platforms and/or existing VSP-client communication habits will have the greatest success in adoption.
- Prior work on domestic violence has shown that taking steps to remove an abuser's access to an account can be dangerous [11]. We suggest research on ways to support secure communication *within* otherwise compromised accounts, e.g., via a hidden secondary messaging interface.
- Finally, the client's ability to easily access and re-access the intended information is crucial — but must be balanced with protecting the same information from a potential local or remote adversary.

Publicly Available Information. The survivor leaders we spoke with are already extremely cautious in terms of digital security, yet there are data sources and tools/services that they have no control over that can compromise their safety. For example, traffickers utilize public records (e.g., DMV records) to track down former victims. To try to combat this, government address-confidentiality programs [23] provide qualified individuals with an alias mailing address. However, this is not a panacea; there are an unknown number of third-party services that pull public data and market themselves as an easy way to find people on the Internet. Even if a survivor qualifies for the confidentiality program, sensitive data could already exist on these people-finding services, and it is unclear how long it takes for new information to replace old.

Furthermore, this problem affects the general public as it enables a host of other crimes such as stalking and "doxxing" (releasing sensitive information publicly). We suggest future work study this ecosystem and develop solutions for helping people protect themselves — for example, streamlining the process of opting out of these people-finding services and helping users renew the opt-outs when/if they expire.

Supporting Safe Technology Use in Shelters. We believe that the computer security and privacy community can work with VSPs to develop ways for shelter residents to safely utilize technology. This is especially imperative for VSPs working with youth. These VSPs are in a difficult position as the youth they work with tend to be avid technology users but may not understand all the risks inherent in active technology use. In addition, they may be using technology to communicate with unsafe individuals such as their trafficker or potential trafficker. We found that, in response, VSPs tend to take an approach of strict regulation of technology use for young clients in particular, locking up phones at night and heavily regulating and monitoring computer use. While this is done out of the best intentions to protect the clients, it is (as discussed) commonly circumvented by clients.

Thus, it is clear that technology abstinence is not a reasonable solution for client safety planning. These findings highlight the need for members of the computer security and privacy community to work with VSPs to develop solutions and/or provide education to help strike the right balance.

Integrate VSPs and Survivors in Solution Development. Finally, echoing an increasingly common refrain in usable security, we note that it is critical to design technologies in a way that is deeply informed by the needs, constraints, and use cases of target populations. Though our work provides a foundation for technologists working in this space, future researchers should continue to connect with VSPs and survivors to design and evaluate any technology interventions.

> P14: Getting advice from the people who are using the technology that y'all are creating is a big part of moving forward. Because the moment y'all stop listening...to those of us who us are on these front lines using this technology to help ... the moment that that stops happening is the moment that y'all stop growing and being effective.

5.3 Limitations

Our study is qualitative, not quantitative; thus, our sample size is small and does not allow us to draw quantitative, generalizable conclusions. Self-reported data also has limitations such as recall and observer bias. Additionally, our participants are based primarily in urban areas in the U.S. and our results relate mainly to sex trafficking and female survivors. Thus, our results do not represent all possible VSP-client interactions. We believe there is more to uncover with regards to providing services to survivors of labor trafficking and survivors of other genders. For example, P15 talked about how her organization opened a shelter for all genders, allowing them to take in labor trafficking survivors and families. These are areas that call for further exploration with regards to how technology plays into these new dynamics.

6 Conclusion

Victim service providers play a critical role in the recovery of survivors of human trafficking. In this work, we conducted 17 semi-structured interviews with VSP staff members and survivor leaders, surfacing the ways technology is involved in VSP-client interactions, as well as the computer security and privacy concerns and mitigation strategies associated with those interactions. Key contributions of this work include detailing the various tensions that VSPs face when using technology in their interactions with clients and providing concrete recommendations for technologists who wish to support VSPs and trafficking survivors.

7 Acknowledgements

We are deeply grateful to our participants for taking the time to share their experiences and perspectives. We thank Kirsten Foot and Tadayoshi Kohno from the University of Washington for their invaluable help and advice throughout this project. We thank Robert Beiser, Sherrie Caltagirone, Kelly Mangiaracina, Lauren Moussa, Taylor Naber, Johnna White and other members of the anti-trafficking community for their assistance in developing the interview protocol and facilitating connections within the VSP community. Tiffany Chen and Kiron Lebeck kindly read the draft of the paper. Finally, we thank the anonymous reviewers for their feedback. This work is supported in part by the National Science Foundation under Awards CNS-1463968 and IIS-1748903.

References

[1] Disappearing Message. `https://onetimesecret.com/`.

[2] Vanessa Bouché and Thorn. Survivor Insights: The Role of Technology in Domestic Minor Sex Trafficking. Thorn, 2018. `https://www.wearethorn.org/survivor-insights/`.

[3] Virginia Braun and Victoria Clarke. Using Thematic Analysis in Psychology. *Qualitative Research in Psychology*, 3(2):77–101, 2006.

[4] Caliber. Evaluation of Comprehensive Services for Victims of Human Trafficking: Key Findings and Lessons Learned, 2007. `https://www.ncjrs.gov/pdffiles1/nij/grants/218777.pdf`.

[5] Rahul Chatterjee, Periwinkle Doerfler, Hadas Orgad, Sam Havron, Jackeline Palmer, Diana Freed, Karen Levy, Nicola Dell, Damon McCoy, and Thomas Ristenpart. The Spyware Used in Intimate Partner Violence. In *IEEE Symposium on Security and Privacy*, 2018.

[6] DARPA. Memex (Domain-Specific Search). `https://opencatalog.darpa.mil/MEMEX.html`.

[7] Nicola Dell, Vidya Vaidyanathan, Indrani Medhi, Edward Cutrell, and William Thies. "Yours is Better!": Participant Response Bias in HCI. In *ACM CHI Conference on Human Factors in Computing Systems*, 2012.

[8] Martin Emms, Budi Arief, and Aad van Moorsel. Single Use URL Access Codes. `http://research.cs.ncl.ac.uk/cybercrime/no-follow-url.php`.

[9] Martin Emms, Budi Arief, and Aad van Moorsel. Electronic footprints in the sand: Technologies for assisting domestic violence survivors. In *Annual Privacy Forum*, pages 203–214. Springer, 2012.

[10] Diana Freed, Jackeline Palmer, Diana Minchala, Karen Levy, Thomas Ristenpart, and Nicola Dell. A Stalker's Paradise: How Intimate Partner Abusers Exploit Technology. In *ACM CHI Conference on Human Factors in Computing Systems*, 2018.

[11] Diana Freed, Jackeline Palmer, Diana Elizabeth Minchala, Karen Levy, Thomas Ristenpart, and Nicola Dell. Digital technologies and intimate partner violence: A qualitative analysis with multiple stakeholders. *Proceedings of the ACM on Human-Computer Interaction*, 1(CSCW):46, 2017.

[12] Felicity Gerry, Julia Muraszkiewicz, and Niovi Vavoula. The role of technology in the fight against human trafficking: Reflections on privacy and data protection concerns. *Computer Law & Security Review*, 32(2):205–217, 2016.

[13] Catherine Glenn and Lisa Goodman. Living with and within the rules of domestic violence shelters: A qualitative exploration of residents experiences. *Violence against women*, 21(12):1481–1506, 2015.

[14] Tamy Guberek, Allison McDonald, Sylvia Simioni, Abraham H Mhaidli, Kentaro Toyama, and Florian Schaub. Keeping a Low Profile?: Technology, Risk and Privacy among Undocumented Immigrants. In *ACM CHI Conference on Human Factors in Computing Systems*, 2018.

[15] Kashmir Hill. How Facebook Figures Out Everyone You've Ever Met, November 2017. `https://gizmodo.com/how-facebook-figures-out-everyone-youve-ever-met-1819822691`.

[16] ILO and Walk Free Foundation and IOM. Global Estimates of Modern Slavery, 2017. `https://www.ilo.org/global/topics/forced-labour/statistics/lang--en/index.htm`.

[17] Mark Latonero, Genet Berhane, Ashley Hernandez, Tala Mohebi, and Lauren Movius. *Human trafficking online: The role of social networking sites and online classifieds*. University of Southern California, Center on Communication Leadership & Policy, 2011.

[18] Mark Latonero, Jennifer Musto, Zhaleh Boyd, Ev Boyle, Amber Bissel, Kari Gibson, and Joanne Kim. *The rise of mobile and the diffusion of technology-facilitated trafficking*. University of Southern California, Center on Communication Leadership & Policy, 2012.

[19] Mark Latonero, B Wex, M Dank, and S Poucki. *Technology and labor trafficking in a network society*. University of Southern California, Center on Communication Leadership & Policy, 2015.

[20] Nelson Lim, Sarah Michal Greathouse, and Douglas Yeung. The 2014 Technology Summit for Victim Service Providers. RAND Corporation, 2014. https://www.rand.org/pubs/conf_proceedings/CF326.html.

[21] Tara Matthews, Kathleen O'Leary, Anna Turner, Manya Sleeper, Jill Palzkill Woelfer, Martin Shelton, Cori Manthorne, Elizabeth F Churchill, and Sunny Consolvo. Stories from survivors: Privacy & security practices when coping with intimate partner abuse. In *ACM Conference on Human Factors in Computing Systems (CHI)*, 2017.

[22] Susan E McGregor, Polina Charters, Tobin Holliday, and Franziska Roesner. Investigating the Computer Security Practices and Needs of Journalists. In *USENIX Security Symposium*, 2015.

[23] National Network to End Domestic Violence (NNEDV). Address Confidentiality Programs. https://nnedv.org/mdocs-posts/state-by-state-listing-of-address-confidentiality-programs-2016/.

[24] Polaris. On-Ramps, Intersections, and Exit Routes: A Roadmap for Systems and Industries to Prevent and Disrupt Human Trafficking. Polaris, 2018. https://polarisproject.org/a-roadmap-for-systems-and-industries-to-prevent-and-disrupt-human-trafficking.

[25] Rebecca S Portnoff, Danny Yuxing Huang, Periwinkle Doerfler, Sadia Afroz, and Damon McCoy. Backpage and Bitcoin: Uncovering human traffickers. In *23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017.

[26] Samantha Raphelson. Cyntoia Brown Case Highlights How Child Sex Trafficking Victims Are Prosecuted, December 2017. https://www.npr.org/2017/12/01/567789605/cyntoia-brown-case-highlights-how-child-sex-trafficking-victims-are-prosecuted.

[27] Dominique E Roe-Sepowitz, Kristine E Hickle, Jaime Dahlstedt, and James Gallagher. Victim or whore: The similarities and differences between victim's experiences of domestic violence and sex trafficking. *Journal of Human Behavior in the Social Environment*, 24(8):883–898, 2014.

[28] Lucy Simko, Ada Lerner, Samia Ibtasam, Franziska Roesner, and Tadayoshi Kohno. Computer Security and Privacy for Refugees in the United States. In *IEEE Symposium on Security and Privacy*, 2018.

[29] Jennifer Stoll, W. Keith Edwards, and Kirsten A. Foot. Between Us and Them: Building Connectedness Within Civic Networks. In *ACM Conference on Computer Supported Cooperative Work (CSCW)*, 2012.

[30] United Nations Office on Drugs and Crime (UNODC). What is Human Trafficking? https://www.unodc.org/unodc/en/human-trafficking/what-is-human-trafficking.html.

[31] Michael Wilson, Nate Schweber, and Ashley Southall. Brothels, Gambling and an Ex-Detective Mastermind: Officials Detail N.Y. Police Scandal, September 2018. https://www.nytimes.com/2018/09/13/nyregion/nypd-officers-arrested-prostitution-gambling.html.

[32] youth.gov. Transition & Aging Out. https://youth.gov/youth-topics/transition-age-youth.

A Interview Protocol

Our study involved qualitative semi-structured interviews. As such, the questions below served as a guide for our interviews, but individual interviews varied based on participants' responses and specific experiences. We let the participant's responses direct the conversation, asking relevant follow-up questions and skipping irrelevant questions as appropriate.

General Background

1. What is your job title and role? In your job, what are the main services you provide to clients?
2. Would you describe your technology comfort level as high, medium, or low? Why?
3. How many years old is your organization? How many employees are there? What are the main services your organization provides?
4. How do you refer to the individuals you work with (e.g., clients, survivors, participants, guests, etc.)?
5. What kinds of clients do you mainly work with? What type of exploitation have they endured? What is their nationality, their age range?
6. What are the most common pathways you see for your clients to be exploited? How has the rise of technology made the situation worse or better? What are your fears and hopes looking ahead?
7. FOSTA (Allow States and Victims to Fight Online Sex Trafficking Act) just recently passed. It amends section 230 of the Communications Decency Act so that laws relating to sexual exploitation of children or sex trafficking can apply to third-party content providers. How do you think the legislation might affect your work, if at all? How do you think it might affect the broader trafficking ecosystem, if at all?
8. What are the key factors that cause revictimization? How do you think technology facilitates revictimization, if at all? If you are worried a client has been revictimized, how (if at all) do you stay in contact with them?
9. How do you get help when you encounter issues with technology and computer security at work?

Client First Contact

1. How do cases typically come to your attention (e.g., law enforcement, community members, homeless shelters, victims directly reaching out, direct outreach, etc.)?
2. What means do you have for victims to directly contact your organization (e.g., phone, email, text, web form, online chat, etc.)?
3. Do clients ever reach out on their own? If so, how do you think they find you (e.g., word of mouth, Google, etc.)? What evidence do you have of this?
4. Walk me through what happens when an individual reaches out to your organization through any of those means. What do you do or not do to protect the individual's identity? How do you vet potential clients?
5. Is there an instance of someone contacting your organization that sticks out to you as particularly memorable?
6. Sometimes victims of trafficking are forced to recruit new victims. With that in mind, does your organization do anything to reach the traffickers, as some of them may be victims themselves?

Client Intake

1. How do you record and store client information? What do you or your organization consider confidential? What worries you most in terms of the security and privacy or confidentiality of the information? Are you worried that the information may be subpoenaed?
2. What do clients usually have with them when they arrive (e.g., devices)? Are there any rules regarding what clients are or are not allowed to have with them before receiving services and/or entering the shelter?

Day-to-Day Interactions

1. In your day to day work as you're interacting with clients, what are common (technology and non-technology) frustrations, worries, or fears? What would you say are the common frustrations, worries, or fears of your clients?
2. How do you communicate with clients in general (e.g., phone, email, SMS, social media, etc.)? For each mode of commuication:
    - How did you choose this mode?
    - What do you commonly communicate about?
    - Have you ever been afraid that someone might be eavesdropping on the conversation?
    - Are there things you purposefully avoid talking about or don't feel comfortable talking about via this mode of communication?
3. What access to technology do survivors have through your organization? Do you provide any devices, apps, software, web programs, wifi access? If there is a computer for clients: What do people use it for?

Additional Organization and Client Safety

1. Is the location of your organization and/or shelter confidential?
2. Do you have rules or guidelines about technology that clients have to follow (e.g., turning off location services on their phones)? Do you have rules or guidelines for visitors? How, if at all, do you enforce these rules?
3. What steps do you take to keep yourself safe? What are things you've noticed your clients doing to keep themselves safe?

Reactions to Prototypes

When presenting these prototypes, in order to minimize participants response bias [7], we explicitly told participants that we had not created the prototypes, that we were interested in both positive and negatives reactions, and that our goal in presenting them was to make the conversation around potential technology solutions more concrete.

We presented to participants two prototypes: single-use URLs and disappearing messages [1, 8, 9]. For each prototype, we asked participants:

1. Would you or your organization ever use something like this? What potential benefits, if any, do you see?
2. What would you change about this tool? Are there any new threats or concerns that it would raise for your clients or your organization?

Closing

1. If you had a magic wand and could solve any issue in this space, what would you do first?
2. What do you want to tell the computer security and privacy community to focus on with regards to helping victim service providers?
3. What drew you to participate in this study?
4. Is there anything you'd like to add about technology use in your job that I didn't ask about?

Clinical Computer Security for Victims of Intimate Partner Violence

Sam Havron*,[1] Diana Freed*,[1] Rahul Chatterjee[1] Damon McCoy[2]
Nicola Dell[1] Thomas Ristenpart[1]
[1] *Cornell Tech* [2] *New York University*

Abstract

Digital insecurity in the face of targeted, persistent attacks increasingly leaves victims in debilitating or even life-threatening situations. We propose an approach to helping victims, what we call *clinical computer security*, and explore it in the context of intimate partner violence (IPV). IPV is widespread and abusers exploit technology to track, harass, intimidate, and otherwise harm their victims. We report on the iterative design, refinement, and deployment of a consultation service that we created to help IPV victims obtain in-person security help from a trained technologist. To do so we created and tested a range of new technical and non-technical tools that systematize the discovery and investigation of the complicated, multimodal digital attacks seen in IPV. An initial field study with 44 IPV survivors showed how our procedures and tools help victims discover account compromise, exploitable misconfigurations, and potential spyware.

1 Introduction

As computers and other digital technologies take an increasingly central role in people's lives, computer insecurity has for some people become debilitating and even life-threatening. Activists and other dissidents are monitored [7, 23, 25, 26], journalists are harassed and doxed [10], gamers are subjected to bullying [9], and abusers are exploiting technology to surveil and harass their intimate partners [35]. Traditional security mechanisms most often fail in the face of such targeted, personalized, and persistent attacks.

A different approach for helping targeted individuals is what we call *clinical computer security*. The idea is to provide victims of dangerous attacks the opportunity to obtain personalized help from a trained technologist. Just like people visit doctors for health problems, seek out lawyers when suffering legal troubles, or hire accountants for complex tax situations, so too should victims of dangerous digital attacks have experts to assist them. But while these other examples of professional services have a long history leading to today's best practices, for computer security we are essentially starting from scratch: existing technology support services are ill-suited for helping victims in dangerous situations. The research challenge is therefore to develop rigorous, evidence-based best practices for clinical approaches to computer security, as well as design the supporting tools needed to help victims.

In this paper we explore clinical computer security in the important context of intimate partner violence (IPV)[1]. IPV is widespread, affecting one out of three women and one out of six men over the course of their lives [32]. Prior work has shown how abusers exploit technology to harass, impersonate, threaten, monitor, intimidate, and otherwise harm their victims [8, 14, 19, 20, 27, 35, 43]. Prevalent attacks include account compromise, installation of spyware, and harassment on social media [20,27]. In many cases digital attacks can lead to physical violence, including even murder [34]. Unfortunately, victims currently have little recourse, relying on social workers or other professionals who report having insufficient computer security knowledge to aid victims [19].

Working in collaboration with the New York City Mayor's Office to End Domestic and Gender-Based Violence (ENDGBV), we designed, prototyped, and deployed a clinical computer security service for IPV survivors.[2] Doing so required not only developing first-of-their-kind protocols for how to handle face-to-face consultations while ensuring safety for both clients (the term we use for IPV victims in this context) and technology consultants, but also the design and implementation of new technical and non-technical instruments that help to tease apart the complicated, multifaceted digital insecurities that clients often face.

We designed a first-of-its-kind consultation procedure via a careful, stakeholder-advised process that made client safety paramount. Initial designs were refined over two months via 14 focus groups with a total of 56 IPV professionals, including

*These authors contributed equally to the paper.

[1] A full version of this paper and materials is available at: `https://www.ipvtechresearch.org`

[2] Our initial and refined research protocols were approved by our institution's IRB and the ENDGBV leadership.

social workers, police, lawyers, mental health professionals, and more. This led to substantive feedback and refinements, culminating in a consultation design that appropriately takes into account the socio-technical complexity of IPV and the unique risks that clients face.

Our consultation procedure starts with a referral from an IPV professional, and then proceeds through a face-to-face discussion where we seek to *understand* the client's technology issues, *investigate* their digital assets via programmatic and manual inspections, and *advise* them and the referring professional on potential steps forward. This last step, importantly, involves procedures for clearly communicating new found information about technology abuse so that professionals can help clients with safety planning. Supporting this understand-investigate-advise framework are a number of tools that we created, including: a standardized technology assessment questionnaire (the TAQ); a diagrammatic method for summarizing a client's digital assets called a technograph; succinct guides for helping consultants and clients manually check important security configurations; and a new spyware scanning tool, called ISDi, that programmatically detects whether apps dangerous in IPV contexts are installed on a client's mobile devices.

After completing our design process, we received permission to meet with clients in order to both help them and field test our consultation procedures and tools. Thus far, we have met with 44 clients and our consultations have discovered potential spyware, account compromise, or exploitable misconfigurations for 23 of these clients. The tools we developed proved critical to these discoveries, and without them our consultations would have been significantly less effective. For clients with discovered issues, we provided advice about improving security, in parallel with appropriate safety planning guided by case managers knowledgeable about their abuse history and current situation. Many other clients expressed relief that our consultations did not discover any problems.

Professionals at the FJCs have uniformly responded positively to our field study, and reported that the consultations are helpful to their clients. Demand for consultations has increased and we are performing them on an ongoing basis. More broadly, our tools, including ISDi, will be made open-source and publicly available, providing a suite of resources for testing the replicability of our clinical approach in other locations. Whether our approaches and methods can be useful for other targeted attack contexts beyond IPV is an interesting open question raised by our work. We discuss this question, and others, at the end of the paper.

2 Towards Clinical Computer Security

This paper considers targeted attacks in the context of intimate partner violence (IPV). Prior work indicates that IPV victims are frequently subject to technology abuse [8, 14, 19, 20, 27, 35, 43], and a taxonomy by Freed et al. [20] includes four broad categories: (1) ownership-based attacks in which the abuser owns the victim's digital accounts or devices, giving them access and control; (2) account or device compromise; (3) harmful messages or posts (e.g., on social media); and (4) exposure of private information online. Abusers use access to victim devices or accounts to setup dangerous configurations, such as adding their fingerprints to be accepted for device login, configuring devices to synchronize data with an abuser-controlled cloud account, or setting up tools such as Find My Phone to send location updates to an abuser's email address. Another avenue is installation of spyware apps that provide powerful features for monitoring devices [8].

Technology abuse in IPV is certainly complex in the aggregate, but even specific individuals suffer from complex, multifaceted threats. To concretize this, we give an example. For privacy reasons it is not any particular person's story. However, it is representative of many of the actual client situations we have encountered in our work.

Example scenario, Carol's experience: *Carol's now ex-husband subjected her to several years of increasing physical, emotional, and technology abuse before she obtained an order of protection, physically moved out, and filed for divorce. They are in a custody battle over their two children, ages four and ten, who live with the ex-husband part of the time.*

Carol knows that he installed spyware on at least one of her devices, because she found the purchase of mSpy on their joint credit card statement. Additionally, he had access to her private photos that he then posted on Facebook. He would also routinely, over the period of a year, "hack" into her online accounts, posing as her in efforts to further alienate her from her friends and family. He even locked her out of her GMail account by changing the recovery emails and phone number to his, which was devastating to her career in sales because it contained her business contacts.

Carol currently has five devices: a new Apple iPhone that is her primary device, two Android phones used by her children, an Apple iPad tablet bought for her children by her ex-husband, and a several-year-old Apple iPhone originally bought for her by her ex-husband. She routinely uses Facebook, a new GMail account (since her old one was stolen by her ex-husband), and a variety of other social media apps that are important for her work in sales.

This representative example highlights the complexities faced by IPV victims. Carol has a complicated *digital footprint* that includes a wide variety of devices and online accounts, some of which may be linked (e.g., different devices may have stored authentication credentials for different online accounts). She has complicating *entanglements*, meaning digital or personal relationships that may enable or complicate tech abuse, or render its mitigation more difficult. In Carol's case, the abuser has access to the children's devices, owns some of the devices in her digital footprint, and her need to use social media for her career limits options for preventing

harassment via it. The complex timeline of events, such as when she physically moved out and when the children visit the abuser, may be directly relevant to the tech problems she is facing. Finally, there is also the risk that blocking digital attacks causes an *escalation* of abuse, such as triggering physical violence as the abuser seeks to regain his control.

One avenue for improving on the status quo is pursuit of new technology designs that better resist such targeted attacks. While doing so is very important, future designs will not help IPV victims in the near term. More pessimistically, it may in fact *never* be possible to rule out damaging attacks by highly resourced, determined adversaries against lower-resource victims. We therefore need complementary socio-technical approaches to helping victims.

Unfortunately, existing victim support services struggle to help with complicated tech abuse situations [19, 27]. The case workers, lawyers, police, and other professionals that work with victims report having insufficient tech expertise to help victims with digital threats [19]. There currently are no best practices for how to discover, assess, and mitigate tech issues [19]. Existing tools for programmatically detecting spyware are ineffective [8], and the state-of-the-art in practice is that professionals assume spyware on phones if a victim reports that the phone is acting strangely [20].

Commercial tech support services (e.g., Geek Squad [36] or phone stores) are unfortunately not a ready solution for addressing tech abuse. Prior work reports that victims occasionally use these services [19, 27], but that even when used they often fail to effectively diagnose problems [20]. We believe this is because commercial IT support professionals do not have context-specific training needed to identify and handle complex tech abuse situations prevalent in IPV. In the worst case, they put victims into more danger due to a lack of appropriate safety planning. Finally, victims with lower socio-economic status may find such services hard to access.

Clinical computer security. We target new approaches for victims to obtain personalized and appropriately contextualized support from a trained technologist. There are a handful of existing efforts from which we drew some inspiration. The Citizen Lab [13] and related Citizen Clinic [1] have been working for several years with targets of government persecution, a recent Technology-Enabled Coercive Control (TECC) clinic was established for IPV victims in Seattle [2], and individual computer security experts have long informally volunteered to aid those suffering attacks [24]. However, there has been little research into how such personalized security services should be systematically designed and deployed.

We propose an approach that we call *clinical computer security*. The goal is to develop, in a rigorous, evidence-based way, a set of best practices for how a technology *consultant* can assist a victim — called the *client* in such a service context — with digital insecurity. Best practices will need to encompass a range of issues, including how to setup and run clinics, recruit and train volunteers or paid professionals to staff them, deal with the many legal issues that will inevitably arise, and how consultations with clients should proceed. In this initial work we focus on designing and prototyping consultations, the fundamental element of any clinical approach. We discuss other aspects of running a clinic in Section 8.

The challenges faced in client consultations. As seen in Carol's example, individual IPV victims often experience a wide range of tech problems. They have a complex digital footprint, including multiple devices and online accounts, each of which can be a vector for abuse. They often have many nuanced entanglements. Existing tools for detecting spyware have a high false negative rate [8]. To improve outcomes for IPV victims, we need to design a protocol for face-to-face consultations that can integrate into existing victim support infrastructure, help us understand the client's problems from their point of view, discover tech risks they may not be aware of, and safely advise them about what steps they could take to improve their computer security.

Of course, we can look to other disciplines that use clinical interventions for guidance, including medicine, social work, mental health counseling, and even legal practice. These areas have long histories leading to today's best practices, including common interview procedures such as standards for psychiatric assessments [28] or client-centered advocacy [31]. However, none of these disciplines speak to procedures for computer security, so while we incorporate ideas from them when useful, overall, we need new approaches.

3 Methods, Client Safety, and Ethics

We designed a client consultation protocol and associated instruments to improve computer security outcomes for IPV victims via face-to-face discussions and both programmatic and manual investigations of their digital assets (i.e., their computing devices and online accounts). Here we discuss our iterative design methods that optimized for client safety.

IPV victims can be in dangerous and even life-threatening situations, and we made client safety and well-being central to our methodological approach. No consultation process will ever be perfect, in the sense that one could guarantee that all of the client's technology problems will be discovered, accurately assessed, and successfully mitigated. Indeed, the current status quo is reportedly missing many issues, accurately assessing few of them, and only sometimes properly mitigating them [19]. To make progress, we must develop research protocols that respect client well-being, are cognizant of safety risks, weigh the relative benefits of research to those risks, and, overall, minimize the potential for harm.

We therefore put into place a multifaceted strategy for performing this research responsibly. We partnered with the New York City Mayor's Office to End Domestic and Gender-Based Violence (ENDGBV) [16], which runs Family Justice

Centers (FJCs) [17] in each borough of New York City (NYC). The FJCs provide a diverse array of resources for IPV victims, including police, legal, mental health, housing assistance, and more. All research protocols were approved not only by our institutional IRB but also by the ENDGBV leadership.

Our consultation protocols went through a thorough, iterative design process that: (1) started with initial designs grounded in findings from prior work [19, 20, 27]; (2) a two-month process of iterative and incremental refinements driven by focus groups with relevant IPV professionals; (3) a review and approval process with the ENDGBV leadership of our refined protocols and instruments for client consultations; and (4) an ongoing refinement process that was responsive to needs that arose during client consultations.

This process maximized the amount of meaningful research we could do *before* interacting with clients. In step (2) we conducted 14 rounds of iterative design with a total of 56 IPV professionals. Each round involved a 60–90 minute focus group held at one of the FJCs, in which we summarized the current consultation design, demonstrated our methods, and gave participants copies of our questionnaires and materials. They were encouraged to edit, rewrite, and redesign them. We took detailed notes. Data analysis was performed immediately after each focus group, consisting of a detailed assessment of our notes with a specific focus on suggestions for improvements or changes. In subsequent sections, we give examples of quotes emanating from focus groups that help explain, or led to changes in, our consultation protocol. These quotes are illustrative and not intended to represent a comprehensive thematic analysis of the focus groups.

After nine rounds of changes based on participant feedback, we had several consecutive focus groups that did not elicit any new suggestions. We therefore determined our procedure and methods were ready for a review and approval process with the ENDGBV. This involved presentations to, and discussions with, ENDGBV leadership about our protocol. Ultimately, we and the ENDGBV concluded that it was ready for use with clients due to (i) the sufficiency of safety procedures we put in place to minimize potential harm to clients, and (ii) the fact that the ENDGBV leadership concluded that our research would benefit their clients. We discuss our safety procedures for consultations in detail in Section 6.

Finally, we note that safety issues extend also to the well-being of the participating researchers. In addition to the potential for vicarious trauma or other emotional strain, spyware could in theory leak recordings of consultations to abusers. We discuss self-care and researcher safety in Section 6.

4 A Consultation Protocol for IPV Victims

We created and refined a first-of-its-kind protocol for conducting a tech consultation in which a trained volunteer with expertise in technology meets face-to-face with an IPV victim. We refer to the volunteer as the tech consultant, or simply consultant, and the victim as the client. A diagrammatic overview of our consultation procedure appears in Figure 1. We give a high level walk-through, and provide more details about various aspects of the procedure starting in Section 4.1. Throughout we give examples of how the iterative design process with stakeholders impacted our design.

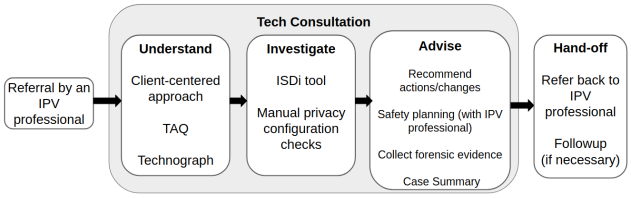


Figure 1: Summary of how a client participates in a tech consultation, beginning with a referral from an IPV professional.

We use a referral model that ensures safe integration of our consultations into NYC's existing survivor support services. Upon setting an appointment and meeting with a client, we use a procedure that we refer to as *understand-investigate-advise* (UIA). This emphasizes three important aspects of the consultation: understanding tech risks in the client's digital footprint, investigating their root causes, and providing advice about how they might improve their digital security.

To maximize the efficacy of the UIA procedure, we developed a number of non-technical and technical instruments to aid consultants, including: a technology assessment questionnaire, a diagrammatic approach called a technograph for mapping a client's digital footprint, guides for reminding consultants how to check security settings for common services and devices, and a software tool called ISDi (**I**PV **S**pyware **Di**scovery) that can safely detect the kinds of spyware reported as used in IPV settings by prior work [8]. We also developed a number of training materials, checklists, and associated protocols to help prepare consultants for meeting with clients. These instruments were refined via focus groups with professionals as well as in an ongoing manner as we gained experience working with clients.

4.1 Integration into Client Support Services

One of the first questions we faced is how to make tech consultations fit into the broader landscape of victim support services, such as legal, financial, and social services. Although consultants will be qualified to provide assistance with technology security and privacy, they will not necessarily be qualified to help with overall safety planning, legal advice, mental health, or other aspects of a client's case. It is therefore essential that other IPV professionals are able to assist the client before, during, and after a consultation.

To ensure all clients have appropriate support from an IPV professional, we use a **referral model** in which consultants only see clients that are referred to them by other IPV professionals for potential tech problems. Using a referral model

has significant safety and procedural benefits over alternative models. In particular, the referring professional will know the client's background and abuse history and be qualified to help them safety plan around the results of the consultation (e.g., if it is safe to change their privacy settings, remove apps, etc.). If possible, and if the client is comfortable, we encourage the referring professional (or client case manager) to be present during the consultation so that they can also discuss their questions or concerns with the consultant.

Referral models have other benefits as well. They allow us to balance client anonymity with continuity of care, since the professional can serve as a safe communication channel between the consultant and client. This specifically enables consultants to perform **followups** for issues that cannot be fully investigated during a consultation. For example, we saw clients asking about esoteric or non-English apps, having browser extensions that are not on the extension market, and describing seemingly inexplicable tech situations. In such cases, we perform further research on the topic after the consultation, and communicate any discoveries back via the referring professional. If appropriate, the client may elect to participate in a second consultation, which happened a couple times so far in our work.

Regardless of followup requirements, when a consultation is complete (and with client permission) the consultant performs a **hand-off** procedure that communicates relevant findings to the referring professional. If the professional is in the room, this may happen at the end of the consultation. Otherwise, it happens via email or phone call. This hand-off is vitally important. First because it facilitates proper safety planning, as we discuss later in the section. In addition, it provides some reassurance to clients potentially frightened by a consultation's discoveries. As one professional described, our hand-off procedure:

> *"...might help the client feel a little bit more comfortable. 'Oh my gosh, I'm being tracked. At least I know there's an officer that can help me with this situation.' You're also aware of what's going on as a screener, as well as a case manager. I have three different backups. I think it was very well done."* (P36, Case Manager)

4.2 Understand-Investigate-Advise Procedure

When the client arrives for a consultation, we follow standard IPV advocacy practices and take a **client-centered approach** [31], which assumes the client knows best regarding their own situation and will be the one to make decisions. One professional described client-centered practice as:

> *"having a conversation with the client and ... letting the client formulate their decisions, their answers. [Professionals] cannot provide them with [answers] because they're the only ones who know what risks are being posed."* (P36, Case Manager)

Therefore, taking a client-centered approach, the consultant begins by asking the client what their main concerns are and/or what caused them to seek out a consultation. We refer to these as their chief concerns[3] and a primary goal of the consultant is to try to accurately identify them. For example, we heard clients express fear that spyware was installed on their devices, that their "phones were tapped", or that their abuser had access to information they should not have (e.g., a client's photos). In some cases the chief concerns are not very clear and take some gentle questioning to ascertain.

From this starting point, the tech consultant will utilize a wide range of instruments and tools that we have created to (1) **understand** the client's digital footprint and entanglements to identify potential avenues for harm; (2) **investigate** their devices, apps, and services to look for, and assess the root cause(s) of, their tech problems; and (3) **advise** clients on how they might move forward. See Figure 1.

Understanding footprint and entanglements. Prior work on tech and IPV [14, 19, 27, 34, 43] indicates that there are no best practices or standard procedures for asking about tech risks or understanding the root cause(s) of client concerns. The lack of standardized procedures may contribute to serious, on-going tech abuse being overlooked. We therefore created several instruments that help systematize the discovery and assessment of tech problems in IPV.

To systematize problem discovery, we created and refined a Technology Assessment Questionnaire, or **TAQ** (Figure 5 in the Appendix). We started with questions that aimed to uncover common problems surfaced in prior work [20], such as risk of device/account compromise if the abuser knows or can guess the client's passwords (e.g., their password is their child's birthday), or ownership-based risks, when the abuser is the legal owner of the client's devices or accounts. Feedback from focus groups helped us refine question wording, and include additional questions that professionals thought would be helpful. As one example, we received many suggestions on the importance of asking about children's devices. As one professional told us,

> *"[For parents] with younger kids, I think another question that might be important is asking if your children go on visits and if they take their electronics with them on visits."* (P40, Social Worker)

We added five questions about risks with children's devices. This feedback was particularly helpful, as we saw several cases in our field study of children's devices being the likely avenue by which the abuser had access to client data.

To support a client-centered approach, the TAQ is designed to be used as a reference to ensure consultants cover important

[3] In medicine, this would be called a chief complaint, but we feel that 'concern' is more client-centered.

topics, rather than as a prescribed interview format. The consultant lets the client lead the conversation and discuss topics they find important, which often touches on a subset of the TAQ. The consultant uses the TAQ to remember to raise remaining topics that the client may not have thought about. We arrived at this approach after early feedback from professionals that it is more empowering to let clients drive conversations, rather than peppering them with questions.

A challenge that came up in early consultations is building a mental map of the client's digital footprint and entanglements. Carol's example in Section 2 illustrates the potential complexity of client technology use. In the field, clients often came with half a dozen devices, many accounts, an involved abuse timeline, and various pieces of (often circumstantial) evidence of account or device compromise (e.g., the abuser keeps tracking or calling them despite changing phones). It is easy for consultants to lose track of relevant details.

We therefore created the **technograph**, a visual map loosely inspired by genograms, a technique used by clinicians in medicine and behavioral health to map family relationships and histories [22]. The technograph uses shapes and symbols to visually document relationships between (1) devices, (2) accounts, and (3) people (usually the client's family). Drawing connections between entities gives the consultant a clearer picture of potential sources of compromise. An example that may have been created discussing Carol's situation appears in the full version of this paper.

The technograph is particularly helpful to identify when abusers may have indirect access to a client's digital assets. For example, two-factor authentication for iCloud accounts can be bypassed if a child's device is a contact for the account. Another example is when family plans synchronize data across devices and accounts. The technograph allows tracing these potential indirect access routes more easily.

Investigating devices, accounts, and services. After using the TAQ and technograph to construct a clearer picture of the client's situation, the next phase of the consultation is to thoroughly investigate devices, accounts, or services that may be compromised by the abuser. We created tools that investigate in two ways: (1) by scanning the client's mobile devices for spyware or other unwanted surveillance apps using a new IPV Spyware Discovery (ISDi) tool that we built, and (2) by manually checking the privacy configurations of the client's devices, apps, and accounts. We discuss each in turn.

As we detail later, most clients have hundreds of apps on their devices. In addition to the threat of spyware-capable apps being installed surreptitiously, many otherwise legitimate apps may be configured by the abuser to act as spyware. For example, Google maps can be configured to update an abuser about the client's location, and while it provides various notifications that tracking is ongoing, their effectiveness is uncertain. We therefore have a dichotomy between unwanted and wanted apps, with the mere presence of the former being sufficient for a safety discussion whereas the latter require investigation into their configuration.

Detecting unwanted apps manually via the user interface (UI) will not work: many IPV spyware apps can effectively hide their presence from the UI [8]. Indeed, current state-of-the-art practice by non-technologist professionals is to use circumstantial evidence to conclude spyware is installed, e.g., if a phone acts "glitchy" it most likely has spyware and should be reset if not discarded [20]. We therefore constructed an IPV Spyware Discovery (**ISDi**) tool for detecting unwanted apps on a client's iOS or Android devices. It also checks if the device has been jailbroken (for iOS) or rooted (for Android), which may indicate that dangerous spyware is installed. With the client's permission, the consultant uses ISDi to programmatically obtain via USB connection the apps installed on their devices, highlighting ones that are known to be risky in IPV. Should the device be detected as rooted/jailbroken or any risky apps found, the consultant can discuss whether the client rooted the phone, recognizes the app, etc.

Our focus groups with professionals helped us iterate on the user flow and understand how best to integrate the tool into client consultations. We learned that clients and professionals want to view and understand the steps required to use the tool as well as visually examine the scan results. Professionals expressed concern about communicating to clients appropriately about privacy issues. One professional suggested that, during a consultation, we say that:

> *"We will see and go through every application on your phone, we will not see any information in your social media, texts, photos. We will only see the names of all the applications but not see anything inside any of the apps and give an example, such as, if you have WhatsApp, we will not see any conversation inside."* (P41, Case Manager)

Focus groups also led us to realize that both clients and consultants are consumers of the ISDi UI (see Figure 2). We therefore avoided language that would be too confusing or scary to a client. Finally, while we have not yet done a thorough user study of the tool, we have begun some initial user studies with IPV support organizations (e.g., TECC [2]) interested in integrating ISDi into their own procedures. We discuss this further in Section 8.

That leaves checking configurations of common apps that are often wanted but potentially dangerous, as well as checking built-in system services (e.g., "find my phone" features), account backup mechanisms, and authentication lists (e.g., registered fingerprints), all of which may be sources of vulnerability. The same holds for online accounts deemed important by the client (e.g., email and social media accounts). Unfortunately, checking the privacy of these accounts cannot be easily automated, not only due to lack of device or web interfaces to support querying this kind of data, but also because one needs to understand the context and have the client help identify

dangerous configurations. For example, in several cases we saw that the client's Facebook or GMail accounts had been accessed by devices the client could confirm as the abuser's.

To assist the consultant with these **manual investigations**, we constructed simple-to-follow guides for popular apps, device settings, and online service settings. For instance, our Google privacy configuration guide lists steps to check a device's login history, location sharing, photo sharing, and Google Drive backup settings. On iCloud we check family sharing, backups to iCloud, and if the abuser still has access to the account. We continue to expand the list of apps and services for which we have guides in response to ongoing work with clients, and currently cover Android (including Google maps and GMail), Apple (including iCloud and location sharing), Facebook, Instagram, and Snapchat. Unfortunately such guides may become out-of-date if software updates change configuration features. Future work on how to sustainably keep guides up-to-date will be needed (see Section 8).

Another benefit of performing manual investigations during consultations is that they serve as impromptu computer security training for clients, which prior work indicated is sorely needed [19]. In fact, many clients we met with did not know about security configuration features, and we were able to show them for the first time that, for example, they could tell what devices were logged into their GMail or Apple accounts. Clients often asked followup questions about security best practices during this part of the consultation, leading into an open-ended discussion about computer security.

Advising clients on potential next steps. In the final phase of the consultation, the consultant combines information gleaned from the understanding and investigation phases to assess the client's situation and, based on this assessment, discuss with the client (and professional, if present) what might be causing tech problems the client is experiencing. If the investigation phase yields any spyware, risky software, or privacy problems with the client's accounts and devices, these are discussed calmly with the client, including how the breach may have happened and potential actions that might remedy the situation. In these cases, the consultant can offer the client a printout that explains what was found and how it may be causing problems (see examples in the full version).

Before taking actions or changing any settings, it is essential that the client discuss their consultation results with a professional to perform **safety planning**. Ideally the professional should be familiar with the client's situation and abuse history, since this is necessary to highlight potential safety issues related to tech abuse. One professional said:

> *"Safety planning is such an individualized thing. I can think of some cases where it would be advantageous to leave the spyware on. I can think of some where we would want it gone immediately. If you can, just find a way to integrate it into the normal safety planning protocol."* (P37, Paralegal)

If the client's case manager is not present, the consultant asks the client if they would like to contact their case manager and/or receive immediate assistance from another on-site professional. Thus, even if the consultation has identified tech problems that are the likely causes of the client's concerns, in many cases, the client may leave the consultation with their devices and accounts unchanged. For a few clients we met with who had complicated scenarios, we encouraged them to schedule a follow-up consultation via their professional, so we could help them further after safety planning.

Consultations also provide new opportunities for collecting **forensic digital evidence**. The need for clients to document evidence of tech abuse is an issue that legal professionals discussed at length in our focus groups. If properly collected, such evidence may help a client secure an order of protection or aid a criminal investigation. Although clients may want to delete suspicious apps or reconfigure settings, our protocol has the consultant discuss with clients the potential benefits of documenting any discoveries before taking action. We asked professionals about how to handle forensic evidence, and they suggested various approaches, such as:

> *"I would definitely take photos. Because ultimately [a detective] will be investigating that report, but I will definitely take photos, write down the name of the app on my report."* (P39, Police Officer)

We therefore settled on the expedient approach of having the client (or a lawyer acting on their behalf) take a photo or screenshot of any discovered spyware, evidence of compromises, etc. As suggested in the quote above, this is actually the standard of evidence currently, at least in family court, and several clients we met with have ongoing court cases in which they plan to use evidence discovered via our consultations.

In many cases the consultation will not yield any tech problems or causes for concern, in which case the consultant may reassure the client that, at least, our approaches did not find any problems. We are careful to not dismiss any problems that remain unaddressed or unexplained by our consultation. If additional investigation is warranted, the consultant explains to the client that they will do more work and follow-up via the referring professional (as explained in Section 4.1).

Finally, at the end of a consultation, the consultant completes a **case summary** that documents (1) the client's chief concerns (in their own words), (2) the consultant's assessment of problems, (3) the results of the ISDi scan and manual configuration check-ups, and (4) advice or recommendations discussed with the client. This case summary is for internal use only[4] and provides useful documentation for the consultant (or other consultants) that can be used should the client request another consultation or need followup.

[4] In some contexts such written documentation may be ill-advised due to the potential threat of hostile subpoena by lawyers working for the abuser. In our work, FJC professionals felt this threat was remote since our consultations take place within a research study that maintains client anonymity.

4.3 Replicability

An important question for our consultation protocol is how to ensure a standard of care that can be maintained across different locations and by different consultants. Many of the tools we created help by systematizing the assessment and investigation of tech problems. To complement these, prior work in disease diagnosis [15], surgery [42], and aviation [11] suggests that simple checklists are a valuable tool for systematizing procedures. Checklists help consultants follow a systematic procedure despite the complexity of many client cases, from both an emotional and technological standpoint. We created three checklists: one each for before, during (see Appendix of full version), and after the consultation.

We also developed a process for training researchers involved in consultations. We wrote a 13-page training manual that includes a detailed description of our protocol with example situations. It also discusses consultant emotional well-being and safety considerations (e.g., that consultants not give their full names until after spyware scans are complete). Training included reading and understanding this manual, along with guided introductions to our instruments, including ISDi.

To gain experience in face-to-face consultations before interacting with clients, we performed mock consultations in which researchers role-play as clients (including setting up, beforehand, a realistic scenario possibly involving spyware or other misconfigurations) and others role-play as consultants (that do not a priori know the scenario). After each mock consultation, the group analyzes how it went, revealing the scenario and constructively discussing how to improve. These are valuable for consultants to gain confidence in their ability to handle consultations as well as for the research team to gather feedback on the usability of various instruments.

Although clearly more research can be done to further refine our instruments, our field evaluation, discussed in Section 6, indicates their immediate practical value. We have publicly released all training materials, instruments, and open-source tools as resources that other advocacy organizations might find useful in their work supporting survivors[5]. We have already been collaborating with the TECC group in Seattle [2], sharing materials and getting feedback. They have adopted some TAQ questions for use in their clinical settings, and we are working towards prototyping ISDi at their clinic.

5 The IPV Spyware Discovery (ISDi) Tool

We now discuss the technical design and testing of ISDi, our IPV Spyware Discovery tool designed for IPV contexts. While technologically ISDi currently only uses, relative to modern anti-virus tools, simpler techniques such as blacklists and other heuristics, the innovation is in tailoring it to IPV: (1) flagging apps that in other contexts are not necessarily dangerous and, importantly, (2) mitigating potential discoverability by existing IPV spyware. Both issues necessitated a new tool, as existing ones fail on both accounts.

[5] https://www.ipvtechresearch.org/resources

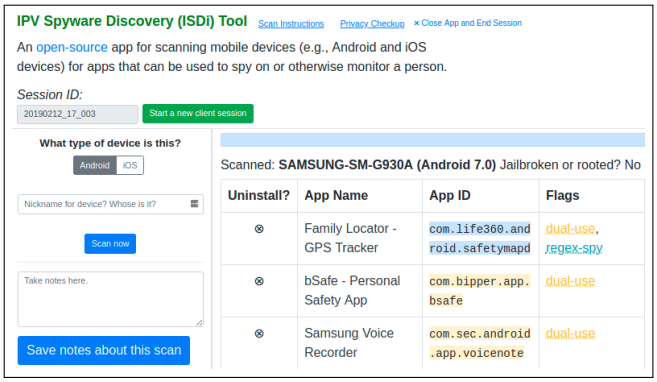


Figure 2: Screen capture of the ISDi tool's main interface after scanning an Android testing device.

Regarding (1), in IPV harmful apps may include both spyware and what are called dual-use apps: otherwise legitimate apps that may be repurposed to act as spyware. We use the term 'IPV spyware' for both types of apps. Prior work showed how existing tools do not detect dual-use apps [8], whereas ISDi was designed to flag all spyware apps, including dual-use apps. Regarding (2), installing an existing anti-virus app is detected by current spyware, potentially endangering victims, while ISDi was designed to be more covert.

ISDi is a Python application with a browser-based user interface (Figure 2) that is used by the consultant to scan a client's devices and identify potentially harmful apps. The tool shows the scan results and serves as a starting point for discussion with the client about any discovered apps. During the investigation phase of a consultation, the consultant, with the client's permission, helps connect the client's device via USB to a laptop running ISDi. A benefit of this design architecture is that it does not require an app to be installed on the device, making it minimally invasive and leaving little to no trace of its execution. We discuss the safety of connecting to client devices below. Further details about how ISDi works are provided in Appendix A.

Detectability of ISDi. A key design consideration is that ISDi does not endanger victims due to being detectable by abusers. As discussed above, we chose to not make ISDi a downloadable app since we know some spyware reports any new apps that are installed. Instead we use the USB interface to connect the device to a laptop running ISDi.

In theory a sophisticated spyware tool might be able to detect ISDi's use of USB interfaces on iOS or Android. Therefore, we conducted additional risk assessments. We installed six highly capable, overt spyware apps found by Chatterjee et al. [8] on an iPhone 6 (running iOS 11.4) and also on a rooted Moto G4 Play phone (running Android 6.0.1). The six

apps are: mSpy, Cerberus, FlexiSpy, SpyToApp, SpyZie, and Trackview. We inspected the features and descriptions of the less sophisticated apps reported on in [8], and decided they were unlikely to support detection of USB connections.

For each of the six considered spyware apps, we created an account (simulating the role of an abuser) and manually investigated capabilities that might allow the app to detect the scanning process (including those tailored to rooted Android devices). We then simulated normal use of the device for several minutes (e.g., opening apps, scrolling) and ran ISDi while network connectivity was enabled. We repeated this process with network connectivity disabled for the scan (and then re-enabled), the intuition being that spyware apps exfiltrate device activities and data to an external cloud-based account configured by the abuser, only some of which may be monitored in real time. We examined the information that the abuser obtains in both cases, and found that for five of the apps there was no way to infer that ISDi was used.

The remaining app, Cerberus, allows exfiltrating system logs on Android, although this capability must be manually invoked by the abuser. These system logs include entries about USB connections to the device and that the device connected to a power source, but nothing beyond that. A technically sophisticated abuser aware of our tool and who carefully analyzed these logs might suspect, but would not have conclusive evidence, that the device was scanned.

Finally, spyware might reveal that the client came to an FJC, and there have been reports of abusers physically confronting victims at FJCs or shelters [20]. However, our consultations and ISDi do not exacerbate this risk given that our clients already visit FJCs for other reasons.

Data collection. Although it is possible to use ISDi without collecting any data, for research and safety reasons we choose to store some information, including the list of apps on a device. Importantly, we do not collect any personally identifiable information or content, such as phone number, emails, photos, etc. See Appendix A for more details.

6 Field Study

After developing and refining our consultation protocol and instruments, we performed a six-month field evaluation with IPV survivors. The study was conducted in collaboration with the ENDGBV, who helped recruit participants, provided safe space for consultations, and ensured the availability of IPV professionals to help with safety planning. Before beginning our study we obtained ethics approval for all procedures from our university's IRB and from the ENDGBV.

Recruitment. We distributed fliers to all five FJC locations (one in each borough of NYC). These fliers advertised the study as a way for clients to obtain a tech safety and privacy consultation, making both clients and professionals aware of the opportunity. Interested clients were asked to speak with their case manager who, after consulting with the client, created a referral and an appointment with our team. Consultations were typically scheduled for days when our team arranged to be at the FJC, with a minimum of one and a maximum of four consultations on a single day. At the suggestion of ENDGBV staff, we gave participants $10 compensation to cover the cost of transportation to/from the FJCs.

Procedure. Consultations took place in a private room at one of the FJCs. Each consultation was done by a team of two or three researchers: one person focused on communication with the client, another on the technical parts of the consultation (ISDi scan, manual privacy checks), and a third (when available) to take notes. Consultations were done individually.

Clients scheduled for a consultation were advised to bring any digital devices that they used or that they wished to have checked. However, two participants did not bring all their devices to their first consultation and therefore made an appointment to return so as to have additional devices checked. Thus, two clients participated in two consultations.

Consultations lasted between 30 minutes and two hours. We began by introducing the team members to the client, explaining the purpose of the study, outlining the activities that would be performed, and discussing the data that would be collected about them and from their devices. We then obtained the client's verbal consent to participate. We also asked participants for permission to audio record the consultation for data collection purposes and received permission to record 36 out of 46 consultations. If the participant did not want to be audio recorded, we instead took detailed notes.

After receiving the client's consent to participate, we followed the consultation procedure detailed in Section 4, including questions from the TAQ, constructing a technograph, scanning the client's devices with ISDi, and performing manual privacy configuration checks. Whenever possible, we suggested it may be advantageous for the client to have their case manager or another IPV professional present during the consultation so they could assist with safety planning and/or documenting relevant findings. In total, 16 out of 44 clients had a professional present during their consultation. After performing all procedures and discussing relevant findings with the client (and professional, if present) we thanked the client for their time. For clients requiring followup, we discussed what that followup would be and confirmed the relevant professional to contact when the followup was complete.

Data collection and analysis. We collected detailed handwritten notes and audio recordings (when permitted) that document each consultation, including client answers to TAQ questions, discussion of their digital footprint, details of manual privacy checks, results from ISDi device scans, the advice or recommendations discussed with the client, and any followups that were done. All audio recordings were professionally transcribed and collated by consultation with the relevant handwritten notes, completed technograph, and ISDi data.

We manually went through all this data multiple times to carefully summarize each consultation and produce the descriptive and aggregate statistics presented in Section 7. The data was stored securely with controls limiting access to only the subset of the research team that performed analysis.

Safety protocols. As discussed in Section 3, IPV presents a sensitive landscape within which to conduct research and survivors are a vulnerable, at-risk population. Our research procedures were carefully designed to protect clients' privacy and safety. For example, we did not ask participants to sign a consent form since we did not want to know or collect any identifying information (e.g., names), and all communication with clients took place through the referring professional, including scheduling and any post-consultation followups.

Although we offered participants a variety of printed handouts to help them understand their digital safety and privacy, we explained there may be risks with taking such materials home, especially if they still lived with their abuser, since someone may discover they had received a consultation. In addition, since changing privacy settings or uninstalling surveillance apps could lead to potentially dangerous escalation of abuse, whenever possible we encouraged participants to have a trusted IPV professional present during their consultation. When this was not possible, we made sure that another experienced case worker was available to help develop safety plans that accounted for any detected tech abuse and/or discuss new protection strategies that participants may want to adopt.

We also considered safety and well-being for our research team. Part of our training included ways to balance the need to properly inform participants about who we were and our affiliation, while avoiding giving out detailed identifying information about the individual researchers. For example, we introduced ourselves by first name only. This was because of the risk that spyware on devices was recording conversations.[6] In addition, working with IPV survivors and hearing their stories may be mentally and emotionally challenging. We regularly met as a team after consultations to debrief and encouraged team members to discuss their feelings, experiences, or anything they were struggling with. Moreover, an experienced IPV case worker was available at all times to speak with researchers and help them process any upsetting experiences that occurred during the consultations.

7 Results of the Field Study

The main goal of our study was to evaluate the utility of our consultation protocol for IPV victims. Our tools and instruments uncovered important, potentially dangerous security problems that we discussed with clients and professionals. This preliminary data suggests our consultation protocol provides benefits. Given the small sample size taken from a single city, we warn that our results should not be interpreted as statistically representative of problems faced by IPV survivors. We discuss limitations of our results more in Section 8.

For the sake of client anonymity, we necessarily cannot report on the full details of our consultations. Instead, we give aggregate results, or when we discuss a particular situation we only do so in a way that makes it coincide with widely reported IPV technology abuse situations, as per prior work [8, 14, 19, 20, 27, 35, 43] and our experiences.

Participants and devices. We conducted a total of 46 consultations with 44 IPV survivors (43 female, 1 male) who were all clients at the FJCs. Two clients received second consultations (at their request) to scan additional devices. All participants were adults and one still lived with their abuser.

As shown in Figure 3 (left table), clients brought a total of 105 devices to the consultations. Of these 82 were Android or iOS and we scanned 75 of these with ISDi. Two unscanned devices were iPhone Xs, which initially caused an error in ISDi when Apple changed the format of device IDs (updates to ISDi fixed this for subsequent scans). In two cases, ISDi could not scan a very old iPhone, potentially due to an error in the libimobiledevice tool we use to communicate with devices. One iPhone was not scanned due to a client leaving early and two other phones were not scanned either because the client was locked out of the device or stated they were not concerned about scanning it. All devices that were not scanned with ISDi were checked manually, except two where clients were locked out of the device (a phone and laptop).

We performed manual checks on 97 out of 105 devices brought in by clients. Clients brought a number of devices for which we did not have a protocol for manual privacy check up, including Internet-of-Things devices such as Amazon Echos, gaming systems, a Blackberry phone, and a flip phone. We performed a best-effort inspection in such cases, except the flip phone for which the client had no privacy concerns.

Participants' chief concerns. Clients expressed a range of chief concerns, as shown in Figure 3 (middle table). The descriptions here, such as "abuser hacked accounts" reflect terminology used by clients. A relatively large number of clients (20) described experiences that suggest abusers had access to clients' online accounts (often described as "hacking") or reported evidence indicative of such access (e.g., abuser knows information only stored in an account). The second most prevalent chief concern (18 clients) were general concerns about their abuser tracking them or installing spyware, but without specific reasons for suspecting it. Other clients were concerned that their location was being tracked, their phone was acting suspiciously, and more. Finally, a few clients wanted to learn more about tech privacy and had no specific concerns about tech abuse directed towards them.

Chief concerns were often connected to the security issues we detected, discussed more below. For example, chief

[6] We explored other ways to protect researchers, such as leaving client devices outside or placing them in sound-insulated containers or Faraday bags, but these proved impractical.

Clients & Devices	
Clients seen	44
Consultations performed	46
Devices seen	105
Devices manually inspected	97
Devices scanned w/ ISDi	75
Median devices per client	2
Max devices per client	7
Median apps per scanned device	170

Chief Concerns	
Worried about tech abuse/tracking/spyware	18
Abuser hacked accounts or knows secrets	20
Worried abuser was tracking their location	10
Phone is glitchy	10
Abuser calls from unknown numbers	9
Unrecognized app on child's phone	1
Money missing from bank account	1
Curious and want to learn about privacy	4

Detected Issues	
Clients w/ vulnerabilities	23
Clients w/ unsolved problems	2
Clients w/ no problems detected	19
Potential spyware detected	3
Potential password compromise	14
Presence of unknown "trusted" devices	12
Shared family/phone plan	4
Rooted device	1

Figure 3: Summary of field study results. **Left:** Breakdown of the number of clients seen, consultations performed, and devices encountered. **Middle:** The chief concerns, as described by the clients (some had multiple chief concerns). **Right:** The problems detected during consultations, including vulnerabilities, security risks, and spyware (some clients had multiple problems).

concerns involving illicit access to accounts were often best explained by poor password practices, family sharing, or confirmation of account access by abuser devices. In one case the chief concern was entirely unrelated to the discovered security issue, however. All this confirms the importance of both identifying the chief concerns, but also using instruments and procedures that may surface unexpected problems.

Security vulnerabilities discovered. For 23 of 44 clients (52%), our consultations identified important security risks, vulnerabilities, or plausible vectors for tech abuse. Before describing our findings, it is important to note that, in most cases, we do not have definitive proof that the vulnerabilities discovered are the root causes of clients' problems. For example, if a client's password is the name of a child they share with the abuser, or if their phone is part of a shared family plan, these provide plausible theories for, but not hard evidence of, how compromises may be occurring.

Results from ISDi: ISDi flagged a total of 79 apps as problematic across all device scans. The majority of these (61) were dual-use apps, with "find my phone" and child monitoring apps the most prevalent categories. For all but one of these dual-use apps, discussions with clients confirmed that they recognized the apps and were aware of their presence. For one dual use app, the client said that they did not install or recognize the app, which was a controller for remote home surveillance systems with WiFi, camera, and motion detection capabilities. We treated this case as a true positive result. The other 18 apps detected by ISDi were false positives (i.e., clearly not relevant to IPV) that the consultant easily dismissed as such. The number of false positives in any individual consultation was low, the maximum number of flagged apps on a client's device was five. This meant that, thus far, we have not had any issues with consultants being overwhelmed by large numbers of apps flagged by ISDi.

The relatively low rate of actual spyware detection may be because, as discussed below, many abusers are seemingly able to surveil clients via compromised accounts, and so may not need to install spyware. In addition, almost all clients no longer lived with the abuser, had changed or reset their devices since leaving (which would remove spyware in most cases), and for many devices the abuser no longer had physical access needed to (re-)install spyware. Finally, ISDi detected that one client's Android tablet was rooted. Subsequent discussion revealed that the abuser bought this tablet for the client, had physical access to it during the relationship, and had insisted the client log into her accounts with it. As a result of our conversation, the client decided to stop using the tablet.

Results from TAQ and technograph: For many clients, we discovered security vulnerabilities through combined use of the TAQ, technograph, and/or manual privacy checks. In some cases, the TAQ and technograph were the primary (or only) way to uncover a potential problem. For example, four clients reported that they were still part of a shared family plan or that their abuser pays for their phone plan, vulnerabilities that could give the abuser access to, for example, the location of the client's device and call and text history. Another common problem that the TAQ and technograph revealed for 14 clients was the use of passwords that the client said were known, or could be guessed, by their abuser. In several of these cases, a compromised password provided a plausible explanation for how the abuser may be gaining access to the client's accounts.

Results from manual checks: Combining TAQ and technograph information with subsequent manual privacy checks often yielded evidence of malicious account access. For example, during manual checks of iCloud account settings for four clients, we discovered that their iCloud accounts listed "trusted" devices that the client either did not know or recognized as belonging to the abuser. Similarly, manual checks of client email and social media accounts showed unknown or abuser device logins for another eight clients.

iCloud and email account access, whether by password compromise or via unauthorized "trusted" device access, also yielded plausible explanations for a range of other problems. For example, three clients reported that they kept written records of passwords for all their accounts in files that were then synced with their compromised iCloud, potentially resulting in the abuser obtaining all these passwords. Similarly, several clients emailed copies of their new passwords to themselves via potentially compromised email accounts. Another prevalent avenue for compromise that we saw happened when

clients used a compromised account as the backup account for other services (e.g., social media), with clients unaware of how this might result in abuser access to these services.

For two clients, manual checks of laptops revealed browser extensions that the clients did not install or know about. In one case, the extension was "off store" (not available via the official Chrome Web Store), may have been sideloaded (installed via developer mode), and had permission to read and write all browser data. We regarded this as possible spyware. For the other case, the extension is available via the Chrome Store and is used to monitor access to web content. This extension provides a plausible explanation for the client's chief concern, which was that her abuser knew about her online activities, and we regarded it as probable spyware.

No problems detected. For 21 out of 44 clients, our instruments did not surface any evidence of potential tech issues. For 19 of these, the lack of discovered problems was reassuring and many left the consultation visibly relieved and more at ease. However, in two cases, the consultation's inability to address their chief concerns left the client unsatisfied. In these cases we performed follow-up research, including reaching out to other tech experts for second opinions about their concerns (in an anonymized fashion) but unfortunately still have no plausible explanation for what they were experiencing.

Hand-off and followup. For the 23 clients with discovered problems and two clients with unresolved issues, we conducted a hand-off in which we discussed our results with the referring professional. For 12 of these, the professional was onsite and hand-off occurred immediately. For the other 13, we followed up with the professional via email and/or a phone call. Although many clients did not resolve discovered problems immediately because of the need to safety plan, they said that it was helpful and empowering to at least know how the abuser was plausibly obtaining information about them.

Eleven cases required further research after the consultation. Six of these were client requests for information about specific apps we were unfamiliar with (e.g., can app X track my location?). For the remaining five we found something during the consultation that needed further analysis to assess its danger. In 10 cases, the consultant researched the issue at length and provided a comprehensive answer to the referring professional within a few days of the consultation. In the remaining case, we could not provide a satisfactory explanation for what the client was describing even after significant research, which we explained to the referring professional.

8 Discussion

Although the results from our field study are preliminary, they suggest that our consultation protocol is already valuable to clients in dangerous situations. Encouragingly, the ENDGBV have asked our team to schedule more consultations with clients at the FJCs. This in turn raises new open questions about how to sustain and scale our clinical computer security approach. In this section we discuss: (1) limitations of our current study; (2) open questions that it raises about how to realize the vision of clinical computer security for IPV victims more generally; and (3) open questions that our work raises about clinical approaches to computer security beyond IPV.

Limitations. This first study on clinical computer security interventions has several limitations that we acknowledge. First, our study was restricted to a single municipality and our participants were not representative of all people who suffer IPV. Although New York City has a large and diverse population, and our sample does include socioeconomic and cultural diversity, all but one of our participants were women, all but one were no longer living with their abuser, and the majority had been in heterosexual relationships. As a result, our study may fail to capture some of the nuances associated with abusive relationships for LGBTQIA+ people or those who may still live with their abuser.

Another limitation is our sample size. Although 44 clients may be sufficient to verify the utility of our consultations, it certainly does not yield statistically significant estimates of, for example, likelihood of spyware or other harms being seen in practice. Further, our study context purposefully biases our sample towards victims that are specifically worried about tech problems. Still, our results provide guidance on what a tech clinic is likely to see, and our experiences are consistent with prior work on tech attacks in IPV [20, 27].

Our consultations may not catch all issues, either due to consultant error (e.g., forgetting to ask a TAQ question) or technical error (e.g., ISDi mislabeling an app). Indeed, one of the fundamental challenges faced in this area is dealing with complex, multifaceted attacks, and it is not possible to be perfect. That said, our new approach vastly improves over the current status quo in practice, which is essentially nothing. Moving forward, future research will need to assess if, and how, our protocol and instruments impact client lives in the longer term, determining, for example, whether our interventions measurably decrease illicit account accesses.

Should a client change their behavior as a result of our consultations, abusers may change behavior, retaliate against the victim, or otherwise escalate abuse. We designed our protocols to try to minimize the potential for this, but no procedures can eliminate such risks entirely. That said, we are in active communication with FJC leadership and have not received any indication that a client has faced retribution as a consequence of participating in a consultation.

Clinical computer security for IPV. Our work focused on client consultations, which are a fundamental component of realizing our vision of clinical computer security. Given the success of our initial field study, we are faced with a range of open questions. The most obvious is that our design and evaluation so far did not perform in-depth investigation of issues related to scalability and sustainability.

A sustainable computer security clinic will likely need a supporting organization, outside the scope of a research study, to handle recruitment, screening, and training of sufficiently many volunteer consultants (or paid professionals, should there be funding to pay them). Although the assessments and materials we developed in this work will help with training future tech consultants, they do not yet speak to challenges that are outside the context of the consultation. In a referral model like the one we used, just scheduling consultations took many hours per week and, more broadly, how best to organize delivery of clinical computer security for IPV victims raises a host of questions for future research.

As a financially sustainable recruitment strategy, we might draw on existing models like pro-bono legal services [30], and initial conversations with tech professionals and companies suggest that some may be willing to offer their time free of charge. (This model is used by the TECC clinic [2].) Another approach is student-run clinics, similar to law school legal clinics [12] or medical school free clinics [37]. In any such model, it will be essential to develop strong protocols for screening consultant applicants, particularly to ensure that abusers are prevented from enrolling as consultants. Advocacy groups have protocols for screening applicants, and one could start by adopting these. In parallel, future research will be needed to localize clinical techniques to geographic locations with different support organizations and laws.

Another pressing issue is maintenance of instruments. ISDi's coverage currently relies on labor-intensive updating of blacklists, based on web crawling and manual analysis. Like malware detection in other contexts, maintaining accuracy over time and staying ahead of emerging threats is an immense challenge [40]. It is also important to consider the longer-term implications of making ISDi's existence and methods public. While current spyware does not infer ISDi was used, if it becomes widespread enough to become a target, spyware developers might turn to more sophisticated methods that monitor USB-related system processes. Similarly, spyware vendors may start attempting to avoid detection. As such, we keep ISDi's blacklist private, allowing access via legitimate requests from those working to help victims.

Our other instruments will also require updating at various time intervals. By design, the TAQ should maintain relevance for quite a while to come, requiring updating only when technology changes suggest new, broad classes of threats we must consider. But our manual investigation guides for checking security or privacy settings may need to be updated more frequently as companies change their products. Future work might evaluate the right balance between generalizability and actionability of such guides (c.f., [19]), or infrastructure for maintaining them (e.g., expert crowdsourcing [29]).

Clinical computer security beyond IPV. IPV is not the only context in which victims suffer targeted, persistent, and personalized attacks. Some examples include the dissidents, activists, and NGO employees targeted by nation-state hacking campaigns [7, 23, 25, 26], or the gamers [9], journalists [10], politicians [6], and researchers [21] who are at high-risk of being harassed online [18, 33]. As in IPV, in all these cases the attacker wants to harm their particular target. There is also an asymmetry between the victim and attacker, with the latter having more resources, time, and/or technological sophistication. Indeed, in some cases the adversary in these other contexts has significant technical prowess.

Clinical approaches to computer security may be of utility in these other contexts. In the near term, adapting our techniques to other communities of victims similar to IPV — such as victims of elder, parental, or child abuse, or victims of sex trafficking (which are also served by FJCs) — could constitute important research directions. Despite the similarities, research will be needed to understand how the nuances emanating from particular circumstances or demographics change best practices for clinical interventions.

Further afield are contexts that are less similar to IPV. For example, those targeted by government agencies as mentioned above might benefit from systematized clinical approaches. One could perhaps start with the work done by the Citizen-Lab [13] and Citizen Clinic [1], and determine to what extent, if any, our methodologies for stakeholder-driven design could help improve clinical interventions.

9 Conclusion

This paper lays out a vision for clinical computer security and explores it in the context of IPV. Through an iterative, stakeholder-driven process, we designed a protocol for conducting face-to-face tech consultations with victims of IPV to understand their tech issues, investigate their digital assets programmatically and by hand to discover vulnerabilities, and advise on how they might proceed. Our preliminary study with 44 IPV victims surfaced vulnerabilities for roughly half our participants, including account compromise, potential spyware, and misconfiguration of family sharing plans. Our consultations also provided advice and information to victims and professionals on ways to document such discoveries and improve computer security moving forward. Our clinical approach provides immediate value, while also laying a foundation for future research on evidence-based refinements to clinical tech interventions in IPV and, potentially, beyond.

Acknowledgments

We would like to sincerely thank all our study participants, the Family Justice Centers, and the NYC ENDGBV. This work was funded by the NSF through grants CNS-1717062 and CNS-1558500, and by gifts from Comcast and Google.

References

[1] Citizen clinic. https://cltc.berkeley.edu/citizen-clinic/.

[2] Technology-enabled coercive control working group, Seattle, WA, USA. https://tecc.tech/.

[3] iOS jailbreak detection (OWASP). https://git.io/fj4te, 2017.

[4] Libimobiledevice: a cross-platform software protocol library and tools to communicate with iOS devices natively. https://www.libimobiledevice.org/, 2017.

[5] Android debug bridge (adb). https://developer.android.com/studio/command-line/adb, 2019.

[6] Maggie Astor. For female candidates, harassment and threats come every day. https://www.nytimes.com/2018/08/24/us/politics/women-harassment-elections.html, 2018.

[7] S. Le Blond, A. Cuevas, J. Ramón Troncoso-Pastoriza, P. Jovanovic, B. Ford, and J. Hubaux. On enforcing the digital immunity of a large humanitarian organization. In *2018 IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 302–318.

[8] Rahul Chatterjee, Periwinkle Doerfler, Hadas Orgad, Sam Havron, Jackeline Palmer, Diana Freed, Karen Levy, Nicola Dell, Damon McCoy, and Thomas Ristenpart. The spyware used in intimate partner violence. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 441–458. IEEE, 2018.

[9] Despoina Chatzakou, Nicolas Kourtellis, Jeremy Blackburn, Emiliano De Cristofaro, Gianluca Stringhini, and Athena Vakali. Hate is not binary: Studying abusive behavior of #gamergate on twitter. In *Proceedings of the 28th ACM Conference on Hypertext and Social Media*, HT '17, pages 65–74, New York, NY, USA, 2017. ACM.

[10] Gina Masullo Chen, Paromita Pain, Victoria Y Chen, Madlin Mekelburg, Nina Springer, and Franziska Troger. "you really have to have a thick skin": A cross-cultural perspective on how online harassment influences female journalists. *Journalism*, 2018.

[11] Robyn Clay-Williams and Lacey Colligan. Back to basics: checklists in aviation and healthcare. *BMJ Qual Saf*, 24(7):428–431, 2015.

[12] Robert J. Condlin. "tastes great, less filling": The law school clinic and political critique. *Journal of Legal Education*, 36(1):45–78, 1986.

[13] Ronald J. Deibert. The Citizen Lab. https://citizenlab.ca/.

[14] Jill P Dimond, Casey Fiesler, and Amy S Bruckman. Domestic violence and information communication technologies. *Interacting with Computers*, 23(5):413–421, 2011.

[15] John W Ely, Mark L Graber, and Pat Croskerry. Checklists to reduce diagnostic errors. *Academic Medicine*, 86(3):307–313, 2011.

[16] NYC ENDGBV. NYC mayor's office to combat domestic and gender-based violence. https://www1.nyc.gov/site/ocdv/about/about-endgbv.page, 2019.

[17] NYC FJCs. NYC family justice centers. https://www1.nyc.gov/site/ocdv/programs/family-justice-centers.page, 2019.

[18] Antigoni-Maria Founta, Constantinos Djouvas, Despoina Chatzakou, Ilias Leontiadis, Jeremy Blackburn, Gianluca Stringhini, Athena Vakali, Michael Sirivianos, and Nicolas Kourtellis. Large scale crowdsourcing and characterization of twitter abusive behavior. *CoRR*, abs/1802.00393, 2018.

[19] Diana Freed, Jackeline Palmer, Diana Minchala, Karen Levy, Thomas Ristenpart, and Nicola Dell. Digital technologies and intimate partner violence: A qualitative analysis with multiple stakeholders. *PACM: Human-Computer Interaction: Computer-Supported Cooperative Work and Social Computing (CSCW)*, Vol. 1(No. 2):Article 46, 2017.

[20] Diana Freed, Jackeline Palmer, Diana Minchala, Karen Levy, Thomas Ristenpart, and Nicola Dell. "A Stalker's Paradise": How intimate partner abusers exploit technology. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, 2018.

[21] Virginia Gewin. Real-life stories of online harassment — and how scientists got through it. https://www.nature.com/articles/d41586-018-07046-0, 2018.

[22] Philip J Guerin and Eileen G Pendagast. Evaluation of family system and genogram. *Family therapy: Theory and practice*, pages 450–464, 1976.

[23] Seth Hardy, Masashi Crete-Nishihata, Katharine Kleemola, Adam Senft, Byron Sonne, Greg Wiseman, Phillipa Gill, and Ronald J Deibert. Targeted threat index: Characterizing and quantifying politically-motivated targeted malware. In *USENIX Security Symposium*, pages 527–541, 2014.

[24] Leigh Honeywell. Personal communication, 2019.

[25] Stevens Le Blond, Adina Uritesc, Cédric Gilbert, Zheng Leong Chua, Prateek Saxena, and Engin Kirda. A look at targeted attacks through the lens of an ngo. In *USENIX Security Symposium*, pages 543–558, 2014.

[26] William R Marczak, John Scott-Railton, Morgan Marquis-Boire, and Vern Paxson. When governments hack opponents: A look at actors and technology. In *USENIX Security Symposium*, pages 511–525, 2014.

[27] Tara Matthews, Kathleen O'Leary, Anna Turner, Manya Sleeper, Jill Palzkill Woelfer, Martin Shelton, Cori Manthorne, Elizabeth F Churchill, and Sunny Consolvo. Stories from survivors: Privacy & security practices when coping with intimate partner abuse. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 2189–2201. ACM, 2017.

[28] APA Work Group on Psychiatric Evaluation. *Practice Guidelines for the Psychiatric Evaluation of Adults*. The American Psychiatric Association, third edition, 2016.

[29] Daniela Retelny, Sébastien Robaszkiewicz, Alexandra To, Walter S. Lasecki, Jay Patel, Negar Rahmati, Tulsee Doshi, Melissa Valentine, and Michael S. Bernstein. Expert crowdsourcing with flash teams. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST '14, pages 75–85, New York, NY, USA, 2014. ACM.

[30] Deborah L Rhode. Cultures of commitment: Pro bono for lawyers and law students. *Fordham L. Rev.*, 67:2415, 1998.

[31] Carl R Rogers. Significant aspects of client-centered therapy. *American Psychologist*, 1(10):415–422, 1946.

[32] Sharon G Smith, Kathleen C Basile, Leah K Gilbert, Melissa T Merrick, Nimesh Patel, Margie Walling, and Anurag Jain. The national intimate partner and sexual violence survey (NISVS): 2010-2012 state report. 2017.

[33] Peter Snyder, Periwinkle Doerfler, Chris Kanich, and Damon McCoy. Fifteen minutes of unwanted fame: Detecting and characterizing doxing. In *Proceedings of the 2017 Internet Measurement Conference*, IMC '17, pages 432–444, New York, NY, USA, 2017. ACM.

[34] Cindy Southworth, Shawndell Dawson, Cynthia Fraser, and Sarah Tucker. A high-tech twist on abuse: Technology, intimate partner stalking, and advocacy. *Violence Against Women*, 2005.

[35] Cynthia Southworth, Jerry Finn, Shawndell Dawson, Cynthia Fraser, and Sarah Tucker. Intimate partner violence, technology, and stalking. *Violence against women*, 13(8):842–856, 2007.

[36] Geek Squad. Geek Squad services. `https://www.geeksquad.com`, 2019.

[37] Lindsey Stephens, Nicole Bouvier, David Thomas, and Yasmin Meah. Voluntary participation in a medical student-organized clinic for uninsured patients significantly augments the formal curriculum in teaching underrepresented core competencies. *Journal of Student-Run Clinics*, 1(1), Jun. 2015.

[38] San-Tsai Sun, Andrea Cuadros, and Konstantin Beznosov. Android rooting: Methods, detection, and evasion. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 3–14. ACM, 2015.

[39] Jing Tian, Nolen Scaife, Deepak Kumar, Michael Bailey, Adam Bates, and Kevin Butler. SoK: "Plug & Pray" Today – Understanding USB insecurity in versions 1 through C. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 1032–1047. IEEE, 2018.

[40] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas. SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *2015 IEEE Symposium on Security and Privacy*, pages 659–673, May 2015.

[41] Zhaohui Wang and Angelos Stavrou. Exploiting smartphone usb connectivity for fun and profit. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 357–366. ACM, 2010.

[42] Thomas G Weiser, Alex B Haynes, Angela Lashoher, Gerald Dziekan, Daniel J Boorman, William R Berry, and Atul A Gawande. Perspectives in quality: designing the who surgical safety checklist. *International journal for quality in health care*, 22(5):365–370, 2010.

[43] Delanie Woodlock. The abuse of technology in domestic violence and stalking. *Violence against women*, 23(5):584–602, 2017.

A More Details about ISDi

How ISDi works. ISDi uses the libimobiledevice tool [4] for iOS or Android Debug Bridge (adb) [5] for Android to programmatically access the connected device. On Android, the device must be configured to allow USB debugging, which is done by enabling developer mode for the scan and revoking it again after the scan is complete. When a scan is initiated, ISDi pairs with the connected device and queries it for a list of all installed apps, including those that are hidden from the app drawer on Android (c.f., [8]). ISDi then runs additional queries on the device to obtain the OS version, hardware model, and manufacturer. It also performs heuristic checks to infer if the device is jailbroken (iOS) or rooted (Android). ISDi displays information about the outcome of these checks via the tool's UI, along with a list of all installed apps with potentially dangerous apps listed first. We compute each app's threat score by combining several heuristics.

First, we created a blacklist of potential IPV spyware and dual-use apps using techniques from Chatterjee et al. [8]. To ensure the list was not stale, we re-ran their measurements several times and added the results to the blacklist. We applied the machine learning classifier used in [8] to remove the obviously irrelevant apps. However, we did not manually prune the list further to reduce the falsely flagged apps, as during consultation a consultant can check those apps and ignore if not relevant for IPV. The most recent update was shortly before we initiated meetings with clients. Our current blacklist contains over 500 iOS and 5,000 Android apps. A second heuristic is a set of regular expressions that app names are checked for, including substrings such as "spy" or "track". Lastly, on Android, ISDi checks whether any apps were installed outside of the Play Store. A threat score is then computed for each app so that the apps can be listed in decreasing order of potential risk.

Clicking on an app name in ISDi's UI displays more information about that app, including installation date, developer description of the app, requested permissions, and when permissions were last invoked (on Android). ISDi is also capable of uninstalling apps (after appropriate safety planning) via its interface, which is especially useful for hidden apps on Android that cannot be located using the device's UI.

ISDi is not perfect and may have both false positives and false negatives. The former are less dangerous, and in our experience were easily dealt with by the consultant in the field. False negatives are of course potentially dangerous, and so we purposefully designed ISDi to have a low false negative rate by allowing for more false positives.

ISDi collects the following information of each app: the app ID, permissions, installation date (Android only), and package files (Android only). ISDi also generates and stores a keyed cryptographic hash of the device's serial number. The latter is useful to ensure we can determine if we scan the same device twice, since clients may have multiple consultations, without explicitly storing the device identifier. Collected data is linked to a random client identifier. Storing a list of apps is helpful not only for our research, but also because it allows us to further examine, via followup if necessary, any suspicious apps discovered during a consultation. In addition, whenever we update the blacklist, we retroactively scan the apps from past consultations to ensure that no newly found IPV spyware apps were on a previously scanned devices. (Fortunately, we have not yet detected any spyware retroactively.) All data is stored securely and accessible only to our team.

Detecting potential IPV spyware. A core feature of ISDi is its detection of IPV spyware apps (either overt or dual-use) on iOS or Android devices. To do so, ISDi integrates various heuristics into a rank-ordered list of apps by an internal threat score. After querying the device for a list of installed apps, ISDi assigns a threat score to each app, that score derived from summing the weights of heuristics.

The main heuristic is two blacklists of app IDs, one for overt spyware apps and one for dual-use apps. The blacklists deployed with ISDi was seeded with the list of apps discovered in [8], but then updated by using their snowball searching techniques on the Google Play store and iTunes store. Note that Google Play occasionally bans apps and reportedly banned some in response to the results of [8]. We do not remove apps from a blacklist should they be removed from the play store — they could have been downloaded and installed by an abuser before removal. We additionally included any apps we discovered via manual searches or that we discovered in any other way. Following [8], we aggressively added apps to a blacklist, at risk of creating false positives. This favors having a low false negative rate, and we built into our protocol the ability for consultants to handle false positives when they arose. To help with ordering, we kept a separate blacklist of overt spyware, with other apps appearing on the dual-use blacklist.

In addition to blacklists, ISDi uses a few other heuristics. First are regular expressions applied to application names, as described in Section 5. Second was that we marked any off-store app as potentially dangerous. Third was whether the device is a system app, meaning it was pre-installed on the device by the cellular provider or OS vendor.

We then gave a weighted score to each app according to the values shown in Figure 4. The score of an app is equal to the sum of the weights for the set of heuristics that apply to the app. A higher score denotes being potentially more dangerous. The weights are admittedly somewhat arbitrary, but roughly correspond to our perception of the danger each heuristic indicates. In practice, the number of apps on a device that were assigned risk signals by ISDi were sufficiently small that our choice of weights and rank-ordering did not make much of a difference during consultations.

App detection accuracy. While ISDi lists all apps on the device, and the consultant is encouraged to visually inspect

Heuristic	Weight	Description
Overt spyware blacklist	1.0	Known, overt spyware
Dual-use blacklist	0.8	Legitimate uses, but possibly harmful in certain situations
Offstore app	0.8	Not installed through an official app marketplace
Regex match	0.3	App name or ID contains 'spy', 'track', etc.
System app	-0.1	Pre-installed by device vendor

Figure 4: The ISDi heuristics for ordering apps. Each app is assigned a score that is the sum of the weights for each heuristic that applies to it.

the entire list, we would still consider it a false negative if a dangerous app was not flagged by one of the four heuristics (excluding the system app heuristic).

As discussed in Section 5, ISDi's accuracy depends in part on labor intensive web crawling and manual pruning. Our blacklist of dual-use apps included all 2,474 seed apps from Chatterjee et al. [8], as well as 3,263 new apps from our own periodic crawls since May 2018 and filtering using the ML classifier given in [8]. Unlike in [8], we do not manually prune the 3,263 apps we added to the blacklist to further remove apps falsely flagged by the machine learning classifier. During consultation, the consultant ignores apps that are not relevant, which was not a problem during our consultations.

Most overt spyware apps we have encountered (and certainly all dual-use apps we have inspected) do not try to hide their presence from a programmatic scan. However, for a few of the overt spyware apps we have observed that they chose innocuous-looking app IDs (such as "com.android.system"). This reiterates the need for programmatic scans, which are not fooled by this. However, if apps change their app IDs frequently to avoid detection, our blacklists may not cover the full set of app IDs associated with a spyware. We have observed that one overt offstore spyware app, mSpy, has published versions of its Android APK with different app IDs: sys.framework and core.framework, while others such as SpyToApp, FlexiSpy, and SpyZie have not changed their app IDs to our knowledge (we re-downloaded them in September 2018 and in February 2019). We have found no evidence that onstore dual-use apps change their app IDs, though Trackview has published their app twice on the Google Play Store, as both net.cybrook.trackview and app.cybrook.trackview under different developer IDs. We have added all of the changed app IDs to our blacklist as we have discovered them.

Finally we note that ISDi is not designed to detect more sophisticated malware, such as that used by national intelligence agencies. We believe such malware is unlikely to arise in IPV settings, since it requires special access to obtain it. For a client for which it is plausible that her abuser might have access to such capabilities (e.g., the abuser works as a computer security expert), a discussion about potential remediations, such as obtaining new devices, would be appropriate.

App reports. Upon clicking on an app, ISDi gives a number of details about the app. This includes a developer description (if available), when the app was installed (Android only), the permissions the app has requested, and the time of all the permissions recently used by the app (Android only), including dangerous permissions such as microphone, camera, or GPS. It also provides a link to a Google search on the app ID, which allows the consultant to quickly attempt to look up more information about the app should it be unfamiliar.

Detecting jailbroken or rooted phones. ISDi attempts to determine if the scanned device is jailbroken (iOS) or rooted (Android), since such devices are at much greater risk for installation of powerful spyware. For example, most spyware vendors enable for sophisticated features if the device is jailbroken/rooted. Moreover, it is unlikely that a client purposefully jailbreaks or roots their phone.

Thus ISDi uses a set of heuristics to determine whether a device is jailbroken/rooted. If any heuristic comes back positive, ISDi considers the device to be jailbroken or rooted and indicates this along with the results of the scan. Detecting jailbroken/rooted devices is under active discussion for both Android and iOS because app developer communities want to prevent their apps from being illegitimately being used on a jailbroken/rooted device. We therefore collected different heuristics from such community forums. For both iOS and Android, ISDi checks whether common jailbreak/rooting applications are installed on the device [3]. On Android devices, ISDi checks whether or not the su tool is installed on the system "shell" application [38]. On iOS devices, ISDi attempts to mount the filesystem at the root directory.

To the best of our understanding, ISDi will detect any jailbroken or rooted device. However, it is possible that a device could evade detection by ISDi using techniques that are not publicly known. We regularly look into app developer forums for new heuristics and update ISDi accordingly.

Possible attack vectors on ISDi. We have considered that spyware installed by an abuser on a client's phone may attempt to use its USB connection to ISDi as a possible attack vector [39, 41]. We are not aware of any overt spyware apps that try to misuse USB connections to a host computer. We ensured that all commands used by ISDi to communicate with iOS and Android devices, over libimobiledevice and adb, respectively, were run over least privilege (i.e., without sudo).

Technology Assessment Questionnaire (TAQ)

Start with the most pressing concern widely expressed by clients thus far

- Do you worry that your device(s) is being used to track you?
    - Does the abuser show up unexpectedly or know things they shouldn't know?

Probe for risks of device compromise

- What devices do you use in your home or carry with you?
  (e.g., smartphone, iPad, tablet, desktop, laptop, kindle, echo, etc.)
- Do you currently (or have you in the past) share(d) your devices with your abuser?
- Is there any chance that your abuser has (or had) physical access to your devices?
    - Does (Did) your abuser ask or demand physical access to your devices?
- Who set up the screen locks or passwords on your devices?
    - Do you use fingerprint or facial recognition to unlock your devices?

Probe for risks from ownership-based attacks

- Do have a shared family plan?
- Do you or does someone else pay for your phone plan or Internet access plan?

Probe for risks of account compromise

- Who set up your email account or other online accounts?
- Have you ever shared any passwords with your abuser (or anyone)?
    - When did you last update your passwords for your email or other online accounts?
    - How do you remember your passwords?
    - Do you ever take photos of your passwords?
    - Is there a chance your abuser knows (or could guess) the answers to your password reset questions?
- Do you think your abuser has access to your accounts online?
    - Do you have an iCloud or Google account?
    - Do you think the abuser knows the password or has access to your bank account?
    - Do you think the abuser knows the password or has access to your email accounts?
    - Do you think the abuser knows the password or has access to your social media accounts? (Facebook, Instagram, WhatsApp, etc.)

Probe for risks from children's devices

- Do you have any children?
    - Do you share devices with your children?
    - Do you or does someone else pay for your children's devices?
    - Who gave your child their device?
    - Does the abuser have access to the child's device?
    - Does your child bring their device to visitation with the other parent?

Figure 5: The current version of the Technology Assessment Questionnaire (TAQ).

Evaluating the Contextual Integrity of Privacy Regulation: Parents' IoT Toy Privacy Norms Versus COPPA

Noah Apthorpe
Princeton University

Sarah Varghese
Princeton University

Nick Feamster
Princeton University

Abstract

Increased concern about data privacy has prompted new and updated data protection regulations worldwide. However, there has been no rigorous way to test whether the practices mandated by these regulations actually align with the privacy norms of affected populations. Here, we demonstrate that surveys based on the theory of contextual integrity provide a quantifiable and scalable method for measuring the conformity of specific regulatory provisions to privacy norms. We apply this method to the U.S. Children's Online Privacy Protection Act (COPPA), surveying 195 parents and providing the first data that COPPA's mandates generally align with parents' privacy expectations for Internet-connected "smart" children's toys. Nevertheless, variations in the acceptability of data collection across specific smart toys, information types, parent ages, and other conditions emphasize the importance of detailed contextual factors to privacy norms, which may not be adequately captured by COPPA.

1 Introduction

Data privacy protections in the United States are enforced through a combination of state and federal legislation and regulatory action. In Europe, the General Data Protection Regulation (GDPR) is currently the best example of strong, centralized privacy legislation. The GDPR has inspired similar laws in other countries, such as the Brazilian General Data Privacy Law. According to the United Nations Conference on Trade and Development [51], 57% of countries have data protection and privacy legislation as of 2018.

Although data privacy protections vary across countries in terms of details and implementation, many share a common provenance: public pressure to protect sensitive personal data from unauthorized use or release. Surveys report that consumers worldwide were more concerned about online privacy in 2016 than 2014 [7] and that over 60% of U.S. survey respondents in 2018 are concerned about data privacy in general [34]. However, there has been no rigorous, quantifiable, and scalable way to measure whether existing legal privacy protections actually match the privacy expectations of affected individuals. Without such data, it is difficult to know which aspects of privacy regulation effectively align company behaviors with social and cultural privacy norms and which necessitate further revision.

In this paper, we demonstrate that an existing survey technique [3] based on the formal privacy theory of contextual integrity (CI) [32] can be directly adapted to test the conformity of specific regulatory requirements to privacy norms, providing much-needed data to policymakers and the privacy research community. The survey technique can be applied to any privacy regulation that defines guidelines for data collection and transfer practices. Importantly, the survey technique involves questions describing privacy scenarios that are concrete and understandable to respondents from all backgrounds. It also allows straightforward longitudinal and cross-sector measurements to track the effectiveness of regulatory updates over time.

We present a rigorous case study of this technique evaluating the U.S. Children's Online Privacy Protection Act (COPPA), which provides a federal legal framework to protect the online privacy of children under the age of 13. Specifically, we investigate whether parents' opinions about the acceptability of data collection practices by Internet-connected "smart" children's toys match COPPA mandates. Since the Federal Trade Commission (FTC) only updated its guidance on COPPA to explicitly include "connected toys or other Internet of Things devices" in June 2017 [16], our results provide the first indication as to whether COPPA aligns with parents' privacy expectations.

This question is particularly relevant given the recent high-profile security breaches of smart toys, ranging from the theft of personal information of over 6 million children from toy manufacturer VTech to vulnerabilities in Mattel's Hello Barbie [13]. More recently, Germany banned children's smart watches and Genesis Toys' My Friend Cayla doll, citing security risks and "spying concerns" [17, 31].

We survey a panel of 195 U.S. parents of children from

ages 3 to 13, the largest sample size for a study of parent opinions of smart toy data collection in the literature to date. We find that parents generally view information collection predicated on requirements specified by COPPA (e.g., "if the information is used to protect a child's safety") as acceptable, while viewing equivalent information collection without COPPA-specified conditions as unacceptable. This indicates that the existing conditions COPPA places on information collection by smart toys are generally in line with parents' privacy norms, although there may be additional data collection requirements which could be added to regulation that were not tested in our study.

Additionally, we find that COPPA requirements for notification and consent result in more acceptable data collection practices than requirements related to confidentiality and security. This corroborates previous work indicating the primary importance of consent to user privacy norms [3]. We also find variations in the acceptability of COPPA-permitted data collection practices across specific smart toys, types of information, certain information use cases, parent ages, parent familiarity with COPPA, and whether parents own smart devices. These variations emphasize the importance of detailed contextual factors to parents' privacy norms and motivate additional studies of populations with privacy norms that may be poorly represented by COPPA.

We conclude by noting that COPPA's information collection criteria are broad enough to allow smart toy implementations that compromise children's privacy while still adhering to the letter of the law. Continuing reports of smart toys violating COPPA [6] also suggest that many non-compliant toys remain available for purchase. Further improvements to both data privacy regulation and enforcement are still needed to keep pace with corporate practices, technological advancements, and privacy norms.

In summary, this paper makes the following contributions:

- Demonstrates that an existing survey method [3] based on contextual integrity [32] can be applied to test whether privacy regulations effectively match the norms of affected populations.
- Provides the first quantitative evidence that COPPA's restrictions on smart toy data collection generally align with parents' privacy expectations.
- Serves as a template for future work using contextual integrity surveys to analyze current or proposed privacy regulation for policy or systems design insights.

2 Background & Related Work

In this section, we place our work in the context of related research on contextual integrity, COPPA, and smart toys.

2.1 Contextual Integrity

The theory of contextual integrity (CI) provides a well-established framework for studying privacy norms and expectations [32]. Contextual integrity defines privacy as the appropriateness of information flows based on social or cultural norms in specific contexts. CI describes information flows using five parameters: (1) the subject of the information being transferred, (2) the sender of this information, (3) the attribute or type of information, (4) the recipient of the information, and (5) the transmission principle or condition imposed on the transfer of information from the sender to the recipient. For example, one might be comfortable with a search engine (*recipient)* collecting their (*subject & sender)* Internet browsing history (*attribute)* in order to improve search results (*transmission principle*), but not in order to improve advertisement targeting, which is a different transmission principle that places the information in a different context governed by different norms. Privacy norms can therefore be inferred from the reported appropriateness and acceptability of information flows with varying combinations of these five parameters.

Previous research has used CI to discover and analyze privacy norms in various contexts. In 2012, Winter used CI to design an interview study investigating Internet of things (IoT) device practices that could be viewed as privacy violations [54].

In 2016, Martin and Nissenbaum conducted a survey with vignette questions based on CI to understand discrepancies between people's stated privacy values and their actions in online spaces [27]. Rather than straightforward contradictions, they find that these discrepancies are due to nuanced effects of contextual information informing real-world actions. This result motivates the use of CI in our study and others to investigate privacy norms in realistic situations.

In 2016, Shvartzshnaider et al. used the language of CI to survey crowdworkers' privacy expectations regarding information flow in the education domain [46]. Survey respondents indicated whether information flows situated in clearly defined contexts violated acceptability norms. The results were converted into a logic specification language which could be used to verify privacy norm consistency and identify additional acceptable information flows.

In 2018, we designed a scalable survey method for discovering privacy norms using questions based on CI [3]. We applied the survey method to measure the acceptability of 3,840 information flows involving common connected devices for consumer homes. Results from 1,731 Amazon Mechanical Turk respondents informed recommendations for IoT device manufacturers, policymakers, and regulators.

This paper adapts the survey method from our previous work [3] for a specific application: comparing privacy norms to privacy regulation. Our use of language from regulation in CI survey questions, direct comparison of discov-

ered privacy norms to policy compliance plans, and survey panel of special interest individuals (parents of children under age 13) distinguishes our work from previous uses of the survey method and previous CI studies in general.

2.2 COPPA & Smart Toys

Previous research has investigated Internet-connected toys and COPPA from various perspectives. Several studies have focused on identifying privacy and/or security vulnerabilities of specific smart toys [45, 48, 53], some of which are expressly noted as COPPA violations [6]. Our work uses these examples to inform the information flow descriptions included on our survey.

Researchers have also developed methods to automate the detection of COPPA violations. In 2017, Zimmeck et al. automatically analyzed 9,050 mobile application privacy policies and found that only 36% contained statements on user access, editing, and deletion rights required by COPPA [59]. In 2018, Reyes et al. automatically analyzed 5,855 Android applications designed for children and found that a majority potentially violated COPPA [43]. Most violations were due to collection of personally identifiable information or other identifiers via third-party software development kits (SDKs) used by the applications, often in violation of SDK terms of service. These widespread violations indicate that COPPA remains insufficiently enforced. Nevertheless, COPPA remains the primary legal foundation for state [30] and federal [12] action against IoT toy manufacturers and other technology companies for children's privacy breaches.

Additional work has investigated parents' and children's relationships with Internet-connected toys. In 2015, Manches et al. conducted observational fieldwork of children playing with Internet-connect toys and held in-school workshops to investigate parents' and children's cognizance of how IoT toys work [26]. They found that most children and caregivers were unaware of IoT toys' data collection potential, but quickly learned fundamental concepts of connected toy design when instructed.

In 2017, McReynolds et al. conducted interviews with parents and children to understand their mental models of and experience with Internet-connected toys [28]. Parents in this study were more aware of and concerned about IoT toy privacy than in [26], likely due to the intervening two years of negative publicity about connected toy privacy issues. The parents interviewed by McReynolds et al. provided feedback about desired privacy properties for connected toys, such as improved parental controls and recording indicators. The researchers urge ongoing enforcement of COPPA, but do not evaluate the parents' responses in light of the law.

Our work builds on past research by obtaining opinions about smart toy information collection and transfer practices from a much larger pool of parents (195 subjects). We use these data to evaluate whether privacy protections mandated by COPPA align with parents' privacy norms.

3 CI Survey Method

This study adapts a CI-based survey method first presented in our previous work [3] to evaluate whether specific requirements in privacy regulations align with user privacy norms. We chose this particular survey method because it is previously tested, scalable to large respondent populations, and easily adaptable to specific domains. The survey method works as follows, with our modifications for regulation analysis marked in italics:

1. Information transfers ("flows") are defined according to CI as sets of five parameters: subject, sender, attribute, recipient, and transmission principle (described in Section 2.1).
2. We select lists of values for each of these parameters *drawn from or directly relevant to a particular piece of privacy regulation.* Using these values, we generate a combinatorial number of information flow descriptions *allowed or disallowed by the regulation.*
3. Survey respondents rate the acceptability of these information flows, each of which describe a concrete data collection scenario in an understandable context.
4. Comparing the average acceptability of flows *allowed or disallowed by the regulation* indicates how well they align with respondents' privacy norms.
5. Variations in acceptability contingent upon specific information flow parameters or respondent demographics can reveal nuances in privacy norms *that may or may not be well served by the regulation.*

The following sections provide detailed descriptions of our survey design (Sections 3.1–3.2), deployment (Section 3.3), and results analysis (Section 3.4) for comparing parents' privacy norms about smart toy data collection against COPPA regulation. Many of these steps mirror those in our previous work [3], but we include them here with specific details from this study for the sake of replicability.

3.1 Generating Smart Toy Information Flows

We first selected CI information flow parameters (Table 1) involving smart toys and specific data collection requirements from COPPA. We then programmatically generated information flow descriptions from all possible combinations of the selected CI parameters.

We next discarded certain information flow descriptions with unrealistic sender/attribute pairs, such as a toy speaker (sender) recording a child's heart rate (attribute). Unrealistic sender/attribute pairs were identified at the authors' discretion based on whether each toy could reasonably be expected to have access to each type of data during normal use. This decision was informed by smart toy products currently available on the market. The use of exclusions to remove

unrealistic information flows is a core part of the CI survey method [3] for reducing the total number of questions and the corresponding cost of running the survey. This process resulted in 1056 total information flow descriptions for use in CI survey questions (Section 3.2).

The degree to which these flows are rated as acceptable or unacceptable by survey respondents indicate agreement or disagreement between COPPA and parents' privacy norms. This rest of this section describes how we selected values for each information flow parameter in detail.

Transmission Principles from COPPA. We used the Federal Trade Commission's Six Step Compliance Plan for COPPA [10] to identify transmission principles. Some of these transmission principles match those in our previous work [3], facilitating results comparison.

We converted steps 2–4 of the Compliance Plan into four transmission principles regarding consent, notification, and privacy policy compliance (Table 1). COPPA dictates that parents must receive direct notice and provide verifiable consent before information about children is collected. Operators covered by COPPA must also post a privacy policy that describes what information will be collected and how it will be used. Our corresponding transmission principles allow us to test whether these requirements actually increase the acceptability of data collection from and about children.

The fifth step of the Compliance Plan concerns "parents' ongoing rights with respect to personal information collected from their kids" [10]. Operators must allow parents to review collected information, revoke their consent, or delete collected information. We translated this requirement into the transmission principle "if its owner can at any time revoke their consent, review or delete the information collected."

The sixth step of the Compliance Plan concerns operators' responsibility to implement "reasonable procedures to protect the security of kids' personal information" [10] and to only release children's information to third party service providers who can do likewise. We translated this step into five transmission principles involving confidentiality, security, storage and deletion practices (Table 1).

The Compliance Plan also lists a set of exclusions to COPPA. We converted the exclusions that were most applicable to Internet-connected children's devices into four transmission principles (Table 1). We also added the transmission principle "if it complies with the Children's Online Privacy Protection Rule" to test parents' trust and awareness of COPPA itself.

Importantly, we also included the *null* transmission principle to create control information flows with no COPPA-based criteria. Comparing the acceptability of flows with the *null* transmission principle against equivalent flows with COPPA-based transmission principles allows us to determine whether the COPPA conditions are relevant to parents' privacy norms.

Smart Toy Senders. The senders included in our survey represent five categories of children's IoT devices: a smart speaker/baby monitor, a smart watch, a toy walkie-talkie, a smart doll, and a toy robot. We chose these senders by searching for children's Internet-connected devices mentioned in recent press articles [13, 17, 20, 29, 31, 35], academic papers [5, 25], blogs [9, 19, 37], IoT-specific websites [21, 23, 38], and merchants such as Toys "R" Us and Amazon. All of the selected senders are devices that are reasonably "directed towards children" [10, 11] in order to ensure that they are covered by COPPA. We excluded devices such as smart thermometers or other smart home devices that might collect information about children but are not directly targeted at children.

It is important to note that the selected devices do not represent the full breadth of smart toy products. However, information flow descriptions involving specific devices or device categories evoke more richly varied privacy norms from survey respondents than flows describing a generic "smart toy." This is supported by existing interview data [58] noting that IoT device owners often have very different privacy opinions of specific entities than of their generic exemplars (e.g., the "Seattle government" versus "government").

Information Attributes. We reviewed academic research [25], online privacy websites [38], toy descriptions [15], and privacy policies [18, 36] to compile a list of information attributes collected by the toys in our sender list. The final selected attributes include heart rate, frequently asked questions, the times the subject is home, frequently traveled routes, the times the device is used, location, sleeping habits, call history, audio of the subject, emergency contacts, video of the subject, and birthday. These attributes cover a variety of personally identifiable or otherwise sensitive information with specific handling practices mandated by COPPA.

First- and Third-party Recipients. We included device manufacturers and third-party service providers as recipient parameters. This allowed us to examine variations in privacy between first and third parties while limiting the total number of information flows and the corresponding cost of running the survey.

Children as Information Subjects. The only subject parameter included in the survey is "its owner's child." This wording emphasizes that the child is not the owner of the device and acknowledges the parental role in ensuring children's privacy. It also accounts for devices that may not be used directly or exclusively by the child (e.g., a baby monitor). We indicated in the survey overview that respondents should think about their own children's information when interpreting this subject.

Sender	Transmission Principle
a smart speaker/baby monitor	*COPPA Compliance Plan Steps 2-3*
a smart watch	if its privacy policy permits it
a toy walkie-talkie	if its owner is directly notified before the information was collected
a smart doll	
a toy robot	*COPPA Compliance Plan Step 4*
	if its owner has given verifiable consent
Recipient	if its owner has given verifiable consent before the information was collected
its manufacturer	
a third-party service provider	*COPPA Compliance Plan Step 5*
	if its owner can at any time revoke their consent, review or delete the information collected
Subject & Attribute	
its owner's child's heart rate	*COPPA Compliance Plan Step 6*
its owner's child's frequently asked questions	if it implements reasonable procedures to protect the information collected
	if the information is kept confidential
the times its owner's child is home	if the information is kept secure
its owner's child's frequently traveled routes	if the information is stored for as long as is reasonably necessary for the purpose for which it was collected
	if the information is deleted
the times it is used	
its owner's child's location	
its owner's child's sleeping habits	*COPPA Exclusions*
its owner's child's call history	if the information is used to protect a child's safety
audio of its owner's child	if the information is used to provide support for internal operations of the device
its owner's child's emergency contacts	if the information is used to maintain or analyze the function of the device
video of its owner's child	if the information is used to serve contextual ads
its owner's child's birthday	
	Other
	if it complies with the Children's Online Privacy Protection Rule
	null

Table 1: Contextual integrity parameter values selected for information flow generation. The *null* transmission principle is an important control included to generate information flows with no explicit conditions. The transmission principles were derived from the FTC's Six Step Compliance Plan for COPPA [10].

3.2 Survey Design

We created and hosted the survey on the Qualtrics platform [39]. The survey was split into six sections: consent, demographic questions I, overview, contextual integrity questions, awareness questions, and demographic questions II. This section provides details about each section. The survey did not mention COPPA, privacy, security, nor any potential negative effects of smart toy information flows prior to the contextual integrity questions to prevent priming and framing effects.

Consent. Respondents were initially presented with a consent form approved by our university's Institutional Review Board. Respondents who did not consent to the form were not allowed to proceed with the study.

Demographic Questions I. The first set of demographic questions asked respondents for the ages of their children under 13. We chose this age limit because COPPA only applies to data collection from children under 13. We randomly selected one of the ages for each respondent, n, which was piped to the survey overview.

Overview. Respondents were then presented with a survey overview containing a brief description of Internet-connected devices and instructions for the contextual integrity questions (Appendix A). This overview also explained how respondents should interpret the recurring phrase "its owner's child," and instructed them to keep their n-year-old child in mind while taking the survey (where n was selected for each respondent from their responses to the demographics questions I).

Contextual Integrity Questions. The core of the survey consisted of 32 blocks of questions querying the acceptability of our generated information flows (Section 3.1). Each question block contained 33 information flows with the same sender, same attribute, varying recipients, and varying transmission principles. For example, one block contained all information flows with the sender "a smart doll" and the at-

tribute "the times it is used." Each question block also included one attention check question.

Each respondent was randomly assigned to a single question block. Answering questions about flows with the same sender and attribute reduced cognitive fatigue and ensured independence across recipients and transmission principles.

The information flows in each block were divided into matrices of individual Likert scale multiple choice questions. The first matrix in each block contained questions about information flows to different recipients with the *null* transmission principle (Figure 1). The remaining matrices each contained questions about information flows to a specific recipient with varying transmission principles (Figure 2). The order of the information flows in each block was randomized for each respondent.

Each individual multiple choice question in the matrices asked respondents to rate the acceptability of a single information flow on a scale of five Likert items: Completely Acceptable (2), Somewhat Acceptable (1), Neutral (0), Somewhat Unacceptable (-1), Completely Unacceptable (-2). We also included the option "Doesn't Make Sense" to allow respondents to indicate if they didn't understand the information flow.

Awareness Questions. Respondents then answered questions about their general technological familiarity and Internet use, ownership of Internet-connected devices, ownership of children's Internet-connected devices, and previous knowledge of COPPA.

Demographic Questions II. Finally, respondents answered standard demographic questions from the United States Census. This allowed us to check the representativeness of our sample (Appendix B, Section 5.2) and account for demographic variables in our analysis.

3.3 Survey Deployment

We tested the survey on UserBob [52] once during the survey design process and again immediately prior to deployment. UserBob is a usability testing service for obtaining video screen capture of users interacting with a website while recording audio feedback. Each survey test involved creating a UserBob task with a link to the survey, brief instructions for users,[1] and settings to recruit 4 users to take the survey for 7 minutes each. UserBob automatically recruited users through Amazon Mechanical Turk at a cost of $1 per user per minute. The resulting video and audio recordings of users interacting with the survey informed changes to our survey design. In particular, we reduced the number of questions per block and increased the number of pages over which the questions were presented. This reduced the amount of scrolling necessary to complete the survey and improved engagement. This practice of using pre-deployment "cognitive interviews" to test and debug survey design is common in survey research [49]. UserBob responses were not included the final results.

We used Cint [8], an insights exchange platform, to deploy our survey to a panel of 296 adult parents of children under the age of 13 in the United States. We selected respondents with children younger than 13 because COPPA applies to "operators of websites or online services directed to children under 13" [11]. Our surveyed population therefore consisted entirely of individuals affected by COPPA. We chose not to set a minimum age for respondents' children, because there is a lack of readily available information on the minimum age of use of Internet-connected children's devices. While certain manufacturers list recommended minimum ages for their connected toys and devices, this was not the case for the majority of the devices we considered. Additionally, many devices such as wearable trackers, water bottles, baby monitors, are targeted towards very young children. Lastly, not restricting the minimum age allowed us to relax the demographic requirements for survey deployment.

Respondents were paid $3 for valid responses where the attention check question was answered correctly. Each respondent was only allowed to answer the survey once. The survey responses were collected over an 18 hour time frame. We chose Cint to deploy our survey instead of Amazon Mechanical Turk, because Cint allowed us to directly target a specific panel of respondents (as in Zyskowski et al. [60]) without requiring a preliminary screening questionnaire to identify parents [44].

3.4 Response Analysis

We began with 296 responses. We removed the responses from 8 respondents who did not consent to the survey (none of their information was recorded) as well as those from 85 respondents who did not correctly answer the attention check question. We removed 2 responses in which over 50% of the information flows were characterized as "Doesn't make sense." We also removed 2 responses where not all information flow questions were answered. Finally, we removed 1 response where the respondent self-reported over 10 children and 3 responses that were completed in less than 2 minutes. This resulted in a final set of 195 responses with an average of 6 responses per information flow (standard deviation 1.4).

The responses to all contextual integrity questions (Section 3.2) were on a Likert scale with the following Likert items: "Completely acceptable" (2), "Somewhat acceptable" (1), "Neutral" (0), "Somewhat unacceptable" (-1), and "Completely unacceptable" (-2). We call this value the "acceptability score" of each information flow for each respondent.

In order to generalize privacy norms beyond individual respondents and information flows, we averaged the accept-

[1] UserBob task instructions: "This is a survey that will be given to a group of parents with children younger than 13. Take the survey, pretending you have one or more children younger than 13. Record your thoughts on the user interface and whether the questions do/don't make sense."

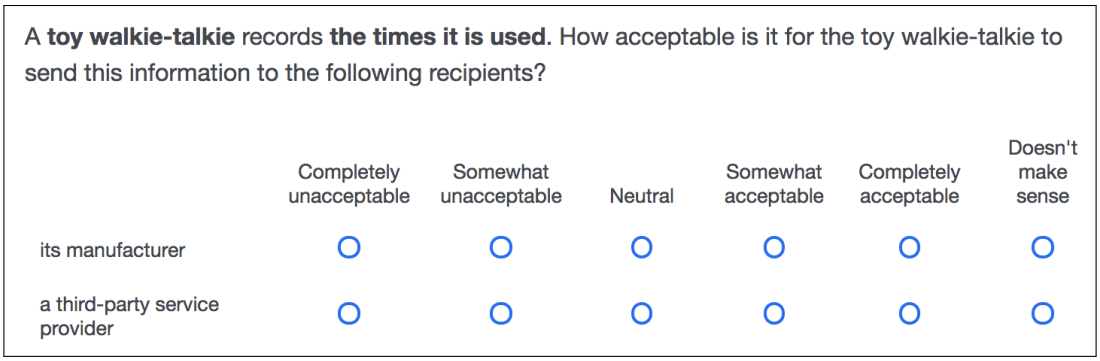


Figure 1: Example CI question matrix with information flows to different recipients and the *null* control transmission principle.

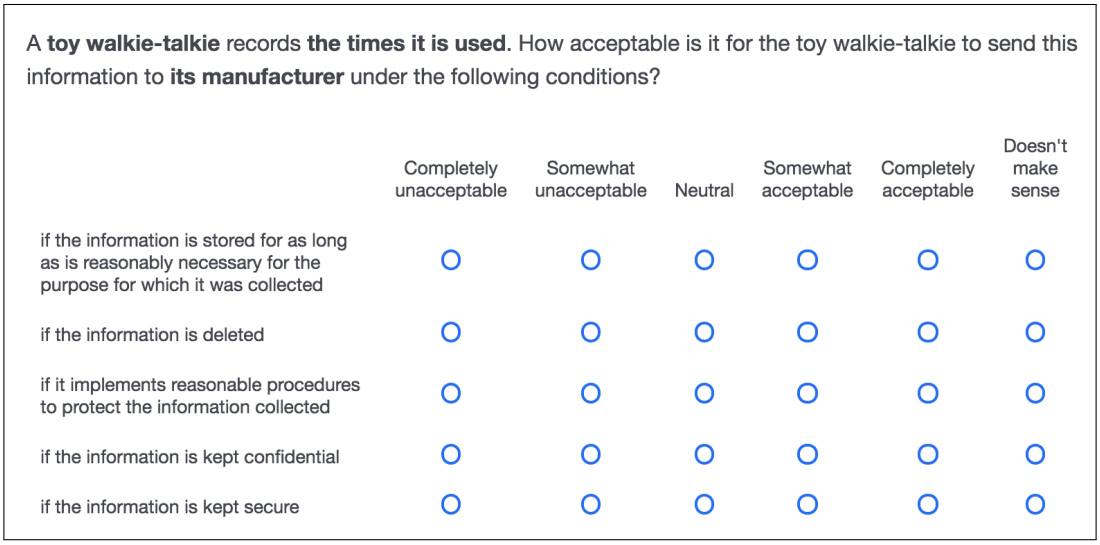


Figure 2: Example CI question matrix with information flows to a fixed recipient and varying transmission principles.

ability scores of flows grouped by CI parameters or respondent demographics. For example, we averaged the acceptability scores of all information flows with the recipient "its manufacturer" and the transmission principle "if the information is deleted" in order to quantify the pairwise effects of these two parameters on privacy norms. We then plotted these pairwise average acceptability scores as heatmaps to visualize how individual CI parameters or respondent demographic factors affect the overall alignment of information flows with privacy norms (Figures 3 & 4).

We statistically compared the effects of different COPPA provisions (Sections 4.1–4.4) by averaging the acceptability scores of all information flows grouped by transmission principles. For example, one group contained the average score given by each of the 195 respondents to information flows with non-*null* transmission principles, while a second group contained the average score given by each respondent to information flows with the *null* transmission principle. We then applied the Wilcoxon signed-rank test to find the likelihood that these two groups of scores come from the same distribution. We performed three such tests with different transmission principle groups and set the threshold for significance to $p = 0.05/3 = 0.016$ to account for the Bonferroni multiple-testing correction.

We statistically compared the effects of smart device awareness, COPPA familiarity, and demographic factors (Sections 4.5–4.8) by averaging the acceptability scores of all information flows grouped by respondent category of interest. For example, one group contained the average score given by each respondent who owned a smart device across all answered CI questions, while the second set contained the average score given by each respondent who did not own a smart device. We then applied the Wilcoxon signed-rank test to find the likelihood that these two groups of scores come from the same distribution. We performed five such tests with groupings based on COPPA familiarity, age, smart device ownership, education, and income and set the threshold for significance to $p = 0.05/5 = 0.01$ to account for the Bonferroni multiple-testing correction.

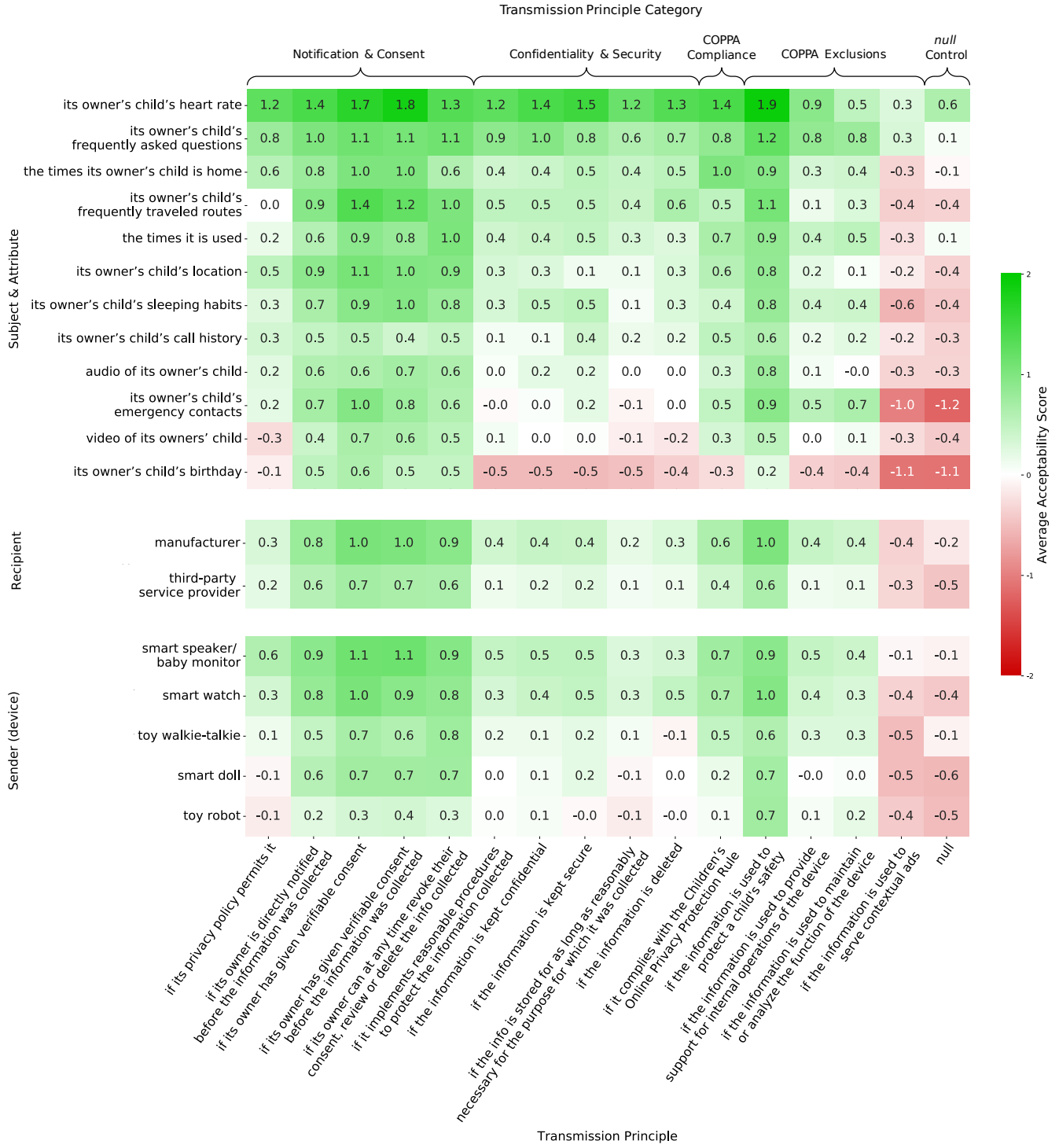


Figure 3: Average acceptability scores of information flows grouped by COPPA-derived transmission principles and attributes, recipients, or senders. Scores range from −2 (completely unacceptable) to 2 (completely acceptable).

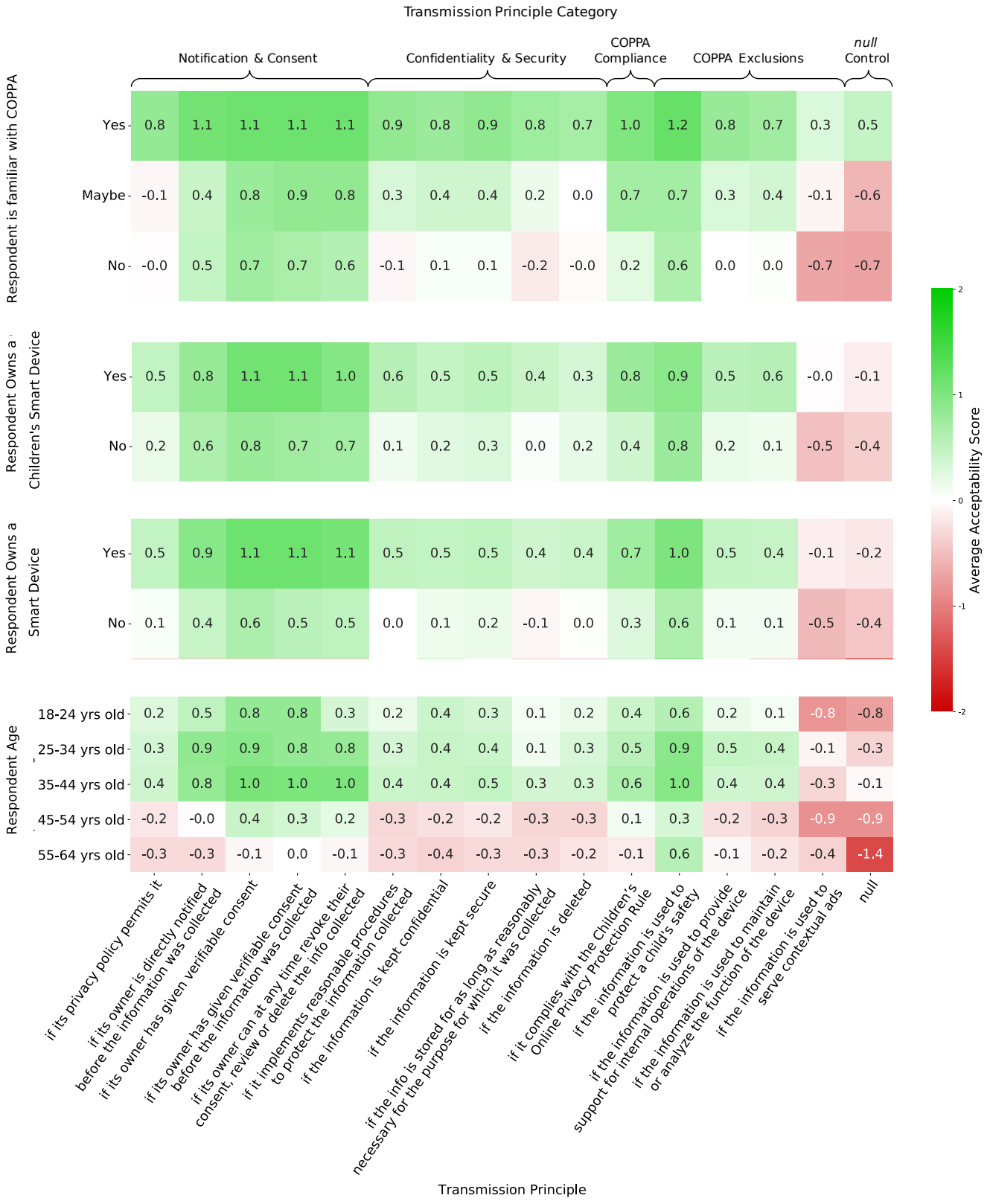


Figure 4: Average acceptability scores of information flows grouped by COPPA-derived transmission principles and respondent ages, familiarity with COPPA, or ownership of smart devices. Scores range from −2 (completely unacceptable) to 2 (completely acceptable)

4 Results

Overall, surveyed parents view information flows meeting COPPA data collection guidelines as acceptable while viewing equivalent information flows without COPPA criteria as unacceptable (Figures 3 & 4). This supports the conclusion that COPPA-mandated information handling practices generally align with parents' privacy norms. In this section, we elaborate on this finding and explore additional trends in our survey responses to further compare COPPA to parents' privacy norms regarding children's smart toys.

4.1 COPPA Data Collection Requirements Align with Parents' Privacy Norms

COPPA requirements were incorporated in the survey as information flow transmission principles derived from the FTC's Six-Step Compliance Plan for COPPA [10] (Section 3.1). The average acceptability scores of information flows explicitly obeying these requirements are mostly non-negative (Figures 3 & 4). This indicates that most surveyed parents consider these flows as "completely acceptable" or "somewhat acceptable." In comparison, the average acceptability scores of information flows with the control *null* transmission principle are mostly negative (Figures 3 & 4), indicating that most surveyed parents consider these flows without COPPA criteria as "completely unacceptable" or "somewhat unacceptable."

This difference between information flows with no explicit conditions versus flows with COPPA requirements holds regardless of information sender, recipient, attribute, or parents' demographics (apart from a few specific exceptions which we discuss below). On average, information flows with COPPA-derived transmission principles are 0.73 Likert-scale points more acceptable than their *null* transmission principle counterparts ($p < 0.001$).

Our research provides the first quantitative evidence that COPPA guidelines generally match parents' privacy norms for Internet-connected toys. This indicates that regulation can mandate meaningful transmission principles for information flows and supports further creation and fine-tuning of regulation to keep Internet data collection within the bounds of consumer privacy preferences.

4.2 Parents View Data Collection for Contextual Advertising as Unacceptable

Information flows with the transmission principle "if the information is used to serve contextual ads" have negative average acceptability scores across almost all senders, recipients, and attributes (Figure 3). Unlike all other information flows on our survey with non-*null* transmission principles, these flows are actually prohibited by COPPA. The "contextual ads" transmission principle is a "limited exception to COPPA's verifiable parental consent requirement" as listed in the COPPA Compliance Plan [10]. This exception only applies to the collection of persistent identifiers (such as cookies, usernames, or user IDs) and not to any of the attributes included on our survey. Our respondents generally agree that collecting the attributes on our survey for contextual (targeted) advertising would be unacceptable, providing further support for COPPA's alignment with parents' norms.

This result indicates that the CI survey technique can detect regulatory provisions that reduce alignment with privacy norms, essential for future applications of the method (Section 6.2). It also provides evidence that the mere presence of a transmission principle doesn't necessarily improve the acceptability of information flows.

This result relates to existing work about opinions of data collection for advertising. Zheng et al. [58] interviewed owners of non-toy Internet-connected home devices and found mixed opinions of targeted advertising with data from these devices depending on the perceived benefit to the user. Combined with our results, this suggests that parents do not believe that relaxing COPPA to allow contextual advertising from more types of children's toy data would have enough benefit to outweigh privacy concerns.

4.3 Parents View Children's Birthdays as Especially Private

Information flows including the subject and attribute "its owner's child's birthday" are an exception to the trend described in Section 4.1. The average acceptability scores of information flows with this attribute and 10 of the 15 COPPA-derived transmission principles are negative (Figure 3). This discrepancy could be attributed to the relatively small number of parents (11 parents or 5.6% of total respondents) who were asked to score flows with this attribute. Parents may also view their children's birthdays as more personal than the other surveyed attributes or as less necessary for some of the surveyed transmission principles (such as "to maintain or analyze the function of the device"). Follow-up qualitative studies could focus on specific attributes, such as children's birthdays, to understand parents' rationales behind corresponding privacy norms.

4.4 Notification & Consent Versus Confidentiality & Security

Our results also provide insights into the relative importance of different sections within COPPA to parents' privacy norms. This could help regulators prioritize certain forms of non-compliant information collection for legal action.

Our COPPA-derived transmission principles can be divided into categories based on their topic and the section of the COPPA Compliance Plan [10] from which they were drawn (Section 3.1). One category consists of transmission principles from the Compliance Plan steps 2–5 regarding notification and consent (Table 1). These transmission princi-

ples involve device privacy policies, the collection of verified consent, and the ability to revoke consent or review collected information. Another category consists of transmission principles from the Compliance Plan step 6 regarding information confidentiality and security (Table 1). These transmission principles involve reasonable data protection, confidential and secure storage, and limited information lifetime.

Across all senders, attributes, and recipients, information flows with transmission principles in the notification/consent category have significantly higher acceptability scores than flows with transmission principles in the confidentiality/security category by an average of 0.43 Likert scale points ($p < 0.001$) (Figure 3). One notable exception to this trend is the transmission principle "if its privacy policy permits it." The acceptability scores for this transmission principle are an average of 0.53 Likert points lower than for others in the notification/consent category ($p < 0.001$). We suspect this reflects the general distrust of privacy policies evidenced in previous research [50]. Privacy policies are typically dense, lengthy, and difficult to interpret even for experts [42]. It therefore makes sense that parents would not view the disclosure of information collection in privacy policies as acceptable as other notification methods.

The greater acceptability of information flows with notification or consent criteria versus flows with confidentiality or security criteria corroborates previous research using the CI survey method to discover privacy norms of non-toy consumer IoT devices [3]. This provides longitudinal data indicating that users of Internet-connected products continue to prioritize consent over built-in security when reasoning about the appropriateness of information collection practices. This motivates continued work to improve the state of notification and consent mechanisms for Internet data collection. The most prevalent mechanisms, privacy policies and mobile application permissions, are widely understood to be ineffective for informing users or providing meaningful privacy control options [47]. As policies change to nuance the definitions of informed consent to include ideas of intelligibility, transparency and active opt-in, among others, it is important to continue to study and evaluate consumer's privacy expectations regarding consent.

4.5 COPPA Compliance and Familiarity Increase Data Collection Acceptability

Information flows with the transmission principle "if it complies with the Children's Online Privacy Protection Rule" received a positive average acceptability score of 0.49 across all senders, recipients, and attributes. As expected, flows with this transmission principle were rated as more acceptable by the 67% of respondents familiar with COPPA than by the 33% of respondents unfamiliar with the rule.

Furthermore, respondents who indicated that they were familiar with COPPA rated all information flows 0.75 Likert points more acceptable on average than respondents who were not familiar with the rule ($p < 0.001$) (Figure 4).

In both cases, stated compliance and/or familiarity with COPPA may increase parents' acceptance of smart toy data collection by reassuring them that their children's privacy is protected by regulation. However, this may be a false sense of security, as COPPA guidelines are relatively broad and COPPA violations are likely widespread in practice (Section 6.1) [6, 43].

4.6 Younger Parents are More Accepting of Smart Toy Data Collection

Parents younger than 45 gave an average acceptability score of 0.48 to all rated flows, following the trend discussed in Section 4.1 (Figure 4). In comparison, parents 45 years and older gave an average acceptability score of -0.17 to all rated flows. This difference in the acceptability scores of these two groups is significant ($p < 0.01$). Nevertheless, context still matters, as information flows specifically "to protect a child's safety" are viewed as generally acceptable to all surveyed parents regardless of age.

Previous work indicates that young American adults are more aware of online privacy risks and more likely to take steps to protect their privacy online than older adults [40]. Future studies could investigate why this awareness of online privacy risks makes younger parents more accepting of smart toy data collection.

4.7 Parents Who Own Smart Devices are More Accepting of Data Collection

Parents who own generic smart devices or children's smart devices were more accepting of information flows than respondents who do not own these devices on average, but the difference in scores (0.34 Likert scale points) between these two groups is not significant ($p = 0.12$).

Nevertheless, this difference corroborates previous work using the CI survey method, in which owners of non-toy consumer IoT devices were found to be more accepting of information flows from these devices than non-owners [3]. This difference likely reflects a self-selection bias, in which those more uncomfortable with Internet data collection are less likely to purchase Internet-connected toys or other devices. However, the small effect size in both this study and the previous work may be due to parents purchasing smart toys unaware of their data collection potential [26] or willing to trade-off privacy concerns for other benefits provided by the products [58].

4.8 Education & Income have Little Effect on Parents' Smart Toy Privacy Norms

Parents' education and income did not have significant effects on acceptability scores. Parents earning more than

$100,000 per year gave an average acceptability score of 0.46 to all rated flows, not significantly different from the average score of 0.37 from parents earning less ($p = 0.77$). Similarly, parents with at least some college education gave an average acceptability score of 0.37, not significantly different from the 0.33 average score of parents with a high school diploma or less ($p = 0.58$). This is perhaps unexpected given previous work indicating that parents with more resources are more likely to engage with children on privacy issues [41] and is a topic for follow-up research.

5 Limitations

Our results must be considered in the context of the following limitations.

5.1 Privacy Attitudes Versus Behaviors

Individuals often self-report greater privacy awareness and concerns than reflected in actual privacy-related behaviors [1, 22]. This "privacy paradox" is well-documented and poses a challenge for researchers. The CI survey method is vulnerable to privacy paradox effects. However, there is a reasonable argument that privacy regulation should prioritize the expressed norms of users (measured by the survey instrument) over norms evidenced through behaviors, which are influenced by external factors (such as confusing user interfaces) that could be affected by the regulation. For example, it is often difficult for consumers to determine the data collection practices of IoT devices, including Internet-connected children's toys, due to poor company disclosure practices [42] and limited auditing by third parties. Just because many parents purchase smart toys does not mean that they approve of the toys' data collection practices and wouldn't support new regulation to improve privacy.

5.2 Respondent Representativeness

The self-reported demographics of our respondents (Appendix B) indicate that the sample, while diverse, is non-representative in ways that may influence measured privacy norms.

Females and high-income individuals are notably overrepresented in our sample compared to the United States population. The literature on gender differences in online privacy concerns suggests that women may generally perceive more privacy risks online than men [4, 14, 56], but some studies contradict this conclusion, reporting no significant gender effect [55]. The effect of income on online privacy concerns is similarly unsettled, with some reporting that high-income individuals are less concerned about privacy [24, 33], others reporting that high-income individuals are more likely to engage in privacy-preserving behaviors [41], and still others finding no significant income effect [57].

Limiting our survey to parents also ignores the opinions of other parties, including school and daycare teachers and extended family members, who also purchase Internet-connected toys for children but may have different privacy norms. These individuals are also affected by COPPA and have legitimate justification for their opinions and interests to be reflected in children's privacy regulation. Likewise, we did not ask whether our respondents were members of communities that may have less common privacy norms, but our respondent panel, drawn from across the United States, certainly missed smaller demographics.

Finally, our respondent panel consisted entirely of parents living in the United States, as COPPA only applies to products sold in the U.S. These respondents are therefore influenced by American attitudes toward privacy, which may vary from those of parents in other countries. We hope that future work will apply the CI survey method used in this paper to evaluate the alignment between privacy norms and privacy regulation in non-U.S. contexts.

5.3 Goals of Privacy Regulation

Our use of CI surveys to evaluate privacy regulation assumes that the underlying value of such regulation is to better align data collection practices with privacy norms. This makes an implicit normative argument about the purpose of privacy regulation, which does not necessarily hold, especially for the norms of majority populations. For example, privacy regulation may seek to protect minority or otherwise vulnerable populations. In these cases, surveys of all individuals affected by the regulation may reflect a majority view that does not value the norms or appreciate the situation of the target population. CI surveys could still be applied in these contexts, but care would need to be taken to identify and recruit respondents from populations differentially affected by the regulation in order to uncover discrepancies between the regulation and the norms of these groups.

Additionally, some regulation may be created with the goal of changing existing norms. In these cases, the CI survey method will indicate that the regulation does not match current privacy expectations upon enactment. However, CI surveys would still be useful for conducting longitudinal measurements to track whether the regulation has the desired effect on privacy norms over time.

6 Discussion & Future Work

We would like this study to serve as a template for future work using contextual integrity to analyze current or pending privacy regulation for policy or systems design insights. This section discusses our COPPA findings and presents suggestions for future applications of our method by policymakers, device manufacturers, and researchers.

6.1 COPPA Insights & Concerns

Previous research indicates that parents actively manage the information about their children on social media platforms to avoid oversharing [2], and that owners of IoT home appliances view most data collection by these devices as inherently unacceptable [3]. We expected that these domains would overlap, resulting in skepticism of smart toy data collection that even the restrictions in COPPA could not ameliorate. Surprisingly, it seems that the COPPA criteria assuaged parents' privacy concerns on average.

While we are encouraged that COPPA generally aligns with parents' privacy expectations, we are also concerned that the existence of COPPA may give parents an unreasonable expectation that their children's data is protected, especially since parents familiar with COPPA were less critical of smart toy information flows. In fact, several online services and Internet-connected toys have been found to violate COPPA [6,43], and many more non-compliant toys are likely available for purchase. Additionally, the information collection guidelines in COPPA are relatively broad, leaving room for technical implementations that adhere to the letter of the law but still compromise children's privacy. This motivates continued work by regulators and researchers to identify toys that place children's privacy at risk, as well as healthy skepticism by parents before purchasing any particular toy.

As an additional policy insight, variations in information flow acceptability across recipients[2] corroborate previous work [58] indicating that privacy norms are deeply contingent on the perception of entities that collect online data. COPPA distinguishes between first- and third-parties, but does not further categorize data recipients. This increases the flexibility of the law, but raises the potential that some recipients, which may viewed as completely unacceptable by privacy norms, could still legally get access to children's data. This suggests that incorporating a more contextual framing of entities could improve the ability of future regulation to prevent unwanted data collection practices.

6.2 Further Policy Analysis Applications

The CI survey method is not limited to COPPA. We would like to see the results of follow-up studies focusing on different regulation, such as the Health Insurance Portability and Accountability Act (HIPAA), the Family Educational Rights and Privacy Act (FERPA), the National Cybersecurity Protection Advancement Act, the European General Data Protection Regulation (GDPR), and others from the local to international level, to see if their requirements result in similarly acceptable information flows for members of their target populations. As most privacy regulation encompasses information transfer or exchange, the theory of contextual integrity is an appropriate framework for this research. Further studies would also allow cross-regulatory analysis to find common factors that affect alignment with privacy norms.

The CI survey method could also be incorporated into the policymaking process. Policy formulation and resource allocation could be guided by surveying a wide-variety of information flows allowed under current regulation and identifying egregious or unexpected norm violations that require attention. Policymakers could test whether previous regulatory approaches will be applicable to new innovations by conducting surveys with CI parameters describing new technologies and existing regulation (e.g., smart toys and COPPA prior to the 2017 inclusion of IoT devices [16]). Policymakers could also perform A/B tests of policy drafts with different stipulations and/or language by conducting multiple parallel surveys with varying CI parameters. These and other use cases would improve quantitative rigor in data-driven policy development and facilitate the design of regulation responsive to the privacy norms of affected populations.

6.3 Systems Design Applications

The application of CI surveys to guide systems and product design is covered in detail in our previous work [3]. To summarize, device manufacturers can conduct CI surveys to determine whether information collection practices of devices or new features under development will violate consumer privacy norms. This allows modifications during the design process to prevent consumer backlash and public relations debacles.

Applying CI surveys to evaluate privacy regulation can also yield valuable insights for systems research and development. For example, learning that parents value the ability to revoke consent or delete information (Figure 3) motivates research into verifiable deletion of cloud data from IoT platforms. Such insights are especially relevant as neither privacy norms nor regulations are necessarily tied to technical systems feasibility. Discovering that a particular CI parameter value is crucial to privacy norm adherence could launch several research projects developing efficient implementations or correctness proofs. We expect future applications of the CI survey method will generate many such results.

7 Conclusion

Increased interest in data privacy has spurred new and updated regulation around the world. However, there are no widely accepted methods to determine whether this regulation actually aligns with the privacy preferences of those it seeks to protect. Here, we demonstrate that a previously developed survey technique [3] based on the formal theory of contextual integrity (CI) can be adapted to effectively measure whether data privacy regulation matches the norms of affected populations. We apply this methodology to test whether the Children's Online Privacy Protection Act's re-

[2] Information flows to first-party manufacturers have higher average acceptability scores than flows to third-party service providers (Figure 3).

strictions on data collection by Internet-connected "smart" toys align with parents' norms. We survey 195 parents of children younger than 13 about the acceptability of 1056 smart toy information flows that describe concrete data collection scenarios with and without COPPA restrictions.

We find that information flows conditionally allowed by COPPA are generally viewed as acceptable by the surveyed parents, in contrast to identical flows without COPPA-mandated restrictions. These are the first data indicating the general alignment of COPPA to parents' privacy norms for smart toys. However, variations in information flow acceptability across smart toys, information types, and respondent demographics emphasize the importance of detailed contextual factors to privacy norms and motivate further study.

COPPA is just one of many U.S. and international data privacy regulations. We hope that this work will serve as a template for others to adopt and repeat the CI survey method to study other legislation, allowing for a cross-sectional and longitudinal picture of the ongoing relationship between regulation and social privacy norms.

Acknowledgments

We thank Yan Shvartzshnaider and our survey respondents. This work was supported by the Accenture Fund of the School of Engineering and Applied Science at Princeton University.

References

[1] ACQUISTI, A., BRANDIMARTE, L., AND LOEWENSTEIN, G. Privacy and human behavior in the age of information. *Science 347*, 6221 (2015), 509–514.

[2] AMMARI, T., KUMAR, P., LAMPE, C., AND SCHOENEBECK, S. Managing children's online identities: How parents decide what to disclose about their children online. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems* (2015), ACM, pp. 1895–1904.

[3] APTHORPE, N., SHVARTZSHNAIDER, Y., MATHUR, A., REISMAN, D., AND FEAMSTER, N. Discovering smart home internet of things privacy norms using contextual integrity. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol. 2*, 2 (July 2018), 59:1–59:23.

[4] BARTEL SHEEHAN, K. An investigation of gender differences in on-line privacy concerns and resultant behaviors. *Journal of Interactive Marketing 13*, 4 (1999), 24–38.

[5] CHAUDRON, S., DI GIOITA, R., GEMO, M., HOLLOWAY, D., MARSH, J., MASCHERONI, G., PETER, J., AND YAMADA-RICE, D. Kaleidoscope on the internet of toys: Safety, security, privacy and societal insights. Tech. rep., EU Science Hub, Feb 2017.

[6] CHU, G., APTHORPE, N., AND FEAMSTER, N. Security and privacy analyses of internet of things children's toys. *IEEE Internet of Things Journal* (2018).

[7] CIGI-Ipsos global survey on internet security and trust. `https://www.cigionline.org/internet-survey-2016`, 2016. Centre for International Governance Innovation.

[8] Cint. `https://www.cint.com/`, 2018.

[9] Consumerist Archives: COPPA. `https://consumerist.com/tag/coppa/index.html`, 2018. Consumer Reports.

[10] FEDERAL TRADE COMMISSION. Children's Online Privacy Protection Rule: a six-step compliance plan for your business. `https://www.ftc.gov/tips-advice/business-center/guidance/childrens-online-privacy-protection-rule-six-step-compliance`, July 2017.

[11] FEDERAL TRADE COMMISSION. Children's Online Privacy Protection Rule ("COPPA"). `https://www.ftc.gov/enforcement/rules/rulemaking-regulatory-reform-proceedings/childrens-online-privacy-protection-rule`, Aug 2017.

[12] FEDERAL TRADE COMMISSION. Electronic toy maker vtech settles FTC allegations that it violated children's privacy law and the FTC Act. `https://www.ftc.gov/news-events/press-releases/2018/01/electronic-toy-maker-vtech-settles-ftc-allegations-it-violated`, Jan 2018.

[13] FINKLE, J., AND WAGSTAFF, J. Vtech hack exposes id theft risk in connecting kids to internet. `https://www.reuters.com/article/us-vtech-cyberattack-kids-analysis/vtech-hack-exposes-id-theft-risk-in-connecting-kids-to-internet-idUSKBN0TP0FQ20151206`, Dec 2015. Thomson Reuters.

[14] FOGEL, J., AND NEHMAD, E. Internet social network communities: Risk taking, trust, and privacy concerns. *Computers in human behavior 25*, 1 (2009), 153–160.

[15] Gator kids smart watch. `http://gatorsmartwatch.com/index.php/kids-gps-watch-supply/`, 2018.

[16] GRAY, S. Federal Trade Commission: COPPA Applies to Connected Toys. *Future of Privacy Forum* (June

2017). https://fpf.org/2017/06/26/federal-trade-commission-coppa-applies-connected-toys/.

[17] HEATER, B. Germany bans smartwatches for kids over spying concerns. *TechCrunch* (Nov 2017). https://techcrunch.com/2017/11/17/germany-bans-smartwatches-for-kids-over-spying-concerns/.

[18] Hello barbie privacy policy. https://toytalk.com/hellobarbie/privacy/, April 2017. PullString.

[19] IoTList – Discover the Internet of Things. http://iotlist.co/tag/kids, 2018.

[20] JOHNSTON, P. Toy-telligence the smart choice for children at Christmas. https://www.reuters.com/article/us-britain-christmas/toy-telligence-the-smart-choice-for-children-at-christmas-idUSKBN1CH283, Oct 2017. Thomson Reuters.

[21] Kidsafe seal program: Member list. https://www.kidsafeseal.com/certifiedproducts.html, 2018. Samet Privacy, LLC.

[22] KOKOLAKIS, S. Privacy attitudes and privacy behaviour: A review of current research on the privacy paradox phenomenon. *Computers & security 64* (2017), 122–134.

[23] LAUGHLIN, A. Smart toys - should you buy them? ttps://www.which.co.uk/reviews/smart-toys/article/smart-toys-should-you-buy-them. Which? Digital Blog.

[24] MADDEN, M. Privacy, security, and digital inequality. *Data & Society* (2017).

[25] MAHMOUD, M., HOSSEN, M. Z., BARAKAT, H., MANNAN, M., AND YOUSSEF, A. Towards a comprehensive analytical framework for smart toy privacy practices. In *International Workshop on Socio-Technical Aspects in Security and Trust (STAST)* (2017).

[26] MANCHES, A., DUNCAN, P., PLOWMAN, L., AND SABETI, S. Three questions about the internet of things and children. *TechTrends 59*, 1 (2015), 76–83.

[27] MARTIN, K., AND NISSENBAUM, H. Measuring privacy: an empirical test using context to expose confounding variables. *Colum. Sci. & Tech. L. Rev. 18* (2016), 176.

[28] MCREYNOLDS, E., HUBBARD, S., LAU, T., SARAF, A., CAKMAK, M., AND ROESNER, F. Toys that listen: A study of parents, children, and internet-connected toys. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (2017), ACM, pp. 5197–5207.

[29] MOON, A. FBI warns parents of privacy risks associated with internet-connected toys. https://www.reuters.com/article/us-usa-toys-fbi/fbi-warns-parents-of-privacy-risks-associated-with-internet-connected-toys-idUSKBN1A22AW, Jul 2017. Thomson Reuters.

[30] NEW YORK STATE ATTORNEY GENERAL'S PRESS OFFICE. A.G. Schneiderman announces $100,000 settlement with TRUSTe over flawed privacy certification program for popular children's websites, 2017.

[31] NIENABER, M. Germany bans talking doll Cayla, citing security risk. https://www.reuters.com/article/us-germany-cyber-dolls/germany-bans-talking-doll-cayla-citing-security-risk-idUSKBN15W20Q, Feb 2017. Thomson Reuters.

[32] NISSENBAUM, H. Privacy as contextual integrity. *Wash. L. Rev. 79* (2004), 119.

[33] O'NEIL, D. Analysis of internet users level of online privacy concerns. *Social Science Computer Review 19*, 1 (2001), 17–31.

[34] PAUL, K. Here's the no. 1 issue that consumers want corporate America to fix today. https://www.marketwatch.com/story/heres-the-no-1-reason-people-dislike-us-companies-2018-11-13, Nov 2018. MarketWatch.

[35] PEACHMAN, R. R. Mattel pulls aristotle childrens device after privacy concerns. *The New York Times* (Oct 2017).

[36] pi lab privacy policy. http://www.edwintheduck.com/privacy-policy, 2018. pi lab.

[37] POLK, R. Internet of things: When it comes to smart toys, it pays to shop smart. https://www.internetsociety.org/blog/2017/11/comes-smart-toys-pays-shop-smart/, November 2017.

[38] Privacy not included. https://advocacy.mozilla.org/en-US/privacynotincluded/, 2018. Mozilla.

[39] Qualtrics. http://www.qualtrics.com/, 2018.

[40] RAINIE, L. The state of privacy in post-Snowden America. *Pew Research Center FactTank* (September 2016).

[41] REDMILES, E. Net benefits: Digital inequities in social capital, privacy preservation, and digital parenting practices of U.S. social media users. *International AAAI Conference on Web and Social Media* (2018).

[42] REIDENBERG, J. R., BREAUX, T., CRANOR, L. F., FRENCH, B., GRANNIS, A., GRAVES, J. T., LIU, F., MCDONALD, A., NORTON, T. B., AND RAMANATH, R. Disagreeable privacy policies: Mismatches between meaning and users' understanding. *Berkeley Tech. LJ 30* (2015), 39.

[43] REYES, I., WIJESEKERA, P., REARDON, J., ON, A. E. B., RAZAGHPANAH, A., VALLINA-RODRIGUEZ, N., AND EGELMAN, S. "Won't somebody think of the children?" Examining COPPA compliance at scale. *Proceedings on Privacy Enhancing Technologies 2018*, 3 (2018), 63–83.

[44] SCHLEIDER, J. L., AND WEISZ, J. R. Using Mechanical Turk to study family processes and youth mental health: A test of feasibility. *Journal of Child and Family Studies 24*, 11 (2015), 3235–3246.

[45] SHASHA, S., MAHMOUD, M., MANNAN, M., AND YOUSSEF, A. Smart but unsafe: Experimental evaluation of security and privacy practices in smart toys. *arXiv preprint arXiv:1809.05556* (2018).

[46] SHVARTZSHNAIDER, Y., TONG, S., WIES, T., KIFT, P., NISSENBAUM, H., SUBRAMANIAN, L., AND MITTAL, P. Learning privacy expectations by crowdsourcing contextual informational norms. In *Proceedings of the 4th AAAI Conference on Human Computation and Crowdsourcing (HCOMP16)* (2016).

[47] SOLOVE, D. J. Introduction: Privacy self-management and the consent dilemma. *Harv. L. Rev. 126* (2012), 1880.

[48] STREIFF, J., KENNY, O., DAS, S., LEETH, A., AND CAMP, L. J. Who's watching your child? exploring home security risks with smart toy bears. In *IEEE/ACM Third International Conference on Internet-of-Things Design and Implementation (IoTDI)* (2018), pp. 285–286.

[49] SUDMAN, S., BRADBURN, N. M., SCHWARZ, N., ET AL. *Thinking about answers: The application of cognitive processes to survey methodology*. Jossey-Bass, San Francisco, CA, 1996.

[50] TUROW, J. Privacy policies on children's websites: Do they play by the rules?

[51] UNITED NATIONS CONFERENCE ON TRADE AND DEVELOPMENT. Summary of Adoption of E-Commerce Legislation Worldwide, January 2018.

[52] Userbob: Usability testing. `https://userbob.com/`, 2018.

[53] VALENTE, J., AND CARDENAS, A. A. Security & privacy in smart toys. In *Proceedings of the 2017 Workshop on Internet of Things Security and Privacy* (2017), ACM, pp. 19–24.

[54] WINTER, J. S. Privacy and the emerging internet of things: using the framework of contextual integrity to inform policy. In *Pacific telecommunication council conference proceedings* (2012), vol. 2012.

[55] YAO, M. Z., RICE, R. E., AND WALLIS, K. Predicting user concerns about online privacy. *Journal of the American Society for Information Science and Technology 58*, 5 (2007), 710–722.

[56] YOUN, S., AND HALL, K. Gender and online privacy among teens: Risk perception, privacy concerns, and protection behaviors. *Cyberpsychology & behavior 11*, 6 (2008), 763–765.

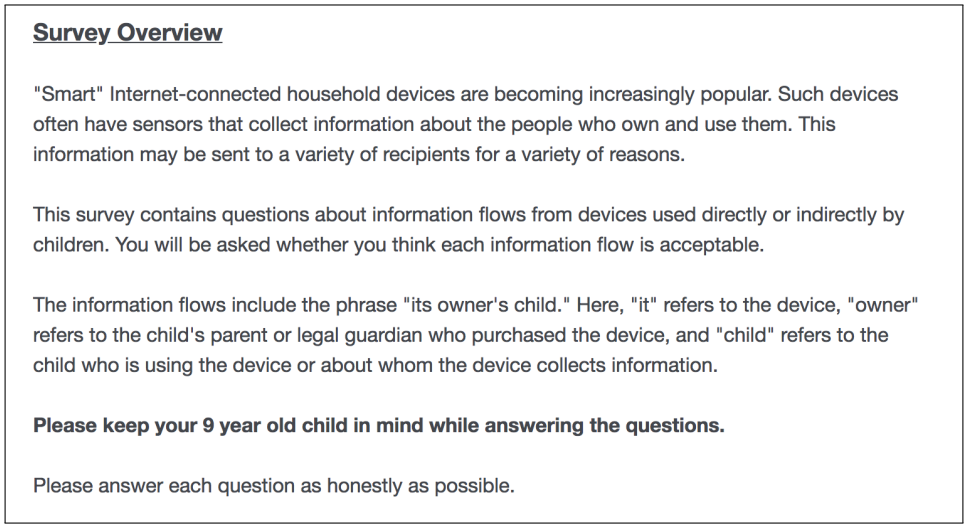
[57] ZHANG, Y. ., CHEN, J. Q., AND WEN, K.-W. Characteristics of internet users and their privacy concerns: A comparative study between China and the United States. *Journal of internet Commerce 1*, 2 (2002), 1–16.

[58] ZHENG, S., APTHORPE, N., CHETTY, M., AND FEAMSTER, N. User perceptions of smart home iot privacy. *In Proceedings of the ACM on Human-Computer Interaction, Computer-Supported Cooperative Work and Social Computing (CSCW)* (November 2018).

[59] ZIMMECK, S., WANG, Z., ZOU, L., IYENGAR, R., LIU, B., SCHAUB, F., WILSON, S., SADEH, N., BELLOVIN, S. M., AND REIDENBERG, J. Automated analysis of privacy requirements for mobile apps. In *24th Network & Distributed System Security Symposium (NDSS 2017), NDSS* (2017).

[60] ZYSKOWSKI, K., MORRIS, M. R., BIGHAM, J. P., GRAY, M. L., AND KANE, S. K. Accessible crowdwork?: Understanding the value in and challenge of microtask employment for people with disabilities. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing* (2015), ACM, pp. 1682–1693.

Appendix A: Survey Overview



Survey overview shown to participants before contextual integrity information flow questions. Participants are asked to keep one child in mind when answering the survey questions. The age of this child (9 in above example) is selected randomly for each participant from the self-reported ages of each of their children younger than 13.

Appendix B: Self-Reported Demographics and Technical Background of Survey Respondents

Metric	Sample	Metric	Sample
Female	61%	18-24 years old	3%
Male	39%	25-34 years old	31%
Other/Prefer not to disclose	-	35-44 years old	48%
		45-54 years old	13%
9th, 10th, 11th, 12th - no diploma	1%	55-64 years old	4%
High school graduate	14%	65 years or older	<1%
Some college but no degree	22%		
Associate degree in college - Vocational	6%	Has 1 child	33%
Associate degree in college - Academic	4%	Has 2 children	45%
Bachelor's degree	30%	Has 3 children	15%
Master's degree	16%	Has 4 or more children	7%
Professional school degree	2%		
Doctorate degree	5%	answers based on 0-3 yr old child	14%
		answers based on 4-7 yr old child	36%
Less than $25,000	13%	answers based on 8-12 yr old child	50%
Between $25,000 and $50,000	22%		
Between $50,000 and $75,000	21%	0-3 hours of internet use per day	15%
Between $75,000 and $100,000	21%	4-7 hours of internet use per day	45%
Between $100,000 and $200,000	17%	8-12 hours of internet use per day	25%
More than $200,000	4%	>12 hours of internet use per day	14%
Prefer not to disclose	2%		
		Uses a personal computer	97%
Asian	7%	Uses a smartphone	94%
Black or African American	11%	Uses a tablet device	78%
Native Hawaiian or Other Pacific Islander	1%		
White	76%	Owns a smart device*	49%
White, American Indian or Alaska Native	<1%	Does not own a smart device	50%
White, Asian	<1%	Unsure	<1%
White, Black or African American	1%		
Other	3%	Owns a children's smart device**	33%
		Does not own a children's smart device	66%
Hispanic	14%	Unsure	1%
Not Hispanic	85%		
Prefer not to disclose	<1%	Familiar with COPPA	63%
		Not familiar with COPPA	33%
		Maybe familiar with COPPA	4%

* Question text: "Do you own any 'smart' (Internet-connected) devices or appliances besides a smartphone, tablet, laptop, or desktop computer?"

** Question text: "Do you own any 'smart' (Internet-connected) devices or appliances used directly or indirectly by children besides a smartphone, tablet, laptop, or desktop computer?"

Secure Multi-User Content Sharing for Augmented Reality Applications

Kimberly Ruth
kcr32@cs.washington.edu

Tadayoshi Kohno
yoshi@cs.washington.edu

Franziska Roesner
franzi@cs.washington.edu

Paul G. Allen School of Computer Science & Engineering, University of Washington

https://ar-sec.cs.washington.edu/

Abstract

Augmented reality (AR), which overlays virtual content on top of the user's perception of the real world, has now begun to enter the consumer market. Besides smartphone platforms, early-stage head-mounted displays such as the Microsoft HoloLens are under active development. Many compelling uses of these technologies are multi-user: e.g., in-person collaborative tools, multiplayer gaming, and telepresence. While prior work on AR security and privacy has studied potential risks from AR applications, new risks will also arise among multiple human users. In this work, we explore the challenges that arise in designing secure and private content sharing for multi-user AR. We analyze representative application case studies and systematize design goals for security and functionality that a multi-user AR platform should support. We design an AR content sharing control module that achieves these goals and build a prototype implementation (ShareAR) for the HoloLens. This work builds foundations for secure and private multi-user AR interactions.

1 Introduction

Augmented reality (AR) technologies, which overlay digitally generated content on a user's view of the physical world, are now becoming commercially available. AR smartphone applications like Pokemon Go and Snapchat, as well as smartphone-based AR platforms from Apple [5], Facebook [6], and Google [4], are already available to billions of consumers. More sophisticated AR headsets are also available in developer or beta editions from companies like Magic Leap [37], Meta [41], and Microsoft [24]. The AR market is growing rapidly, with a market value projected to reach $90 billion by 2022 [15].

The power that AR technologies have to shape users' perceptions of reality — and integrate virtual objects with the physical world — also brings security and privacy risks and challenges. It is important to address these risks early, while AR is still under active development, to achieve more robust security and privacy than would be possible once systemic issues have become entrenched in mainstream technologies. The computer security and privacy community has already taken steps towards identifying and mitigating potential risks from malicious or buggy AR apps. These efforts — e.g., limiting untrusted apps' access to sensor data [28, 49, 54] or restricting the virtual content apps can display [32, 34] — are reminiscent of recent work on access control for untrusted apps on other platforms, such as smartphones [16, 53]. Despite this valuable initial progress, we observe a critical gap in prior work on security and privacy for AR: though past efforts are valuable for protecting *individual* users from untrusted *applications*, prior work has not considered how to address potentially undesirable interactions between *multiple human users* of an AR app or ecosystem.

The need to consider security for multi-user AR. Despite this gap in prior work, we observe that many compelling use cases for AR will involve multiple users, each with their own AR device, who may be physically co-located or collaborating remotely and who may be interacting with shared virtual objects: for instance, in-person collaborative tools [63], multi-player gaming [3], and telepresence [18]. As one concrete example already available to AR users, Ubiquity6 has released a beta version of its smartphone platform in which all users can view and interact with all AR content within the app [67], as shown in Figure 1.

In these contexts, the potential security, privacy, and safety risks for AR users come not only from the apps on their own devices but also from *other users*. For example, one user of a shared AR app might accidentally or intentionally spam other users with annoying or even disturbing virtual objects, or manipulate another person's virtual object (e.g., artwork) without permission. Indeed, even though multi-user AR technologies are not yet ubiquitous in the consumer market, precursors of such issues have already begun to emerge in the wild and in research settings today. In AR specifically, for example, there have been reports of "vandalism" of AR art in Snapchat [38], and a recent study found that pairs of AR users often positioned virtual objects in each other's personal space [35]. Similar findings have been made in virtual reality (VR), where users have blocked each other's vision



Figure 1: Sample screenshots from Ubiquity6 multi-user application (taken from [67]). Users can, for instance feed a virtual cat (left) or tend a virtual garden (right).

with virtual objects [66] and invaded each other's personal space [1]. In earlier work on digital tabletop displays, researchers observed conflicts between users closing or stealing each others' documents [44]. As a final example, Apple's AirDrop scheme for sharing files between physically co-located Apple devices has been misused to send inappropriate content to strangers in public spaces [11].

Thus, we can and should expect conflicts and tensions to arise between multiple AR users, which raises the critical question: how should AR platform and app designers handle these issues? Existing AR platforms provide limited or no support to app developers on this front. For example, though HoloLens supports running an app shared between multiple device users, it surfaces only basic cross-device messaging features, providing no framework for developers to reason about or support complex multi-user interactions.

This work: Sharing control for multi-user AR. In this work, we thus address the challenge of *providing secure and private content sharing capabilities for multi-user augmented reality applications*. Unlike prior AR security work that focused on protecting users from untrusted apps, we aim to limit undesirable interactions between mutually distrusting human users — similar to how traditional file system permissions attempt to separate mutually distrusting human users from each other. By addressing this issue before multi-user AR becomes widespread, we aim to inform the design of future AR systems, thereby preventing such multi-user concerns from manifesting broadly.

In our exploration of this space, however, we find that controlled sharing for AR content raises unique challenges not present in traditional settings such as file systems or shared online documents. The root cause of this complexity is AR's integration with the physical world. Because people share the same physical world, they may have certain expectations about how AR content is shared. Indeed, prior work has found that users often expect co-located users to see the same virtual content [35]. For example, a user might want to have control of their personal physical space, not allowing another user to place too many virtual objects near them. Fulfilling this request requires that either the second user is restricted in augmenting his or her own view of the world, or that the two users see different things. Diverging views of the world can violate expectations, however: consider watching or interacting with an AR object that only you can see while another AR user unknowingly steps into the space between you and your object.

AR's integration with the physical world further complicates many access control design decisions. Consider the seemingly simple notion of Alice sharing an AR object with Bob: for instance, giving him read access to a piece of virtual artwork. When this object is shared, does Bob see the artwork in the same place as Alice (e.g., on a particular wall), or does Bob see his own instance of the object in a different place? The answer may depend on the semantics of the app and whether Alice and Bob are physically co-located, and the answer interacts with many other design choices.

In our work, we thus explore a set of multi-user AR case study apps that represent different points in the possible design space (co-located and remote users, opt-in versus opt-out sharing) to surface functionality and security goals for an AR sharing control module. We then present the design of such a module, which we envision as an app-level library or OS interface that can be leveraged by AR application developers. This module supports app developers in (1) allowing users to *share* AR content with other (co-located or remote) users, (2) allowing users to *control* both inbound and outbound AR content, while (3) addressing fundamental challenges raised by AR's integration with the physical world.

One key challenge is to define and manage different ways that AR content might be mapped into the physical world — we do so by supporting both location-coupled objects (which all users see in the same physical place) and location-decoupled objects (for which users see their own copies in separate locations), and by managing the resulting impact of these types of objects on sharing and access control functionality. Another key challenge is to address potential discontinuities in user expectations around private content in a shared physical world — we do so by introducing "ghost" objects that allow users to share with other AR users *that* they are interacting with a virtual object without sharing sensitive details about that object. Finally, a third key challenge is to respect users' possible sense of ownership of physical space (e.g., personal space) — to that end, our design supports policies for how to handle AR objects that are within a protected region (e.g., making objects closer to a user more transparent to that user). Through our exploration, we find that no single sharing control model will work for all apps, but that our proposed module can provide key features to support app developers in creating multi-user AR apps that meet users' expectations.

Contributions. In summary, our contributions include:

1. We are the first to rigorously explore the design space for secure and private AR content sharing between users. Through an exploration of multi-user AR case study apps, we identify (in Section 2) key design goals, challenges, and features that app developers require to support secure and private multi-user AR experiences.
2. Building on our design space exploration (Section 2), we present the design (in Section 4) of a multi-user AR sharing control module. Our design addresses key challenges and enables app developers to meet our design goals: supporting users in controlling how they share AR content with others and how AR content is shared with them, while taking into account the ways in which AR content might integrate with the physical world.
3. We provide a concrete prototype implementation (ShareAR, in Section 5) and evaluation (in Section 6), iteratively refining our design and demonstrating its feasibility in practice. Our source code will be made available at the project website.[1]

This work lays a foundation for future secure and private multi-user AR apps. Mitigating undesirable interactions between users can facilitate user adoption of AR and help the technology reach its full potential.

2 Problem Formulation and Design Goals

We begin by formulating, for the first time, the problem space and goals for secure and private multi-user AR content sharing. To do so systematically, we consider four case study apps (Section 2.1) that we selected to explore unique points in the multi-user AR design space and that we envisioned might exercise a broad range of functionality and security needs. From these case studies, we then derive our security and functionality goals (Sections 2.2 and 2.3).

In exploring possible apps, we observe that the key aspect of AR that differentiates it from previous technologies is its tight *physical-world integration*: virtual content appears to the user to be situated in 3D space among physical objects (e.g., the examples in Figure 1). Thus, one key axis is (1) co-location: are the users sharing virtual content co-located or not? A second key axis is (2) opt-in versus opt-out sharing: is sharing a deliberate opt-in action between specific people (as the HoloLens developer guidelines prioritize [43]) or are virtual objects public by default, requiring a deliberate opt-out (as the Meta developer guidelines advocate [40])? The example case studies we highlight explore these dimensions.

2.1 Case Study Applications

Paintball: Co-located, opt-in. In this app, users in the same physical space can play a game of paintball with virtual paint. All users can see the game objects (weapons, paint, etc.). Users may also have a dashboard where they can see the game status. This type of AR multiplayer gaming is already emerging in smartphone apps [20].

Multi-Team Whiteboards: Not (necessarily) co-located, opt-in. We envision a collaborative AR whiteboard app in which a user, possibly in a co-located group, may choose to share a whiteboard with other users who may be in the same or different physical locations. Although each co-located group of users sees the same whiteboard in the same location, different groups may see the whiteboard instantiated in different locations; furthermore, a user in a group may split off an individual copy of the whiteboard in order to leave the room and still collaborate from another remote space. The contents of all users' copies are synchronized in real-time. Since different whiteboards may have different levels of sensitivity, access control must be at least at whiteboard-level granularity. Unlike in *Paintball*, where a shared game state is core to app function, users of this app may encounter users with whom they don't want to share a sensitive whiteboard. This case study, also, is grounded in existing work: a pending patent application by Apple [29] describes a GUI for AR document editing, though it does not mention access control.

Community Art: Co-located, opt-out. We now consider an example in which co-located users automatically see each other's objects by default. We consider a virtual art app, where users can create and view sculptures, free-drawn markup, and other artistic artifacts made by other, potentially unknown AR users in the same physical (and virtual) space. Variants of *Community Art* might be used to decorate for a celebration so that guests or passersby will see the content, or to place advertisements outside one's shop. Though we consider *Community Art* as an example of a public-by-default app, some use cases may necessitate more fine-grained access control. For instance, artists may choose to keep their art private while constructing it or allow the public to view but not edit their sculptures. This case study is similar to Ubiquity6's smartphone app [67], in which all content is public.

Soccer Arena: Not co-located, opt-out. Finally, we consider an app that lets the user watch a virtual replica — e.g., on the user's living room table — of the soccer game that it is currently broadcasting. By default, all users of this app see all aspects of the playing field, commentator annotations, and ads. Some users may watch the game together in the same physical space, while others may be in separate physical spaces. While using the app, a user may wish to block a distracting ad or turn off annotations. The ability to form AR reconstructions of soccer games from monocular video footage, demonstrated in [51], shows that this app is within reach of today's technology. We find that *Soccer Arena* does not surface new security, privacy, or functionality requirements not covered by the other case studies. In particular, it raises the same spam-related concerns as *Community Art* does and the same non-colocation challenges as *Multi-Team Whiteboards* does. However, we include it for completeness.

[1] `arsharing.cs.washington.edu` or `arsharingtoolkit.com`

2.2 Functionality Goals

From the above case studies, we now derive a set of functionality design goals for multi-user AR apps and platforms. Any sharing control solution must coexist with these functionality goals — while one could trivially meet the security and privacy goals outlined in the next section by allowing *no* shared content, supporting sharing functionality is critical to the success of emerging multi-user apps.

- **Support physically-situated sharing.** For both *Paintball* and *Community Art*, physically co-located users will want to see the same virtual objects. The multi-user AR platform must support a way of sharing virtual state, and a mapping between virtual objects and the physical world, among the collaborating users.
- **Support physically-decoupled sharing.** *Multi-Team Whiteboards* requires that AR content be synchronized for each person's copy, regardless of the users' relative location — when they're in the same room, or adjacent rooms, or halfway across the world. Thus, the platform must support sharing virtual content decoupled from the physical world as well.
- **Support real-time sharing.** Users of *Paintball* will expect for their interactions with other players to occur in real time. Real-time state changes are also desirable for the other case studies. Thus, the platform must support low latency updates of shared state among multiple users, and any sharing control solution should not impose an undue performance burden. (Note that real-time performance also confers a security benefit, since access control changes can propagate quickly.)

2.3 Security Goals

A trivial solution that provides all of the above functionality would make all AR content public by default. However, interaction between multiple users may not always be positive. For example, a malicious or careless user may attempt to:

1. *Share unwanted or undesirable AR content* with another user. For example, in *Multi-Team Whiteboards*, a user may plaster a wall with offensive messages, or in *Community Art*, violate another user's personal space by attaching virtual objects to them as a practical joke. Such behavior has already manifested in shared VR settings [1, 66].
2. *See private AR content* belonging to another user. For example, in *Multi-Team Whiteboards*, a user may attempt to read another user's private whiteboard.
3. *Perform unwanted manipulations on AR content* created by or belonging to another user. For example, in *Community Art* or *Multi-Team Whiteboards*, a user may delete or vandalize another user's virtual creations. Such behavior has already appeared in the wild, with vandalism of AR art in Snapchat [38].

In response to such multi-user threats, we develop the following security and privacy goals for an AR sharing module.

Control of outbound content. Sharing of AR content involves two parties: the originator of the content and the recipient. We decompose our security goals along this dimension, beginning with control of outbound content, i.e., managing the permissions of other users to access shared content.

Three canonical access control rights are "read," "write," and "execute." Extending "read" and "write" to the AR domain (and deferring "execute" to Section 7):

- **Support granting/revoking per-user permissions.** The multi-user AR platform should support setting edit and view permissions for different users. A user of *Paintball* may wish to share a game session only among a specified friend group instead of allowing any nearby user to join, and may wish to retain full control of game administration even among the set of players.
- **Support granting/revoking per-object permissions.** A user of *Community Art* may wish to leave one piece of art publicly visible while working privately on another. Thus, regulating permissions at the granularity of the app is not sufficient to cover all use cases; object-level permissions must be supported as well.

The consequence of the above goals is that users in the same physical space may not share the same view of the virtual space. This is in sharp contrast to current technologies, where the physical presence of a device — e.g., a smartphone or a television set — enables the user of that device both (1) to signal to others that they are busy with that device, and (2) to establish a dedicated spatial region upon which their use of the device depends. The physicality of the device, then, serves as a scaffold around which interpersonal norms have developed. For instance, a person might avoid standing in front of a television set when a user is watching it, and might refrain from blocking the line of sight from a user to the smartphone they are holding.

AR content has no such physicality. Consider, for instance, *Multi-Team Whiteboards*: as thus far stated, a user looking at or interacting with a private whiteboard will appear to a nearby user as staring into or manipulating empty space. There is no mechanism by which the user can exercise agency over their working space in the presence of other users, and no mechanism by which other users passing by can determine whether the object is five feet away from its owner or up against a more distant wall. As a result, one user may inadvertently stand in front of content that a second user is interacting with. Further adding to this issue, prior work has also shown that people can be uncomfortable with the presence of AR users due to not knowing what the AR user is doing [14,35], and private content causes this rift even between users of the same AR platform. The Meta developer guidelines [40] thus recommend that developers build public-by-default content in accordance with human intuition about a shared physical world. Indeed, novice users in the same *physical* space may expect to also see the same *virtual* content [35]. It is possible that social behaviors

will adapt to this physicality disconnect over time, particularly around the current social discomfort of bystanders. But although social norms may change, and although mitigating these issues for bystanders — non-AR users, or AR users of a different and non-compatible platform — is difficult and beyond the scope of this work, we still seek to address this physical-world disconnect at least in the near term for multiple AR users of compatible platforms. Specifically, we wish to achieve the above content privacy goals while at least partially supporting a shared-world physical intuition:

- **Support physically intuitive access control.** An app may wish to signal to a nearby user that another user is (for example) drawing on a whiteboard, without revealing the content being drawn.

Control of inbound content. We next consider security properties from the perspective of the recipients of shared content. Since shared content can have serious implications for the receiver, such as spam that obscures important real-world information [34], we derive the following goals:

- **Support user control of incoming virtual content.** For instance, users of *Community Art* may wish to filter content to only that which is age-appropriate or that does not contain foul language.
- **Support user control of owned physical space.** In the case of *Community Art*, a user may not want arbitrary other users to attach content to their heads without consent, a homeowner may wish to prevent house guests from placing virtual content inside private rooms, and the keepers of a public monument may not want the monument to be vandalized with virtual graffiti. We note that users may want control over their physical space even when they cannot see the object in question: for instance, an app may wish to prevent a virtual "kick me" sign from being attached to a user's back such that the user cannot see and cannot control the sign. We consider the question of determining who controls a particular physical space to be out of scope for the design we present in Section 4 (see Section 4.4 for further discussion), and instead focus on enforcing owned physical space; however, we urge future work to also address this complementary issue.

2.4 Supporting Flexibility

Stepping back, in defining the above functionality and security goals, we observe that not all multi-user AR apps will have the same needs. For example, AR content that is shared with all users by default is suitable for some apps (e.g., *Community Art*) but not others (e.g., *Multi-Team Whiteboards*). Likewise, not all security and privacy goals are relevant in all cases: for instance, enforcing personal space for shared AR content may conflict with the functionality needs of *Paintball*, which requires that virtual paint stick to players upon a hit. Even in an app that is otherwise simple from a sharing control perspective, user needs may warrant a degree of added sharing control complexity: for instance, an AR assistive technology object that transcribes spoken words for deaf users may be exempt from the app's general rules for the enforcement of owned physical space so that it always remains visible to the deaf user who needs it.

Because the right sharing control model is app-specific, AR app developers will need the ability to implement multi-user features with their chosen model. To that end, we identify the need for a flexible AR sharing control module that can be leveraged by app developers. We envision this module as either an app-level library or an OS interface (i.e., set of APIs) that provides sharing control features. The advantage of an app-level library is that it does not require explicit changes to the platform. That is, an app developer could create an app that runs on different AR platforms and, by including the platform-appropriate version of a library, support interactions among users with different types of AR devices. For example, although we prototype our design as an app-level library for HoloLens, in principle it could be adapted for compatibility with Meta or Magic Leap apps.

3 Threat Model and Non-Goals

We aim to design a flexible module that helps app developers create multi-user AR apps that incorporate shared AR content while mitigating the risk of undesirable interactions between multiple users. We focus on the case of a developer building a single app and mediating the interactions of its multiple users, deferring to future work the problem of cross-app communication. We now present the threat model under which we develop our design in Section 4, as well as specify non-goals of this work.

Threat Model. Our primary focus in this work is on *untrustworthy users*. That is, we aim to help app developers create multi-user AR apps that are resilient to security and privacy threats between multiple users of the *same* app. In that context, we assume that two or more users are using the same AR app, written by the same developer and incorporating our sharing module. We thus assume that users trust both the developers of the apps that they install as well as their AR operating system, but that users may *not* trust each other. This trust dynamic is akin to traditional desktop environments — e.g., where two users of a Unix operating system might both trust the underlying system and installed apps, but might not trust each other and hence might not make their home directories world readable or world writable. A key difference, as noted earlier, is that in our model we only consider sharing of content between users of the same app.

Under this threat model, we do not consider malicious apps that omit or misuse our sharing module. We explicitly trust app developers to incorporate our module (e.g., as an app-level library) into their apps; a malicious app developer might choose to simply not use our sharing module, implementing their own adversarially-motivated sharing functionality, or use our module but violate security or pri-

vacy properties through out-of-band means. Though a user may install malicious apps alongside legitimate ones that use our module, these malicious apps cannot interfere via our module: we consider (and our module supports) AR content sharing only among multiple users of the *same* app, rather than also considering sharing *across* apps. This is consistent with the capabilities of current AR technologies, which are either single-app or do not allow multiple concurrently running apps to communicate [33]. We also assume that all users are running legitimate, uncompromised versions of the app; strategies for verifying [36,75] or enforcing [73,74] this assumption are significant research challenges of their own.

Finally, we assume that communication between devices is secured with today's best practices, e.g., end-to-end encrypted. Thus, we rule content eavesdropping and content modification attacks as out of scope. Current network best practices still suffer from denial-of-service attacks and traffic analysis attacks, but we do not aim to protect against such attacks in this work, focusing instead on the app-level security and privacy issues.

Non-Goals. We consider the following design questions to be non-goals of our present effort:

- **Non-goal: UI/UX design.** Although we propose underlying mechanisms for the sharing control needs of app logic, and although those mechanisms in some instances have implications for what developers are empowered to surface at the UI level, we do not aim to define exactly how those mechanisms should manifest to users in the specific interaction modality or look-and-feel of an app. Thus, we leave the design of specific interfaces — including how much of our module's control should be surfaced directly to users versus shouldered by the app — to future efforts by researchers and app developers. Our work is similar in spiritspirit to work on user interface toolkits (e.g., [25, 27]) in that our goal is to enable app developers to easily create and innovate on a range of user interfaces and experiences, rather than to design and iterate on these interfaces directly.
- **Non-goal: Network architecture design.** It remains an open question whether multi-user AR will ultimately be enabled by client-server, peer-to-peer, or other network architectures; we thus design our platform to be agnostic to network architecture. Additionally, we do not consider how two AR devices initially bootstrap communication; prior, complementary work considers how to securely pair two HoloLens devices [60].
- **Non-goal: App-specific choices about sharing control properties.** We do not aim to recommend to apps which sharing control properties and functionalities might make sense in the context of the app, instead enabling app developers to choose the appropriate subset of properties for their specific use cases.

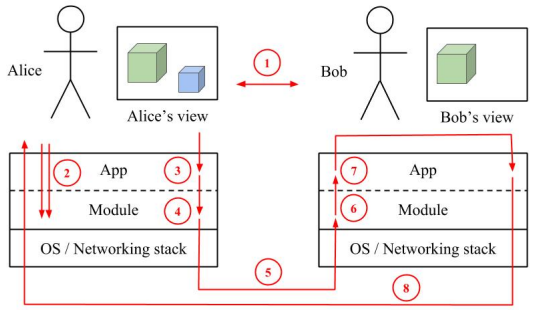


Figure 2: Basic object sharing flow: Alice creates the blue and green boxes and then chooses to share the green box with Bob. See Section 4.1 for details.

- **Non-goal: Accurate spatial localization of AR users and content.** We do not aim to design a system by which the location of an AR user can be accurately and securely determined. Prior work has studied how to localize devices accurately [23, 31], how to verify location claims [10, 69], and how to verify co-location claims [21, 55]. We note that even without further sharing controls, future location-based AR apps will benefit from secure location and co-location verification methods. Thus, we consider this topic to be orthogonal and of independent in.

4 Design

We now present the design of a module that AR developers can use to support secure and private sharing of AR content among multiple users. Compatible with our threat model of untrusted users but trusted developers, we envision this module as an app-level library or an OS interface.

4.1 Module Design Overview

To illustrate the relationships between the OS, the sharing control module, the app, and multiple users, we begin by walking through a simple case of Alice creating two objects and sharing one with Bob (Figure 2).

1. *Precondition:* Alice and Bob are both running an app that incorporates the sharing module and, as such, already have an open communications channel between their devices.
2. *Object creation:* Alice creates two AR objects, a small blue box and a large green box. Her app calls the sharing module's `InstantiateShared()` API for both objects, allowing the module to track permissions at the granularity of those objects (in this case, beginning with view and edit permissions for only Alice).
3. *Outbound sharing (app-level):* Through some user interface provided by the app, Alice chooses to share the green box with Bob.
4. *Outbound sharing (module-level):* On Alice's device, the app calls the sharing module's `SetPermission()`

API. As a result, the module updates its internal permission map, adding Bob to the list of users with view permissions for the green box.

5. *Communication:* The sharing module sends a message (via the device's OS and networking stack) containing object content and metadata to Bob's device, whose OS and networking stack dispatch it to the sharing module in Bob's instance of the app.
6. *Inbound sharing (module-level):* The sharing module surfaces a `SetPermission` event to Bob's app.
7. *Inbound sharing (app-level):* On Bob's device, the app shows some user interface to allow Bob to accept or deny the shared object. (Other apps may skip this step and show the object to Bob automatically, and/or respect Bob's previously-set preferences for shared objects from Alice.) Bob chooses to accept the shared object from Alice; the app updates his view of the world to include the green box.
8. *State update and communication:* The app calls the sharing module's `AcceptObject()` API, which in turn transmits that message back to Alice's device.

Following this transaction, Bob can now see a shared copy of Alice's green box and, depending on the sharing settings, can manipulate that box in ways that are also visible to Alice.

This sharing flow might seem simple: the sharing control module provides APIs that help an app keep track of which users can access which AR objects — i.e., view and edit permissions — and syncs this information across the devices of all users of the app. However, as surfaced in Section 2, sharing in the AR context requires thoughtful consideration — particularly in the face of users' expectations of and interactions within the physical world.

Key design challenges. While striving to achieve the functionality and security goals identified in Section 2, our design space exploration surfaced several key questions which do not arise for sharing and access control in traditional systems (e.g., file systems). These challenges are deeply connected with AR's integration with the physical world, and although they do not on the surface appear to be security-centered questions, they affect the security and privacy mechanisms we design, and so we must address them:

- *Integration of shared AR objects with the physical world (Section 4.2):* How is a shared object integrated into the physical world? In the above example, do Alice and Bob see the green box in the same physical location or in different physical locations? Are Alice and Bob themselves in the same physical location, and what happens when their co-location status changes?
- *Private content in a shared physical world (Section 4.3):* How should the sharing module handle or help shape users' expectations of private AR content, such as Alice's blue box, when they interact in a shared physical world?

	Outbound sharing controls	**Inbound sharing controls**
What and with whom	Permission management	Two-party sharing consent
Where	Location coupling (§4.2)	Personal space (§4.4)
How much	Ghosting (§4.3)	Clutter management

Table 1: Summary of the components of our design for controlling the outbound and inbound sharing of AR content.

- *Ownership of physical-world spaces (Section 4.4):* How can a sharing module help apps respect people's existing ownership of physical spaces? For example, users may wish to control AR content that they or others see in front of their homes or on their own bodies.

Effective solutions to these challenges must integrate with the system design components that have more direct analogues in current technologies. In particular, we incorporate the following established control structures into our design:

- *Permission management.* We leverage classic access control work [30] to track and enforce per-object and per-user permissions. Although we aim to be compatible with whichever access control model a particular app chooses to layer atop our module — e.g., a model akin to Google Docs for the *Multi-Team Whiteboards* case study — we note that this alone is not enough to support the 3D experience of AR, and that the above key design challenges must also be addressed.
- *Two-party sharing consent.* Some existing sharing models require that both the sharer and the receiver of digital content authorize a sharing event before its completion (e.g., Google Drive, Apple AirDrop). We use this principle in our design, with one twist: to help developers avoid decision fatigue in apps with high-volume content sharing, we allow the app to authorize a sharing event without the user in the loop. For instance, an app might automatically authorize content under some contexts but not others [70], use a notification UI that minimally disrupts the user's workflow, or allow users to always trust content from a specific other user. We advise developers to be conscious of habituation and interruption in their app designs.
- *Clutter management.* Our design supports temporarily or permanently removing an object from the user's field of view, as we discuss further in Section 4.2.

We summarize these aspects of sharing control, both new and precedented, in Table 1. We categorize the design points along two axes: (1) where in the above sharing flow the control occurs (*outbound* on the sharer's end, or *inbound* on the receiver's end), and (2) what type of control is enforced (*what* object is shared and *with whom*, *where* a shared object can be, or *how much* information from that object is shared).

4.2 Physical World Integration

The sharing flow in Section 4.1 demonstrates the basic building blocks of a sharing module, with view and edit permis-

sions for users at the granularity of AR objects (e.g., a virtual cat or virtual browser window). We now explore how these notions become significantly more complicated when shared AR objects are integrated into the physical world.

Location-coupled and -decoupled sharing. Recall from Section 2 that we aim to support both physically co-located sharing (i.e., two users in the same physical place and seeing the same AR objects in the same physical locations) and remote sharing (i.e., two users physically separated but seeing the same AR objects in their own physical spaces).

Accordingly, our design supports two notions for how an AR object can be shared with respect to the physical world: (1) *Location-coupled* objects, which all users see in the same physical location, and (2) *Location-decoupled* objects, where all users see the same object but in different physical locations. In the coupled case, if one user moves the object, other users also see the object's location update; in the decoupled case, the two instantiations of the object can be moved independently.

We intend for these notions not to be mutually exclusive for an object but rather to apply between sets of users. For example, an AR object (say, a virtual whiteboard) may be shared (1) in a location-coupled way between Alice and Bob co-located in New York, and (2) in a location-coupled way between Guanyou and Huijia in Beijing, and simultaneously (3) in a location-decoupled way between the two groups.

Handling people moving around the physical world. A challenge for location coupling and decoupling of shared objects arises when we consider that users' co-location can change as they move around the physical world.

For example, suppose that Alice and Bob share a location-coupled AR object — say, a whiteboard — both seeing it in the same physical location. Alice may *also* share it in a location-decoupled way with collaborator Carol working in another room — i.e., Carol will see an instantiation of the whiteboard in her own physical space. Initially, this appears to meet our goals: all users see the whiteboard in their own physical space in the same location as other co-located users.

What happens, however, when Carol moves into the same physical space as Alice and Bob? Since the whiteboard is shared among all of them, they will likely assume that all three of them can now see the same AR whiteboard object on the same wall [35]. This is not the case, however: Alice and Bob see one instantiation of the whiteboard, and Carol sees a separate instantiation in a slightly different location.

To resolve this potential inconsistency, our design keeps track of all copies of a shared object, allowing the app to show *all* of these copies to all users. Thus, when Carol joins Alice and Bob in the same room, all users see *both* versions of the whiteboard. All users then share the same view of the augmented physical world. Note that this location information may have privacy implications, though none in scope for this work; we discuss this point further in Section 7.

Moreover, since users' co-location may change over time, their desired location-(de)coupling of objects may change over time too. For example, in the above scenario, Alice and Bob may wish to merge their whiteboard object with Carol's instantiation, so that all three indeed see the same, single whiteboard object in the same place. Conversely, users may wish to collaborate on a location-coupled object while they are physically co-located but to both take their work with them when they physically separate. Thus, we also provide mechanisms to *merge* two location-decoupled instances into a location-coupled object and to *separate* a location-coupled object into two location-decoupled instances.

Another way to think about shared AR objects, then, is that there is one conceptual object and potentially multiple views of it. Each user sees a view in the same location as do all other users, and users in different physical spaces will see different views. If a user is in the same space as multiple views, that user will see all present views. The object's views may be manipulated separately in space; a single view may be split into two, or two may be merged into one, but the underlying object that all views represent remains the same.

Implications of location coupling for object deletion. A shared object's location coupling or decoupling has design implications for other features as well. For example, our design lets users delete AR content that they have created; it is not clear, however, that this decision should propagate to other users with whom the object has been shared, and location coupling or decoupling affects how deletion is handled.

We design the module to support three cases, which can be chosen by the app developer as appropriate:

1. *Case 1: Local Deletion: Affect user's local view of object only.* This option allows Alice to delete her object without affecting other users. If, in *Multi-Team Whiteboards*, Alice and Bob share a whiteboard in a location-coupled manner, Alice can delete her instance of the object while Bob keeps working.
2. *Case 2: Global Location-Coupled Deletion: Affect all users' views of location-coupled object.* Here, a deleted object is also deleted for all other users with whom that object was shared in a location-coupled way. That is, when Alice deletes her document, Bob also sees that document disappear. However, if Alice has also shared the document in a location-decoupled way with remote collaborator Carol, this option allows Alice to delete her and Bob's location-coupled instantiation without affecting Carol's remote, location-decoupled instantiation.
3. *Case 3: Global, Location-Independent Deletion: Affect all users' view of object, independent of location coupling.* In this case, *all* instantiations of the object for all users are deleted. Continuing the previous example, Alice, Bob, *and* Carol will all see the object deleted.

Which of these cases is most appropriate depends on the semantics of an app and each use case within that app.

Location coupling and decoupling have other design and implementation implications as well. For instance, hiding content with which the user does not currently wish to interact requires considering the same set of options as deletion.

4.3 Private Content in Shared Physical World

As raised in the *Multi-Team Whiteboards* case study in Section 2 and in prior work [35], the fact that AR supports per-user private content can have benefits, but it can also fail to provide a signal about the use of physical space (e.g., leading to one user inadvertently standing in front of another's virtual content, or causing social tension due to one user not knowing what another user is doing).

Thus, in this section we propose a design that allows users to socially signal to other AR users that they are busy interacting with private content without sharing the details of their activities. Further, we aim for this design to align with users' intuitions about a shared physical world.

Strawman designs. Consider two incomplete solutions:
Status quo. A solution with no further intervention would cause private content to be completely invisible to other AR users. A user interacting with private content thus appears to others as if the user were staring off into and manipulating an undefined region of empty space, giving no cue to other users about how far away the object is if they want to walk around it as well as no sense for what the user might be doing.
Occlusion by virtual barrier. Meta's developer guidelines recommend that sensitive content be shared publicly but occluded by a virtual barrier such as a curtain [40]. Although this does provide a shared-world physical intuition and social cue, it is not a robust privacy-preserving mechanism. Consider a user who places a virtual curtain around sensitive content so that the content is visible only from the user's point of view. A curious other user can surreptitiously look over the user's shoulder and observe the sensitive content, similar to shoulder-surfing with current mobile devices [56, 64].

Our approach: "Ghosts". We propose an alternate design that achieves our goal while avoiding the above drawbacks. The key idea is to allow users to share *that* they are interacting with an AR object, without sharing the *details* of that object. This idea is analogous to how a user interacting with a smartphone implicitly signals to bystanders that they are engaged in another activity located in the palm of their hand, without the contents of that activity being directly revealed, or to how users may share free/busy information about specific time blocks on their calendar to avoid double-booking without sharing the details of their calendar events.

To support this interaction, we introduce a new partial-visibility state for shared AR objects that we call *ghosting*. Ghost objects show only their location in space, not the sensitive content they contain, no matter at which angle they are viewed. As such, unlike the above smartphone analogy, they are not susceptible to shoulder-surfing. Furthermore, a user with whom a ghost object is shared receives from the sharer only the data needed to instantiate the ghost, rather than the full object data; this further insulates the private content.

Ghost shape granularity. For non-planar objects, we encounter the following question: How does the sharing module determine the appropriate level of granularity to expose in the ghosted object, given that object shapes may contain app-specific information content?

For instance, in *Community Art*, the ghost of a sculpture that is private during its development should not mimic the original sculpture's shape too closely. However, a shape with too coarse of a granularity — e.g., a large fixed-size cube regardless of the sculpture — no longer gives meaningful physical-world information to nearby users: e.g., by marking an unreasonably large physical space as occupied.

To balance this dilemma,we allow app developers to specify what the ghosted view of an object should look like. This approach allows for non-planar objects to assume an obscured shape appropriate for the app-specific information they carry. For instance, in the *Community Art* case study, a sculpture that is private during its development might be displayed to others as an appropriately sized cylinder.

4.4 Respecting Ownership of Physical Spaces

Finally, we turn to the question of how the sharing control module can help apps respect people's existing ownership of physical-world space. We refer to both personal space (near one's body) and static owned space (e.g., one's home) as *owned physical space* in this section.

Helping users protect their owned physical spaces requires several components: (1) Determining who owns a region in space, (2) Determining what the boundaries of that region are, and (3) Enforcing some kind of policy on shared AR objects in that region.

We defer to future work (1), how to determine *who owns* a region in space. This in itself is complex; for instance, Google has analyzed abuse of location ownership in its Maps app [26], and prior work has considered physical world ownership for restricting continuous sensing apps (including AR) [54]. Accounting for different types of space ownership is also nontrivial: in particular, we identify (a) fixed-location physical spaces (e.g., a house, a room, a storefront, or a public park), (b) person-relative spaces (e.g., within 5 feet of a user), and (c) object-relative spaces (e.g., within 35 feet of a virtual art object). A complete solution to this issue should also consider non-AR users, and we offer the following suggestion as a starting point for future work: locally on the AR user's device, employ computer vision techniques to identify the spatial positions of bystanders visible by the AR user, estimate the rough pose for each bystander, and use that information to mark bystanders' forms as protected regions of space (e.g., using techniques from [2, 39]). However, we do not pursue this topic further in this work, since it is a complex area of investigation in its own right; instead, we assume a prior definition of owned physical space, and we focus on

the challenges of enforcing that space.

For (2), we observe that defining the boundaries of a protected region of physical space — e.g., around a person or house — is not a simple binary determination. A user might perceive an object two feet away to be too close, for instance, but may consider an object nine inches away to be even more so; this definition may also vary across different users [22]. Building on this observation, our solution is to model owned physical space as a *continuum*, where violations become more severe — and thus policies could become stricter — as virtual content approaches the protected region.

The key question, then, is (3): what should an app do when one user's shared AR content overlaps another user's personal physical space, or a physical region (e.g., a house) that another user owns?

Policies for AR content violating owned spaces. To answer this question, we propose that the sharing control module can provide a variety of policies that an app's developer could choose to apply in such a case. These policies can be enforced either such that the result of enforcement is only visible to the user whose personal space is in question, or such that *all* users with access to the shared object see the result of enforcement. App developers may choose which policies to enable based on the needs of the app.

A simple, binary policy might make objects *invisible inside a fixed radius*: for instance, define a three-foot radius around a person within which AR content should not be visible (at least to that user). Such a policy can be exploited by surrounding the person with AR content just outside the boundary; thus, there is a tension between a large radius to minimize the effects of such an attack and a small radius to enable legitimate functionality.

To help balance this tension, and to take advantage of the continuum in the boundary described above, our design provides a *transparency gradient* policy, by which the module makes shared objects more transparent the closer the objects are to the boundary of any protected region (e.g., around a person). Under this policy, objects would start becoming transparent much farther than three feet away, avoiding blocking the person's vision but still being useful.

By having owned-space policies be enforced by apps themselves (via the sharing module), rather than by users, they can be applied without changing the underlying permissions on the shared object. This avoids two pitfalls. First, it prevents malicious users from exploiting the policy, e.g., by gaining control of an object simply by walking up to it. Second, it enables policy enforcement even on objects that the physical space owner cannot see (protecting even non-AR user bystanders from the "kick me" sign from Section 2).

5 Implementation

We now describe our prototype, which we implement as an app-level library for the Microsoft HoloLens, demonstrating the feasibility of our design for a currently available head-mounted AR platform. Our prototype, called ShareAR, is implemented in C# and uses the HoloLens Unity development kit. We implemented the concept of an AR object using the Unity GameObject primitive, which is a virtual entity comprising shape, texture, location, physics properties, script-controlled behavior, etc. Our implementation consists of a core module (1888 lines of code), a network shim layer (1137 lines of code), and a short supplementary script to accompany any object shared using the toolkit (45 lines of code), totalling 3070 lines of code.[2]

The ShareAR core comprises:

- Data and meta-data, including an access control matrix [30] and options for how objects are shared (e.g., location coupled or decoupled).
- Methods to instantiate objects, manually or automatically accept shared objects, change permissions on objects, and sync data between users. Table 2 summarizes sample corresponding message types in our prototype.
- Simple fixed-radius personal space controls in the form of Unity's *plane clipping*, where the portion of an object closer to the user than the fixed plane-clipping distance is not rendered. We did not implement more nuanced controls, like our proposed transparency gradient.

Though network architecture is out of scope of our design, in practice we must choose some way to connect between HoloLens devices. In our prototype, we used the MixedRealityToolkit Sharing toolkit, an open-source library from Microsoft.[3] MixedRealityToolkit does not provide any sharing control or access control functionality; we use it only as a basic tunnel to send messages between HoloLens devices.

We build a network shim layer that serializes and deserializes ShareAR messages and uses MixedRealityToolkit Sharing to send them between devices. A developer who wishes to use a different networking solution — e.g., one relying further on a central server for data storage, or one implementing a more rigorous consensus protocol — may write a replacement network shim layer satisfying the same interface with the ShareAR core.

Users may join, leave, and re-join the network. To be robust to access control changes occurring while a user is offline, we include in our implementation a means for a newly reconnected user to receive a "digest" version of an object containing only the information needed for consistency with the other online users. Since consensus is best done with network architecture in mind, we provide a means to create this object "digest" as a higher-level functionality but relegate consensus operations to the networking shim layer.

[2] To calculate lines of code, we use the CLoC tool version 1.80 available at `https://github.com/AlDanial/cloc/releases/tag/v1.80`. We omit lines of code solely related to our performance evaluation.

[3] `https://github.com/Microsoft/MixedRealityToolkit-Unity`

Message name	Sent when	Bytes
`InstantiateShared`	A new shared object is created	104
`AcceptObj`	A newly-shared object is accepted	22
`SetPermissionNew`	A newly-instantated public object is accepted and there are more than 2 users present	38
`SetPermissionObject`	A permission change is made or offered on an existing object	92
`SetPermission`	A permission change on an existing object is accepted and there are more than 2 users present	54
`UpdateLocation`	A shared object's location in space is updated	62
`DeleteShared`	A shared object is deleted	22

Table 2: Example message types and sizes in our prototype. Messages are relatively small because they do not include full AR object meshes but rather an ID corresponding to the type of object in question and a string of object data that fully specifies the particular object of that type. Sizes are for basic objects with no additional object data.

Feature	*Paintball*	*Cubist Art*	*Doc Edit*
Location-coupled sharing	✓	✓	
Location-decoupled sharing			✓
Public permission settings	✓	✓	✓
Ghost-only permission settings			✓
Private permission settings		✓	✓
Auto-accepting content	✓	✓	
Accepting content ad hoc			✓
Local deletion		✓	✓
Global location-coupled deletion		✓	
Global location-indep. deletion			✓
Updating object location		✓	✓
Updating object data		✓	✓

Table 3: ShareAR sharing control features in case study apps.

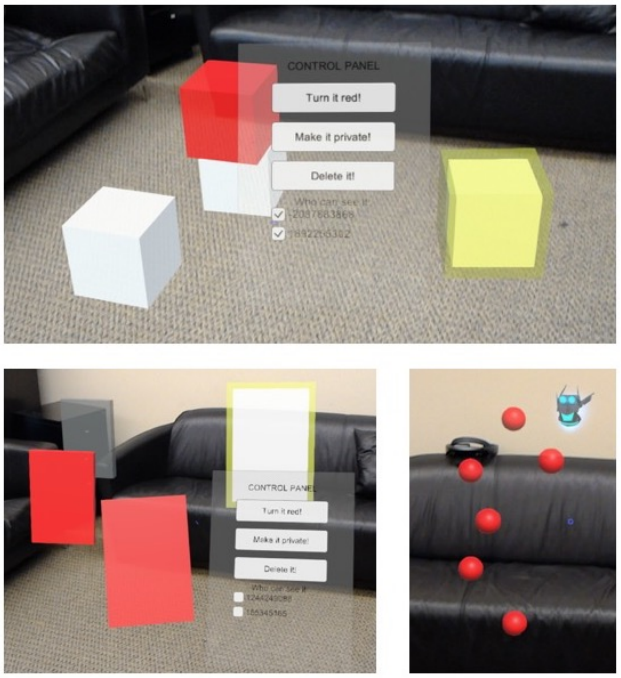


Figure 3: Screenshots of prototype apps: Cubist Art (top), Doc Edit (bottom left), and Paintball (bottom right). In the Doc Edit app, the semitransparent gray box in front of the file cabinet in the upper left is a ghost view of another user's document, and the two red boxes are two users' separate instantiations of the same shared document.

6 Evaluation

We now evaluate our prototype's functionality (Section 6.1), security (Section 6.2), and performance (Section 6.3).

6.1 Functionality Evaluation

We evaluate the functionality of our prototype by implementing case study apps and by comparing against existing AR design guidelines. We find that our prototype is flexible enough to support a range of app sharing control needs and is compatible with all considered existing design guidelines.

Case study applications. To evaluate the flexibility of our design to support our functionality goals, and the associated developer effort, we built bare-bones prototype versions of our case studies from Section 2.1: Paintball, Doc Edit (a variant of *Multi-Team Whiteboards*), and Cubist Art. (We did not implement *Soccer Arena*, since it does not surface new security, privacy, or functionality requirements not covered by the other case studies. Section 2 provides further analysis.) Our prototypes are intended to cover a broad spectrum of sharing control functionalities; see Appendix A for detailed descriptions of the apps. Screenshots of the apps are in Figure 3, the range of sharing control features each exercises is in Table 3, and the sharing-related lines of code for each is in Table 4.

We use lines of code as a proxy measure for developer effort (see Table 4). For each app, we count the total lines of code in the app and the lines of code specific to sharing functionality. The low number of sharing-related lines of code suggests that the burden on app developers to use our toolkit in practice is reasonable. Furthermore, we conjecture that fully fledged apps are likely to contain many more lines of code unrelated to sharing, further reducing the comparative proportion of sharing-related lines of code in the app. We note also that the repetition of some prototyped features across multiple apps (such as location-coupled sharing in both the Paintball and Shared Blocks apps) suggests that ShareAR's features are composable, and that developers can choose an app-appropriate subset of functionality.

Compatibility with existing guidelines. We also consider the compatibility of ShareAR with existing design guidelines from AR headset manufacturers. We focus on guidelines related to multi-user interactions, asking: Does ShareAR allow an app developer to meet these guidelines? We in-

App	Sharing LoC	Total LoC
Paintball	13	240
Doc Edit	173	1236
Cubist Art	153	1131

Table 4: Lines-of-code counts for the three prototype applications. We report both the total lines of code for the application and the lines of code dedicated to interfacing with the ShareAR toolkit.

vestigate the Microsoft HoloLens guidelines [43] and Meta guidelines [40]; we find that ShareAR is compatible with all of them. The results are summarized in Table 5; see Appendix B for additional information.

6.2 Security Evaluation

We examine the security and privacy of our ShareAR-enabled apps under our threat model of untrusted users (but trusted OS and apps). As described in Section 3, we rely on app developers to use the ShareAR APIs that are appropriate for their use case. For the sake of exposition, we focus on the Doc Edit app since it invokes all of the restrictions our ShareAR prototype supports; our observations also extend to Paintball and Cubist Art where applicable.

We find that ShareAR's security and privacy restrictions function as intended and meet the security goals in Section 2.3. Considering first the outbound security goals:

- *Support granting/revoking per-user permissions*: The Doc Edit app includes a menu that allows a user to grant and revoke per-user permissions. A user who never received or no longer has view permissions on a document cannot see it.
- *Support granting/revoking per-object permissions*: The aforementioned menu controls permissions on a per-document basis: only the currently selected document is affected by a permission change.
- *Support physically intuitive access control restrictions*: Doc Edit provides a ghost version of a document object (as a flat gray box). A user with permission only to view the ghost cannot tell if the original document is red or not; but the user sees the ghost in the same location as the document's owner sees the original document, and this location remains synchronized when the document's owner moves the document in physical space.

Considering the inbound security goals from Section 2.3:

- *Support user control of incoming virtual content*: The Doc Edit app surfaces an incoming permission-granting message to the user via a small menu, through which the user can choose to accept or decline. If the user accepts, a new (location-decoupled) instantiation of the document appears in front of the user, and the user can also see the sharer's instantiation of the document in its full (rather than ghost) form. If the user declines the document, no such change occurs.
- *Support user control of owned physical space*: As described in Section 5, our prototype leverages a Unity plane-clipping feature to implement simple owned physical space enforcement. We clip parts of any object closer to the user than 0.85 m (with the distance chosen to match a HoloLens recommended setting [42]).

6.3 Performance Evaluation

We now evaluate ShareAR's performance, measuring its operations (and comparing them to baseline operations where possible) and studying how it scales with numbers of virtual objects and users. We find that ShareAR imposes only a modest overhead on interdevice communication even as the number of objects and users increases.

Experimental setup. We build an app (1506 lines of C# code) to exercise ShareAR's components and measure its performance. In our test app, a test device creates objects that are location-coupled or location-decoupled, sharing them publicly, as ghosts, or keeping them private. One or more other test devices auto-accept or manually accept objects. The first device changes the objects' permissions, updates the objects' location, and finally deletes the objects.

Our experimental setup consists of five HoloLens devices communicating on the same local network, in two experimental scenarios: (1) for each $n \in \{1,2,3,4,5\}$, we select n devices and fix $h = 1$ shared AR object; (2) for each $h \in \{2^0, 2^1, 2^2, 2^3, 2^4, 2^5\}$, we set h to be the number of objects present and fix $n = 2$ devices. All devices run our evaluation app with the same n and h parameters; all devices join the network sequentially, and then the last device to join the network triggers the evaluation app.

The operations we measure are `Create`, `Accept Create`, `Change Permission`, `Accept Change`, `Update Location`, and `Delete`. Each operation involves work done on User A's device to initiate the operation, a message sent across the network from User A to User B, and work done on User B's device to process the operation. Note that in some cases (for `Create` and `Change Permission`) User B's device reacts by initiating an operation that we also measure (`Accept Create` and `Accept Change`).

In addition to measuring the within-module time for A's initiating action and B's receiving action, we measure and report on *operation completion time*, i.e., the time it takes from A's initiating action until B has finished processing. To correct for clock skew between the two devices, we add into the evaluation script a message from B to A containing B's timestamps (note that this is not part of our module's protocol, but exists solely for the purposes of the evaluation). Device A then combines this information with its own timestamps to compute the final timing numbers.

Finally, for the sharing module operations that clearly correspond to a primitive Unity operation (`Create`, `Update Location`, and `Delete`), we also measure as a baseline the timing of the Unity operation.

We repeat each evaluation point — defined by operation, configuration (e.g., location coupled or decoupled), number

Guideline	Short description	Source	Support?
How are they sharing?	Design for app's purpose of sharing: presentation, collaboration, guidance	HoloLens Developer Guidelines [43]	✓*
What is the group size?	Accommodate as many users as the app expects to need	HoloLens Developer Guidelines [43]	✓*
Where is everyone?	Support users in the same or different physical spaces as needed	HoloLens Developer Guidelines [43]	✓
When are they sharing?	Design for asynchronous or real-time sharing as appropriate	HoloLens Developer Guidelines [43]	✓*
How similar are their physical environments?	Place objects appropriately in non-co-located users' environments	HoloLens Developer Guidelines [43]	✓
What devices are they using?	Integrate with VR as needed	HoloLens Developer Guidelines [43]	✓*
Clip planes near user	Set minimum visible distance for object to 0.85 m	HoloLens Hologram Stability Guidelines [42]	✓*
Do not disturb	Avoid incessant notifications to user	Meta Developer Guidelines [40]	✓*
The holographic campfire	Allow users to see each other	Meta Developer Guidelines [40]	✓
Public by default	Support shared-world intuition by making content publicly visible	Meta Developer Guidelines [40]	✓*

*Table 5: Summary of ShareAR's compatibility with existing multi-user AR design guidelines. For the check marks with a * appended, see Appendix B for additional details.*

of users, and number of objects — for 2 warm-up trials and 5 measured trials, reporting the mean and standard deviation.

Basic profile of operations. We begin by considering ShareAR's operations with a single pair of users ($n = 2$) sharing a single object ($h = 1$).

The overall operation completion time is between approximately 70 ms and 250 ms, depending on the operation and configuration. This overall time is significantly dominated by factors external to the ShareAR module, and that *any* multi-user sharing solution would encounter: i.e., the network latency and the HoloToolkit Sharing library on either end of it. The overhead specific to ShareAR is minimal: we find that `Create` and `Change Permissions` operations are most expensive on average, still taking less than 5 ms in the worst case for the computation on each device; all other operations take less than 1 ms on each device. For the operations that we can compare directly with Unity baselines, we also find that ShareAR's overhead is minimal: the operations stay within 2.5 ms of the baseline in the worst case.

Scaling with the number of users. Next, we consider how ShareAR scales as the number of users increases.

In terms of network traffic, a user sharing an object needs to send object create and update messages to $n-1$ others; additionally, once a user accepts a sharing offer, their device sends an acceptance message back to the sharer and an informational message to all other $n-2$ users to stay in sync. The total number of messages in the interaction thus scales quadratically. For updates to already-shared objects (location change, deletion), the sharing user sends one message per other user, and no replies or additional messages are sent (overall scaling linearly with users).

In terms of timing, all operations under all test conditions took less than 5 ms. For all but `Create` and `Change Permission`, the operations on average remained under 1 ms. These overheads are reasonable, especially given their small additional overhead beyond to the corresponding baseline operations where present (shown with a dashed line for `Create`, `Update Location`, `Delete`). More detailed performance data is in Appendix C.

In terms of scaling, `Create` and `Change Permission` scale approximately linearly with the number of users; all other operations remain approximately constant. Different configurations for an operation (e.g., location coupled versus decoupled sharing, or different object deletion modes) may slightly affect performance (as reflected in the differently colored lines in the graphs), typically taking longer for location-decoupled objects due to the overhead of processing multiple instantiations of the same object.

Scaling with the number of objects. Finally, we measure ShareAR with increasing numbers of AR objects.

In terms of network traffic, we observe that it scales linearly with the number of objects, as each operation and associated message is independent per object.

In terms of timing, all operations took less than 3 ms (and often less than 1 ms). These overheads are reasonable, especially given their relation to the corresponding baseline operations where present. (For the module operations for which a baseline Unity operation is plotted — `Create`, `Update Location` and `Delete` — the relevant module operation timing is very close to that of the baseline Unity operation.) Additional details are in Appendix C.

In terms of scaling, for all operations, the time taken is approximately constant per object as the number of objects scales: in other words, an operation on one object registered with the module is independent of how many other objects are also registered with the module. Some operations exhibit a slight slope downward, suggesting caching benefits.

Performance evaluation summary. From our measurements, we see that object creation and permission changes are the most computationally expensive operations. However, we anticipate that in practice these operations will only occur during a small fraction of the frame updates in an app.

Even so, the greatest observed time taken for an operation was under 5 ms, and most measurements remained under 1 ms. Furthermore, since these measurements were of our unoptimized research prototype, continued code optimization may bring the performance overhead down even further.

7 Discussion

This work presents the first systematic investigation of multi-user sharing control for AR apps. We propose a module that is flexible enough to support many different decisions by app developers. Below we discuss several examples of future directions enabled by our work.

Execute permissions. Although multi-user AR systems are still primitive, we envision that future systems will support not only read and write but also execute permissions. One possible manifestation may be to allow a user to execute *predefined actions* on another user's object without having full edit control. For instance, an app may allow other users to make a virtual dog wag its tail without allowing them to make the dog arbitrarily large. Our module can be extended to include additional permissions, including this one.

Asynchronous sharing. Our design exploration assumes that both users are online when a sharing action occurs; extensions of our work could explore removing this assumption. For example, consider a user who places publicly visible virtual decorations outside their home. We may want (1) the objects to still be visible to a passerby when the user is not home, but (2) the passerby's device to only become notified of the objects' existence and public visibility when the passerby is physically proximate to the home. Such a design may require an alternate network architecture than peer-to-peer; our module's network agnosticism would support this.

Minimizing developer errors. We emphasize that one consequence of our module's flexibility is that developers must be cautious to use it in a way that supports their app use case. Some potential user-to-user threats may be subtle: for example, if app developers chose to share ghost objects automatically with no way to refuse or delete them, one user might intentionally or accidentally clutter another user's view with ghost objects (an example of a denial-of-service attack). Or, if an app developer implements a personal space policy that makes AR objects invisible to all users but does not provide a way to interact with or retrieve an invisible object, then a malicious user could walk up to others' objects to force them to become invisible and non-interactible. Still other pitfalls may depend on app semantics: for instance, if the developer of an app such as *Community Art* does not put limits on users, a user could monopolize a common space and prevent other users from placing objects there. Future work, then, may explore ways to support app developers in using the features from our system that are most suitable for their overall goals.

Analysis in the wild. More broadly, our work lays a foundation for future empirical studies on how developers use our module's components in practice and how users respond to concrete usage of these components. Such an evaluation is nontrivial since evaluating the usability of a single app does not generalize well to the usability of others [45], for the same reason described in Section 2 that a sharing control module cannot be one-size-fits-all. However, we note the importance of follow-up studies considering user perceptions when making specific design decisions, and we encourage future work to leverage our technical foundation to examine under which circumstances certain sharing mechanisms are appropriate.

Location privacy. Much multi-user app functionality, including our design, requires that users share their location with the app: sharing at least where one is within a physical space is necessary for location-synchronized virtual content. Some users may not anticipate or agree with such location sharing, even for trustworthy apps, though such sharing may be fundamental to the design of location-based AR apps. Additional location privacy concerns could be introduced by app developers, if app developers mishandle and accidentally or intentionally expose a user's location to *other users*. This threat, however, is dependent upon app-level semantics, and is neither unique to nor preventable by the underlying sharing framework. We encourage future work to explore this point further.

Inherently conflicting goals. Finally, we conjecture that there may be fundamental tensions in some aspects of secure and private content sharing between users. For example, consider the case of a shared space in which one user owns a publicly visible ball object and another user owns a private wall object. When the public ball is thrown at the private wall, it is not obvious which user(s) should see the ball rebound. If the ball rebounds for both users, then the ball owner gains information about the presence of the wall; if the ball does not rebound for either user, then the wall owner sees the ball go through the wall, defying physics; if only the wall owner sees the ball rebound, then the two users no longer have a synchronized view of the shared space. Determining how physics-obeying virtual objects should interact to minimize information leakage via this side channel while maximizing physical intuition is a subtle area for future work, and we conjecture that *no* content sharing solution can simultaneously achieve both goals perfectly.

8 Related Work

Although AR has a long history (e.g., [61]), the computer security community has only recently begun examining the space [13, 52]. Prior efforts on AR security and privacy include filtering raw real-world input [12, 28, 49, 54, 65] and regulating untrusted AR output [32, 34]. These efforts focus on the case of a single user interacting with an AR device.

Literature on multi-user AR security and privacy is just beginning to emerge. Some prior work has proposed methods

for secure device pairing via out-of-band channels [19, 60]; our work is complementary. Other prior work has proposed specific multi-user interaction modalities, such as location-based interfaces for making virtual content private and auditing content visibility [8, 9], mediating shared experiences with remote collaborators [50], and using personal tablets in shared spaces to separate private and public content [62, 72]. While these works present specific multi-user AR systems, our work is the first to systematically and broadly consider the design space for AR sharing control and our module could be leveraged when implementing these prior ideas.

There is a rich literature on access control (see, e.g., [7] for an overview). Our work does not assume what access control model is best for a particular app. Our implementation leverages an access control matrix [30] as a simple and flexible model for per-user and per-object permissions; we intend for other established access control models in specific app contexts (e.g., [17, 68]) to be layered on top of our toolkit, and we instead focus on the challenges of managing the implications of access control in the 3D physical AR setting.

Work in AR user experience has surfaced security- and privacy-relevant themes for multi-user contexts. Lebeck et al. [35] surface multi-user concerns such as physiological attacks, virtual clutter, and the obscurity of other users' actions. Poretski et al. [48] examine normative tensions in AR, emphasizing enforcing personal space and designing for user control. Olsson et al. [46,47] identify user needs such as control over privacy, socially appropriate ways to interact with devices, and solutions for abuse of public content by other users. These studies shed light on desired system properties and user concerns but do not directly address system design; our work builds concretely on these findings.

Multi-user digital interactions that take place in a physical space have also been studied in the context of tabletop interfaces and large computerized displays [44, 57–59, 71]. Our work addresses similar needs for and tensions around public versus private content arising in the AR setting, where immersive 3D content can be situated anywhere in the physical world rather than constrained to a shared display.

9 Conclusion

Multi-user AR technologies hold much promise, but also raise security and privacy risks in the potentially undesirable interactions between human users. These risks should be addressed while AR ecosystems are being actively developed rather than after sub-optimal ad hoc conventions have taken root. To that end, we are the first to systematically develop a set of security and functionality goals for multi-user AR. We present the design of a sharing control module for AR content, which we envision as an app-level library or OS interface that can be leveraged by app developers. Our work identifies and addresses key challenges that stem from AR's tight integration into the physical world. Our prototype, ShareAR,[4] for the Microsoft HoloLens demonstrates the feasibility of our design, and our evaluation suggests that it meets our design goals and imposes minimal performance overhead. By addressing multi-user AR sharing control systematically now, we are taking steps toward securing the fully fledged multi-user AR applications of the future.

Acknowledgments

We thank Ivan Evtimov, Earlence Fernandes, Kiron Lebeck, Lucy Simko, and Anna Kornfeld Simpson for valuable discussions and feedback on previous drafts; we thank James Fogarty for his advice on tabletop interface related work. This work was supported in part by the National Science Foundation under awards CNS-1513584, CNS-1565252, and CNS-1651230, and by the Washington Research Foundation.

References

[1] J. Alexander. 'Ugandan Knuckles' is overtaking VRChat, Jan. 2018. https://www.polygon.com/2018/1/8/16863932/ugandan-knuckles-meme-vrchat.

[2] R. Alp Güler, N. Neverova, and I. Kokkinos. DensePose: Dense human pose estimation in the wild. In *CVPR*, 2018.

[3] E. Alvarez. Facebook's next big augmented reality push is multiplayer games, Sept. 2018. https://www.engadget.com/2018/09/07/facebook-ar-games-multiplayer-first-look/.

[4] ARCore. https://developers.google.com/ar/.

[5] ARKit. https://developer.apple.com/arkit/.

[6] AR Studio. https://developers.facebook.com/products/camera-effects/ar-studio/.

[7] M. Bishop. *Computer Security: Art and Science*. Addison-Wesley Professional, 2nd edition, 2018.

[8] A. Butz, C. Beshers, and S. Feiner. Of vampire mirrors and privacy lamps: Privacy management in multi-user augmented environments. In *ACM UIST*, 1998.

[9] A. Butz, T. Höllerer, S. Feiner, B. MacIntyre, and C. Beshers. Enveloping users and computers in a collaborative 3D augmented reality. In *IEEE/ACM International Workshop on Augmented Reality*, 1999.

[10] J. T. Chiang, J. J. Haas, and Y.-C. Hu. Secure and precise location verification using distance bounding and simultaneous multilateration. In *WiSec*, 2009.

[11] S. Curtis. Sex pests are using Apple AirDrop to send explicit pictures to unsuspecting commuters, Aug. 2017. https://www.mirror.co.uk/tech/sex-pests-using-apple-airdrop-10987968.

[12] L. D'Antoni, A. Dunn, S. Jana, T. Kohno, B. Livshits, D. Molnar, A. Moshchuk, E. Ofek, F. Roesner, S. Saponas, et al. Operating system support for augmented reality applications. *HotOS*, 2013.

[13] J. A. de Guzman, K. Thilakarathna, and A. Seneviratne. Security and privacy approaches in mixed reality: A literature survey, 2018. http://arxiv.org/abs/1802.05797.

[4] See arsharing.cs.washington.edu or arsharingtoolkit.com

[14] T. Denning, Z. Dehlawi, and T. Kohno. In situ with bystanders of augmented reality glasses: Perspectives on recording and privacy-mediating technologies. In *CHI*, 2014.

[15] Digi-Capital. Ubiquitous $90 billion AR to dominate focused $15 billion VR by 2022, 2018. https://www.digi-capital.com/news/2018/01/ubiquitous-90-billion-ar-to-dominate-focused-15-billion-vr-by-2022/.

[16] A. P. Felt, S. Egelman, M. Finifter, D. Akhawe, and D. Wagner. How to ask for permission. In *HotSec*, 2012.

[17] D. Ferraiolo and R. Kuhn. Role-based access controls. In *NCSC*, 1992.

[18] C. Fink. The trillion dollar 3D telepresence gold mine, Nov. 2017. https://www.forbes.com/sites/charliefink/2017/11/20/the-trillion-dollar-3d-telepresence-gold-mine/#42b8f0a12a72.

[19] E. Gaebel, N. Zhang, W. Lou, and Y. T. Hou. Looks good to me: Authentication for augmented reality. In *TrustED*, 2016.

[20] J. Gallagher. Upcoming game easily shows you how to master paintball, Aug. 2017. https://mobile-ar.reality.news/news/apple-ar-upcoming-game-easily-shows-you-master-paintball-0179651/.

[21] G. Hancke and M. Kuhn. An RFID distance bounding protocol. In *SECURECOMM*, 2005.

[22] L. A. Hayduk. Personal space: Where we now stand. *Psychological Bulletin*, 94(2):293–335, 1983.

[23] T. He, C. Huang, B. M. Blum, J. A. Stankovic, and T. Abdelzaher. Range-free localization schemes for large scale sensor networks. In *MobiCom*, 2003.

[24] Microsoft HoloLens. https://www.microsoft.com/microsoft-hololens/en-us.

[25] S. Houben and N. Marquardt. WatchConnect: A toolkit for prototyping smartwatch-centric cross-device applications. In *CHI*, 2015.

[26] D. Y. Huang, D. Grundman, K. Thomas, A. Kumar, E. Bursztein, K. Levchenko, and A. C. Snoeren. Pinning down abuse on google maps. In *WWW*, 2017.

[27] S. E. Hudson, J. Mankoff, and I. Smith. Extensible input handling in the subArctic toolkit. In *CHI*, 2005.

[28] S. Jana, D. Molnar, A. Moshchuk, A. M. Dunn, B. Livshits, H. J. Wang, and E. Ofek. Enabling fine-grained permissions for augmented reality applications with recognizers. In *USENIX Security*, 2013.

[29] S. W. Kim. 3D document editing system. U.S. Patent Application 20180081519, 2016, https://bit.ly/2N5Dt2S.

[30] B. W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8(1):18–24, Jan. 1974.

[31] L. Lazos and R. Poovendran. SeRLoc: Secure range-independent localization for wireless sensor networks. In *WiSe*, 2004.

[32] K. Lebeck, T. Kohno, and F. Roesner. How to safely augment reality: Challenges and directions. In *HotMobile*, 2016.

[33] K. Lebeck, T. Kohno, and F. Roesner. Enabling multiple applications to simultaneously augment reality: Challenges and directions. In *HotMobile*, 2019.

[34] K. Lebeck, K. Ruth, T. Kohno, and F. Roesner. Securing augmented reality output. In *IEEE Symposium on Security & Privacy*, 2017.

[35] K. Lebeck, K. Ruth, T. Kohno, and F. Roesner. Towards security and privacy for multi-user augmented reality: Foundations with end users. In *IEEE Symposium on Security & Privacy*, 2018.

[36] L. Li, D. Li, T. F. Bissyandé, J. Klein, Y. Le Traon, D. Lo, and L. Cavallaro. Understanding Android app piggybacking: A systematic study of malicious code grafting. *IEEE TIFS*, 12(6):1269–1284, 2017.

[37] Magic Leap. https://www.magicleap.com/#/home.

[38] L. Matney. Jeff Koons' augmented reality Snapchat artwork gets 'vandalized', Oct 2017. https://techcrunch.com/2017/10/08/jeff-koons-augmented-reality-snapchat-artwork-gets-vandalized/.

[39] D. Mehta, O. Sotnychenko, F. Mueller, W. Xu, S. Sridhar, G. Pons-Moll, and C. Theobalt. Single-shot multi-person 3D pose estimation from monocular RGB. In *3DV*, 2018.

[40] Meta. Spatial interface design: Public by default. https://devcenter.metavision.com/design/spatial-interface-design-principles-public-by-default.

[41] https://www.metavision.com/.

[42] Microsoft. Hologram stability. https://docs.microsoft.com/en-us/windows/mixed-reality/hologram-stability.

[43] Microsoft. Shared experiences in mixed reality. https://developer.microsoft.com/en-us/windows/mixed-reality/shared_experiences_in_mixed_reality.

[44] M. R. Morris, A. Cassanego, A. Paepcke, T. Winograd, A. M. Piper, and A. Huang. Mediating group dynamics through tabletop interface design. *IEEE CGA*, 6(5):65–73, 2006.

[45] D. R. Olsen, Jr. Evaluating user interface systems research. In *UIST*, 2007.

[46] T. Olsson, T. Kärkkäinen, E. Lagerstam, and L. Ventä-Olkkonen. User evaluation of mobile augmented reality scenarios. *Journal of Ambient Intelligence and Smart Environments*, 4:29–47, 2012.

[47] T. Olsson, E. Lagerstam, T. Kärkkäinen, and K. Väänänen-Vainio-Mattila. Expected user experience of mobile augmented reality services: A user study in the context of shopping centres. *Personal and ubiquitous computing*, 17(2):287–304, 2013.

[48] L. Poretski, J. Lanir, and O. Arazy. Normative tensions in shared augmented reality. *CSCW*, 2018.

[49] N. Raval, A. Srivastava, A. Razeen, K. Lebeck, A. Machanavajjhala, and L. P. Cox. What you mark is what apps see. In *MobiSys*, 2016.

[50] D. Reilly, M. Salimian, B. MacKay, N. Mathiasen, W. K. Edwards, and J. Franz. Secspace: Prototyping usable privacy and security for mixed reality collaborative environments. In *ACM SIGCHI EICS*, 2014.

[51] K. Rematas, I. Kemelmacher-Shlizerman, B. Curless, and S. Seitz. Soccer on your tabletop. In *CVPR*, 2018.

[52] F. Roesner, T. Kohno, and D. Molnar. Security and privacy for augmented reality systems. *Communications of the ACM*, 57(4):88–96, 2014.

[53] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *IEEE Symposium on Security and Privacy*, 2012.

[54] F. Roesner, D. Molnar, A. Moshchuk, T. Kohno, and H. J. Wang. World-driven access control for continuous sensing. In *ACM CCS*, 2014.

[55] N. Sastry, U. Shankar, and D. Wagner. Secure verification of location claims. In *WiSe*, pages 1–10, 2003.

[56] F. Schaub, R. Deyhle, and M. Weber. Password entry usability and shoulder surfing susceptibility on different smartphone platforms. In *MUM*, 2012.

[57] S. D. Scott, M. S. T. Carpendale, and K. M. Inkpen. Territoriality in collaborative tabletop workspaces. In *CSCW*, 2004.

[58] S. D. Scott, K. D. Grant, and R. L. Mandryk. System guidelines for co-located, collaborative work on a tabletop display. In *ECSCW*, 2003.

[59] C. Shen, K. Everitt, and K. Ryall. Ubitable: Impromptu face-to-face collaboration on horizontal interactive surfaces. In *UbiComp*, 2003.

[60] I. Sluganovic, M. Serbec, A. Derek, and I. Martinovic. HoloPair: Securing shared augmented reality using Microsoft HoloLens. In *ACSAC 2017*, 2017.

[61] I. E. Sutherland. A head-mounted three-dimensional display. In *Fall Joint Computer Conference, American Federation of Information Processing Societies*, 1968.

[62] Z. Szalavári, E. Eckstein, and M. Gervautz. Collaborative gaming in augmented reality. In *VRST*, 1998.

[63] D. Takahashi. Spatial raises $8 million for augmented reality collaboration platform, Oct. 2018. `https://venturebeat.com/2018/10/24/spatial-raises-8-million-for-augmented-reality-collaboration-platform/`.

[64] F. Tari, A. Ant Ozok, and S. Holden. A comparison of perceived and real shoulder-surfing risks between alphanumeric and graphical passwords. In *SOUPS*, 2006.

[65] R. Templeman, M. Korayem, D. Crandall, and A. Kapadia. PlaceAvoider: Steering first-person cameras away from sensitive spaces. In *NDSS*, 2014.

[66] R. Tilton. Daydream Labs: positive social experiences in VR. Google, Aug. 2016. `https://www.blog.google/products/google-vr/daydream-labs-positive-social/`.

[67] Ubiquity6. `https://ubiquity6.com`.

[68] L. Wang, D. Wijesekera, and S. Jajodia. A logic-based framework for attribute based access control. In *FMSE*, 2004.

[69] X. Wang, A. Pande, J. Zhu, and P. Mohapatra. STAMP: Enabling privacy-preserving location proofs for mobile users. *IEEE/ACM ToN*, 24(6):3276–3289, 2016.

[70] P. Wijesekera, A. Baokar, L. Tsai, J. Reardon, S. Egelman, D. Wagner, and K. Beznosov. The feasibility of dynamically granted permissions: Aligning mobile privacy with user preferences. In *IEEE Symposium on Security and Privacy*, 2017.

[71] M. Wu and R. Balakrishnan. Multi-finger and whole hand gestural interaction techniques for multi-user tabletop displays. In *UIST*, 2003.

[72] Y. Xu, M. Gandy, S. Deen, B. Schrank, K. Spreen, M. Gorbsky, T. White, E. Barba, I. Radu, J. Bolter, and B. MacIntyre. Bragfish: Exploring physical and social interaction in co-located handheld augmented reality games. In *ACE*, 2008.

[73] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu. ViewDroid: Towards obfuscation-resilient mobile application repackaging detection. In *WiSec*, 2014.

[74] W. Zhou, X. Zhang, and X. Jiang. AppInk: Watermarking Android apps for repackaging deterrence. In *ASIA CCS*, 2013.

[75] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party Android marketplaces. In *CODASPY*, 2012.

A Prototype Application Descriptions

In Section 6.1, we describe our assessment of our design's functionality by its ability to flexibly support our representative case study apps. Here, we provide further details on our implemented prototype case study apps.

Paintball. This is a minimalist implementation of the *Paintball* case study. Players can launch red spheres that, upon contact with another player, attach to that other player. All users in the game session can see all of the red spheres. For the purposes of the prototype, we leave out scorekeeping and more advanced game features.

Doc Edit. This is a basic version of *Multi-Team Whiteboards* in which each user accessing content has a personal instantiation of it. Users, by interacting with a simple control panel, create flat rectangular boxes as documents. Documents are location-decoupled, and though they are private and ghosted by default, users can choose to share individual documents with individual users. A user can also turn a document red, modifying the document's contents in a way that ghost documents do not display (for the prototype, this emulates arbitrary content entry, which we do not implement); the user can also delete the document in a group-extended way (i.e., all other users' instances of the document are also deleted).

Cubist Art. This is a simplified version of the *Community Art* case study. Rather than making and manipulating arbitrary objects, users create and control cubes and can choose to share them or not. Although many of the user's possible actions via the control panel are similar to those of Doc Edit, there are several key differences: (1) Cubes are public by default rather than private. (2) Cubes are shared in a location-coupled way rather than location-decoupled. (3) Cubes obey real-world physics instead of being entirely script-controlled. (4) Object deletion is global location-coupled rather than

global location-independent (note, though, that the semantics of location-coupled sharing make these two cases visually equivalent for solely location-coupled objects).

We did not implement the *Soccer Arena* case study, since it does not surface new security, privacy, or functionality requirements not covered by the other case studies. Section 2 provides further analysis.

B Interaction with Existing Design Recommendations

Below we include a further analysis of our module's compatibility with existing design recommendations.

How are they sharing? The HoloLens guidelines list possible sharing scenarios as consisting of presentation, collaboration, or guidance. Our design supports all of these: for instance, a developer can use ShareAR to set appropriate controls such as view-only permissions when a presenter shares content with an audience. Besides the opt-in scenarios that the HoloLens guidelines describe, our design also supports opt-out public content sharing, which we argue should be treated as another important use case for AR.

What is the group size? The HoloLens guidelines remind developers to design for as many users as needed. Our design can support an arbitrary number of users. (In practice, our implementation stores both object IDs and user IDs as 32-bit integers, providing a generous upper bound on its capacity. We examine performance questions in Section 6.3.)

When are they sharing? The HoloLens guidelines ask whether sharing is asynchronous or synchronous. Although we explicitly design ShareAR to support real-time sharing, we do not preclude the possibility of asynchronous sharing. A developer could, for instance, write a replacement network shim layer that relies on a central server for data storage and periodically queries the server for updates.

What devices are they using? In particular, the HoloLens guidelines ask whether AR users might share content with VR users. Although this is outside the scope of this work, we note that there is nothing in principle that fundamentally prevents developers from extending our work into VR as well. More broadly, we note that in principle, the ShareAR design is compatible across any AR HMD platforms that satisfy basic assumptions such as a shared notion of 3D space. (Our implementation is built for the HoloLens and has not been ported to other platforms as of this writing.)

Clip planes near user. HoloLens recommends setting a "plane clipping" distance of 0.85 m so that a user does not see any portions of AR objects that are closer than that in the user's field of view [42]. Plane clipping may conceptually be considered a partial way of enforcing personal space; however, it only affects the view of the user whose space is invaded, and other users still see the object as being close to the user. ShareAR's treatment of owned physical space

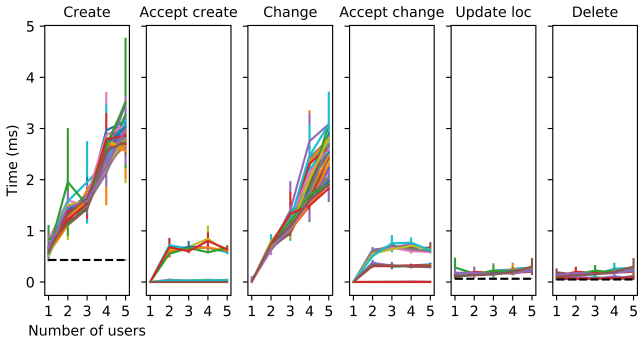


Figure 4: Timing measurements for all steps in the evaluation protocol, each from the perspective of the device initiating the step, as the number of present users scales. Acceptance times are measured on the receiver's device; all other times are measured on the sharer's device. Black dashed lines denote a corresponding baseline Unity operation where one exists.

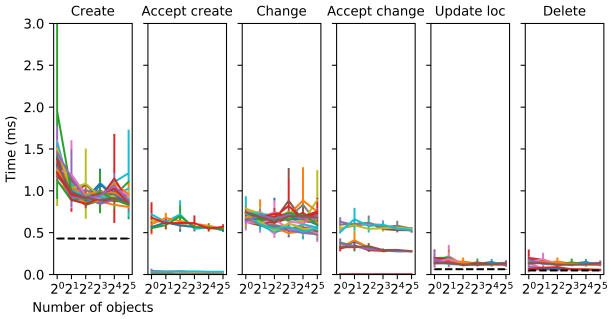


Figure 5: Timing measurements for all steps in the evaluation protocol, each from the perspective of the device initiating the step, on a per-object basis as the number of objects scales. Acceptance times are measured on the receiver's device; all other times are measured on the sharer's device. Black dashed lines denote a corresponding baseline Unity operation where one exists.

encompasses this recommendation and is a more complete solution. (Our implementation does not yet include personal space; however, it does include basic plane clipping.)

Do not disturb. Meta cautions against showing the user incessant notifications. Our design does not specify the user interface: notifications from other devices are passed as an event to the app but not displayed to the user. Thus, ShareAR is flexible enough to support this design choice.

Public by default. This recommendation is similar in spirit to our goal of supporting shared physical-world intuition. Although our design does support a purely public virtual world, we do not recommend it for all circumstances; our ghosting mechanism maintains a basic shared-world intuition while preserving a degree of privacy.

C Detailed Performance Data

Results as the number of users scales are shown in Figure 4; results as the number of objects scales are shown in Figure 5.

Understanding and Improving Security and Privacy in Multi-User Smart Homes: A Design Exploration and In-Home User Study

Eric Zeng
ericzeng@cs.washington.edu

Franziska Roesner
franzi@cs.washington.edu

Paul G. Allen School of Computer Science & Engineering
University of Washington

Abstract

Smart homes face unique security, privacy, and usability challenges because they are multi-user, multi-device systems that affect the physical environment of all inhabitants of the home. Current smart home technology is often not well designed for multiple users, sometimes lacking basic access control and other affordances for making the system intelligible and accessible for all users. While prior work has shed light on the problems and needs of smart home users, it is not obvious how to design and build solutions. Such questions have certainly not been answered for challenging adversarial situations (e.g., domestic abuse), but we observe that they have not even been answered for tensions in otherwise functional, non-adversarial households. In this work, we explore user behaviors, needs, and possible solutions to multi-user security and privacy issues in generally non-adversarial smart homes. Based on design principles grounded in prior work, we built a prototype smart home app that includes concrete features such as location-based access controls, supervisory access controls, and activity notifications, and we tested our prototype though a month-long in-home user study with seven households. From the results of the user study, we re-evaluate our initial design principles, we surface user feedback on security and privacy features, and we identify challenges and recommendations for smart home designers and researchers.

1 Introduction

Smart devices and smart home platforms, such as Samsung SmartThings, Philips Hue lights, Google Home, the Amazon Echo, and Nest thermostats and cameras, are being increasingly adopted and deployed in the homes of end users. These devices and platforms allow users to remotely control and monitor their devices as well as to create automations (e.g., automatically locking the door when the user leaves home).

Security and Privacy in Multi-User Smart Homes. Smart homes are fundamentally multi-user platforms. Multiple people living in or accessing a home — including partners, roommates, parents and children, guests, and household employees — may want or need the ability to use and configure the smart devices within the home. As prior work (e.g., [14, 36, 41]) has begun to show, conflicts and tensions may arise between these multiple stakeholders — even in generally non-adversarial (e.g., non-abusive) households. For example, the more tech-savvy users who install smart devices in their homes may intentionally or unintentionally restrict other users from accessing home functions (like thermostats) that were previously physically accessible [14, 41]; privacy concerns and violations may arise between co-occupants [3, 41]; and remote control of devices can be used for harassment [3].

Unfortunately, current smart homes are not yet thoughtfully designed for interactions between and use by multiple people. Though prior work has surfaced the need for additional access control options [16], transparency, and privacy features [41], many commercial smart home platforms present only simple security and privacy controls, or even none at all [22]. For example, Samsung SmartThings, a popular smart home platforms, forces home administrators to choose between provisioning additional accounts with administrator privileges or not provisioning additional accounts at all [33].

Designing to Address These Challenges. Providing multi-user smart home security, privacy, and usability is not a straightforward matter of simply building it, but rather requires careful consideration of a complex design space. We take a step back to ask: What security, privacy, and other goals *should* a multi-user smart home design aim to achieve? How might it achieve those goals? And do those goals and their implementation meet the needs of end users in practice?

In this work, we focus specifically on answering these questions for generally functional households without explicitly adversarial relationships. That is, we consider "typical" tensions that may arise between roommates, partners, and parents and children as they interact with and through a smart home; we do not consider explicitly adversarial relationships, such as domestic abuse. Addressing such challenging situations is of course also critically important, but we observe that even the seemingly "easy" case has not yet been sufficiently addressed

in prior work or today's commercial platforms.

To begin answering these questions, we thus systematized prior work to develop an initial set of design principles for smart homes in generally non-adversarial multi-user households: *access control flexibility, user agency, respect among users, and transparency of smart home behaviors*. Based on these initial principles, we designed and prototyped a mobile app for smart home control, which includes concrete features such as location-based access controls, supervisory access controls, and activity notifications.

In-Home User Study and Design Recommendations. To evaluate how well our proposed design principles and our prototype's specific design choices meet the use cases of real (non-adversarial) households — and to gain a deeper understanding of the multi-user smart home access control needs and use cases of this class of users — we conducted a month-long *in situ* user study. We deployed our prototype with seven households in the Seattle metropolitan area.

The empirical findings from our user study allow us to evaluate and refine our proposed principles for security and privacy in multi-user smart homes, and we surface technical directions and open questions for platform designers and researchers, which were not apparent in prior work that did not conduct *in situ* design evaluations.

Among our multiple findings, we found that for some of our participants, positive household social norms and relationship dynamics obviated the need for technical access controls. This finding suggests directions and questions for future work, including: How can a smart home platform design leverage or scaffold these social norms rather than simply existing alongside them? And how can the platform simultaneously support use cases and user groups where these social norms and relationship dynamics are not as positive [3] or (as in the case of our participant families with teenagers) in tension?

Another finding surfaced through our user study was that participants' varied access control desires required our prototype to support complex combinations of access control options. Unfortunately, when we increased complexity, it decreased usability, potentially discouraging less motivated or savvy users from using access controls. This finding raises the question: how can smart home designers increase the flexibility of smart home access control systems while making the complexity manageable for all users?

Contributions. Our work makes the following contributions:

1. *Design Principles and Prototype:* We systematize from prior work a set of possible design principles for security and privacy in multi-user smart homes, and we develop a prototype based on these principles targeting generally cooperative households.
2. *In-Home User Study:* We use our prototype to conduct a month-long in-home user study with seven (non-adversarial) households, including couples, roommates, and families with children of various ages. Our study serves to both test our proposed design principles in practice and to more generally enrich the literature on people's security and privacy needs, concerns, and priorities in a multi-user smart home.
3. *Lessons and Recommendations:* Based on our design experience and in-home study, we reflect on our proposed design goals for multi-user smart homes and surface future technical directions as well as open questions for designers and researchers.

2 Background and Motivation

Smart homes raise significant potential security and privacy challenges. These challenges include, for instance, vulnerabilities in the devices themselves (e.g., [29]) and privacy concerns due to ubiquitous recording in the home [6, 21, 42].

In this work, we focus primarily on multi-user security and privacy: how peoples' behavior and usage of the smart home can impact each others' security and privacy. We begin by systematizing the multi-user security and privacy issues prior work has identified for smart homes, as well as the shortcomings of existing approaches in addressing these issues.

2.1 Multi-User Challenges in Smart Homes

Prior work suggests that smart homes can cause or intensify conflicts or tensions between people living in the home — even when relationships between people are not explicitly adversarial (e.g., abusive).

Power and Access Imbalances. One negative dynamic that emerges from smart homes is a power imbalance between the person(s) who install(s) and configure(s) the home, and the other users who are more passively involved. In the worst-case — in the context of intimate partner violence — abusers may have total control over the smart home, enabling harassment and abuse [3]. However, power imbalances also arise in more benign relationships. For example, Geeng et al. observed how more tech-savvy users have more agency in the home, including more access to device functionality, more information about what devices and people in the home are doing, and the power to restrict others from using devices [14].

Privacy Violations. Smart homes can also intentionally or unintentionally used to expose privacy sensitive information about one user to another. Zeng et al. found examples of such situations, like users being unaware of automated notifications sent by cameras to their landlord, and users feeling a loss of privacy because others could view their behavior through smart home logs [41]. Choe et al. studied how devices that capture video, audio, and other behavioral traces could cause tensions between household members, or between guests and household members, who would object to being recorded [6].

Direct Conflict. Lastly, smart homes can be focal points of conflict between people in the home, both due to explicit

malice (e.g., abuse) and due to ordinary conflicts between household members. For example, prior work has documented conflicts arising due to differences in opinion on thermostat setting [14, 41], due to conflicting goals between parents and teens in the context of entryway surveillance [36], or due to the potential use of recorded evidence in household disputes [6].

2.2 Additional Actors: Apps and Automations

The above multi-user issues are compounded by the presence of additional "actors" in smart home systems: third-party apps and integrations that users may install (such as SmartApps or IFTTT), as well as end-user programmed automations. These apps and automations can range from simple rules (such as automatically locking the door or turning off lights when leaving home) to more complex "smart" features that integrate with other cloud services, e.g. weather data and calendars.

These applications and automations can expose users to physical security risks and privacy violations. Third-party applications and automations may be expressly malicious, or buggy and exploitable (e.g., [11]). Moreover, end users themselves may make mistakes programming automations, leading to unexpected behavior, bugs, and potential security and privacy risks [27, 34, 39]. In a multi-user smart home, this combination of actors means that when something unexpected happens in the home, it may be challenging or impossible for a user to determine whether it was the result of another user actuating the smart home remotely, a buggy application or automation, a legitimate application or automation that another user installed, or explicitly malicious activity.

2.3 Shortcomings of Existing Approaches

Though many commercial smart home platforms exist, and a growing body of research literature supports the need to address the above challenges, we are not aware of existing approaches that succeed at addressing them and/or have been rigorously evaluated with end user — neither for explicitly adversarial settings nor in generally cooperative households.

There are many types of access control policies that could be used in smart homes, including time-based policies, location-based policies, per-user policies, and per-device policies. However, Mare et al. found that adoption of these techniques in smart home platforms is uneven and limited [22]. Some platforms support a subset of these policies, e.g., Apple Homekit has location-based access controls, and Vera has multiple privilege tiers for admins and guests. However, some popular platforms have minimal or no access control at all: Samsung SmartThings has only a single privilege level for all users and no access control policies, while Google Home and Amazon Echo do not authenticate voice commands. He et al. [16] and Ur et al. [35] found similar fragmentation of access control and authentication policies between individual devices: some devices like door locks had many access controls, while others like smart thermostats had none.

While having no access controls or user roles at all is clearly insufficient for user needs (e.g., [33]), the jury is still out on what are the *right* access control designs for multi-user smart homes. To that end, He et al. [16] surveyed hundreds of participants to understand their smart home access control preferences, such as which device capabilities people felt need restrictions (like "deleting door lock logs") and which types of device capabilities and people could use special contextual controls (e.g., allowing children to control devices only when parents are around). These survey results provide a valuable basis for future smart home access control designs, but they still only represent a theoretical view of people's preferences. To the best of our knowledge, there have been no direct, *in situ* evaluations of multi-user smart home access controls designs with end users. We aim to close that gap in this work.

3 Scope and Research Questions

Prior work has surfaced many multi-user security and privacy challenges in current smart home systems. However, this body of research lacks concrete design proposals that have been evaluated with end users. We aim to advance our understanding of this space.

We focus in this work on generally functional multi-user households, rather than on explicitly adversarial situations (e.g., domestic abuse) or cases where users do not belong to a household together (e.g., Airbnb-style rentals). Understanding and designing for these cases is also critical, but different (and significant) challenges exist in designing systems that are resilient to motivated adversaries with malicious intent, elevated privileges, and physical device access [23]. We discuss the ways in which our work may address — but also falls short of addressing these challenges, in Section 7.4.

Yet prior work has not answered the question of how to design multi-user smart homes for "typical" households; thus, in this work, we seek to answer two research questions:

RQ1: How should a smart home be designed to address multi-user security and privacy challenges (in generally functional households)? What design principles and concrete features may help mitigate tensions and disagreements among otherwise cooperative (e.g., non-abusive) co-habitants that stem from multi-user security and privacy issues?

RQ2: What security and privacy behaviors and needs do these smart home users exhibit in practice? Prior work has provided some understanding of users' security and privacy preferences in the smart home, like preferences for access controls [16], or examples of undesirable situations [14, 41]. However, these preferences could conflict with other priorities, such as utility and convenience. We ask: when presented with a smart home with more advanced security and privacy features, how do people (in non-adversarial households) use them in practice? Do users' security and privacy related be-

haviors differ from their stated preferences? Do our initial design principles match their needs?

To answer these research questions, we take a two-part approach. First, we design and implement a multi-user smart home interface, based on design principles (Section 4.1) that we distill from prior work. Second, we conduct an in-home user study using our prototype, to evaluate whether these design principles meet user needs in practice, and to improve our understanding of users' behaviors given improved multi-user security and privacy features in a smart home.

4 Prototype Design and Implementation

To support the investigation of our research questions, we prototyped a mobile application for controlling smart homes that provides multi-user security and privacy features such as access controls, designed for households in which members are generally motivated to cooperate. We now describe the guiding design principles for our prototype.

4.1 Initial Design Principles

We developed our prototype based on lessons from prior work, which suggested that the following design principles may be important for multi-user smart homes:

Access Control Flexibility. Prior work [16] has suggested that smart home access control and authentication systems should be flexible enough to support a wide variety of use cases, people, and types of relationships that exist in homes. We aimed to support a variety of relationships, like couples, roommates, children, guests, and domestic workers, and also different contextual factors, like location. These factors can be combined to create the policy that suits the user.

User Agency. Prior work [14] found power imbalances among smart home users that reduce the agency of users with less (technical or interpersonal) power. We aimed to support a feeling of agency for all users in the smart home, by making the smart home more accessible and discoverable. For example, for access controls, our prototype allows people to "ask for permission", rather than to be locked out entirely. We aim to make smart home functionality more discoverable, by showing users which devices are nearby and accessible. We also aimed to simplify the process of on-boarding new users.

Respect Among Users. Prior work has surfaced significant potential for tensions and conflicts among users of a smart home (e.g., [14,41]). We aimed to encourage respectful usage of the smart home by minimizing conflict points: for example, making it harder for one user to remotely control or automate devices in a way that would surprise or disturb another.

Transparency of Smart Home Behaviors. Prior work suggests that smart home automations and apps may malfunction or act maliciously (e.g., [11,34]), violate the privacy of unaware users (e.g., [41]), or confuse users who did not configure them. When smart homes are used for domestic abuse, abusers have harassed victims with remote control, masking it as automatic behavior [3]. We aimed for the smart home to transparently surface its behavior to all people in the home (realizing that there may be privacy implications, as we discuss below), especially when people are remotely controlling it, or when an automation/third-party app is acting on its own.

4.2 General Design Description

We designed a mobile application that allows multiple users to control their smart home devices. In terms of threat model, we assume that the control application and the underlying smart home (SmartThings, in our study) are trustworthy and uncompromised. We assume that third-party smart home automations or applications may be buggy or compromised, but our design does not aim to prevent such issues. We assume that users may use or configure the smart home in ways that are undesirable to others in the home, though we focus on cases in which this behavior is accidental or mildly malicious (e.g., "trolling"); we do not attempt to defend against a determined, malicious adversary (e.g., an abuser).

The basic interface of our app is similar to other mobile apps for controlling smart homes (e.g., Samsung SmartThings). The main view of the app displays a list of devices and their current status (Figure 1a). Devices can be organized by room for convenience. The state of a device can be adjusted by tapping its status, and tapping its name reveals options for access controls and notifications (described below).

We aimed to simplify the process of onboarding additional users, towards meeting the "user agency" principle. The first user must create an account with a username and password, but they can add other users by scanning a QR code on the new user's phone. These additional users do not need a login, instead using public key authentication tied to their device.

4.3 Access Controls

Towards meeting the "access control flexibility" and "respect among users" principles, we designed access controls for accessing device capabilities, based on access control preferences and use cases surfaced in prior work (e.g., [10, 16]).

Role-Based Access Control. Each household member has a separate user account. Users can be restricted from using a device via the 'Allowed Users" setting (Figure 1b). Users are also assigned to roles (admin, child, guest). Only admins have the ability to make configuration changes: changing access control policies, adding new users, organizing the devices.

Location-Based Access Control. Users can also be restricted from controlling device capabilities if they are not physically near the device, or not at home, using the "Remote Control" permission (Figure 1c). This access control can be

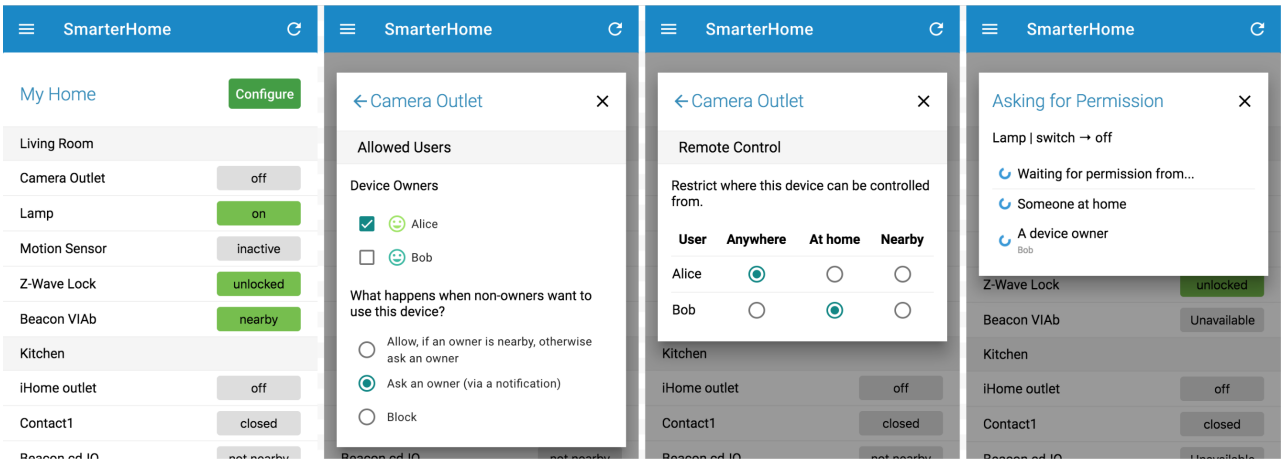


Figure 1: **Access Control UI.** From left to right: (a) The main interface for controlling devices. (b) Interface for setting access controls on devices, by role, and options for reactive/supervisory access control. (c) Interface for setting location-based access controls on a device, for each user. (d) Reactive access control prompt: what users see while waiting for approval.

set per-user, to accommodate use cases like only allowing guests and domestic workers to access smart home devices while in the house. It could also be used to promote respect among users by preventing them from remotely controlling devices like lights when other people are in the room.

Supervisory Access Control. Access controls are in some ways antithetical to user agency. For example, parents may want to use parental controls to keep children from causing trouble, but may not want to block children from using the smart home at all times, like when the parents are at home and are able to supervise. To serve this potential use case, we implemented supervisory access control (first proposed by He et al. [16]): if a user is restricted from controlling a device, they can still be permitted to control it if another (authorized) user is nearby (Figure 1b).

Reactive Access Control. Access control policies based on role and location could be too rigid for every situation. There may be occasional edge-cases where it does not make sense to enforce a policy. Towards the principles of increasing flexibility and supporting user agency for restricted users, we implemented reactive access control [10, 24]. If a user attempts to access a capability they do not have permission to use (Figure 1d), the app will ask a more privileged user for permission in real-time, by sending a notification to asking them to approve or deny the request (Figure 2c).

4.4 Activity Notifications

Towards meeting the "transparency of smart home behaviors" principle, i.e., to make it more transparent when the smart home is being remotely controlled, or controlled by automations and apps, we designed notifications that alert users when the states of home devices change. Each notification displays the name of the device, the change in state, and the user or process responsible for causing the change (Figure 2).

We chose to use notifications over other designs that focused on visualizing automations and events in-app [5, 26], to explore a different point in the design space. Rather than having users navigate to a particular interface when motivated to investigate activity in their smart home, we hypothesized that real-time notifications could provide information in a more timely and relevant manner.

Because the number of notifications from the smart home could be overwhelming, we allowed users to disable notifications on a per-device basis, or to only receive notifications from physically nearby devices.

4.5 Discovery Notifications

Prior work (e.g., [14, 41]) suggests that one challenge with multi-user smart homes is that less technically savvy or engaged users may struggle with accessing smart devices. Thus, towards meeting the "user agency" principle, we wanted to make it clear which smart devices were nearby and could be actuated, especially for novice users. We designed a persistent notification which displays the status of nearby devices, and includes action buttons to toggle those devices (Figure 2b). This design makes devices that are nearby (and potentially relevant) accessible without needing to open the app. We designed it to be minimally intrusive — the notification is silent and is minimized at the bottom of the notification tray.

4.6 Implementation

We implemented a prototype mobile app with these features for Android, iOS, and web, using the Cordova framework. Rather than implement our own smart home controller that

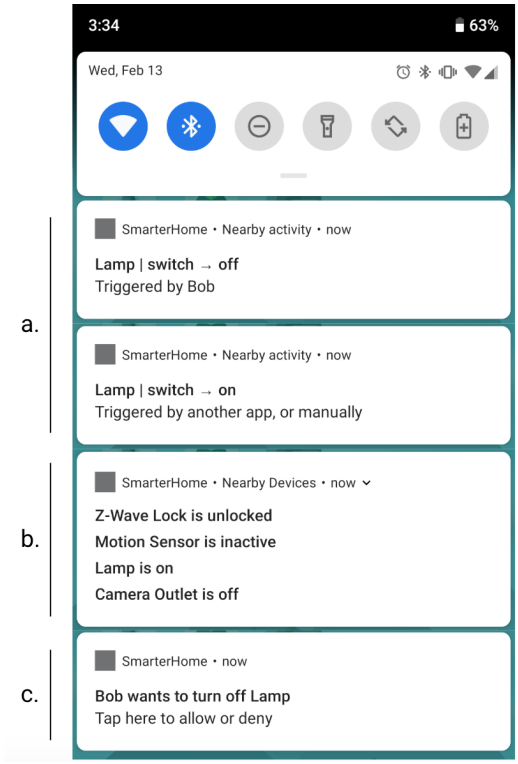


Figure 2: **Overview of Notification Types.**
(a) *Activity Notifications*. When an event occurs in the home, this notification shows the name of device, capability being changed, and who or what caused the change.
(b) *Discovery Notifications*. Persistent, low priority notification that shows nearby devices and their current state; can be expanded to reveal action buttons for controlling devices.
(c) *Reactive Access Control prompt*. Appears when another user asks for permission to use a restricted device capability.

interfaced with hardware devices directly, our prototype connected to devices via the Samsung SmartThings API. Participants set up their smart home devices using SmartThings, and then used our app to control the system. Our prototype did not support automations and third party apps — users accessed this functionality through the SmartThings app. Our prototype consisted of 10257 lines of JavaScript, CSS, and HTML.[1]

Proximity Sensing. To enable room-scale proximity-based features (location-based access controls, proximity-scoped notifications), we incorporated Bluetooth Low Energy beacons into our system. Beacons broadcast an ID that can be scanned by modern smartphones that support Bluetooth 4.0+. Users register physical beacons in our app using an ID printed on the device, and then assign it to a room in the app. When a user's phone detects the beacon, the app infers that the user is near the devices in that room. We chose beacons as our proximity sensing solution out of convenience: they are supported by all

[1]The source code and a demo of the prototype are available at `https://github.com/UWCSESecurityLab/smarter-home`

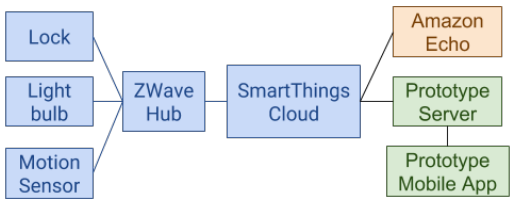


Figure 3: **Prototype Architecture Diagram**. We use the SmartThings API to communicate with smart home devices.

modern Android and iOS devices. However, our design does not require a specific proximity sensing technology; others such as WiFi or ultrasonic sensing would work as well.

SmartThings and iOS Limitations. Due to the limitations of the SmartThings API, activity notifications cannot attribute changes in home state to particular third-party apps, automations, or manual actuation of devices. For state changes in these categories, our implementation only displays "Triggered by an automation or manually". Discovery notifications were only implemented on Android, as the iOS notification center does not support persistent, low priority notifications.

5 User Study: Goals and Methodology

Our prototype allows us to study the research questions we set out in Section 3. To do so, we recruited seven households in the Seattle metropolitan area to use our prototype to interact with their smart homes for a month-long period. We conducted studies between October 2018 and January 2019.

User Study Goals. Our goals in conducting the user study were two-fold, corresponding to our two research questions. First, we aimed to evaluate how participants used and reacted to the specific multi-user smart home features (and corresponding design principles) we implemented in our prototype. Second and more generally, we aimed to understand the multi-user access control and other needs and behaviors of end users, grounded in the use of a specific prototype in real homes.

Our specific evaluation questions, paired with the design principles our prototype intended to embody, included:

1. *Access Control Use Cases:* Is our current combination of access controls sufficient for users' desired access control use cases? If not, what use cases are we missing?
2. *User Agency and Respect:* We envisioned that location-based and reactive access controls could be used to mitigate conflicts and tensions over controlling the home. Can we observe this in practice?
3. *Transparency of Smart Home Behaviors:* We envisioned that notifications could improve users' mental models of smart homes, which would help with understanding privacy implications; and also improve security by creating a simple mechanism for auditing automations and apps. Do notifications provide these benefits to users in

practice? Conversely, do notifications harm privacy by revealing one person's activity to other people?

Study Overview. We conducted a month-long *in situ* user study in the homes of participants. We recruited households in the Seattle metropolitan area. We provided a Samsung SmartThings smart home to households that did not already own smart home devices, or integrated SmartThings with the smart homes of households that owned an existing system. We collected qualitative data about participants' previous experiences with smart homes and feedback on our prototype through interviews, experience sampling, and log data.

Recruitment. We recruited seven households, containing 19 participants who actively participated. Participating households were recruited via Facebook ads, targeted at people interested in smart homes and home DIY projects. People who clicked on the ads filled out a short survey including information about their household composition and interest in smart homes. We did not require participants to own any smart home devices prior to the study. We conducted a screening call with participants that met our criteria to collect additional information. We selected participants who lived within a 45 minute radius from our homes (so that it was feasible to make an in-home visit), and we aimed for a variety of multi-person household compositions, including roommates, families, and couples. Participating households are summarized in Table 1.

A limitation of our recruitment strategy and study design is that it introduces self-selection bias: our participants were likely to be living in generally cooperative households, with one or more technology early adopters. We discuss this, and other limitations, further in Section 7.5.

Initial Interview. We made an initial visit to participants' homes to conduct a semi-structured interview about their existing experiences and attitudes towards multi-user smart home security, privacy, and usability issues (see Appendix A).

Following the interview, we assisted with the setup of any devices if needed, and then we set up our prototype app. We guided them through app installation because it required using the developer mode in SmartThings, which was cumbersome and not representative of a typical install experience for commercial apps. We also assisted participants in adding other household members, to ensure that we could study multi-user interactions (rather evaluating the onboarding barrier).

We also walked through the access control and notification features of the app, and collected their initial impressions of the features. To counteract participant response bias [8] we stressed that we were testing an imperfect prototype, and that we wanted honest, negative feedback on things that were not useful or usable. We used some participant feedback from this stage to iterate on our implementation and push updated features to participants throughout the duration of the study.

Daily Usage. Participants then used the app for 3-4 weeks during their daily lives. During this period, the integrated experience sampling interface in our app prompted participants to provide feedback or to share anecdotes about multi-user interactions in the home. We also collected log data about how users set up access controls, permissions, and notifications.

Exit Interview. At the end of the usage period, we conducted a phone interview with each household. In this semi-structured interview we collected specific feedback about their experience using (or not using) the access control and notification features in our prototype. We also followed up on any interesting data from experience sampling or logs. A list of interview questions is available in Appendix B.

Compensation. Participating households were compensated $250 over the course of the study, in installments. Participants could keep the provided smart home devices after the study, or return them for the equivalent cash value.

Ethics. The study was approved by the University of Washington's human subjects review board. Participants had to be age 18+ to consent to participating; household members under 18 could participate with verbal assent and approval from their parents and guardians. We had approval to collect incidental data via the smart home on children who declined to participate or were too young to actively participate.

During the study, we experienced a security breach due to a firewall and database misconfiguration, resulting in the possible exposure of hashed passwords, log data, and temporary access tokens. Based on access patterns, we believe the data was accessed by port scanners, and not by targeted attackers. We remediated the issue within 24 hours of discovery. We notified our institution's human subjects board, and contacted participants with a description of the issue, protective steps like changing matching passwords on other sites, and the option to opt out of the study. No participants opted out.

Analysis. We transcribed and analyzed 633 minutes of content from the 14 initial and exit interviews. We analyzed the interviews using a collaborative qualitative coding technique. First, two researchers read over all of the data and developed a codebook, using descriptive codes like "access control: use cases", "relationship: guests", "notifications: too noisy", and "access control: trust/respect" (see Appendix C for a full list). Two researchers independently coded two interviews, and then met to resolve differences and clarify ambiguities in the codebook. Then, one researcher coded the remaining interviews based on our revised understanding of the codes. We used a custom code aggregation tool to help identify patterns and extract higher level themes across interviews.

6 User Study: Results

We now present the findings from our user study, including direct feedback on the features implemented in our prototype, and general findings about participants' desired features and use cases, surfaced by their concrete experiences with our prototype and the smart devices in their homes.

Participants		Gender	Age	Devices	Household Info
H1	H1A	25-34	F	Lock*, motion sensors*, contact sensors, thermostat*, security camera*, lights*, Amazon Echo*	Family with two non-participating children (0-6), living in house
	H1B	35-44	M		
H2	H2A	25-34	M	Lights*, Amazon Echo*, contact sensor, door lock (not connected)*	Couple, living in house
	H2B	25-34	F		
H3	H3A	25-34	F	Lights, contact sensor, motion sensor, power outlet	Roommates, living in apartment
	H3B	25-34	F		
	H3C	25-34	F		
H4	H4A	25-34	F	Lights, contact sensor, power outlet, Amazon Echo*	Couple, living in apartment
	H3B	25-34	M		
H6	H6A	35-44	F	Lights, contact sensor, door lock, Amazon Echo*	Family with 2 children (one aged 7-12), living in house
	H6B	45-54	F		
	H6C	13-17	M		
H7	H7A	18-24	F	Lights, contact sensor, motion sensor, power outlet, Ring video doorbell, Amazon Echo*	Roommates, living in house
	H7B	18-24	F		
	H7C	18-24	F		
H8	H8A	45-54	F	Lights, contact sensors, security cameras*, Amazon Echo*	Family with 2 participating children, one non-participating child (7-12), one non-participating relative (13-17), living in house
	H8B	45-54	M		
	H8C	18-24	F		
	H8D	13-17	M		

Table 1: **Summary of Participating Households.** Some children were too young to actively participate in the study. Asterisks (*) indicate devices households owned prior to the study.

6.1 Desired Access Control Use Cases

We begin by exploring the situations where participants *wanted* multi-user access controls, and what form of access control mechanisms participants wanted. In some cases, our prototype was able to fulfill participants' goals, and in others, the ability to explore concrete access control features in the context of their own home evoked hypothetical policies that they felt would better suit their needs.

Location Restrictions for Visitors. H1A wanted an access control setting that would allow visitors like guests and domestic workers to be able to access and control the devices in her home, but only while they were physically present.

> I don't want the nanny, who's here all day — I trust her, obviously, or she wouldn't be with my kids — but at the same time, like I don't necessarily need her to be at her house, being able to control the lights at my house. ...if I have guests coming into my house, I'd like them to control automations, but... I certainly don't want them having admin control. I'd prefer to have them to have geofenced control. (H1A-Initial)

At the time, our prototype's location-based access controls did not quite meet her requirements, because it could only be applied as a blanket policy for all users of a given device. Based on this feedback, we updated the prototype to support location-based access controls both per-user and per-device.

Preventing Configuration Changes. Some participants were concerned about other family members accidentally making changes to access control policies, automations, or device configuration. H1A recalled when they set up their smart home, H1B (her spouse) caused confusion by accidentally pairing some devices multiple times. As a result, H1A set H1B at the child privilege level in our prototype, which prevented him from configuring access controls and rooms.

H8A did not want her children to either change or override the existing automation for the porch light, which turned the lights on automatically at night for security purposes, nor did she want them to be able to change access control policies. As a result H8C/D were set at the child privilege level in our app (and were also not added to the native SmartThings app, from which the automations were created).

Parental Controls for Device Usage. Parents in our participant sample expressed interest in placing restrictions on children to prevent mischief or other undesired uses of devices. For example, H1A and H8A wanted to restrict their children from turning on/off security cameras. However, participants did not use our prototype's features for restricting access to any devices in practice, for reasons we discuss below.

A parental control goal that we did not anticipate was that H1A and H8A were more interested in using the smart home to regulate screen time, e.g. blocking internet access at certain times, and using a smart power outlet to turn off the TV.

Devices in Private Rooms. The roommates of H3 placed smart light bulbs each of their bedrooms, and set an access control policy so that only the room's owner could control the lights. They reported that it was "comforting" and a "good

feature to have" (H3C), but that in practice, they never encountered the access controls because they were respectful of each other and did not ever attempt to control another person's lights. (We discuss similar cases of social norms obviating the use of technical access controls below.)

Preventing Remote Access for Media Devices. H4A/B expressed interest in location-based access controls for their Amazon Echo, based on past experiences where one of them accidentally changed the audio that was playing from outside of the room or house, due to confusion over whose Bluetooth device or Spotify account was playing. We did not see similar interest in location-based access controls for other device types — perhaps because unlike lights or locks, which are useful to remotely control for security and energy saving purposes, media devices are only useful to the people physically in the room.

Access Controls for Voice-Controlled Devices. H8A became aware that their Amazon Echo could be used to bypass the access controls and authentication of our prototype (see Section 7.3 for more detail). In one instance, she used this to allow her mother-in-law to access the smart home without installing our prototype. However, she also wanted the Echo to authenticate users by voice, so that they could use access control policies for to their youngest son, who was too young to have a phone but could control devices via the Echo.

6.2 Reasons for Not Using Access Controls

Based on findings about multi-user smart home tensions in prior works, we expected that households would use our access controls, for at least some of the potential use cases outlined in our design principles (Section 4.1). However, in general, we found that the access controls we implemented did not fit with the participants needs and use cases.

We analyzed participants' usage logs, and found that while most households experimented with using access control policies in the first few days after the initial interview, most of them quickly settled on the least restrictive access control setting, not continuing to use location-based, role-based, or supervisory access controls to restrict access to devices. The only household that kept any access controls enabled was H3, a household of roommates who enabled per-device role-based access controls on the lights in their private bedrooms. However, none of the roommates ever attempted to violate these access controls (i.e., tried to turn on or off each others' lights).

Given this limited long-term use of access control features in practice, we thus focus on our qualitative interview data, to dig into the reasons *why* participants did not use the access controls more than they did. Our findings surface several reasons that are more fundamental than simply reactions to our specific implementation — i.e., reasons that participants may not have used *any* access controls, regardless of design.

Social Norms, Trust, and Respect. The most common reason participants cited for not setting access controls was trusting each other enough that they were not concerned about device misuse, relying instead on established household and interpersonal norms. We observed such trust and norms among relatively equal relationships, like partners and spouses:

> No, we didn't turn [remote control restrictions] on either... We both wanted full permissions to do anything whenever, we weren't worried about the other. I had no concern that H2B, from not nearby, would turn off the lights. (H2A-Exit)

We also observed trust and norms among roommates:

> I think we're all pretty respectful and we wouldn't turn on and off each others' lights. (H7A-Exit)

And even with children:

> If [H6C] were a different kid, I would probably leave [remote control] turned off for him. But for him, it would be useful, I would turn it on for him. ...I think it's going to be very specific to who is using it, and having the option is important, but he's just very responsible, so it could've been handy for him to be able to do something from school, like turn on and off lights. (H6A-Initial)

Participants mentioned similar social norms about multi-user privacy. For example, H1A and H8A/B were aware that it was possible to eavesdrop using devices like the Amazon Echo or security cameras, but chose not to do so.

Interference with Other Functionality. Particularly with location-based access controls, participants often felt that the available access controls were too restrictive and prevented them from accomplishing other goals. In our initial design, we expected that location-based access controls could serve a number of goals, like access control for guests, or preventing mischief or inconsiderate use of remote controls. However, multiple participants wanted unfettered remote control access, particularly for lights, because it was convenient.

> I think a big thing for us was in case we forgot to turn off the lights or something, that was like the appeal, to turn it off remotely. (H7A-Exit)

> Like the times when we would both need access to turn off the light we forgot to turn on, were more frequent than any need to restrict us from being able to remotely control it. (H4B-Exit)

In other words, at least for the smart devices our participants had, the convenience for all members of the household to be able to exercise remote control outweighed any concerns about intentional or accidental remote misuse.

Lack of Concern About Devices. Participants did not feel concerned enough to use access controls for certain types of devices, or for devices in certain locations. For example, participants did not feel that smart lights were sensitive enough for access control (but did want restrictions on more sensitive devices, such as cameras, for guests and children).

We cannot say whether participants would have used more access controls for more sensitive devices — since we allowed our participants to select their own devices, their *a priori* threat models likely influenced their devices selection (i.e., selecting devices they were comfortable having in their home). We discuss this issue further in Section 7. Moreover, we note that these limited multi-user concerns were consistent with participants' overall smart home related threat models (likely due to self-selection bias). Though some participants were aware of potential risks such as password compromise, vulnerabilities in wireless protocols, data collection by companies, or lost phones, they did not consider these risks to overwhelm the utility of the smart home.

Some participants also did not find it necessary to control access to devices located in household common spaces, like locks and lights — again showing physical-world household social norms reflected in the configuration of the smart home.

6.3 Limited Utility from Activity Notifications

We found varied use of activity notifications among our participants. From our log data, we observe that 14 participants had activity notifications on at all times for all devices, while 4 participants used a combination of settings: on, off, and proximity scoped for various devices. This data suggests that proximity scoping provided utility for some participants. (One child participant did not have the app installed.)

But having notifications enabled does not necessarily mean that participants found them useful; we now dig further into our qualitative interview data to understand whether and how the notifications were useful to participants. Our participants found notifications useful for a few specific use cases, like home security and sanity checking their smart home automations. However, we did not find much evidence that our notifications provided benefits for transparency and agency.

Monitoring and Home Security. Participants found notifications to be most useful for home security and monitoring purposes. H1, H6, H7, and H8 used our prototype's notifications in conjunction with sensors on their exterior doors and windows, to passively monitor their home's security. H3C used notifications to monitor devices in their bedroom, to check if others were entering the room.

Proximity Scoping for Activity Notifications. While participant H8D found proximity scoping useful, as she did not want to be notified about devices while away from home, other participants said that the feature would be more useful if they could be notified only when *not* at home — either as a home security measure (H8A), or because they could already tell when their devices changed while at home (H4B).

Confirmation of Home Behaviors. Some participants found the notifications to be comforting because they confirmed that both people and automations were behaving as expected.

> It was nice to know it was at that point in the day, and really what I had it set on were essentially the lights to come on and go off at appropriate times, and so it was a notice that, yes, today is progressing as it should. (H6A-Exit)

Desire for Contextual Notifications. Our activity notifications prompted participants to propose more advanced, context-dependent notifications that would be more useful to them. For example, H3C suggested notifications which would suggest turning off the lights to save energy. H6A wanted more intrusive notifications when something incorrect happens (e.g., a window is open when it should not be).

We were not able to test whether notifications would be helpful for identifying actions caused by specific automations, because limitations of the SmartThings API did not let us see *which* automation caused an event to happen. None of our participants mentioned encountering a situation in which they wanted more specific information about provenance.

Quick Access via Discovery Notifications. Most participants did not notice or see discovery notifications. (Unfortunately, persistent notifications of this sort are not supported on iOS, and few of our participants were Android users.) One participant, H8D, was interested in these notifications, but for convenience, not device discoverability, as it allowed him to toggle the state of the device without opening the app.

Limited Concern about Privacy. No participants reported that the notifications affected their sense of privacy, nor that they changed their behavior as a result of knowing that notifications would be shown to others. Participants also did not report learning new information about others via notifications.

Notifications Were Overwhelming or Not Useful. For some participants, the notifications were annoying and overwhelming. H1A said she just did not care when other people, like her husband and nanny, used devices. H7A complained about redundant notification: each time someone walked through the front door, their doorbell and contact sensor would both trigger notifications, resulting in four notifications.

Other participants said that the notifications were not useful when at home, because it was information that was already apparent. Participants in H3 and H4 lived in small apartments, and could naturally observe all of the information from the notifications (e.g., the sound of others walking around and the glow of lights in other rooms). And H7A said that their dogs already notified them when people were at the front door.

6.4 Usability and Configuration Complexity

Hands-on experience with our app revealed that the complexity of access controls and other smart home features were adversely affecting the usability of the system. The complexity came from both the granularity of the settings, and the number of different devices managed by the home.

Complexity as a Barrier to Access Control Use. While we

aimed to make our prototype's access controls as easy to understand as possible, the inherent complexity in the matrix of options may have still been too much of a barrier for novice users to configure them. For example, usability may have been an issue for H8A/B, where both expressed interest in setting various access controls during the feature walkthrough in the initial interview, but did they did not ending up using them. When we asked about other goals they might have for access controls and the smart home in general, H8A said:

> It interests me, but you have to think it through, what you want to do, how it would benefit you... part of the Smart Things is you're taking on a bit of a responsibility, getting it set up, getting it working, it's kind of like getting a new computer, but there's a bit of the downside, you have more options but it's complicated. (H8A-Exit)

Design Complexity from Combinations of Settings. During the study, participants requested more fine-grained options for the access control and notification features. Based on this feedback, we iterated on the implementation of our prototype and released updates. However, we struggled with adding these features, as each additional access control dimension compounded the complexity of the interface.

One example was for location-based access controls. Initially, these access controls were set per-device. However, H1A and H8B wanted to set these access controls per-user in addition to per-device, so that they could restrict their nanny and kids (respectively), but not themselves. To fulfill this request, we had to surface more options ($3n$ options per device, where n is the number of users, instead of 3 options per device). As another example, if we wanted to add toggles for supervisory and reactive access controls to location-based access control when users are not nearby and try to use the device, there would not be enough space to display these options without an additional submenu, making it more laborious to set policies for each user and device (see Figure 1c).

Usability is fundamentally in tension with the desire to support access control flexibility and surface all of these options to users - we discuss this issue further in Section 7.

Displaying Access Control Policies. Participants remarked that it would have been helpful if the main device control page (Figure 1a) surfaced each device's access control policies. Living in a home with 14 devices, H1B struggled with identifying and remembering which devices had access controls:

> Seeing the list of all of the devices in the room, and knowing which ones he could click, and which ones he couldn't, and which ones had to ask for permission... (H1A-Exit)

H1 suggested an interface for favorite devices (a feature supported by Vera), while H3 suggested that devices that you did not have access to would simply be hidden.

Install Barrier. We attempted to make the install process as painless as possible for our app, implementing a QR-code passwordless public key authentication system for additional users. However, even this barrier was too much for some users — H1A did not want to go to the effort for adding their nanny (despite stating the desire to set access controls for her), and H8A did not feel confident in being able to add her mother-in-law without our guidance. As a result, these household members were either shut out of the smart home, or accessed it via other means (i.e. Amazon Echo), bypassing our prototype's access controls and losing access to notifications.

7 Discussion

7.1 Lessons on Smart Home User Behaviors

Based on our *in situ* prototype evaluation, we surface lessons about users' security and privacy behaviors in smart homes, including how they interact with concrete security and privacy features in practice, and how our observations of actual behavior align with user preferences identified in prior work.

Limited Usage of Access Controls. Though our participants mentioned multiple use cases for access controls in our initial interviews, such as restrictions on guests, domestic workers, and children, in practice, few of them made use of the access controls we implemented. There are several possible reasons for this. In two cases, usability was a barrier; one household was discouraged by the complexity of the access control interface, and the other by the difficulty of onboarding guests. More commonly, participants did not have a strong need to use access controls, either because they were unconcerned about restricting access to mundane devices, or that existing social norms and trust in their household checked against bad behavior. Lastly, some participants chose not to use access controls because it would interfere with other desired functionality, like occasionally allowing children remote access.

These findings suggest that while at first glance there are many user goals that could be achieved with access controls, there are only a few specific use cases that access controls are well suited for in practice, like limiting access for domestic workers. But for other use cases where users have weak or subtle preferences, access controls can be too rigid, complex, or simply not useful, even with reactive and contextual mechanisms, such as parental controls.

Importance of Social Norms. Among our study population, we observed that in circumstances where prior work has shown the potential for multi-user conflicts and privacy issues, our participants often did not experience these problems due to the norms of interpersonal behavior in their home. For example, children were trusted to follow rules, roommates respected each others' spaces, and people were not concerned about information revealed by the smart home when it matched their household's privacy norms. This finding suggests that in generally cooperative households, multi-user security and privacy issues may be able to be addressed in

part by cultivating good norms around usage of the smart home. We discuss this topic further below.

Acceptance of Security and Privacy Tradeoffs. As we expected from prior work [41], participants were willing to accept (multi-user) security and privacy risks posed by usage of the smart home because of the convenience and utility it provided. Participants often explicitly mentioned the tradeoff between convenience and privacy, when asked about their concerns about data privacy. H8 decided against setting up access controls (for parental controls) because the smart home would be less convenient for the household, and H1 decided against using access controls for their nanny because the setup process would be inconvenient. While this finding is not new, it re-emphasizes that when designing security and privacy features for smart homes, these features must work with, and not limit, users' primary use cases for the smart home.

7.2 Revisiting our Design Principles

In Section 4, we proposed a set of design principles which we hypothesized could help address multi-user security and privacy issues. Based on the insights provided by our evaluation and user study, we revisit these principles:

Access Control Flexibility: Important But Not a Panacea. Our results suggest that while access controls might not be suitable for satisfying all user preferences, the flexible access control mechanisms we implemented, such as location-based access controls and per-device ownership, can help users in clear-cut use cases, like guest access. However, we also found that increasing flexibility also increases the complexity of the interface, and as we discuss below, a challenging open question remains how to support such a complex array of options in a usable and useful way.

User Agency and Respect: Dominated by Social Norms. Contrary to our initial hypotheses, we found that our participants relied more heavily on household social norms to support user agency and minimize conflicts than the access control, notification, and device discovery features we designed in our prototype. While such norms would not exist in abusive or adversarial households, for generally cooperative households, we propose a new research and design question that we discuss further below: how can a multi-user smart home be designed to *support and leverage* positive social norms, rather than existing alongside or supplanting them?

Transparency of Smart Home Behaviors: Inconclusive. Our results suggest that smart home transparency features did not provide significant benefits for our participants, in terms of our design principles (user agency and respect among users). Participants were generally indifferent to the information provided by the activity and discovery notifications, though some participants found them to be useful for other reasons: home security and verifying that their automations were working. However, our investigation is not sufficient to conclude that transparency might not be valuable in other contexts, e.g., with cameras or voice assistants, or among people with more adversarial relationships. It is also possible that our implementation of transparency via notifications was not effective, and that another design, like calendar [26] or dashboard [5] interfaces, would provide different reactions.

7.3 Design Recommendations and Challenges

Based on our findings and revised design principles, we surface several design recommendations for multi-user smart home systems, particularly for platforms that can orchestrate access controls and features across all devices of the home.

Support Smart Home-Specific Access Control Needs. Our study highlights a number of use cases for access controls that appear to be common in smart home settings, including restrictions on visitors, and different policies for different rooms. To support these use cases, we recommend that smart home platforms support the following primitives: (1) Location/proximity-based access control, for handling guests and domestic workers, as well as restricting access to media devices, (2) Time-based access control, also for guests, (3) per-device roles for private rooms, (4) and per-user roles, for limiting access to device and access control configuration.

Simplify Access Control Configuration. A system with all of the above access control mechanisms will run into serious usability challenges if it simply surfaces a large matrix of multi-dimensional per-user, per-device options. In fact, such complexity risks increasing the access gap between the smart home's primary user and others with less technical or interpersonal power. It could also put the use of access control out of reach for novice users. Moreover, complex policies could introduce errors or conflicts between access control rules.

A good first step towards simplifying smart home access control could be to use sensible defaults based on data on people's access control preferences, as suggested by He et al. [16]. However, our results suggest that individual factors, social norms, and conflicting use cases may cause household needs to diverge from these broad preferences, so it is still important to have a usable configuration interface. However, it is not clear what kind of interface would be effective in this context. In Section 7.4, we recommend that future work investigate systems for simplifying access control configuration in smart homes, such as natural language-based policy creation.

Incorporate Voice Assistants into Access Control Systems. A major limitation of our prototype was that our access control system could be (intentionally or unintentionally) bypassed by sending a command through a voice assistant, such as the Amazon Echo. This is likewise a challenge for current smart home platforms: in platforms like SmartThings, voice assistants and other third party apps like IFTTT are given unrestricted access to smart home devices via OAuth integrations. Additionally, current voice assistants do not explicitly

perform voice recognition, so a smart home would not be able to identify who is issuing a command. In order for access controls to be consistently applied, voice assistants should support voice-based authentication, and voice assistant manufacturers should work with smart home platforms to develop a federated access control system. This is particularly important as adoption of voice assistants increases and they become a popular way to interact with smart homes.

Reduce User Onboarding Barrier. The smart home control interface still needs to be made more accessible to users. Even by our best efforts, a mobile app was too much to ask for some participants to install without our direct assistance and urging. If a smart home control system provides perfect security and privacy features that are locked up in an app that not all household members install, the benefits of these features will be limited. And in worst-case scenarios, if a household members cannot gain access to the smart home, it can enable domestic abuse by those with control. We suggest several potential approaches to address this issue:

One approach is to lower the installation barrier by making a mobile web version of control interfaces. In our experience, Web APIs were sufficient for all functionality, except for Bluetooth beacon scanning for proximity sensing — though browsers intend to implement this feature in the future [7].

Another approach is to further simplify user authentication. Our prototype required only a QR code rather than a username/password for subsequent users. We suggest exploring even more radical approaches, such as not requiring *any* traditional authentication to use the smart home, and instead granting basic smart device control functions to anyone in physical proximity (just as someone with physical access to a manual light switch can toggle it).

7.4 Directions for Future Research

Our work also suggests research questions that we encourage future work to investigate:

Study and Design for Positive Household Norms. We observed in our study that in cooperative households, social norms were effective at mitigating multi-user security and privacy issues, sometimes more so than the features we implemented in our prototype. Rather than trying to provide features that play the same role as these social norms, like location-based access controls for preventing inconsiderate use of remote access, we suggest (1) studying households that exhibit positive social norms around smart home usage and (2) designing and evaluating smart home systems that encourage the development of these norms in generally cooperative households. Based on the results of our study, we propose a few design "nudges" that could potentially instill better behaviors in smart home users.

First, rather than asking users to design access control policies around considerate usage, smart home platforms could automatically detect commands that are potential norm violations, and then ask the user "Are you sure?", including a reason for why the command might violate a norm. For example, this prompt could be triggered when attempting to control devices in another user's private bedroom, or when remotely controlling devices that would impact other people physically present. Such a prompt could encourage users to think twice about disturbing others, while still allowing for seamless access if necessary.

Another type of nudge could promote user agency: during the setup of a smart home, the app could encourage the person installing the smart home to involve other occupants in the setup process, including encouraging and even guiding the setup of additional accounts and conversations about the different devices, automations, and policies that should be part of the new smart home. How to best design such a conversational guide is an interesting question for future work.

Nudges could also be designed to "scold" users for excessive trolling or other playful behavior, like rapidly flicking lights on and off. While it might be good to allow playful experimentation when the smart home is set up initially, eventually the app could rate limit these behaviors, or display a dialogue box encouraging the user to stop.

While norm-based nudges would of course not protect against users with malicious intent, our study results suggest that promoting positive norms could help reduce friction in the case of generally cooperative households, where conflicts and tension may arise from unfamiliarity with how one's actions affect others in the smart home. Next, we discuss the challenge of designing smart homes for adversarial settings.

Investigate Designs for Adversarial Situations. Smart homes can enable or amplify harms in adversarial living situations, like in households where domestic abuse is occuring, or in homes with Airbnb-style rentals. While some of the design principles we proposed could mitigate some of these harms, such as using notifications to provide more transparency about how surveillance cameras are being used, our prototype would not provide adequate protections against other harmful actions, such as a malicious admin abusing their privileges to deny victims control of the home, or overriding protections against remote harassment that location or role-based access controls could provide. This is a very challenging problem, because some of these security and privacy features are inherently dual use: for example, admin roles and access controls may desirable for parents to prevent children from doing harmful things, but could be used by abusers to exercise power over their victims. A critical but challenging design question for future work is how to design smart home access controls and monitoring that both protects users from abuse, but still enables benign use cases.

Study Transparency Features for Privacy-Sensitive Devices. As discussed above, a limitation of our prototype was that we could not provide activity notifications for privacy

sensitive smart home devices like voice assistants and security cameras, because of the limitations of the SmartThings API. We suspect that surfacing information about when audio and video is being recorded or viewed could change users' perceptions of the privacy risks of these devices, and could help people identify when their privacy is being violated. We propose an *in situ* evaluation of user reactions to a smart home system that notifies people if they are being recorded, or if another user views or listens to a log that they are present in.

Audio/video recording notifications could also be surfaced not just in the smart home, but at a global level with cooperation from mobile operating systems and device manufacturers. Cameras and microphones could emit Bluetooth beacon signals when they are active, so that users could receive notifications whenever they are nearby an active recording device.

Study Natural Language-based Access Control Policy Creation for Smart Homes. During our interviews, we observed that our participants were able to clearly convey their access control preferences and hypothetical policies verbally. Given that these policies are easily comprehensible in natural language, a possible way to simplify configuration is to allow users to craft policies using a natural language interface, rather than menus with drop-down lists and checkboxes. While prior work has found that direct conversion from natural language to policy is possible but imprecise [20, 31, 32], controlled natural language policy creation could be used to constrain the space of usable words and sentence structures. Using a controlled natural language approach, a possible interface could be an autocomplete-style input, which guides users through picking access control mechanisms, possible devices, users, roles, and other conditions. While this approach was found to be relatively usable in a systems administration context [17, 30], future work should evaluate whether it is usable for typical end users in a smart home setting.

Further Study of Automations and Attributions. We were not able to fully study whether notifications could help users with debugging automations, or attributing issues caused by automations and third-party apps, because of technical limitations of our prototype (specifically, that SmartThings does not surface to third-party applications the provenance of programmatic smart device actuations). Other researchers have proposed ways of preventing buggy or malicious behavior by third-party smart home integrations, such as detecting provenance [37] or contextual permission prompts for third-party apps [19]. These research contributions are technically valuable but their usability and utility have not been tested with real end users; we suggest that future work do so.

7.5 Limitations

Though an in-home user study allowed us to study how people used our prototype under realistic circumstances, this study design nevertheless comes with several limitations.

Most importantly, as discussed already, our prototype and user study focused on generally cooperative households, rather than households with adversarial relationships. Since we required consent from all participating household members, our sample is skewed towards households with sufficiently functional interpersonal relationships to agree to participate together in the study. Thus, we were unable to evaluate how our prototype would perform in an adversarial setting, nor did we gain insight into how to design for those settings.

Moreover, our protocol design involved conducting interviews with participants in a group setting, with the entire household. It is possible that participants were unwilling to reveal multi-user conflicts and privacy issues, because it would also reveal these problems to other household members.

Additionally, the devices our participants chose were generally not among the most invasive. This was due both to technical limitations (e.g., our prototype could not integrate with most security cameras using the SmartThings API), and because we gave participants the freedom to choose devices they were comfortable with. While our prototype did not interface with these more privacy sensitive devices, we still learned from participants via hypotheticals about access control grounded in their concrete experiences with our prototype and their past experiences with those devices. Future work should further consider multi-user smart home design in the face of more invasive devices.

Finally, the complexity and cost of an in-home study limited the feasible number of participating households, preventing us from drawing any quantitative conclusions from our results.

Despite these limitations, we believe our study provides valuable insights into how to design multi-user smart home security/privacy features for many (though not all) households.

8 Additional Related Work

Methodologically, our paper drew on a number of other in-home studies of smart homes, from HCI and ubiquitous computing. Most closely related to our work were the design and evaluation of a calendar-based interface for smart home control [26], and of a smart home data visualization dashboard [5]. Other in-home studies in HCI have studied how users interact with commercial smart homes in practice, like general usage patterns and usability [4, 18, 25], setup and configuration [9], and end user programming [39]. Researchers have also studied how users perceive and use privacy sensitive devices like cameras and voice assistants, both in-situ [28, 40], and in interviews or surveys with broader populations [21, 42].

In terms of the security and privacy of smart home devices and platforms, researchers have discovered vulnerabilities in the underlying protocols and technologies (e.g., [2, 15, 29, 38]) and studied the spread and behavior of the Mirai botnet that targeted IoT devices [1]. Other work has analyzed security and privacy weaknesses in smart home platforms that support third-party apps like SmartThings [11]. To address the

risks posed by apps, researchers have proposed and evaluated various defenses, including modifications to trigger-action programming platforms to limit misuse of access tokens [13], restricting apps using flow control [12], using provenance detection to identify anomalies [37], and a contextual access control system to protect against malicious third-party apps [19].

9 Conclusion

Multi-user smart homes face unique security and privacy challenges, such as supporting a wide range of access control preferences, and managing tensions and conflicts between users. Finding the design of current smart home systems to be insufficient for addressing these challenges, and recognizing the gap in knowledge around what designs can meaningfully improve end user experiences, we conducted an in-home user study to investigate possible approaches and solutions. Focusing on generally cooperative (rather than explicitly adversarial) households, we designed a smart home control interface based on design principles of access control flexibility, user agency, respect among users, and transparency of smart home behaviors. We deployed our prototype in seven households in a month-long study to evaluate our proposed design principles, and to improve our understanding of how users interact with security and privacy features in practice. Based on the findings of our user study, we provide design recommendations and identify open challenges for future research. Among our recommendations, we suggest that researchers improve the usability of smart home access controls by developing more usable configuration interfaces (such as natural language policy creation), and design smart home platforms that reduce tensions and conflicts by leveraging and scaffolding positive household norms.

Acknowledgements

We are extremely grateful to our user study participants for making this research possible, as well as our pilot study participant, Greg Akselrod. We would like to thank Christine Geeng, Ivan Evtimov, Kiron Lebeck, and Shrirang Mare for reviewing an earlier draft of this paper. We would also like to thank Tadayoshi Kohno for his feedback in the early stages of this research. We thank Sarah Mennicken for her advice on conducting in-home user studies. Lastly, we thank our anonymous reviewers and our shepherd, Sascha Fahl, for providing us valuable feedback for improving our paper. This research was supported in part by the National Science Foundation under Award CNS-1513584.

References

[1] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou. Understanding the Mirai Botnet. In *26th USENIX Conference on Security Symposium*, 2017.

[2] R. Baldwin. Researcher finds huge security flaws in Bluetooth locks. `https://www.engadget.com/2016/08/10/researcher-finds-huge-security-flaws-in-bluetooth-locks/`, 2016.

[3] N. Bowles. Thermostats, Locks and Lights: Digital Tools of Domestic Abuse. `https://www.nytimes.com/2018/06/23/technology/smart-home-devices-domestic-abuse.html`, 2018.

[4] A. B. Brush, B. Lee, R. Mahajan, S. Agarwal, S. Saroiu, and C. Dixon. Home Automation in the Wild: Challenges and Opportunities. In *SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 2115–2124, New York, NY, USA, 2011. ACM.

[5] N. Castelli, C. Ogonowski, T. Jakobi, M. Stein, G. Stevens, and V. Wulf. What Happened in my Home?: An End-User Development Approach for Smart Home Data Visualization. In *CHI Conference on Human Factors in Computing Systems (CHI)*, 2017.

[6] E. K. Choe, S. Consolvo, J. Jung, B. L. Harrison, S. N. Patel, and J. A. Kientz. Investigating receptiveness to sensing and inference in the home using sensor proxies. In *14th International Conference on Ubiquitous Computing (UbiComp)*, 2012.

[7] Chromium blink-dev mailing list. Intent to Implement: Web Bluetooth Scanning. `https://groups.google.com/a/chromium.org/forum/#!topic/blink-dev/aVxGkVQ2xRk`, 2018.

[8] N. Dell, V. Vaidyanathan, I. Medhi-Thies, E. Cutrell, and W. Thies. "Yours is better!": Participant Response Bias in HCI. In *SIGCHI Conference on Human Factors in Computing Systems (CHI)*, 2012.

[9] A. Demeure, S. Caffiau, E. Elias, and C. Roux. Building and Using Home Automation Systems: A Field Study. In *International Symposium on End User Development (IS-EUD)*, 2015.

[10] Y. Elrakaiby, F. Cuppens, and N. Cuppens-Boulahia. Interactivity for Reactive Access Control. In *International Conference on Security and Cryptography (SECRYPT)*, 2008.

[11] E. Fernandes, J. Jung, and A. Prakash. Security Analysis of Emerging Smart Home Applications. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 636–654, 2016.

[12] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash. FlowFence: Practical Data

Protection for Emerging IoT Application Frameworks. In *USENIX Security Symposium*, pages 531–548, Austin, TX, 2016. USENIX Association.

[13] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash. Decentralized Action Integrity for Trigger-Action IoT Platforms. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[14] C. Geeng and F. Roesner. Who's In Control?: Interactions In Multi-User Smart Homes. In *CHI Conference on Human Factors in Computing Systems (CHI)*, 2019.

[15] J. Granjal, E. Monteiro, and J. Sá Silva. Security for the Internet of Things: A Survey of Existing Protocols and Open Research Issues. *IEEE Communications Surveys Tutorials*, 17(3):1294–1312, thirdquarter 2015.

[16] W. He, M. Golla, R. Padhi, J. Ofek, M. Dürmuth, E. Fernandes, and B. Ur. Rethinking Access Control and Authentication for the Home Internet of Things (IoT). In *USENIX Security Symposium*, 2018.

[17] P. Inglesant, M. A. Sasse, D. W. Chadwick, and L. L. Shi. Expressions of expertness: the virtuous circle of natural language for access control policy specification. In *SOUPS*, 2008.

[18] T. Jakobi, C. Ogonowski, N. Castelli, G. Stevens, and V. Wulf. The Catch(es) with Smart Home: Experiences of a Living Lab Field Study. In *CHI 2017*, 2017.

[19] Y. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, and A. Prakash. ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[20] H.-T. Le, D. C. Nguyen, L. C. Briand, and B. Hourte. Automated inference of access control policies for web applications. In *SACMAT*, 2015.

[21] N. Malkin, J. Bernd, M. Johnson, and S. Egelman. "What Can't Data Be Used For?" Privacy Expectations about Smart TVs in the US. In *European Workshop on Usable Security (Euro USEC)*, 2018.

[22] S. Mare, L. Girvin, F. Roesner, and T. Kohno. Consumer Smart Homes: Where We Are and Where We Need to Go. In *IEEE Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2019.

[23] T. Matthews, K. O'Leary, A. Turner, M. Sleeper, J. P. Woelfer, M. Shelton, C. Manthorne, E. F. Churchill, and S. Consolvo. Stories from survivors: Privacy & security practices when coping with intimate partner abuse. In *CHI Conference on Human Factors in Computing Systems*, 2017.

[24] M. L. Mazurek, P. F. Klemperer, R. Shay, H. Takabi, L. Bauer, and L. F. Cranor. Exploring Reactive Access Control. In *CHI '10 Extended Abstracts on Human Factors in Computing Systems*, 2010.

[25] S. Mennicken and E. M. Huang. Hacking the Natural Habitat: An In-the-Wild Study of Smart Homes, Their Development, and the People Who Live in Them. In *International Conference on Pervasive Computing (Pervasive)*, 2012.

[26] S. Mennicken, D. Kim, and E. M. Huang. Integrating the Smart Home into the Digital Calendar. In *CHI Conference on Human Factors in Computing Systems (CHI)*, 2016.

[27] C. Nandi and M. D. Ernst. Automatic Trigger Generation for Rule-based Smart Homes. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, 2016.

[28] A. Oulasvirta, A. Pihlajamaa, J. Perkiö, D. Ray, T. Vähäkangas, T. Hasu, N. Vainio, and P. Myllymäki. Long-term Effects of Ubiquitous Surveillance in the Home. In *14th International Conference on Ubiquitous Computing (UbiComp)*, 2012.

[29] E. Ronen, A. Shamir, A.-O. Weingarten, and C. O'Flynn. IoT Goes Nuclear: Creating a ZigBee Chain Reaction. *IEEE Symposium on Security and Privacy*, 2017.

[30] L. L. Shi and D. W. Chadwick. A controlled natural language interface for authoring access control policies. In *SAC*, 2011.

[31] J. Slankas and L. A. Williams. Access control policy extraction from unconstrained natural language text. *2013 International Conference on Social Computing*, pages 435–440, 2013.

[32] J. Slankas, X. Xiao, L. A. Williams, and T. Xie. Relation extraction for inferring access control rules from natural language artifacts. In *ACSAC*, 2014.

[33] SmartThings Community Forums. Guest Access - Solution? `https://community.smartthings.com/t/guest-access-solution/97288/26`, 2017.

[34] M. Surbatovich, J. Aljuraidan, L. Bauer, A. Das, and L. Jia. Some Recipes Can Do More Than Spoil Your Appetite: Analyzing the Security and Privacy Risks of IFTTT Recipes. In *26th International Conference on World Wide Web (WWW)*, 2017.

[35] B. Ur, J. Jung, and S. Schechter. The Current State of Access Control for Smart Devices in Homes. In *Workshop on Home Usable Privacy and Security (HUPS)*. HUPS 2014, 2013.

[36] B. Ur, J. Jung, and S. E. Schechter. Intruders Versus Intrusiveness: Teens' and Parents' Perspectives on Home-Entryway Surveillance. In *ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*, 2014.

[37] Q. Wang, W. U. Hassan, A. M. Bates, and C. A. Gunter. Fear and Logging in the Internet of Things. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[38] M. Wollerton. Here's what happened when someone hacked the August Smart Lock. `https://www.cnet.com/news/august-smart-lock-hacked/`, 25 2016.

[39] J. Woo and Y.-K. Lim. User Experience in Do-It-Yourself-Style Smart Homes. In *ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*, 2015.

[40] P. Worthy, B. Matthews, and S. Viller. Trust Me: Doubts and Concerns Living with the Internet of Things. In *ACM Conference on Designing Interactive Systems (DIS)*, 2016.

[41] E. Zeng, S. Mare, and F. Roesner. End User Security and Privacy Concerns with Smart Homes. In *Symposium on Usable Privacy and Security (SOUPS)*, 2017.

[42] S. Zheng, N. Apthorpe, M. Chetty, and N. Feamster. User Perceptions of Smart Home IoT Privacy. *Proceedings of the ACM on Human-Computer Interaction (PACMHCI)*, 2:200:1–200:20, 2018.

Appendices

A Initial Interview Script

Control and Agency

- Who found out about the study? Who wanted to be a part of it?
- Did you set up your smart home together, or did one person take the lead?
- Do all of you have access to all smart home devices right now? If not, why?
- What are you hoping that your smart home will do for you?

Multi-user Privacy

- Have you ever unexpectedly learned anything about someone else, through the smart home?
- Can you think of ways you could "spy" on people using your smart home? Would you do it?
- Do you think it's a good or bad thing that you can find out those things?

Transparency

- Are you having any trouble figuring out how to control your devices? Or figuring out which devices are smart?
- Has there been any confusing moments where you weren't sure what was causing something to happen in your home? How did you figure it out?

Access Control Preferences

- Can you think of any situations where you want to restrict where people could remotely control devices from?
- Can you think of any situations where you want to restrict certain people from controlling certain devices?

General Security and Privacy Questions

- Do you have any security and privacy concerns about smart homes?
- Are there any potential security and privacy issues that you are aware of, but aren't worried about?

B Exit Interview Script

General Usage and Control

- How did you end up using your new devices?
- Did you set up any automations?
- How involved were each of you in configuring the home? Like setting up rooms and permissions?

Notifications and Transparency

- Let's talk about the activity notifications feature - the notifications you can get when someone turns something on or off, or trips one of your sensors. Did you have this feature on? (Why not?)
- How did you set your preferences for notifications? Why? Which devices? Proximity based or not?
- In what situations did you normally see notifications?
- Did seeing notifications provide any useful or interesting information?
- Did any notifications help you understand what your smart home was doing?
- Did you learn something about other people's behavior that you wouldn't have found out about without notifications?
- Did you change your behavior in your home because of the notifications?
- Were the notifications overwhelming, or not useful?
- What changes would you like made to make to this feature?
- Leaving aside the particular capabilities of our app, can you think of any situation where it would it be useful to get notifications, maybe just for particular devices?

Supervisory, Reactive, and Role-based Access Control

- Let's move onto the allowed users feature. This is the feature that lets you designate owners for each device, and have everyone else ask for permission to use it. Did you use this feature?
- If so, who was restricted? What devices and policies did you set? (block, ask, ask if not nearby)
- If not, why?
- How did you all decide on who to set restrictions on?
- In what situations did <restricted user> have to ask for permission to use a device?
- Did anyone try to circumvent restrictions on them? How?
- To blocked user: was it clear to you which devices you needed permission to use? How did you find out?
- To blocked user: How comfortable did you feel pushing the button to ask for permission?
- To blocked user: Did you change your behavior as a result of having to ask for permission?
- To admin users: How did you feel when you got notifications when someone asked for permissions?
- To blocked user: when you asked for permission, did the other person usually respond in time?

- To admin user: did you receive notifications in a timely manner? Were you able to fulfill requests?
- What changes would you like made to make to this feature?

Location-based Access Control

- Now let's talk about permissions for remote control. This is the setting where you can make people ask for permission to use a device if they aren't nearby.Were any devices restricted to remote control in a particular location? If not, why?
- How did you all decide on which devices to set restrictions on?
- In what situations did you have to ask for permission to use a device?
- Did anyone try to circumvent the restrictions on a device? How?
- When you had to ask for permission, did someone respond in time?
- When you got a permission request, did you receive a notification in a timely manner? Were you able to fulfill the request?
- Did the beacons usually accurate put you in the correct room?
- Was it clear which devices were location restricted? How did you know?
- To blocked user: How comfortable did you feel pushing the button to ask for permission?
- To blocked user: Did you change your behavior as a result of having to ask for permission?
- To admin users: How did you feel when you got notifications when someone asked for permissions?
- Did you ever use this feature to check who was home?
- What changes would you like made to make to this feature?
- Hypothetically, imagine we built an app that had every access control scheme and level of granularity you wanted - custom permission tiers, time-based access controls, proximity-based access controls, and device-level granularity. How would you set these for the different people who visit your home? (Spouse, children, guests, domestic workers?)

C Codes Used for Qualitative Analysis

- Access control - ask for permission
- Access control - complexity/discoverability
- Access control - conflicts with other goal
- Access control - desired use cases
- Access control - location-based
- Access control - not useful
- Access control - role-based
- Access control - side channel
- Access control - trust/respect each other
- Access control - unconcerned about device
- Access control - useful
- Multi-user - conflicts
- Multi-user - pranks
- Multi-user - privacy
- Multi-user - unexpected home behavior
- Notifications - checking/debugging automations
- Notifications - desired use cases
- Notifications - not noisy
- Notifications - not useful
- Notifications - privacy
- Notifications - proximity scoping
- Notifications - too noisy
- Notifications - useful
- Relationship - children
- Relationship - couples
- Relationship - domestic workers
- Relationship - guests
- Relationship - roommates
- SecPriv - Accepts risk
- SecPriv - Concern about location/proximity
- SecPriv - Concern about others
- SecPriv - Non concern
- SecPriv - Privacy concerns
- Usability - automation confusion
- Usability - complexity
- Usability - discoverability/naming
- Usability - install barrier
- Usability - need phone
- Usability - setup difficulty
- Utility - automation
- Utility - general convenience
- Utility - provides security
- Utility - remote control
- Utility - time cost

PAC it up: Towards Pointer Integrity using ARM Pointer Authentication*

Hans Liljestrand
Aalto University, Finland
Huawei Technologies Oy, Finland
hans.liljestrand@aalto.fi

Thomas Nyman
Aalto University, Finland
thomas.nyman@aalto.fi

Kui Wang
Huawei Technologies Oy, Finland
Tampere University of Technology, Finland
wang.kui1@huawei.com

Carlos Chinea Perez
Huawei Technologies Oy, Finland
carlos.chinea.perez@huawei.com

Jan-Erik Ekberg
Huawei Technologies Oy, Finland
Aalto University, Finland
jan.erik.ekberg@huawei.com

N. Asokan
Aalto University, Finland
asokan@acm.org

Abstract

Run-time attacks against programs written in memory-unsafe programming languages (e.g., C and C++) remain a prominent threat against computer systems. The prevalence of techniques like return-oriented programming (ROP) in attacking real-world systems has prompted major processor manufacturers to design hardware-based countermeasures against specific classes of run-time attacks. An example is the recently added support for *pointer authentication* (PA) in the ARMv8-A processor architecture, commonly used in devices like smartphones. PA is a low-cost technique to authenticate pointers so as to resist memory vulnerabilities. It has been shown to enable practical protection against memory vulnerabilities that corrupt return addresses or function pointers. However, so far, PA has received very little attention as a general purpose protection mechanism to harden software against various classes of memory attacks.

In this paper, we use PA to build novel defenses against various classes of run-time attacks, including the first PA-based mechanism for data pointer integrity. We present PARTS, an instrumentation framework that integrates our PA-based defenses into the LLVM compiler and the GNU/Linux operating system and show, via systematic evaluation, that PARTS provides better protection than current solutions at a reasonable performance overhead.

1 Introduction

Memory corruption vulnerabilities, such as buffer overflows, continue to be a prominent threat against modern software applications written in memory-unsafe programming languages, like C and C++. Theses vulnerabilities can be exploited to overwrite data in program memory. By overwriting control data, such as code pointers and return addresses, attackers can redirect execution to attacker-chosen locations. *Return-oriented programming* (ROP) [35] is a well known technique that allows the attacker to leverage corrupted control-data and pre-existing code sequences to construct powerful (Turing-complete) attacks without the need to inject code into the victim program. By overwriting non-control data, such as variables used for decision making, attackers can also influence program behavior without breaking the program's *control-flow integrity* (CFI) [1]. Such attacks can cause the program to leak sensitive data or escalate attacker privileges. Recent work has shown that non-control-data attacks can also be generalized to achieve Turing-completeness. Such *data-oriented programming* (DOP) attacks [16] are difficult to defend against, and are an appealing attack technique for future run-time exploitation.

Software defenses against run-time attacks can offer strong security guarantees, but their usefulness is limited by high performance overhead, or requiring significant changes to system software architecture. Consequently, deployed solutions (e.g., Microsoft EMET [26]) trade off security for performance. Various hardware-assisted defenses in the research literature [15, 42, 41, 14, 38, 40, 28, 32] can drastically improve the efficiency of attack detection, but the majority of such defenses are unlikely to ever be deployed as they require invasive changes to the underlying processor architecture. However, the prevalence of advanced attack techniques (e.g, ROP) in modern run-time exploitation has prompted major processor vendors to integrate security primitives into their processor designs to thwart specific attacks efficiently [17, 29, 31]. Recent additions to the ARMv8-A architecture [3] include new instructions for *pointer authentication* (PA). PA uses cryptographic message authentication codes (MACs), referred to as *pointer authentication codes* (PACs), to protect the integrity of pointers. However, PA is vulnerable to *pointer reuse* attacks where an authenticated pointer is substituted with another [31]. Practical PA-based defenses must minimize the scope of such substitution.

Goals and Contributions In this work, we further the security analysis of ARMv8-A PA by categorizing pointer

Extended version of this article is available as a technical report [24].

reuse attacks, and show that PA enables practical defenses against several classes of run-time attacks. We propose an enhanced scheme for *pointer signing* that enforces *pointer integrity* for all code and data pointers. We also propose *run-time type safety* which constrains pointer substitution attacks by ensuring the pointer is of the correct type. Pointer signing and run-time type safety are effective against both control-flow and data-oriented attacks. Finally, we design and implement *Pointer Authentication Run-Time Safety* (PARTS), a compiler instrumentation framework that leverages PA to realize our proposed defenses. We evaluate the security and practicality of PARTS to demonstrate its effectiveness against memory corruption attacks. Our main contributions are:

- *Analysis:* A categorization and analysis of pointer reuse and other attacks against ARMv8-A pointer authentication (Section 3).
- *Design:* A scheme for using *pointer integrity* to systematically defend against control-flow and data-oriented attacks, and *run-time type safety*, a scheme for guaranteeing safety for data and code pointers at run-time (Section 5).
- *Implementation:* PARTS, a compiler instrumentation framework that uses PA to realize data pointer, code pointer, and return address signing (Section 6).
- *Evaluation:* Systematic analysis of PARTS showing that it has a reasonable performance overhead ($< 0.5\%$ average overhead for code-pointer and return address signing, 19.5% average overhead for data-pointer signing in nbench-byte (Section 7)) and provides better security guarantees than fully-precise static CFI (9).

We make the source code of PARTS publicly available at `https://github.com/pointer-authentication`.

2 Background

2.1 Run-time attacks

Programs written in memory-unsafe languages are prone to memory errors like buffer-overflows, use-after-free errors and format string vulnerabilities [39]. Traditional approaches for exploiting such errors by corrupting program code have been rendered largely ineffective by the widespread deployment of measures like data execution prevention (DEP). This has given rise to two new attack classes: *control-flow attacks* and *data-oriented attacks* [11].

2.1.1 Control-flow attacks (on ARM)

Control-flow attacks exploit memory errors to hijack program execution by overwriting code pointers (function return addresses or function pointers). Corrupting a code pointer can cause a control-flow transfer to anywhere in executable memory. Corrupting the return address of a function can be used for ROP attacks, which are feasible on several architectures, including ARM [19].

ARM processors, similar to other RISC processor designs, have a dedicated Link Register (LR) that stores the return address. LR is typically set during a function call by the Branch with Link (`bl`) instruction. An attacker cannot directly influence the value of LR, as it is unlikely for a program to contain instructions for directly modifying it. However, nested function calls require the return address of a function to be stored on the stack before the next function call replaces the LR value. While the return address is stored on the stack, an attacker can use a memory error to modify it to subsequently redirect the control flow on function return. On both x86 and ARM, it is possible to perform ROP attacks without the use of return instructions. Such attacks are collectively referred to as *jump-oriented programming* (JOP) [9].

Control-flow integrity (CFI) [1] is a prominent defense technique against control-flow attacks. The goal of CFI is to allow all the control flows present in a program's control-flow graph (CFG), while rejecting other flows. Widely deployed CFI solutions are less precise than state-of-the-art solutions presented in scientific literature.

2.1.2 Data-oriented attacks

In contrast to control-flow attacks, *data-oriented attacks* can influence program behavior without the need to modify code pointers. Instead, they corrupt variables that influence the program's decision making, or leak sensitive information from program memory. Such attacks are called *non-control-data attacks*. Chen et al [11] demonstrated a variety of non-control-data attacks for forging user credentials, changing security critical configuration parameters, bypassing security checks, and escalating privileges. Recent work on DOP [16] showed that non-control-data corruption can also enable expressive attacks without compromising control-flow integrity. DOP may compromise the input of individual program operations and chain together a chosen sequence of operations to achieve the intended functionality.

A data-oriented attack can in principle corrupt arbitrary program objects, but corrupting data pointers is often the preferred attack vector [12]. In Chen et al.'s attack against the GHTTPD web server [11], a stack buffer overflow is used to corrupt a data pointer used in input string validation in order to bypass security checks on the input under the attacker's control. Data pointers are also routinely corrupted in heap exploitation. For instance, the "*House of Spirit*" attack on Glibc[1], involves corrupting a pointer returned by `malloc()` to trick subsequent `malloc()` calls into returning attacker controlled memory chunks. The DOP attacks in [16] also involve the corruption of pointers as a means to control which data is processed by vulnerable code.

[1] Team Shellphish repository of educational heap exploitation techniques: `https://github.com/shellphish/how2heap`

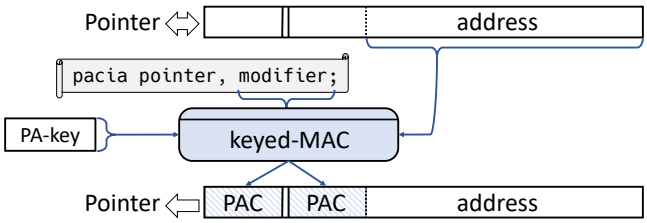


Figure 1: The PAC is created using key-specific PA instructions (`pacia`) and is a keyed MAC calculated over the pointer address and a modifier.

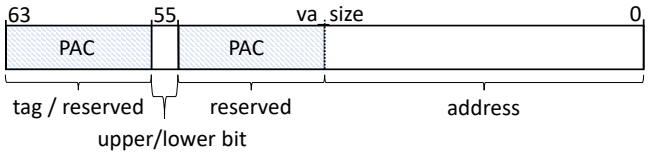


Figure 2: Pointer layout on 64-bit ARM. The PAC is stored in the reserved bits, and its size depends on the used virtual address range. If pointer tagging is disabled, then the PAC can also extend to the tag bits.

2.2 ARM Pointer Authentication

ARMv8.3-A includes a new feature called *pointer authentication* (PA). PA is intended for checking the integrity of pointers with minimal size and performance impact. It is available when the processor executes in 64-bit ARM state (AArch64). PA adds instructions for creating and authenticating *pointer authentication codes* (PACs). The PAC is a tweakable message authentication code (MAC) calculated over the pointer value and a 64-bit *modifier* as the tweak (Figure 1). Different combinations of key and modifier pairs allow domain separation among different classes of authenticated pointers. This prevents authenticated pointer values from being arbitrarily interchangeable with one another.

The idea of using of MACs to protect pointers at run-time is not new. *Cryptographic CFI* (CCFI) [25] uses MACs to protect control-flow data such as return addresses, function pointers, and vtable pointers. Unlike ARMv8-A PA, CCFI uses hardware-accelerated AES for speeding up MAC calculation. Run-time software checks are needed to compare the calculated MAC to a reference value. PA, on the other hand, uses either QARMA [5] or a manufacturer-specific MAC, and performs the MAC comparison in hardware.

64-bit ARM processors only use part of the 64-bit address space for virtual addresses (Figure 2). The PAC is stored in the remaining unused bits of the pointer. On a default AArch64 Linux kernel configuration with 39 bit addresses and without address tagging [3, D4.1.4], the PAC size is 24 bits. However, depending on the memory addressing scheme and whether address tagging is used, the size of the PAC is between 3 and 31 bits [31]. Security implications of the PAC size are discussed in Section 9.

PA provides five different keys for PAC generation: two for code pointers, two for data pointers, and one for generic use. The keys are stored in hardware registers configured to be accessible only from a higher privilege level: e.g., the kernel maintains the keys for a user space process, generating keys for each process at process `exec`. The keys remain constant throughout the process lifetime, whereas the modifier is given in an instruction-specific register operand on each PAC creation and authentication (i.e., MAC verification). Thus it can be used to describe the run-time context in which the pointer is created and used. The modifier value is not necessarily confidential (see Section 4) but ideally such that it 1) precisely describes the context of use in which the pointer is valid, and 2) cannot be influenced by the attacker.

PA is used by instrumenting code with PAC creation and authentication instructions. PA instruction mnemonics are generally prefixed either with `pac` or `aut` for creation and authentication, respectively, followed by two characters that select one of the data or code keys. For instance, the `pacia` instruction in Figure 1 will generate an authenticated pointer (`pac`) based on the instruction (`i`) A-key (`a`). Table 5 in Appendix C provides a list of PA instructions referred to in this paper. An authenticated pointer cannot be used directly, as the PAC embedded in the pointer value intentionally interferes with address translation. The corresponding PA authentication instruction (in this case, `autia`) removes the PAC from the pointer if authentication is successful, i.e., if the current pointer value, key and modifier for `autia` yields a PAC that matches the PAC embedded in the pointer. If authentication fails, the pointer is invalidated such that a dereference or call using the pointer will cause a memory translation fault. Dedicated PA instructions are encoded in NOP space; older processors without PA support will ignore them.

Return address signing. Qualcomm's return address signing scheme [31] is the first to make use of ARMv8-A PA. It was first introduced in Linaro's GCC toolchain, but has been supported by mainline GCC since version 7.0[2]. It thwarts attacks that manipulate function return addresses through stack corruption (see Section 2.1.1) by ensuring that the return address in LR always contains a PAC when written to or retrieved from memory. Listing 1 shows an example.

The instrumentation adds `paciasp` (①) at beginning of the function prologue, before the LR value is stored on the stack. `paciasp` adds a PAC tag using the current Stack Pointer (SP) value as the modifier. Before function return, `autiasp` (②) authenticates the pointer and either removes the PAC or invalidates the pointer. An alternative is to use the combined `autiasp+ret` instruction, `retaa`, but it is not backwards-compatible with older processors.

[2]GCC return address signing and PA support is based on patches provided by ARM, `https://github.com/gcc-mirror/gcc/commit/06f29de13f48f7da8a8c616108f4e14a1d19b2c8`

```
function:
  paciasp                 ; ① create PAC
  stp FP, LR, [SP, #0]    ; store LR
  ; ...
  ldp FP, LR, [SP, #0]    ; load LR
  autiasp                 ; ② authenticate
  ret                     ; return
```

Listing 1: Return address signing using PA. At funtion entry, `paciasp` is used to create a PAC in LR (①). The value is then authenticated with `autiasp` before return (②).

The PAC cryptographically binds the return address to the current SP value. It is valid only when authenticated using the same SP value as on PAC creation. The goal is to limit the validity of the PAC to the function invocation that created it, thus preventing reuse of authenticated return addresses.

3 Attacks on Pointer Authentication

PA prevents an attacker from injecting or forging pointer values. This effectively prevents any attack that relies on corrupting pointers, resisting even attackers with *arbitrary access to program memory*.

The modifier value used in computing a PAC can depend on both static (e.g., a hard-coded value) and dynamic (e.g., the SP) information. We assume that the program code itself is not confidential and that the attacker can learn how dynamic modifiers are generated and may infer their values.

PA also relies on the security of the underlying cryptographic primitives. In particular, an attacker may attempt to brute-force either the PA keys themselves, or individual PAC values. Sophisticated adversaries may even attempt cryptanalysis attacks based on known PAC values, or side-channels attacks against the hardware circuitry for computing PACs. The security of the QARMA block cipher has already been analyzed [43, 23]. We leave the scrutiny of the cryptographic building blocks outside the scope of this paper. Nevertheless, the limited PAC size means that guessing attacks are a potential concern. We discuss the feasibility of brute-forcing PACs in Section 7.2.4. Assuming proper precautions for the lifetime of PA keys (see Section 2.2), we do not consider guessing attacks the primary attack vector against PA. However, the following concerns for the security of PA-based defenses remain: 1) an attacker controlling the creation of PAC values, or 2) an attacker *reusing* previously authenticated pointers.

Malicious PAC generation. Attackers can potentially control PAC values in three ways, by controlling:

1. *the unauthenticated pointer value before PAC creation*: get an arbitrary authenticated pointer for any context with the same modifier and PA key.
2. *control the PA modifier value*: get an authenticated pointer for a context with the same PA key, but with an attacker-chosen modifier.
3. *both*: get arbitrary authenticated pointers for a context with attacker-chosen modifier, and the same PA key.

To prevent the attacker from generating arbitrary authenticated pointers, the program must not contain PA creation instructions with attacker controlled inputs. Also, a control-flow attack could be mounted by chaining together instruction sequences to prepare the PA operand registers with attacker controlled input and then jump to a PA instruction at another part of the program. This suggests that PA-based defenses *must provide, or be combined with, CFI guarantees* that prevent the use of individual authentication instructions as attacker-controlled gadgets.

Reuse attacks. The attacker can read authenticated pointers (including PAC values), and later reuse them to either:

- *rollback* an authenticated pointer to a previous value, or
- *substitute* an authenticated pointer with another using the same PA modifier.

For instance, in GCC's return address signing scheme (Section 2.2), the return address is bound to the location of the stack frame by using the current SP value as the PA modifier. However, the SP value is not necessarily unique to a specific function invocation. Consequently, an attacker can reuse the authenticated return addresses value from one function when a different vulnerable function executes with a matching SP value. Given that typical programs offer no guarantees on the uniqueness of SP values between different function invocations, this approach exposes a large attack surface for pointer reuse attacks. Therefore, a concern for any PA-based defense is partitioning authenticated pointers into distinct classes based on different <PA key, modifier> pairs.

Attackers can reuse only those pointers they can observe (as opposed all possible values a function pointer can take). Even with full read access to memory (and hence the ability to observe any pointer value that has been generated so far), attackers are still limited to authenticated pointer values the program has already generated.

4 Adversary Model and Requirements

4.1 Pointer Integrity

Kuznetsov et al. [21] introduced the idea of *code pointer integrity*: ensuring precise memory safety for all code pointers in a program. Since control-flow attacks depend on the

manipulation of code pointers, guaranteeing code pointer integrity will render *all* control-flow attacks impossible [21].

The notion of *pointer integrity* is generalizable to both code and data pointers. In Section 9.1, we provide a more rigorous definition of pointer integrity. Intuitively, pointer integrity aims to prevent unintentional changes to pointers while they remain in program memory so that the value of a pointer at the time it is "used" (e.g., dereferenced or loaded from memory) is the same as when it was created or stored on memory. In particular, integrity-protected pointers reference the intended target objects. As explained in Section 2.1, all control-flow attacks, all known DOP attacks and many other data-oriented attacks rely on the manipulation of vulnerable pointers. Consequently, ensuring pointer integrity will prevent these attacks.

4.2 Attacker Capabilities

To reason about how effectively PA defends against state-of-the-art attacks we assume attacker capabilities consistent with prior work on run-time attacks (Section 2.1). Our adversary model assumes a powerful attacker with arbitrary memory read and write capabilities restricted only by DEP. The attacker can thus read any program memory and write to non-code segments. We further assume that the attacker has no control of higher privilege levels, i.e., an attacker targeting a user space process cannot access the kernel or higher privilege levels. Specifically, we assume that the attacker cannot infer the PA keys, as they are in registers not directly readable from user space (Section 2.2). We discuss protection of kernel code using PA in Section 10. The attacker's ability to read arbitrary memory precludes the use of randomization-based defenses that cannot withstand information disclosure (e.g., address space layout randomization [36] or software shadow-stacks [1]). PA was specifically designed to remain effective even when the entire memory layout of the victim process is known.

4.3 Goal and Requirements

Our goal is to thwart control-flow and data-oriented attacks by preventing the attacker from forging pointers used by a vulnerable program. We identify the following requirements that our solution should satisfy:

R1 *Pointer Integrity*: Detect/prevent the use of corrupted code and data pointers.

R2 *PA-attack resistance*: Resist attempts to control PAC generation, and pointer reuse attacks.

R3 *Compatibility*: Allow protection of existing programs without interfering with their normal operation.

R4 *Performance*: Minimize run-time and memory overhead and gracefully scale in relation to the number of protected pointers and dereferences/calls.

5 Design

To meet our requirements (Section 4.3) we must solve a number of challenges which we elaborate below.

5.1 Instrument program with PA instructions

To meet requirement R1, the program executable must be instrumented with PA instructions to create and authenticate PACs when needed. For this, we designed and implemented *Pointer Authentication Run-Time Safety* (PARTS), a compiler enhancement that emits PA instructions to sign pointers in memory as required. Specifically, it protects:

- return addresses;
- local, global and static pointers; and
- pointers in C structures.

Figure 3 shows the overall architecture of the PARTS-enhanced compiler. PARTS analyzes the compiler's intermediate representation (IR) to identify any pointers used by the program and then emits PA instructions at points in the program where pointers are (a) created or stored in memory, and (b) loaded from memory or used.

5.2 Create PACs in statically allocated data

Programs may contain pointers which are initialized by the compiler, e.g., defined global variables. However, PAC values for authenticated pointers cannot be calculated before program execution, as PA keys are set only at program launch. Consequently, initialized pointers in the program's data segment pose a challenge, as their values are normally initialized by the linker and loaded into memory separately. PARTS solves this problem by generating a custom initializer function for pointers requiring PACs. At run-time, the PARTS runtime library, PARTSlib, processes the relocated variables and invokes the generated initializer function to ensure that any defined pointers are furnished with a PAC.

5.3 Pointer compartmentalization

As described in Section 3 the attacker may attempt to reuse previously signed pointers. To meet requirement R2 PARTS therefore limits the scope of such reuse attacks by compartmentalizing pointers in three different ways, as shown in Table 1.

Code / Data Pointer Compartmentalization: Recall from Section 2.2, that PA provides separate key sets for data and code pointers making it possible to limit reuse attacks.

Table 1: For code and data pointers PARTS uses a static PA modifier based on the pointer's *ElementType* as defined by LLVM. Return address signing uses a 48-bit `function-id` and the 16 most-significant bits of the SP value.

		key	Modifier type	Modifier construction
①	Data pointer signing	Data A	static	`type-id` = SHA3(ElementType)
②	Code pointer signing	Instr A	static	`type-id` = SHA3(ElementType)
③	Return address signing	Instr B	dynamic + static	SP | `function-id` = compile-time nonce

Run-time type safety: Pointer compartmentalization, while effective, is coarse-grained. To address this, PARTS adds run-time type safety for data and code pointers. Run-time type safety records the pointer's type by encoding it in the PA modifier. Then, it checks that pointer dereferences or indirect calls take place using a pointer with a recorded type that matches the type expected at the use site. PARTS assigns pointers a unique id, `type-id`, based on the pointer's LLVM *ElementType* which depends on the pointed-to data, structure, or function signature. Two pointers are *compatible* (have the same `type-id`) if their ElementType is the same. PARTS uses a deterministic scheme, detailed in Section 6.1 and shown in Table 1, to calculate `type-id`s during compilation. This ensures that separate compilation units generate equivalent `type-id`s for compatible objects, and different `type-id`s for non-compatible ones.

Improved Return Address Signing: While run-time type safety could also be applied for return addresses, it would result in an over-permissive policy for backward edges. As described in Section 3, binding the authenticated return address to the current stack pointer value alone is insufficient because the stack pointer may not be unique to a specific function invocation. Instead, PARTS uses a combination of the current stack pointer value, and a compile-time nonce (`function-id`) ensuring that the authenticated return address cannot be reused across invocations of *different functions*, while the stack pointer values effectively compartmentalizes return addresses to callers with different stack layouts.

5.4 On-load data pointer authentication

Pointers with PACs can be authenticated either as they are loaded from memory, or immediately before they are used. We refer to these as *on-load* and *on-use* authentication, respectively. Data pointers are often dereferenced frequently without intervening function calls, i.e., they will not be cleared after use. This allows the compiler to optimize memory accesses such that, for instance, temporary values might never be written to memory. PARTS accommodates this behavior by only using on-load authentication for data pointers. The combined PA instructions can be used for on-use authentication of code pointers, which are typically loaded to a register, used once, and cleared. On-load authentication always uses the standalone authentication instructions. An attacker could attempt to exploit either the standalone authentication or the separate pointer dereference by diverting control flow to either. However, as mentioned in Section 3, PA solutions must be combined with CFI guarantees, which prevent this type of attacks.

5.5 Handling pointer conversions

A data pointer to an object of a specific type may be converted to a pointer to a different object type. When run-time type safety is applied to authenticated pointers, special care must be taken to not interfere with legitimate pointer conversions to meet requirement R3. For instance, if a struct pointer is cast to a pointer to its first field, it will change the `type-id` and hence the expected PAC.

If the source and destination object types are compatible, no special consideration is needed. If not, PARTS must convert the authenticated pointer to the correct `type-id`. Because data pointer PAC creation and authentication is done at store/load, PARTS handles conversions by; (a) if loading the pointer from memory, validating and stripping the PAC using the `type-id` of the original object, and (b) on store, creating a new PAC using the destination object `type-id`.

A pointer to a function of one type may be converted to a pointer to a function of another type. However, the behavior when calling a function pointer cast to a non-compatible type is undefined [18][6.3.2.3§8]. Hence, PARTS does not need to convert the pointer's PAC to match the destination function's `type-id`. If the converted pointer is converted back, the result is expected to be the same as the original pointer [18][6.3.2.3§8]. PARTS satisfies this as it does not modify the pointer's PAC.

6 Implementation

The PARTS compiler is based on LLVM 6.0 but modifies and adds new passes to the optimizer and the AArch64 backend (Figure 3). The optimization passes (❶) generate necessary metadata for PA modifiers, inserts wrappers for compatibility with legacy code, and prepares initializers for statically allocated pointers. The AArch64 Frame Lowering emits function prologues and epilogues and is modified to include instructions for authenticating the LR value (❷). The

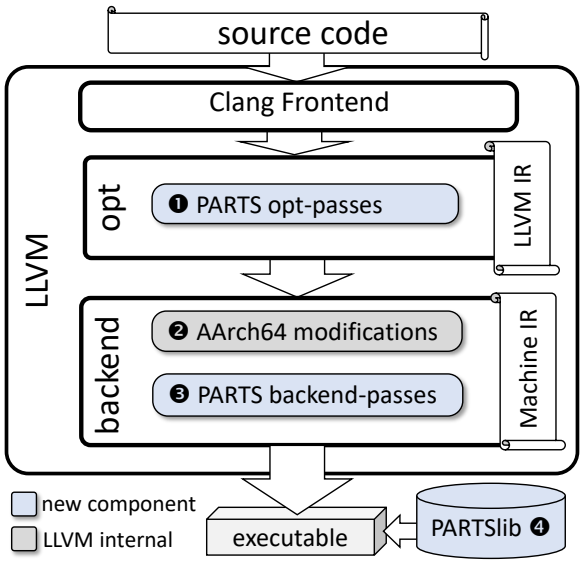


Figure 3: PARTS architecture.

PARTS backend passes (❸) retrieve the PA modifiers and instruments appropriate low-level instructions. The resulting binary is linked with PARTSlib (❹), which at run-time creates PACs for the initialized pointers.

6.1 LLVM Compiler Integration

While the LLVM 6.0 AArch64 backend recognizes PA instructions, they are not used by any pre-existing security feature. Our modifications consist of added optimizer and backend passes, minor modifications to the AArch64 backend, and new PARTS-specific intrinsics. Where applicable, we use optimizer passes that operate on the high-level LLVM *intermediate representation* (IR). Nonetheless, much of the needed functionality is PA-specific and thus implemented in the backend that uses low-level LLVM *machine IR* (MIR), and a register- and instruction set specific to 64-bit ARM.

Determining pointer `type-id`. The compiler backend views the program from a low-level perspective, and the MIR has lost much of the semantics present in C or the high-level IR. Therefore, PARTS must determine `type-ids` during its optimizer passes where this information is still available (Figure 3, ❶). The `type-id` for data consists of a truncated 64-bit SHA-3 hash of the pointer's LLVM `ElementType`. The `ElementType` represents the IR level data type and distinguishes between basic data types, but does not retain `typedef` or other information from the frontend (i.e., clang). Code pointers use the same scheme wherein the `ElementType` consists of the function signature at the same abstraction level. The `type-ids` are passed to the backend either via PARTS-specific compiler intrinsics, or by embedding them as metadata in the existing IR instructions. The AArch64 instruction selection retrieves the information from the IR instructions and transfers it to the emitted MIR (Figure 3, ❷). To facilitate the run-time bootstrap (Section 6.2) PARTS also includes a pass that prepares a custom initializer function that is called at run-time to generate PACs for defined global pointers (Figure 3, ❶).

Return addresses signing. Return address signing is implemented in the AArch64 backend during frame lowering (Figure 3, ❷). Frame lowering emits the function prologues and epilogues, and for non-leaf functions, emits instructions for storing and retrieving the LR value from the stack. PARTS authenticates the value of the LR only if it was retrieved from the stack. The PAC modifier is based on the 16 least-significant bits of the SP value and a 48-bit function-specific `function-id`. The `function-id` is guaranteed to be unique within the current compilation unit or, with link time optimization (LTO), the whole program. To avoid repetition across different compilation units, the `function-id` is generated using a pseudorandom, non-repetitive sequence.

Code pointer signing. PARTS uses the combined PA instructions for branches and converts branch instructions directly to their PA variants (Figure 3, ❸). The PAC for any code pointer is created only once at the time of pointer creation, e.g., when the address of a function is taken. This is instrumented by adding a PAC-creation instruction immediately after the instruction that moves a code pointer to a register. Subsequent load and store operations do not authenticate the signed code pointers, instead they are authenticated only on use.

Data pointer signing. As discussed in Section 5.4, it is not feasible to perform on-use authentication for data pointers. Instead, we authenticate data pointers when they are loaded from memory and create PACs before storing them. In some cases, e.g., using globals, the IR will include explicit load and store operations that can be furnished with the `type-id`. Our modified Instruction Selection then forwards the `type-id` to the emitted MIR (Figure 3, ❷). However, stack-based store and load operations, in particular, are often not present before the backend finalizes the stack-layout and register allocation. Thus, some load and store instructions must be instrumented solely in the backend.

While it would be possible to modify the AArch64 backend (e.g., register allocation), we have instead opted for a less invasive approach. The PARTS backend pass (Figure 3, ❸) finds load and store instructions in the MIR, and uses the attached `type-id` for instrumentation. When the `type-id` is not present, e.g., because the load and store is a register spill, the `type-id` is fetched from surrounding code. For instance, when instrumenting the store due to register spilling

```
MACRO movFunctionId Mod
  movk  Mod, #func_id16, lsl #16
  movk  Mod, #func_id32, lsl #32
  movk  Mod, #func_id48, lsl #48
ENDM

function:
  mov       Xd, SP              ; ① get SP
  movFunctionId Xd              ; ② get id
  pacib     LR, Xd              ; ③ PAC
  stp       FP, LR, [SP, #0]    ; store
  ; function body
  ldp       FP, LR, [SP, #0]    ; load LR
  mov       Xd, SP              ; ⑤ get SP
  movFunctionId Xd              ; ④ get id
  autib     LR, X               ; ⑥ auth
  ret
```

Listing 2: The PARTS return address signing binds the PAC to the SP (①,⑤) and unique function id (②,④). The PA modifier is in register `Xd` during PAC creation (③) and authentication (⑥). The 48-bit `func-id` is split into three 16-bit parts, each moved individually to `Xd` by left-shifting.

```
MACRO movTypeId Mod
  mov   Mod, #type_id00
  movk  Mod, #type_id16, lsl #16
  movk  Mod, #type_id32, lsl #32
  movk  Mod, #type_id48, lsl #48
ENDM

mov       cPtr, #instr_addr   ; load cPtr
movTypeId Xd                  ; ❶ get id
pacia     cPtr, Xd            ; ❷ PAC
; no intermediate cPtr instrumentation
movTypeId Xd                  ; ❸ get id
blraa     cPtr, Xd            ; ❹ branch
```

Listing 3: The PARTS forward-edge code pointer signing uses the code pointer's `type-id` as the PA modifier (❶,❸). The 64-bit `type-id` is split into four 16-bit parts. The PAC is created only once when initially creating the code pointer (❷). Upon use, i.e., indirect call, the PAC is authenticated using the combined branch and authenticated instruction (❹). PARTS does not instrument intermediate store/load operations.

a pointer variable, the correct `type-id` can be fetched from the original load.

6.2 Run-time Bootstrap

Programs may contain pointers in statically allocated data, i.e., pointers stored in global variables or static local variables. These are initialized by the compiler or linker, and therefore cannot include PACs. The PARTSlib runtime library instead invokes the compiler generated custom PAC initializer function at process startup. Our Proof-of-Concept implementation invokes the PARTSlib bootstrap using compiler instrumentation that explicitly calls the functionality when entering `main`.

6.3 Instrumentation

PARTS uses only in-line instrumentation and does not require storage of separate run-time metadata. With the exception of the bootstrap process the original code structure is thus largely unchanged. As discussed in Section 2.2, no explicit error handling is added by PARTS; instead, an authentication failure will set specific high-order bits in the pointer, thus triggering a memory translation fault on subsequent dereference or call using the pointer that failed authentication. The high-order bits ensure that the fault is distinguishable as one caused by authentication failure. Our code listings use two macros for setting up PA modifiers for return address signing and `type-id` based PACs, these are shown in Listing 2 and Listing 3.

Return address signing. The return address signing instrumentation is similar to GCC's implementation [31] but includes an added modifier (Listing 2). The function prologue is instrumented such that it prepares the PA modifier by moving SP (①) value into a free register. The SP value is combined with the `function-id` (②) to form the PA modifier, which is then used with the instruction B key (③). The `function-id` is generated at compile-time using LLVM's random number generator, and is guaranteed to be unique withing the LLVM Module (i.e., the whole program, when using link time optimization). The function epilogues (i.e., any part that ends with a return or a tail-call) are similarly instrumented to generate the same PA modifier (④,⑤) and to verify the PAC in the restored LR (⑥).

Code pointer signing. PARTS instruments code pointers only on creation and use (Listing 3). Specifically, when a code pointer is initially created, PARTS will use the instruction A-key to create a PAC (❷) based on the target `type-id` (❶). The instrumentation will at no point remove the PAC from a code pointer. Instead, PARTS uses the combined authenticate and branch instructions — e.g., `blraa` — to perform the branch directly on an authenticated pointer (❹), again using the same PA modifier (❸).

```
ldr     dPtr, [SP, #0]     ; load dPtr
movTypeId Xd, #type_id     ; ① get id
autda   dPtr, Xd           ; ② authenticate
; dPtr is directly usable
```

Listing 4: PARTS immediately authenticates data pointers loaded from writeable memory. This is done by first loading the `type-id` (①) and then verifying the PAC (②).

Data pointer signing. All data pointer stores and loads are instrumented such that a PAC is created immediate before store and authenticated immediately after load (Listing 4). When a data-pointer is used the instrumentation first sets up the correct PA modifier, i.e., the `type-id` (①). The pointer is then immediately authenticated using the modifier and data A-key (②); this also strips the PAC from the pointer. As long as the data pointer resides in a register it can thus be used without any performance overhead. PARTS creates PACs for pointers immediately before store in the same manner, save for the `pacda` instruction.

7 Evaluation

We develop our Proof-of-Concept implementation of PARTS on the ARMv8-A Base Platform Fixed Virtual Platform (FVP), based on Fast Models 11.4, which supports version 8.0 to 8.4 of the ARMv8-A architecture [4]. At the time of writing, the only PA-capable hardware is the Apple A12 and S4 SoCs featuring ARMv8.3-A CPUs [2]. However, these proprietary SoCs are, to the best of our knowledge, not available in development versions outside Apple. The FVP provides a software simulation of an ARMv8.3-A processor in AArch64 mode, and is, to the best of our knowledge, the only publicly available environment with ARMv8-A PA support.

7.1 ARMv8.3 Emulation and Software Stack

We use GNU/Linux with a 4.14 kernel, modified to support PA . We modified the bootloader and kernel to activate ARMv8-A PA, and allow key configuration during kernel scheduling at Exception Level 1 (EL1 in Figure 4). Our kernel modifications are based on Mark Rutland's 2018 PA patches[3].

PA keys for each task are stored in a process-specific `mm_context_t` structure (in the process' memory descriptor in the kernel) which contains architecture-specific data related to the process address space. Threads within the same process have a common memory descriptor, and thus share the same PA keys. The scheduler will configure the PA key registers using the keys in the process' memory descriptor whenever a task is scheduled to run. When a new child process is forked, the parent's keys are duplicated to the child's memory descriptor. However, when a new executable file is `exec`'d in the context of an existing process, the kernel initializes a new set of PA keys using `get_random_bytes()`. In other words, each new process receives a new set of PA keys which remain unchanged thereafter.

[3] `https://lwn.net/Articles/752116/`

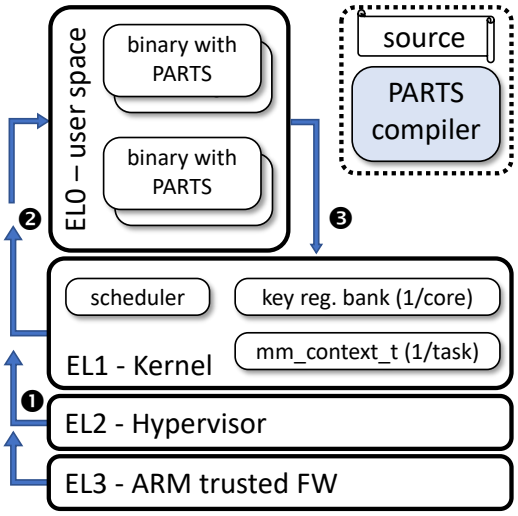


Figure 4: The trapping of PA configuration must be released ❶, in order to allow the kernel to manage the PA keys on process creation and context switches ❷. Faults generated by failed authentications will be trapped by the kernel ❸.

7.2 Security Evaluation

7.2.1 Return address signing

Return address signing in both GCC [31], and PARTS prevents an attacker from introducing forged return addresses to the program stack. Compared to GCC, PARTS augments the PA modifier used for return address signing by combining a function-specific identifier with the SP value (R2). As a result, PARTS return address signing precludes the possibility of reuse of the return address between different functions, irrespective of SP value collisions. It remains susceptible to pointer reuse between distinct invocations of the same function from call sites with same SP value (R1).

7.2.2 Forward-edge code pointer signing

As with PARTS return address signing, forward-edge code pointer signing prevents an attacker from using forged code pointers injected into program memory (R1). This prevents a large class of attacks (e.g., typical ROP/JOP gadgets) that rely on redirecting the control flow to code in the middle of functions, i.e., addresses that never were valid targets of benign control-flow transfers.

PARTS restricts forward-edge code pointer reuse by enforcing *run-time type safety* for signed pointers (R2). Under this scheme, pointers used in a pointer reuse attack must share the same `type-id` (i.e., have a matching type on the LLVM IR level). This prevents large classes of function-reuse attacks. The solution is compatible with common programming patterns involving function pointers (R3), such as callbacks, but allows reuse between code pointers to functions with identical type signatures.

7.2.3 Data pointer signing

PARTS data pointer signing protects all data pointers and prevents an attacker from loading a forged data pointer to program memory (R1). This prevents all non-control data attacks that rely on corrupting data pointers to unintended parts of of memory. This class of attacks includes all currently known DOP attacks [16].

PARTS restricts data pointer reuse by enforcing run-time type safety also for data pointers (R2). Reuse attacks would be more useful to an attacker if they could substitute a vulnerable pointer with one referencing an object of different size or type. Therefore restricting pointer substitution based on the pointer's type restricts the attacker's capability to cause unintended data flows within the program. However, pointer conversions are a challenge for data pointer integrity. As discussed in Section 5.3, PARTS accommodates data pointers that are cast from type A to an incompatible type B by writing the converted pointer using the `type-id` of B. This may expand the effective set of reusable pointers under our threat model; the attacker can record pointers of type A and reuse them at PAC conversion site $A \rightarrow B$, thereby obtaining a pointer of type B to an object of type A. This converted pointer can then be used at de-reference sites that require pointers of type B. If the program also includes a conversion from B to A this makes both types interchangeable.

PARTS data pointer integrity does not guarantee spatial safety of pointer accesses to data objects, nor does it address the temporal safety (e.g., prevent use-after-free conditions). ARMv8-A PA does not provide facilities to directly address these challenges. We discuss orthogonal schemes that can be used in combination with PARTS to provide spatial and temporal safety guarantees in Section 8.

7.2.4 PAC entropy

As explained in Section 3, the PAC size b is a concern for any PA-based scheme. On typical AArch64 Linux systems, b is between 16 and 24. To succeed with probability p, a PAC guessing attack requires $\frac{\log(1-p)}{\log(1-2^{-b})}$ guesses on the assumption that a PAC comparison failure leads to program termination. On our simulator setup where $b = 16$, achieving a 50%-likelihood for a correct guess requires 45425 attempts.

Note that ROP/DOP attacks require an environment where a set of jumps (gadgets) can be set up, each requiring a separate PAC to be broken. Consequently, success probability of a complete attack will decrease exponentially with the number of jumps necessary.

Pre-forked or multithreaded programs will share the same PA key between the parent and all sibling threads/processes. This could allow an attacker to brute force a PAC by targeting a sibling, if PAC failure on a sibling does not result in the termination (and hence PA key reset) of all threads/processes sharing the same PAC key. In this scenario, 2^{b-1} guesses on average are enough to guess a b-bit PAC (32768 guesses for $b = 16$). Multithreaded / pre-forking applications could be hardened against guessing attacks by requiring a full application restart if the number of unexpected terminations of child threads/processes exceeds a pre-defined threshold.

7.3 Performance Evaluation

The FVP processor, peripheral models, and micro-architectural fabric is simplified. Consequently, timing on the FVP model differs from actual hardware. The ARM Fast Models documentation states that *"all instructions execute in one processor master clock cycle"*. We confirm this behavior for PA instructions in the FVP by using microbenchmarks that allow PA instructions to be timed in isolation. As a result, we cannot use the FVP to estimate the expected run-time overhead of PARTS. Instead, we estimate the execution time of PA instructions and develop a PA-analogue that emulates the run-time cost of PA instructions (Section 7.3.1). We then run large-scale benchmarks on real (non-PA) hardware using our PA-analogue (Section 7.3.2).

7.3.1 PA-analogue

From [5, Table 8] we can deduce that on a (1.2GHz) mobile core, the PAC is computable with an approximate overhead of 4 cycles, without accounting for the potential speed benefits of opportunistic pipelining or the inclusion of several parallel PAC computing engines per core. For simplicity, we assume equal cycle counts for all PA instructions. Based on this assumption we construct a *PA-analogue* (Listing 5) as a proxy to measure overhead of PA instrumentation on non-PA CPUs: it consists of four exclusive-or (`eor`) operations to account for the 4 cycles. The final `eor` operates on the modifier and SP to enforce a memory read/write dependency, thus preventing the CPU pipeline from arbitrarily delaying the operations. We have confirmed that our PA-analogue exhibits the expected overhead using our microbenchmarks.

```
eor   Xptr, Xptr, #0x2 ; spend cycles
eor   Xptr, Xptr, #0x3 ; to approximate
eor   Xptr, Xptr, #0x5 ; PA instruction
eor   Xptr, Xptr, Xmod ; overhead
```

Listing 5: PA-analogue simulating PA instructions

7.3.2 nbench-byte benchmarks

For our performance evaluation we use the Linux nbench-byte 2.2.3 synthetic benchmark[4] designed to measure CPU and memory subsystem performance, providing a reasonable prediction of real-world system performance[5]. We follow work such as [6, 10, 22, 33, 37, 10] and use nbench rather than the SPEC CPU standardized applications benchmarks for our evaluation, as nbench allows us verify the functionality of PARTS instrumentation with manageable simulation times on the FVP. The current version of the SPEC CPU benchmark suite, SPEC CPU2017[6], has replaced many tests in the previous, now retired SPEC CPU2006[7] with significantly larger and more complex workloads (up to ~10X higher dynamic instruction counts). As a result, the SPEC simulation times on the FVP proved to be unmanageable; for example, running individual SPEC benchmarks take hours to *days* to complete on the FVP. This is a challenge for both researchers and industry practitioners who rely on hardware simulation for evaluation [30]. We report our results for a subset of SPEC CPU2017 tests in Appendix B.

The nbench benchmarks include 10 different tests. We adopt the same methodology as Brasser et al. [6] and run each test a constant number of iterations for the following cases: a) uninstrumented baseline b) each PARTS scheme (return address signing, forward-edge code pointer integrity, and data pointer integrity) enabled individually, and c) all schemes enabled simultaneously. Compiler optimizations were disabled for all tests. The tests were performed on a 96boards Kirin 620 HiKey (LeMaker version) with a ARMv8-A Cortex A53 Octa-core CPU (1.2GHz) / 2GB LPDDR3 SDRAM (800MHz) / 8GB eMMC, running the Linux kernel v4.18.0 and BusyBox v1.29.2. Figure 5 shows the results, normalized to the baseline. A more detailed description can be found in Appendix A.

Return address signing incurs a negligible overhead of less than 0.5%. This is expected because the estimated per-function overhead of 12 to 16 cycles is typically small compared to the full execution time of the instrumented function. The same holds for indirect calls (6-8 cycle overhead at the call site), although indirect calls are underrepresented in nbench-byte. However, our microbenchmarks for the code pointer integrity instrumentation indicate that a 6 to 8 cycle overhead per indirect function call is reasonable under the assumed QARMA performance.

Data pointer integrity depends largely on the memory profile of the instrumented program. For instance, the floating point emulation test extensively handles data pointers, resulting in a 39.5% overhead. In contrast, the Fourier and neural network benchmarks contain no data pointers and thus incur no discernible overhead. The geometric mean of the overhead of the combined instrumentation for all tests is 19.5%.

7.4 Compatibility Evaluation

Based on our evaluation, PARTS is compatible with standard C code (R3). Because return address signing only affects the instrumented function, it can be safely applied without interfering with the operation of other parts of programs, or uninstrumented code.

PARTS forward-edge code pointer integrity and data pointer integrity can be safely applied to complete code bases. However, if PARTS is applied only to a partial code base, the instrumented code interfacing with non-instrumented (legacy) libraries requires special consideration. In particular pointers used by both instrumented and uninstrumented code cannot be passed directly between them. We discuss solutions for backwards compatibility with legacy libraries in Section 10.

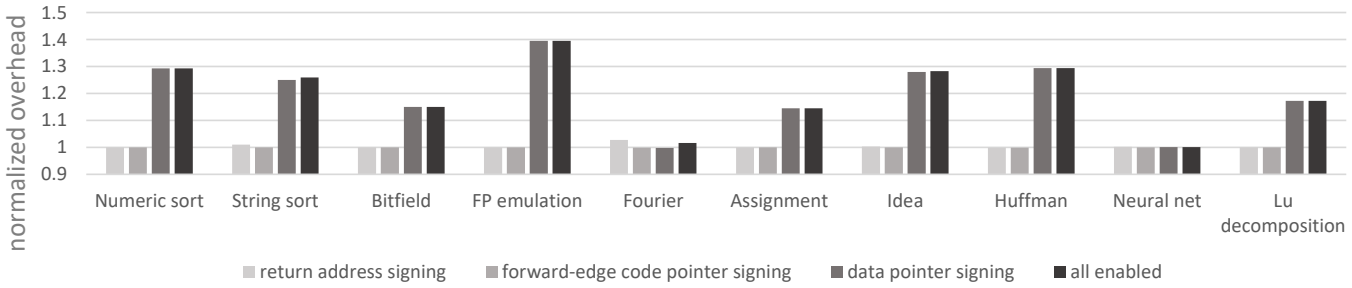
We encountered no compatibility issues with PARTS during our performance evaluation with nbench (Section 7.3).

8 Related Work

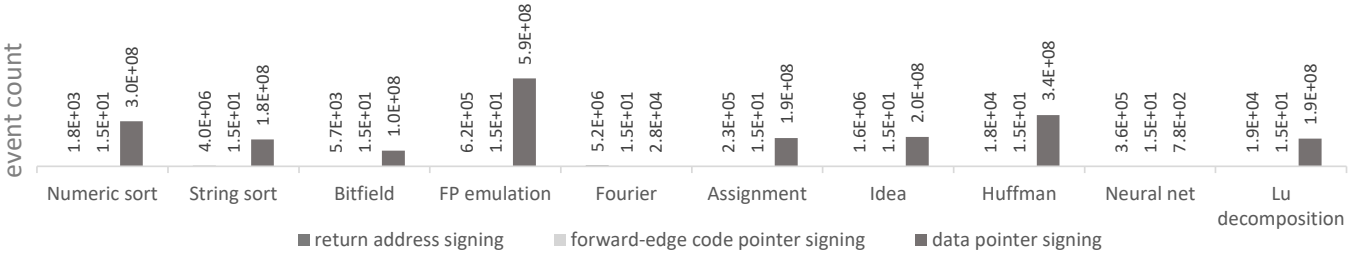
Code-pointer integrity (CPI) [21] protects access to code pointers — and data pointers that may point to code pointers — by storing them in a disjoint area of memory; the *SafeStack*[8]. The SafeStack itself must be protected from unauthorized access. Randomizing the location of the SafeStack is efficient [20], but easily defeated by an attacker who can read arbitrary memory. Stronger protection of the SafeStack using hardware-enforced isolation or software-isolation incurs an average performance overhead of 8.4% or 13.8% in SPEC CPU2006 benchmarks.

Protecting pointers using cryptography. Prior cryptographic defenses against run-time attacks generally assume the attacker cannot read memory. *PointGuard* [12] instruments a program to apply a secret XOR mask to all pointer values. This prevents an attacker from reliably forging pointer values without knowledge of the mask. *Data randomization* [7] extends data masking to cover all data in memory. It uses static points-to analysis and distinct masks to partition memory accesses in separate classes. Neither

[4] http://www.math.utah.edu/~mayer/linux/bmark.html
[5] http://www.math.utah.edu/~mayer/linux/byte/bdoc.pdf
[6] https://www.spec.org/cpu2017/
[7] https://www.spec.org/cpu2006/
[8] https://clang.llvm.org/docs/SafeStack.html



(a) Results of instrumented nbench-byte tests features, normalized to a non-instrumented baseline.



(b) Run-time count of executed locations instrumentable by PARTS. Because the program's memory profile affects performance the benchmark results clearly correlate with observed memory use (e.g., FP emulation has a large data pointer integrity overhead because it uses many data pointers)

Figure 5: nbench benchmark results

PointGuard nor data randomization remain effective under our threat model.

Similarly to ARMv8-A PA, *Cryptographic CFI* (CCFI) [25] uses MACs to protect control-flow data, such as return addresses, function pointers, and vtable pointers. Like PARTS, CCFI uses a function's type signature to separate function pointers to distinct protection domains, but does not protect function pointers embedded in C structures. Unlike PA, CCFI only benefits from hardware-accelerated AES for speeding up MAC, resulting in a high performance overhead (52% overhead on average in SPEC CPU2006 benchmarks). In contrast, PARTS also benefits from hardware-accelerated checks by using ARMv8-A PA instructions, protects both code and data pointers, including pointers embedded in C structures.

Hardware-assisted mechanisms. Various hardware-assisted defenses are described in research literature [15, 42, 41, 14, 38, 40, 28, 32]. The majority of such defenses have only been realized as soft microprocessor prototypes on FPGAs. Here we describe mechanisms available in commercial off-the-shelf processor architectures.

Only a few commercial processors, such as the SPARC M7[9], support tagged memory, which can be used to realize variety of security models (including pointer integrity). ARM recently announced support for memory tagging in the ARMv8.5-A architecture[10]. It enforces that all accesses to memory must be made via a pointer with the correct tag. Pointer tags use the existing address tagging feature in the ARM ISA that partly overlaps with the bits used to store PA PACs, meaning that enabling both features simultaneously reduces the available PAC size by eight bits.

Hardware-assisted memory tagging is designed primarily as a statistical debug aid against use-after-free and other temporal memory errors. *Hardware-Assisted AddressSanitizer* (HWASAN) [34] is an AArch64-specific compiler-based tool that builds upon *AddressSanitizer* (ASAN) — a memory-error detector popular for vetting memory safety bugs during software testing. ASAN can detect both spatial and temporal memory errors. HWASAN can leverage hardware tagged memory, such as SPARC ADI and the upcoming ARMv8.5-A to reduce the performance overhead associated with managing tagged memory checks in software. ASAN / HWASAN are complementary to PARTS, as they provide spatial and temporal safety for data accesses via pointers.

Intel Memory Protection Extensions (MPX) is a hardware feature for detecting spatial memory errors that debuted in the Intel Skylake microarchitecture. MPX is similar to the software based SoftBound [27] and its hardware-based predecessor [15]. Although Intel MPX is a hardware-assisted approach specifically designed to provide spatial memory safety guarantees, it is not faster than software-based approaches [29]. It can cause up to 4x slowdown in the worst

[9] `https://swisdev.oracle.com/_files/What-Is-ADI.html`

[10] `https://community.arm.com/processors/b/blog/posts/arm-a-profile-architecture-2018-developments-armv85a`

case with an average run-time overhead of 50%. It also suffers from other shortcomings, such as the lack of support for multithreading and several common C/C++ idioms. GCC has dropped support for MPX altogether[11].

Control-flow integrity. Carlini et al. [8] define *fully-precise static CFI* as follows: *"An indirect control-flow transfer along some edge is allowed only if there exists a non-malicious trace that follows that edge."* In other words, fully-precise static CFI enforces that execution follows a CFG that contains an edge if and only if that edge is exercised by intended program behavior. Fully-precise static CFI is thus the most restrictive *stateless* policy possible without breaking intended functionality. To date, there exist no implementation of fully-precise CFI; all practical implementations are limited by the precision of CFGs obtained through static *control analysis*.

Carlini et al. further show that all stateless CFI schemes, including fully-precise static CFI are vulnerable to *control-flow bending*; attacks where each control-flow transfer is within a valid CFG, but where the program execution trace conforms to no feasible benign execution trace. For instance, in a stateless policy such as fully-precise static CFI, the best possible policy for return instructions (i.e., backward edges in the CFG) is to allow return instructions within a function F to target any instruction that follows a call to F. In other words, fully-precise static CFI checks if a given control-flow transfer conforms to any of the known control-flow transfers from the current position in the CFG, and does not distinguish between different paths in the CFG that lead to a given control-flow transfer. For this reason CFI is typically augmented with a *shadow call stack* [1, 13] to enforce integrity of return addresses stored on the call stack. We compare PARTS to CFI solutions in Section 9.2.

9 Comparison with other integrity policies

9.1 Fully precise pointer integrity

As discussed in Section 4.1, Pointer Integrity can be loosely defined as a policy ensuring that the value of a pointer at the time of use (dereference or call) corresponds to the value of the pointer when it was created. In this section, we provide a more rigorous definition of Pointer Integrity.

We define *fully-precise pointer integrity* as follows: A pointer dereference is allowed if and only if the pointer is based on its target object. We adopt Kuznetsov et al.'s [21] definition of "*based on*" and say a pointer *P is based* on a target object *X* if, and only if, *P* is obtained at run-time by *"(i) allocating X on the heap, (ii) explicitly taking the address of X, if X is allocated statically, such as a local or global variable, or is a control-flow target (including return locations, whose addresses are implicitly taken and stored on the stack when calling a function), (iii) taking the address of a sub-object y of X (e.g., a field in the struct X), or (iv) computing a pointer expression (e.g., pointer arithmetic, array indexing, or simply copying a pointer) involving operands that are either themselves based on object X or are not pointers."*

Kuznetsov et al's CPI [21] (Section 8) provides fully precise integrity guarantees for *code pointers* by ensuring that accesses to sensitive pointers are safe (sensitive pointers are code pointers and pointers that may later be used to access sensitive pointers). However, CPI requires dedicated, integrity-protected storage for sensitive pointers.

As discussed in Section 7.2, PARTS, and PA solutions in general, achieve an approximation of fully-precise pointer integrity. In particular, PARTS allows the substitution of a pointer P by another pointer P' *based on* object X, if P and P' share the PA modifier. In other words, when PA modifiers are unique to each protected pointer value, PA provides fully-precise pointer integrity. However, ensuring the uniqueness of PA modifiers is not possible in practice due to the following reasons: 1) program semantics may require a set of pointers to be substitutable with each other (e.g., pointers to callback functions) 2) the choice of allowed pointers may depend on run-time properties (e.g., which callback function was registered earlier). In these cases, a unique modifier must be determined at run-time. Fully-precise pointer integrity does *not* imply *memory safety*. In the case of PA, if the modifier is determined at run-time and stored in memory, the PA modifier itself may become a target for an attacker wishing to undermine the integrity policy. To avoid this, modifier values must be derived in a way which leaves the value outside the control of the attacker, e.g., stored in a dedicated hardware register, or read-only program memory.

9.2 Fully-precise static CFI

In contrast to stateless CFI, which allows control-flow transitions present in its CFG regardless of the origin of the code pointer value, PA-based solutions (including PARTS) can preclude forged pointer values from outside the process. The policy that prevents pointer reuse can suffer from limitations similar to those present stateless CFI.

PARTS return address signing provides strong guarantees even when subjected to pointer reuse. In contrast, a stateless CFI policy allows a function to return to any of its call sites. As such, static CFI cannot prevent injection of pointers that are within the expected CFG, i.e., control-flow bending attacks. PARTS additionally requires matching SP values, and that the reused return address originates from a prior function invocation of the same function within the same process for an attack to succeed.

PARTS forward-edge code pointer integrity provides similar guarantees (under reuse attacks) as LLVM's type-based protection (when subjected to any forged pointer). In both

[11] `https://gcc.gnu.org/viewcvs/gcc?view=revision&revision=261304`

cases, attacks are limited to using pointers of the correct dynamic type. PARTS in addition requires that the injected pointer originates from the victim process.

While shadow-stacks protected through randomization can be implemented with minimal performance overhead, our adversary model precludes this approach. Furthermore, software-isolated shadow stack solutions impose impractical performance overheads, and ARM processors do not currently provide direct hardware support for shadow stacks.

10 Conclusion and Future Work

We plan to extend PARTS protection architecture to other protection domains like the OS kernel, or hypervisor. The only significant change for PARTS architecture is to arrange for key configuration for both kernel and EL0 PARTS to be trapped (and managed) on a higher exception level (EL2,3). We are further looking at adding C++ support PARTS. While we do not expect any fundamental problems, some C++ specific features, such as inheritance, cannot be directly handled by our current instrumentation strategy.

Authenticated pointers with PACs cannot be used by legacy code (Section 2.2) while PARTS-instrumented code will trap if pointers without PACs are used. For legacy and PARTS code to interact, we can use wrappers that manipulate function arguments and return values by embedding/stripping PACs. For shared pointers or complex data structures, annotations can disable authentication of selected pointers, allowing programmers to manually adjust pointer conversion to and from legacy code.

Currently, the PARTS compiler assumes shared libraries to be uninstrumented. Instrumented shared libraries must deal with PACs for statically allocated pointers after linking, and thus require changes to the dynamic linker.

Pointer integrity does not imply full memory safety (Section 9.1). Although ARMv8-A PA does not support bounds checking for pointer accesses with authenticated pointers, it has a general-purpose instruction, `pacga`, for producing and validating PACs computed over the contents of two 64-bit registers. This can be used to build authenticated canaries to identify buffer overflow attacks, or to validate the integrity (freshness) of atomic data, such as integer or counter values. In principle, `pacga` instructions can even be chained to validate arbitrary-sized blocks of data.

Finally, effective ways of complementing PA with other emerging memory safety mechanisms like the forthcoming support for memory tagging in ARMv8.5-A is an important line of future work.

Acknowledgments

This work was supported in part by the Academy of Finland under grant nr. 309994 (SELIoT), and the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS).

The authors thank Kostya Serebryany and Rémi Denis-Courmont for interesting discussions and Zaheer Gauhar for implementation assistance.

References

[1] ABADI, M., ET AL. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur. 13*, 1 (Nov. 2009), 4:1–4:40.

[2] APPLE INC. iOS Security — iOS 12. `https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf`, 2018.

[3] ARM LTD. ARMv8 architecture reference manual, for ARMv8-A architecture profile (ARM DDI 0487C.a). `https://static.docs.arm.com/ddi0487/ca/DDI0487C_a_armv8_arm.pdf`, 2017.

[4] ARM LTD. Fast models, version 11.4, fixed virtual platforms (FVP) reference guide. `https://static.docs.arm.com/100966/1104/fast_models_fvp_rg_100966_1104_00_en.pdf`, 2018.

[5] AVANZI, R. The QARMA block cipher family. almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes. *IACR Trans. Symmetric Cryptol. 2017*, 1 (2017), 4–44.

[6] BRASSER, F., ET AL. DR.SGX: Hardening SGX enclaves against cache attacks with data location randomization. `https://arxiv.org/abs/1709.09917`, 2017.

[7] CADAR, C., ET AL. Data randomization. Tech. Rep. MSR-TR-2008-120, Microsoft Research, September 2008.

[8] CARLINI, N., ET AL. Control-flow bending: On the effectiveness of control-flow integrity. In *Proc. USENIX Security '15* (2015), pp. 161–176.

[9] CHECKOWAY, S., ET AL. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 559–572.

[10] CHEN, S., ET AL. Detecting privileged side-channel attacks in shielded execution with DéJà Vu. In *Proc. ACM ASIA CCS '17* (2017), pp. 7–18.

[11] CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., AND IYER, R. K. Non-control-data attacks are realistic threats. In *Proc. USENIX Security '05* (2005), pp. 177–191.

[12] COWAN, C., ET AL. PointGuard™: Protecting pointers from buffer overflow vulnerabilities. In *Proc. USENIX Security '03* (2003), pp. 91–104.

[13] DAVI, L., ET AL. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Proc.NDSS '12* (2012).

[14] DAVI, L., ET AL. HAFIX: Hardware-assisted flow integrity extension. In *Proc. ACM/EDAC/IEEE DAC '15* (2015), pp. 74:1–74:6.

[15] DEVIETTI, J., ET AL. Hardbound: Architectural support for spatial safety of the C programming language. In *Proc. '08* (2008), pp. 103–114.

[16] HU, H., ET AL. Data-oriented programming: On the expressiveness of non-control data attacks. In *Proc. IEEE S&P '16* (2016), pp. 969–986.

[17] INTEL. Control-flow enforcement technology preview. `https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf`, 2016.

[18] ISO/IEC. ISO/IEC 9899:201x committee draft — December 2, 2010. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1548.pdf`, 2010.

[19] KORNAU, T. *Return Oriented Programming for the ARM Architecture*. PhD thesis, Ruhr-Universität Bochum, 2009.

[20] KUZNETSOV, V., ET AL. Poster: Getting the point(er): On the feasibility of attacks on code-pointer integrity. IEEE S&P '15.

[21] KUZNETSOV, V., ET AL. Code-pointer integrity. In *Proc. USENIX OSDI '14* (2014), pp. 147–163.

[22] LEE, S., ET AL. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *Proc. USENIX Security '17* (2017), pp. 557–574.

[23] LI, R., AND JIN, C. Meet-in-the-middle attacks on reduced-round QARMA-64/128. *The Computer Journal 61*, 8 (2018), 1158–1165.

[24] LILJESTRAND, H., ET AL. PAC it up: Towards pointer integrity using ARM pointer authentication. `arXiv:1811.09189 [cs.CR]`, 2019.

[25] MASHTIZADEH, A. J., ET AL. CCFI: Cryptographically enforced control flow integrity. In *Proc. ACM CCS '15* (2015), pp. 941–951.

[26] MICROSOFT. Enhanced Mitigation Experience Toolkit. `https://www.microsoft.com/emet`, 2016.

[27] NAGARAKATTE, S., ET AL. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proc. ACM PLDI '09* (2009), pp. 245–258.

[28] NYMAN, T., ET AL. HardScope: Thwarting DOP with hardware-assisted run-time scope enforcement. `arXiv:1705.10295 [cs.CR]`, 2017.

[29] OLEKSENKO, O., ET AL. Intel MPX explained: An empirical study of Intel MPX and software-based bounds checking approaches. `https://arxiv.org/abs/1702.00719`, 2017.

[30] PANDA, R., ET AL. Wait of a decade: Did SPEC CPU 2017 broaden the performance horizon? In *Proc. IEEE HPCA '18* (2018), pp. 271–282.

[31] QUALCOMM TECHNOLOGIES, INC. Pointer authentication on ARMv8.3. `https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf`, 2017.

[32] ROESSLER, N., AND DEHON, A. Protecting the stack with metadata policies and tagged hardware. In *Proc. IEEE S&P '18* (2018), pp. 1072–1089.

[33] SEO, J., ET AL. SGX-Shield: Enabling address space layout randomization for SGX programs. In *Proc.NDSS '17* (2017).

[34] SEREBRYANY, K., ET AL. Memory tagging and how it improves C/C++ memory safety. `arXiv:1802.09517 [cs.CR]`, 2018.

[35] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proc. ACM CCS '07* (2007), pp. 552–561.

[36] SHACHAM, H., ET AL. On the effectiveness of address-space randomization. In *Proc. ACM CCS '04* (2004), pp. 298–307.

[37] SHIH, M.-W., ET AL. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Proc. NDSS '17* (2017).

[38] SONG, C., ET AL. HDFI: Hardware-assisted data-flow isolation. In *Proc. IEEE S&P '16* (2016), pp. 1–17.

[39] SZEKERES, L., ET AL. SoK: Eternal war in memory. In *Proc. IEEE S&P '13* (2013), vol. 12, pp. 48–62.

[40] TSAMPAS, S., ET AL. Towards automatic compartmentalization of c programs on capability machines. In *Workshop on Foundations of Computer Security 2017* (8 2017), pp. 1–14.

[41] WATSON, R. N. M., ET AL. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *Proc. IEEE S&P '15* (2015), pp. 20–37.

[42] WOODRUFF, J., ET AL. The CHERI capability model: Revisiting RISC in an age of risk. In *Proc. '14* (2014), pp. 457–468.

[43] ZONG, R., AND DONG, X. Meet-in-the-middle attack on QARMA block cipher. *IACR Cryptology ePrint Archive* (2016).

A nbench experimental setup

The nbench benchmarks employs dynamic workload adjustment to allow the tests to expand or contract depending on the capabilities of the system under test. To achieve this, nbench employs timestamping to ensure that a test run exceeds a pre-determined minimum execution time. If a test run finishes before the minimum execution time has been reached, the test dynamically adjusts its workload, and tries again. For example, the Numeric Sort test will construct an array filled with random numbers, measure the time taken to sort the array. If the time is less than the pre-determined minimum time, the test will build two arrays, and try again. If sorting two arrays takes less time than the pre-determined minimum, the process repeats with more arrays.

Since we want to determine the relative overhead in execution time caused by our instrumentation, we employ the methodology described by Brasser et al. [6] and modify nbench to instead run each test a constant number of iterations. The number of iterations was determined individually for each test based on the iteration counts determined by a unmodified nbench run on the FVP. We then instrument the nbench benchmarks using our PA-analogue (Section 7.3.1) and measure the relative execution time between non-instrumented and instrumented nbench tests on the HiKey development platform using the BusyBox `time` utility.

Each individual benchmark test was run 200 times using the pre-determined number of iterations. Figure 5a, in Section 7.3.2 shows instrumentation overhead for individual tests in relation to the uninstrumented test run. Table 3 shows the numeric overhead ratio for each individual test. Because the nbench benchmarks are designed to measure performance in a manner which is operating system agnostic, they are written in ANSI C and only execute in a single thread. We therefore only consider user time when measuring the overhead of the instrumentation, and exclude context switches and system calls.

The run-time overhead of PARTS is dependent on specific run-time events, such as the number of function invocations in the case of return address signing. Figure 5b in Section 7.3.2 shows the order of magnitude of instrumented run-time events in the nbench tests. We also report the user mode run-time for uninstrumented nbench tests, the number of iterations of each individual test, and number of instrumented run-time events in Table 4.

Table 2: Overhead as ratio and standard deviation (σ) for return address signing and (forward-edge) code pointer signing for 505.mcf_r and 519.lbm_r SPEC benchmarks.

Benchmark	Uninstrumented		ret. addr. sign. + code ptr. integrity	
	ratio	σ	ratio	σ
505.mcf_r	1	0.004	1.005	0.004
519.lbm_r	1	0.000	1.000	0.000

B SPEC CPU2017 experimental setup

Due to unmanageable simulation times in the FVP simulator we have verified the correctness of PARTS instrumentation only on a subset of SPEC CPU2017 benchmarks. Specifically, we chose the 505.mcf_r and 519.lbm_r benchmarks from the SPECrate 2017 integer and floating point suites, because these were the smallest C benchmarks in terms of lines of code. The benchmarks were compiled using SPEC `runcpu`, with a AArch64-specific configuration specifying whole-program-llvm[12], with our PARTS-enabled LLVM, as the compiler. We then extracted the bitcode — created by whole-program-llvm during compilation — and used it to instrument and compile the binaries we used for evaluation: one uninstrumented, one instrumented with PA instructions, and one instrumented with our PA-analogue. We enabled both return address and forward-edge code pointer signing for the instrumented binaries.

We run the PARTS-instrumented binaries on the FVP simulator to confirm correct functionality. The simulation time for the tested benchmarks was between 12 and 48 hours. Performance benchmarks, for baseline and PA-enabled binaries, were run on the HiKey devices, using the same setup as our nbench evaluation. The results are shown in Table 2, and are based on five runs of each benchmark. In 505.mcf_r we observed overheads consistent with our results from nbench. We observed no discernible overhead in 519.lbm_r. We attributed this to the following properties of 519.lbm_r: (a) it does not exhibit forward-edge code pointers, and (b) it has few non-leaf function calls in relation to the arithmetic computation performed part of the benchmark.

[12] `https://github.com/travitch/whole-program-llvm`

Table 3: Overhead as ratio and standard deviation (σ) for nbench tests reported separately for uninstrumented, return address signing, (forward-edge) code pointer signing, data pointer signing and all instrumentation enabled.

Test	Uninstrumented		PARTS							
			ret. addr. sign		code ptr. signing		data ptr. signing		all enabled	
	ratio	σ	ratio	σ	ratio	σ	ratio	σ	ratio	σ
Numeric sort	1	0.002	1	0.003	1	0.003	1.293	0.003	1.293	0.003
String sort	1	0.002	1.01	0.002	1	0.002	1.251	0.002	1.259	0.002
Bitfield	1	0.002	1	0.002	1	0.002	1.15	0.002	1.15	0.001
FP emulation	1	0.001	1	0.001	1	0.001	1.395	0.001	1.396	0.001
Fourier	1	0.002	1.027	0.004	0.999	0.003	0.998	0.002	1.016	0.003
Assignment	1	0.001	1	0.002	1	0.002	1.145	0.002	1.145	0.002
Idea	1	0.001	1.004	0.002	1	0.002	1.279	0.002	1.283	0.002
Huffman	1	0.001	0.999	0.001	0.999	0.001	1.294	0.001	1.295	0.002
Neural net	1	0.001	1.002	0.002	1	0.002	1.001	0.002	1.001	0.003
Lu decomposition	1	0.001	1	0.002	1	0.002	1.173	0.002	1.173	0.002
Geometric average	1	-	1.004	-	1.000	-	1.191	-	1.195	-

Table 4: User mode run-time (utime) and standard deviation (σ) in seconds for uninstrumented nbench tests, the pre-determined number of iterations for each individual test, and the number of run-time events that are affected by instrumentation. Non-leaf calls correspond to function invocations protected by return address signing. Leaf calls correspond to function invocations which do no store the value of LR in memory, and thus can be left uninstrumented. Instruction pointers created and indirect calls are instrumented by (forward-edge) code pointer signing, and data pointer loads / stores correspond to events where data pointer instrumentation is active.

Test	Baseline			Instrumented events				
	utime	σ	iterations	non-leaf calls	leaf calls	instr. ptr. created	indirect calls	data ptr. ldr/str
Numeric sort	3.573	0.007	350	1802	7117598	10	5	302212833
String sort	2.971	0.005	125	3977237	1022510	10	5	180105579
Bitfield	2.687	0.004	101647890	5669	4308	10	5	104670943
FP emulation	5.862	0.004	35	616536	37906118	10	5	589518589
Fourier	2.693	0.005	25870	5240188	161	10	5	27504
Assignment	4.414	0.005	10	225602	113353	10	5	190662093
Idea	2.808	0.004	1500	1640184	54420196	10	5	196844406
Huffman	4.212	0.005	1000	17659	46983276	10	5	343176061
Neural net	5.477	0.007	10	359423	441412	10	5	782
Lu decomposition	3.596	0.005	230	18970	441412	10	5	186704928

C ARMv8-A PA Instructions

Table 5: List of PA instructions referred to in the main paper [3]. *PA Key* indicates the PA key the instruction uses. *Addr.* indicates the source of the address to be signed / authenticated (*Xd* indicates that the address is specified using a general purpose register). *Mod.* indicates the modifier used by the instruction (*Xm* indicates that the modifier is specified by a general purpose register.) The *backwards-compatible* column indicates if the instruction encoding resides in the NOP space for pre-existing ARMv8-A processors.

Instruction	Mnemonic	PA Key					Addr.	Mod.	Backwards-compatible
		Instr.		Data		Generic			
		A	B	A	B				
BASIC POINTER AUTHENTICATION INSTRUCTIONS									
Add PAC to instr. addr.	`paciasp`	✓					LR	SP	✓
	`pacia`	✓					*Xd*	*Xm*	✓
	`pacibsp`		✓				LR	SP	✓
	`pacib`		✓				*Xd*	*Xm*	✓
Add PAC to data addr.	`pacda`			✓			*Xd*	*Xm,*	✓
	`pacdb`				✓		*Xd*	*Xm*	✓
Calculate generic MAC	`pacga`					✓			✓
Authenticate instr. addr.	`autiasp`	✓					LR	SP	✓
	`autia`	✓					*Xd*	*Xm*	✓
	`autibsp`		✓				LR	SP	✓
	`autib`		✓				*Xd*	*Xm*	✓
Authenticate data addr.	`autda`			✓			*Xd*	*Xm,*	✓
	`autdb`				✓		*Xd*	*Xm*	✓
COMBINED POINTER AUTHENTICATION INSTRUCTIONS									
Authenticate instr. addr. and return	`retaa`	✓					LR	SP	✗
	`retab`		✓				LR	SP	✗
Authenticate instr. addr. and branch	`braa`	✓					*Xd*	*Xm*	✗
	`brab`		✓				*Xd*	*Xm*	✗
Authenticate instr. addr. and branch with link	`blraa`	✓					*Xd*	*Xm*	✗
	`blrab`		✓				*Xd*	*Xm*	✗
Authenticate instr. addr. and exception return	`eretaa`	✓					ELR	SP	✗
	`eretab`		✓				ELR	SP	✗
Authenticate data. addr. and load register	`ldraa`				✓		*Xd*	*zero*	✗
	`ldrab`					✓	*Xd*	*zero*	✗

Origin-sensitive Control Flow Integrity

Mustakimur Rahman Khandaker
Florida State University
mrk15e@my.fsu.edu

Wenqing Liu
Florida State University
wl16c@my.fsu.edu

Abu Naser
Florida State University
an16e@my.fsu.edu

Zhi Wang
Florida State University
zwang@cs.fsu.edu

Jie Yang
Florida State University
jyang@cs.fsu.edu

Abstract

CFI is an effective, generic defense against control-flow hijacking attacks, especially for C/C++ programs. However, most previous CFI systems have poor security as demonstrated by their large equivalence class (EC) sizes. An EC is a set of targets that are indistinguishable from each other in the CFI policy; i.e., an attacker can "bend" the control flow within an EC without being detected. As such, the large ECs denote the weakest link in a CFI system and should be broken down in order to improve security.

An approach to improve the security of CFI is to use contextual information, such as the last branches taken, to refine the CFI policy, the so-called context-sensitive CFI. However, contexts based on the recent execution history are often inadequate in breaking down large ECs due to the limited number of incoming execution paths to an indirect control transfer instruction (ICT).[1]

In this paper, we propose a new context for CFI, origin sensitivity, that can effectively break down large ECs and reduce the average and largest EC size. Origin-sensitive CFI (OS-CFI) takes the origin of the code pointer called by an ICT as the context and constrains the targets of the ICT with this context. It supports both C-style indirect calls and C++ virtual calls. Additionally, we leverage common hardware features in the commodity Intel processors (MPX and TSX) to improve both security and performance of OS-CFI. Our evaluation shows that OS-CFI can substantially reduce the largest and average EC sizes (by 98% in some cases) and has strong performance – 7.6% overhead on average for all C/C++ benchmarks of SPEC CPU2006 and NGINX.

1 Introduction

The foundation of our software stacks is built on top of the unsafe C/C++ programming languages. C/C++ provides strong performance, direct access to resources, and rich legacy. However, they lack security and safety guarantees of more modern programming languages, such as Rust and Go. Vulnerabilities in C/C++ can lead to serious consequences, especially for low-level software. Many defenses have been proposed to retrofit security into C/C++ programs. Control-flow integrity (CFI) is a generic defense against most, if not all, control-flow hijacking attacks. It enforces the policy that run-time control flows must follow valid paths in the program's control-flow graph (CFG). Since its introduction in the seminal work by Abadi et al. [2], there has been a long stream of research in CFI [1,3,6,9,11–14,16,17,21,25,28,29,31,38,40,41,43,44]. Many earlier systems aim at improving the performance by trading security for efficiency [25,41,43,44], making them vulnerable to various attacks [6, 13, 15, 16]. Recent work focuses more on improving the precision and security of CFI [14,17,21,38], which can roughly be quantified by the average and largest equivalence class (EC) sizes [21]. An EC is a set of targets indistinguishable from each other in the CFI policy; i.e., CFI cannot detect control flow hijacking within an EC. It has been demonstrated the control flow can be "bent" within the ECs without being detected, compromising the protection [6]. Therefore, there is a pressing need to further constrain the leeway of such attacks by reducing the average and largest EC sizes .

One way to improve the security of CFI is to refine the CFG with contextual information, the so-called context-sensitive CFI. Likewise, traditional CFI systems are context-insensitive because they do not collect and use the context information for validating the targets of an ICT. There are many choices of the contextual information. Existing context-sensitive CFI systems use the recent execution history as the context. For example, PathArmor uses the last few branches recorded by Intel processor's Last Branch Record (LBR) [38]; while PittyPat uses the detailed execution paths recorded by Intel processor trace (PT) [14]. Both PathArmor and PittyPat are said to be path-sensitive since they use execution paths as the context. A path-sensitive CFI policy essentially specifies that if the execution comes from this specific path, the ICT can

[1] We use ICT to denote forward indirect control transfers, *excluding* returns. An ICT can be either C-style indirect calls or virtual calls.

only go to that set of targets. There are often multiple paths leading to an ICT. Consequently, the target set of the ICT can be divided into smaller sets by those paths. Another common choice of the context is the call stack [21]. Since the call stack can be represented by its return addresses, such a system is often called call-site sensitive. If the context consists of only one level of return address, it is denoted as 1-call-site sensitive. Similarly, 2-call-site sensitive CFI uses two levels of return addresses as the context.

Execution history based context can substantially reduce the average EC size, but is much less capable in reducing the largest EC size. Unfortunately, the largest EC gives the attacker most leeway in manipulating the control flow without risking detection. For example, PittyPat reports the largest EC size of 218 in SPEC CPU2006, even though it is equipped with the detailed execution history [14]. The fundamental weakness of such context is that most programs only have a small number of execution paths that reach an ICT; i.e., the in-degree of a node (representing an ICT) in the CFG is usually small. If an ICT has hundreds of possible targets, at least one of the ECs will be relatively large. Therefore, such context is more capable in handling small to medium-sized ECs but insufficient for large ones. To address that, we need a more *distributed* context that is not concentrated on the ICT.

In this paper, we propose a new type of context for CFI, origin sensitivity. Origin-sensitive CFI (OS-CFI) takes the origin of the code pointer called by an ICT as the context. It supports both C-style indirect calls and C++ virtual calls with slightly different definitions for them: the origin for the former is the *code* location where the called function pointer is most recently updated; that for the latter is the location where the receiving object (i.e., the object for which the virtual function is called) is created. As usual, returns are protected by the shadow stack, implemented either in software [10,23] or hardware [19]. Our measurement shows that origin sensitivity is particularly effective in breaking down large ECs. For example, it can reduce the largest EC size of a SPEC CPU2006 benchmark from 168 to 2, a reduction of 99% (see Table 1).

We have implemented a prototype of OS-CFI for C and C++ programs. The prototype enforces an adaptive CFI policy that automatically selects call-site or origin sensitivity to protect an ICT in order to improve the system performance without sacrificing security. Its CFG is built by piggybacking on the analysis of a demand-driven, context-, flow-, and field-sensitive static points-to analysis based on SVF (Static Value-Flow Graph) [36]. Its reference monitors are implemented securely and efficiently by leveraging the common hardware features in the commodity Intel processors (MPX and TSX). Our evaluation with SPEC CPU2006, NGINX, and a few real-world exploits shows that the prototype can significantly reduce the average and largest EC sizes, and incurs only a small performance overhead: 7.6% on average for the SPEC CPU2006 and NGINX benchmarks.

In summary, this paper makes the following contributions:

- We propose the concept of origin sensitivity that can substantially reduce both the average and largest EC sizes to improve the security of CFI. Origin sensitivity is applicable to both C-style ICTs and C++ virtual calls. Both types of ICTs are equally important to protect C++ programs.
- We have built a prototype of OS-CFI with the following design highlights: we re-purpose the bound table of MPX to securely store and retrieve origins, and use TSX to protect the integrity of reference monitors; we piggyback on the analysis of SUPA, a precise static points-to algorithm, to built the origin-sensitive CFGs.
- We thoroughly evaluated the security and performance of the prototype with SPEC CPU2006, NGINX, and a few real-world exploits. In particular, we carefully studied the CFGs generated from the points-to analysis and revealed a number of its issues. Detailed CFG generation and measurement are often overlooked in the evaluation of previous CFI systems.

2 Origin Sensitivity

In this section, we first introduce the initial definition of origin sensitivity that is simple, powerful, but potentially inefficient. We then derive a more viable but still effective definition.

2.1 A Simple Definition

OS-CFI takes the origin of the code pointer called by an ICT as the context. If the ICT is a virtual call, the origin is defined as the code location where the receiving object is created, i.e., where its constructor is called; [2] The context of a C-style ICT is similarly defined. A typical example of this type of ICT is an indirect call to a function pointer. The origin of the function pointer is defined as the instruction that initially takes the function address stored in the function pointer.

Next, we use a real-world example from `471.omnetpp` in SPEC CPU2006 to illustrate the concept of the origin (Fig. 1). `471.omnetpp` is a discrete event simulator for large Ethernet networks, written in the C++ programming language. It relies heavily on macros to initialize many objects of the simulated network. Line 1 - 10 shows how simulated networks are initialized: it creates an `ExecuteOnStartup` object for each network to call the network's initialization code; The constructor of `ExecuteOnStartup` sets the private member `code_to_exec` (a function pointer) and adds itself to a linked list (Line 18 - 23). When the program starts, it calls all the queued `code_to_exec` function pointers (`setup` → `executeAll` → `execute`).

The ICT at Line 25 has the largest EC of this program with 168 targets. Call-site sensitivity is not useful here because there is only one call stack to the ICT. Processor-trace-based path

[2] If the object is a global variable, its constructor is conceptually added to a compiler-synthesized function that is called before entering `main()`.

```
1  #define EXECUTE_ON_STARTUP(NAME, CODE)  \
2    static void __##NAME##_code() {CODE;} \
3    static ExecuteOnStartup __##NAME##_reg(__##NAME##_code);
4
5  #define Define_Network(NAME) \
6    EXECUTE_ON_STARTUP(NAME##__net,\
7    (new NAME(#NAME))->setOwner(&networks);)
8
9  Define_Network(smallLAN);
10 Define_Network(largeLAN);
11
12 class ExecuteOnStartup{
13 private:
14   void (*code_to_exec)();
15   ExecuteOnStartup *next;
16   static ExecuteOnStartup *head;
17 public:
18   ExecuteOnStartup(void (*_code_to_exec)()){
19     code_to_exec = _code_to_exec;
20     // add to list
21     next = head;
22     head = this;
23   }
24   void execute(){
25     code_to_exec();
26   }
27   static void executeAll(){
28     ExecuteOnStartup *p = ExecuteOnStartup::head;
29     while (p){
30       p->execute();
31       p = p->next;
32     }
33   }
34 };
35 void cEnvir::setup(...){
36   try{
37     ExecuteOnStartup::executeAll();
38   }
39 }
```

Figure 1: Example to illustrate origin sensitivity

sensitivity can distinguish individual calls to `code_to_exec` (because it records each iteration of the while loop); but it is difficult to decide which target is valid because that depends on the unspecified order in which the constructors are called. Origin sensitivity can handle this case perfectly: the origin of `code_to_exec` is where the related function addresses are initially taken. For example, the macro at Line 9 creates a new function called `__smallLAN__net_code` and passes its address to the constructor of object `__smallLAN__net_reg`. Therefore, Line 9 becomes the origin of this function address. The origin is propagated through the program along with the function address when it is assigned to variables or passed as an argument, in a way similar to how the taint is propagated in taint analysis [33]. At the ICT, the origin is used to verify the target. Because only one function address can be taken at each origin, only one target is possible at the ICT. In other words, origin sensitivity *ideally* can reduce the EC size for this ICT from 168 to 1. The same security guarantee can be achieved for virtual calls because only one class of objects can be created at an origin (Section 2.2).

Execution history based context is limited by the in-degree of an ICT node in the CFG. Assuming the ICT node has n valid targets and m incoming edges, there exists at least one EC with more than $\lceil \frac{n}{m} \rceil$ targets (the pigeonhole principle). For example, the in-degree of the ICT in Fig. 1 is only one for call-site sensitivity; Call-site sensitivity thus cannot reduce this EC at all. The in-degree of this ICT for PathArmor is only 16 because LBR can only record 16 most recent branches. In contrast, origins are associated with the data flow of the program. It traces how function addresses are propagated in the program. Because of this, origin sensitivity can uniquely identify and verify a single target for each ICT. Moreover, this example clearly demonstrates that CFI systems for C++ programs must fully support C-style ICTs because many C++ programs use them (they may even have the largest ECs). Protection of virtual calls alone provides only minimal security.

2.2 A Hybrid Definition

The previous definition of origin sensitivity is conceptually simple but powerful because it can identify a unique target at run-time for each ICT. However, we need to track origins as function addresses are propagated throughout the program in a way similar to how taint is propagated – the origin is the source of the taint, and the ICT is the sink. It is well-known that taint analysis has high overhead, even though the performance of origin tracking could be much better because function addresses are usually not as widespread as the regular data (e.g., a network packet) [23]. This problem is more severe for C-style ICTs because function pointers are frequently copied or passed as arguments. It will not affect virtual calls as much for the following reason: the origin of a virtual call is the location where the receiving object's constructor is called. If an object is copied to another object, we essentially create a new object using its class' copy constructor or copy assignment operator. This creates a new origin for that object. There is thus no need to propagate the origin for objects.

To address the challenge, we propose a hybrid definition of origin sensitivity that combines the origin with call-site sensitivity. More specifically, we relax the definition of the origin as the code location where the related code pointer is most recently updated. In Fig. 1, the only function pointer is `code_to_exec` in the `ExecuteOnStartup` class. It is last updated in the class' constructor at Line 19; i.e., the origin of `code_to_exec` is just Line 19. Clearly, one origin cannot tell Line 9 and 10 (and other places not shown) apart. This can be solved by adding the call-site information to the origin. The origin can now be represented as a tuple of ($\mathcal{CS}$, $\mathcal{I}_o$). $\mathcal{I}_o$ is the instruction that last updates the code pointer; $\mathcal{CS}$ is the immediate caller of the origin function (the function that contains $\mathcal{I}_o$). Under this new definition, the ICT at Line 25 has two origins: (Line 9, Line 19) and (Line 10, Line 19). Note how the two elements of the origin complement each other: $\mathcal{I}_o$ moves the context off the *current* execution path

Benchmarks	Language	Context-insensitive	1-call-site		2-call-site		Origin-sensitive	
		EC_L	EC_L	Reduce by	EC_L	Reduce by	EC_L	Reduce by
445.gobmk	C	427	427	0	427	0	427	0
400.perlbench	C	173	120	31%	113	35%	21	88%
403.gcc	C	54	54	0	54	0	42	22%
464.h264ref	C	10	2	80%	2	80%	1	90%
471.omnetpp	C++	168	168	0	168	0	2	99%
483.xalancbmk	C++	38	38	0	38	0	4	95%
453.povray	C++	11	11	0	11	0	10	10%

Table 1: Effectiveness of hybrid origin sensitivity in reducing the largest EC size (EC_L) as compared to call-site sensitivity

(that reaches the ICT); while $\mathcal{CS}$ adds extra information to $\mathcal{I}_o$ to separate different origins. We use call sites here because they can be directly fetched from the shadow stack in the user space. Other execution contexts such as last-branch record and processor trace can only be accessed in the kernel.

Interestingly, the addition of call sites does NOT make the context for virtual calls more powerful. The origin of a virtual call is where the constructor of the receiving object is called. C++'s constructors cannot be called virtually or indirectly.[3] As such, a call to the constructor can create an object of just one class. There is no ambiguity in the class created, hence no ambiguity in the virtual functions. As such, we keep using the object construction site alone as the origin for virtual calls.

Table 1 demonstrates the hybrid origin sensitivity's capability in reducing the largest EC size as compared to call-site sensitivity [21]. Specifically, we run and recorded the complete execution history of all the C/C++ benchmarks in SPEC CPU2006. We then parsed the history to construct the CFGs for origin and call-site sensitivity. For example, 1-call-site sensitivity uses the most recent return address as the context. For each ICT, we grouped the recorded targets by the last return addresses. Each group was an EC. We report the largest EC sizes in Table 1 for all the benchmarks having the largest EC size greater than or equal to 10 (ten other benchmarks have less than 10 targets for every ICT). The table shows that origin sensitivity consistently out-perform call-site sensitivity. Particularly, we can reduce the largest EC size of `471.omnetpp` by 99%, from 168 to 2. Neither call-site nor origin sensitivity is effective to `445.gobmk` because it contains a loop over a large static array of function pointers (the `owl_defendpat` array). `403.gcc` similarly has a large array (`operand_data`) used in a recursive function (`expand_complex_abs`). Common CFI policies cannot handle such cases because there is not sufficient information in the control flow to separate these targets apart. A similar case is shown in Fig. 4 of Section 4 with the code snippet.

[3]Bjarne Stroustrup's C++ Style and Technique FAQ: "To create an object you need complete information. In particular, you need to know the exact type of what you want to create. Consequently, a 'call to a constructor' cannot be virtual." [8].

3 System Design

In this section, we present the design of our LLVM-based prototype OS-CFI system in detail.

3.1 Overview

Since its inception, many CFI systems have been proposed. To separate OS-CFI from the existing work, we have the following requirements for its design:

- **Precision:** OS-CFI must improve the security by reducing the average and largest EC sizes. Large ECs are the weakest link in a CFI system since they provide the most leeway in "bending" the control flow within the CFI policy
- **Security:** context-sensitive CFI systems, including OS-CFI, have more complex reference monitors to collect and maintain the contextual information. As such, we must protect both the contextual data and the (temporary) data used by reference monitors.
- **Performance:** high performance overhead can severely limit the application of any defense mechanism. OS-CFI must have strong performance relative to the native system.
- **Compatibility:** OS-CFI must support both C and C++ programs. As previously mentioned, any defense for C++ programs must protect both virtual calls and C-style ICTs.

A CFI system consists of three major components: the CFI policy, the CFG generation, and the enforcement mechanism. OS-CFI enforces an adaptive CFI policy that applies either origin or call-site sensitivity for each ICT and adopts the shadow stack to protect returns. OS-CFI's CFG is generated with a precise context-, flow-, and field-sensitive static points-to analysis [36].[4] The enforcement mechanism of OS-CFI uses the hash-table based set-membership test with the hardware acceleration for metadata storage. Next, we describe each component in detail.

[4]Context sensitivity in the points-to analysis is, more precisely, call-site sensitivity. It is named as is for the historical reasons.

3.2 OS-CFI Policy

OS-CFI features an adaptive CFI policy [21] that applies either origin or call-site sensitivity to an ICT, decided by which one is more capable in reducing the EC size. If both have the same effectiveness, we prefer call-site sensitivity because it has lower overhead. If the EC size is already small without context, we just enforce the context-insensitive CFI for this ICT. In addition, call-site sensitivity can use multiple levels of call sites as the context. More levels generally improve the security but incur higher overhead. We limit call-site sensitivity in OS-CFI to at most three call sites. Note that origin sensitivity itself uses 1-call-site on its origins for C-style ICTs (Section 2.2).

We adopt this policy to improve the performance without sacrificing the security: origin sensitivity is a powerful context that can substantially break down large ECs, but it has to collect and maintain more metadata at the run-time. On the other hand, most ICTs in a program have a small number of possible targets. For example, the largest EC size for `400.perlbench` is 173, but its second largest one is only 18. For small ECs, call-site sensitivity is mostly sufficient. We select call-site sensitivity as the secondary policy because last-branch registers (LBR) and processor trace (PT) can only be accessed in the kernel mode, even though they provide more fine-grained execution records. Call-sites instead can be directly fetched from the shadow stack in the user space.

3.3 CFG Generation

A complete and precise CFG is the foundation of any CFI systems. A CFG must be complete to ensure that the resulting CFI system has no false positives (valid control flows reported as invalid). False positives are detrimental to the usability of a security system. Meanwhile, a precise CFG can reduce false negatives, making the system more secure. CFGs can have different levels of precision. For example, a CFG that assumes each ICT can target any address-taken functions is complete but utterly imprecise. Most CFI systems utilize static points-to analysis to construct CFGs because such analysis is (supposedly) conservative and the generated CFGs are complete. The precision of the points-to analysis directly decides the quality of the generated CFGs. A precise points-to analysis is often context- and flow-sensitive, such as SUPA [34].

OS-CFI enforces an adaptive CFI policy that combines call-site and origin sensitivity, which require call-site and origin-sensitive CFGs, respectively. We represent these CFGs as a set of tuples:

- **Call-site sensitive CFG:** each tuple of this CFG has the following form: $(\mathcal{CS}_{1/2/3}, \mathcal{I}_i, \mathcal{T})$. $\mathcal{CS}$ represents the callers of the current function on the call stack. OS-CFI may use up to three call sites. $\mathcal{I}_i$ is the address of the ICT instruction itself. It is either a C-style ICT or a virtual call. $\mathcal{T}$ is the set of valid targets under this context.

```
typedef void (*Format)();
class Base {
protected:
  Format fmt;
public:
  Base(/* Base_o.vPtr, origin */) {
    // store_metadata(Base_o.vPtr, Base::vTavle,
    //                origin);
  }
  ~Base() {}
  virtual void set(Format fp) {
    fmt = fp;
    // store_metadata(fmt.addr, fp.value,
    //                Base:set_loc1, Base::set_ctx);
  }
  void print() {
    // ccall_ref_monitor(fmt.addr, fmt.value);
    fmt();
  }
};
class Child : public Base {
public:
  Child(/* Child_o.vPtr, origin */) {
    // Base(Child_o.vPtr, origin);
    // store_metadata(Child_o.vPtr, Child::vTable,
    //                origin);
  }
  ~Child() {}
  void set(Format fp) {
    fmt = fp;
    // store_metadata(fmt.addr, fp.value,
    //           Child::set_loc1, Child::set_ctx);
  }
  void print() {
    // ccall_ref_monitor(fmt.addr, fmt.value);
    fmt();
  }
};
void exec () {
  Base *bp = new Base(); // call constructor
  // vcall_ref_monitor(Base_o.vPtr,
  //               Base::vTable, Base::set())
  bp->set(&targetA);
  bp->print();

  Child ci;              // call constructor
  ci.set(&targetB);
  ci.print();

  bp = &ci;
  // vcall_ref_monitor(Child_o.vPtr,
  //                Child::vTable, Child::set())
  bp->set(&targetB);
  bp->print();
}
```

Figure 2: An example featuring C-style ICT and virtual call.

- **Origin sensitive CFG for C-style ICTs:** each tuple of this CFG has the form of $((\mathcal{CS}_o, \mathcal{I}_o), \mathcal{I}_i, \mathcal{T})$. $(\mathcal{CS}_o, \mathcal{I}_o)$ is the hybrid origin of the function pointer. In particular, $\mathcal{I}_o$ is the

last store to the related function pointer; while $\mathcal{I}_i$ is where the function pointer is actually called.

- **Origin sensitive CFG for virtual calls:** each tuple of this CFG has the form of $(\mathcal{I}_o, \mathcal{I}_i, \mathcal{T})$. $\mathcal{I}_o$ is the location where the receiving object of a virtual call is constructed.

We use the C++ code in Fig. 2 to illustrate how the CFGs are generated (and later enforced). There are two classes, `Base` and `Child`. `Child` inherits `Base`. `Base` has a protected function pointer `fmt` that can only be set by virtual function `set`. `fmt` is called indirectly by the `print` function, which is overloaded in `Child`. As such, this example has both C-style ICTs (Line 18 and 36) and virtual calls (Line 43 and 53).

Our CFG construction algorithm is based on SVF, a static tool that "enables scalable and precise inter-procedural dependence analysis for C and C++ programs" [36]. SVF constructs a whole-program sparse value-flow graph (SVFG) that conservatively captures the program's def-use chains. SVFG is imprecise because it overestimates the points-to sets when constructing the def-use chains. SUPA is a client of SVF. It is an on-demand context-, flow-, and field-sensitive points-to analysis based on the SVFG. It improves the precision by refining away imprecise value-flows in the SVFG with strong updates [34]. Our CFGs are constructed on top of the refined SVFG of SUPA.

SUPA is a demand-driven points-to analysis. It traverses the program's SVFG reversely to compute the points-to sets. OS-CFI queries SUPA for every ICT in the program. In response, SUPA starts traversing the def-use chains to solve the request. OS-CFI piggybacks on SUPA during this traversal. Specifically, OS-CFI monitors the traversed nodes to identify the origin of the ICT. When SUPA stops the traversal, it has located the targets of the ICT, and OS-CFI has collected all the elements required to generate the tuples for the ICT. Next, we describe how OS-CFI generates the related tuples for the indirect calls in Fig. 2 since they are the more complex cases.

In Fig. 2, `Base` has a protected member function pointer `fmt`, which is called by `Base.print` and `Child.print`. Therefore, OS-CFI requests SUPA to resolve the points-to set for both uses of `fmt`. We describe the resolution of the first call to `fmt` by `Base.print` here. This indirect call to `fmt` is actually a use of the `fmt` filed of the `this` object. SUPA can create a def-use chain from Line 18 to the assignment of the `fmt` field at Line 12 because it is field-sensitive. This def-use chain is linked by the `bp` pointer created at Line 40. When traversing this def-use chain, OS-CFI marks the first store to `fmt` as the origin for the ICT. The traversal continues until SUPA has reached the call to the `set` function at Line 43. OS-CFI then marks Line 43 as the call-site for the origin. Now, SUPA has located the target of `fmt` (`targetA`). Note that SUPA is precise enough to exclude `targetB` from the points-to set of `fmt` at Line 18. The CFG tuple for Line 18 is ((Line 43, Line 12), Line 18, `targetA`). Tuples for other CFGs can be similarly constructed.

3.4 Enforcement Mechanism

Overview: we use a hash-table based set membership test to enforce the OS-CFI policy. Specifically, we create a hash table for each CFG and instrument the program (at the LLVM IR level) to collect the run-time metadata at the origins. OS-CFI verifies the targets at each ICT site by searching the hash table for matches. As mentioned before, the CFGs are encoded as tuples. The hash function simply takes each element of the tuple and xor them together. It is extremely fast and leads to few conflicts in practice. The hash function can be easily replaced if necessary.

In this section, we will describe the instrumentation in detail. Note that OS-CFI adopts both call-site and origin sensitivity. The context for the former is the return addresses on the call stack, which can be fetched from the shadow stack at the ICT sites. As such, call-site sensitivity is enforced (instrumented) only at the ICT sites. However, we need to instrument both the origin and ICT sites for origin sensitivity.

3.4.1 Instrumentation at Origin Sites

OS-CFI has different origins for C-style ICTs and virtual calls. We describe them separately.

C-style ICTs: the origin for this type of ICTs is defined as $(\mathcal{CS}_o, \mathcal{I}_o)$. $\mathcal{I}$o is the address of the origin (i.e., the instruction that last writes to the function pointer), and $\mathcal{CS}_o$ is the most recent return address on the call stack. Since we are instrumenting the origin, $\mathcal{I}_o$ is a known constant. $\mathcal{CS}_o$ can be retrieved directly from the shadow stack. To store the metadata, we use the address of the function pointer as the key and the context, $(\mathcal{CS}_o, \mathcal{I}_o)$, as the value. At the ICT site, we can recover the context with the function pointer address. Fig. 2 has been annotated with the calls to store metadata at Line 13 and 31 for `Base.fmt` function pointer.

Virtual calls: the origin for virtual calls is the location where the object is created $(\mathcal{I}_o)$. $\mathcal{I}_o$ is also a known constant at the origin site. To store this metadata, we use the object's `vPtr` pointer address as the key and $\mathcal{I}_o$ as the value. In C++, every object with virtual functions has a hidden member named `vPtr` that points to its `vTable`. `vTable` is used by the compiler for dynamic dispatching of virtual function calls. It is a table of virtual function pointers. Each virtual function of a class has a fixed offset in `vTable`. A virtual call is thus compiled as an indirect call to the corresponding entry in `vTable`. Initially, a sub-class inherits its base class' `vTable`. If the sub-class overrides a virtual function, it sets the related function pointer in `vTable` to its own function's address. Consequently, the virtual call can call either the base or sub-class' virtual function, decided by the class of the receiving object. COOP attacks essentially compromise the binding of `vPtr` and `vTable` [32]. After an object is created, its `vTable` will not be changed.

The reason we use `vPtr`'s address as the key (instead of the base address of the object, even though they both can

uniquely identify the object) will be clarified as we discuss the metadata storage. The instrumentation is added to each class' constructor so that we only need to insert the code once (instead of once at each location where the constructor is called). Line 7 and 25 of Fig. 2 show the added code. We simply pass the origin from the object allocation site to the constructor as a hidden parameter. Note that the constructor of a sub-class calls the constructor of its base classes first. We thus add the code near the end of the constructor so that the metadata will not be mistakenly overwritten.

Metadata storage: the storage of the contextual information (i.e., the metadata) is a key design component of OS-CFI. The metadata of OS-CFI is organized as (key, value) pairs. The key is the address of the function pointer or the receiving object's `vPtr` pointer. The value is the origin associated with the key. We store the (key, value) pair at each origin site, and query the storage with the same key to retrieve the origin information at each ICT site. The performance and security of the storage is critical to OS-CFI. In our prototype, we uniquely (ab)use the hardware-based bound table of Intel MPX for metadata storage [18].

MPX is a hardware-based bound check system. With the support of the compiler, run-time, and kernel, MPX can check the bounds of memory access to prevent memory errors, such as buffer overflows and over-reads. However, whole program bound check is hard to implement correctly and efficiently, even with the hardware support [26]. In fact, the MPX support will be removed from GCC in version 9.0, after it was just integrated in 5.0 [27]. This leaves the whole MPX hardware free-to-use by OS-CFI and other (security) systems.

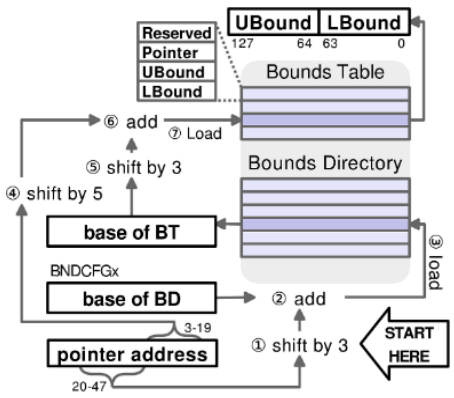


Figure 3: MPX operations, from Intel's manual [30]

MPX 's bound table is indexed by the address of a pointer (i.e., the key). Each key has its own unique bound table entry, which consists of the content of the pointer, the upper bound, and the lower bound. The bound table is organized and operates like a two-level page table, as shown in Fig. 3: the bounds directory points to the second-level bounds tables; each bounds table contains a number of bound entries. The pointer address is divided into two indexes. To locate a bound entry, MPX first indexes into the bounds directory to retrieve the base of the related bounds table, and then uses the second index to locate the related bound entry. If a bounds table does not exist, the kernel allocates a new one and links it to the bounds directory. The base of the bounds directory is stored in a special register, `BNDCFGx`, inaccessible to the user space.

We can store all the origins in the MPX bound table. Even though we are supposed to store the lower and upper bounds in this table, the hardware does not perform any validations on the bounds, as confirmed by both the official document and our experiments. Accordingly, we can store and retrieve arbitrary numbers in the bounds (after doing some simple calculations on these two numbers). This design not only significantly accelerates the access of the metadata but also improves the security: the MPX table stores the content of the key along with the bounds. When querying the table for a key, we need to provide the pointer's address and its content. If the provided pointer content mismatches that in the table, MPX will return an error. Therefore, *we can detect any manipulation of these pointers, after they have been stored, without the extra performance penalty.*

For virtual calls, OS-CFI uses the address of the receiving object's `vPtr` pointer, which points to the object's `vTable`, as the *key*. As such, OS-CFI can readily detect any COOP attack, which compromises the object's `vPtr` pointer [32], similar to how object-type integrity (OTI [4]) works. Note that OTI is not a complete protection for C++ virtual calls because the attacker can still call the "correct" virtual functions of an unintended object. In contrast, OS-CFI provides more comprehensive and complete protection for C++ programs because it not only enforces the precise CFI but also protects both virtual calls and C-style ICTs.

OS-CFI can use other keys for the virtual call. For example, it can use the address of the object itself as the key and a constant number as its content. We can retrieve the origin from the MPX table by this key and its "content", and then enforce the CFI against the unique target decided by the origin. This is because the origin (i.e., the location where the receiving object is constructed) identifies the exact class of the object, hence the unique target of the virtual call (Section 2.2). Our prototype uses the address of the `vPtr` pointer as the key because it is more natural for MPX (i.e., the `vPtr` pointer address and its *real* content). This is not strictly necessary since OS-CFI nevertheless can detect these attacks.

3.4.2 Instrumentation at ICT Sites

The instrumentation at each ICT site is rather straightforward: it first queries the metadata storage with the key and its content to retrieve the origin. If the origin exists, it further checks the corresponding hash table whether the origin and the target are valid for the ICT site. Use the indirect call as an example, we need to reconstruct the tuple of $((\mathcal{CS}_o, \mathcal{I}_o), \mathcal{I}_i, \mathcal{T})$. $(\mathcal{CS}_o, \mathcal{I}_o)$

is the origin fetched from the metadata storage; $\mathcal{I}_i$ represents the address of indirect call instruction; $\mathcal{T}$ is the target, i.e., the value of the function pointer. OS-CFI then queries the hash table whether this tuple is one of its items. If so, the indirect call is allowed. For call-site sensitivity, OS-CFI retrieves the return addresses from the shadow stack and uses a similar method to verify the target under this context.

3.4.3 Protection of Metadata

The MPX table is protected by ASLR. The 64-bit address space provides enough entropy to render brute force attacks difficult, if not impossible. Note that the access to the bounds directory and tables is implemented by the hardware, similar to the access to page tables. Particularly, the base of the bounds directory is stored in a kernel-mode register, inaccessible to the user space. Therefore, the address of the MPX table will not be leaked to the user space. This prevents the attacker from overwriting the metadata stored in the MPX table. We consider side-channel attacks out-of-scope. A number of defenses have been proposed to detect/mitigate them [7, 45]. If a stronger protection of the MPX table is necessary, we can use MPX's bound check to protect it, with a small additional overhead [4, 22]. Note that this use of the bound check does not conflict with OS-CFI's use of the MPX table since the bound check can be performed with just the bound registers.

The hash tables for CFGs are protected as the read-only memory and thus cannot be changed by the attacker. A subtle attack surface is the temporary data used by the reference monitors to search the hash tables. Context-sensitive CFI systems have more complex reference monitors, which have to use the memory (instead of all registers) to store the temporary data. This makes them vulnerable to race conditions in a brief time widow. To address that, we utilize the transactional memory (Intel TSX) to protect the reference monitors [21]. Specifically, TSX keeps tracks of the memory accessed by a transaction and aborts the transaction if any of that memory is changed by others (e.g., attacks). We enclose each reference monitor in a transaction and repeat the transaction if it fails because, with a very low probability, transactions could fail without attacks (e.g., because of cache conflicts).

3.4.4 CFG Address Mapping

Our CFGs are generated using SUPA, a LLVM-based points-to analysis. The resulting CFGs are accordingly encoded as the LLVM IR locations. However, the instrumentation requires the run-time addresses of the CFG nodes. We need to map the IR locations to the run-time addresses. Previous systems often use the debug information for this purpose, which works for function addresses but not as well for call sites because they are not in the symbol table. To address that, heuristics such as the code structure are used to infer the locations of call sites. This approach works most of the time but may not be reliable when the compiler optimization is turned on.

Benchmarks	Out of budget			Empty points-to sets	
	# of ICTs	SUPA	Type	# of ICTs	Type
400.perlbench	54	639	349	2	7
403.gcc	46	544	218	20	107
445.gobmk	22	1645	1637	1	4
447.dealII	0	-	-	23	37
450.soplex	0	-	-	157	11
453.porvray	47	317	79	22	24
471.omnetpp	37	143	44	67	21
483.xalancbmk	0	-	-	349	29
NGINX	141	1066	102	4	34

Table 2: Failed cases of SUPA and the improvements of our type-based matching. Column 3, 4, and 6 show the largest EC sizes for SUPA and the type-based matching. SUPA works for all other benchmarks.

OS-CFI solves this problem without using any heuristics. Specifically, we insert a custom label after each call instruction. We then use the label-as-value extension of Clang to store the label addresses to an array and assign the array to a custom section. The compiler will automatically convert these labels to the addresses. Note that the array has to be marked as used so that the later stages of the compiler will not optimize it away. These extensions are supported by both GCC and Clang/LLVM. A benefit of this approach is that OS-CFI theoretically can support ASLR because the loader will automatically fix these addresses when the program is loaded. This resolves the run-time addresses of call sites. For the rest of the data in the CFGs, we encode the ICT and origin sites as IDs (specifically the hashes of their source code locations) since their concrete values are irrelevant. The target function addresses are obtained from the symbol table. With the address mapping information, we can encode the CFGs in the hash tables.

4 Evaluation

In this section, we evaluate how effectively OS-CFI can improve the security by reducing the largest and average EC sizes and what is the performance overhead for some standard benchmarks. We also experimented with real-world exploits to demonstrate how OS-CFI can block them.

4.1 Improvement in Security

The security of a CFI system can be measured by its CFGs, assuming the enforcement mechanism does not introduce imprecision. Particularly, the average and largest EC sizes reflect the overall quality of the CFGs [21]. OS-CFI's CFGs are derived from SUPA, a static points-to analysis. Therefore, the quality of its CFGs are affected by SUPA.

Advancements and issues of SUPA: SUPA is a scalable and precise context-, flow-, and field-sensitive points-to analysis. Public availability of such algorithms is, to the best of our

knowledge, non-existent before the release of SUPA. Though, SUPA has its own issues. More specifically, SUPA is an on-demand points-to analysis. It allocates a specific amount of (configurable) budgets for each query. We found that, even with a generous budget on a relatively powerful machine (a 16-core Xeon server with 64GB of memory), SUPA can still run out of budgets for complex programs, such as `gcc` and `perlbench`. When that happens, SUPA may return wrong results in the points-to sets (e.g., functions with wrong signatures). In addition, SUPA may return empty results because of the language features it does not yet support (e.g., C++'s pointers to member functions). [5] When these issues were detected, we used a simple type-based matching to fix the points-to sets.

The results are listed in Table 2. Generally speaking, the type-based matching can substantially reduce the target sizes for the failed cases. For example, we can reduce the size of largest EC size of NGINX returned by SUPA from 1,066 to just 102. A noticeable exception is `gobmk`, which has more than 1,600 address-taken functions with the same signature (`(int, int, int, int)`). Our manual examination of the program shows that no ICTs in `gobmk` should have more than 500 targets.

It is no surprising that SUPA has some issues because scalable points-to analysis with multiple types of sensitivity is a hard problem. We suspect SUPA is still more scalable and/or precise than other publicly available points-to analysis algorithms, and expect these problems to be solved soon. However, these issues can put OS-CFI to a disadvantage currently – our CFGs are generated by piggybacking on the SUPA as it traverses the SVFG. For these failed cases, SUPA prematurely stops traversing the graph. Accordingly, we cannot generate call-site or origin sensitive edges for these failed ICTs. We instead have to fallback to the context-insensitive CFI for them. We would like to emphasize that *the issues of SUPA does not invalid the usefulness of origin sensitivity.* These are two orthogonal problems.

Effectiveness of OS-CFI: Table 3 shows how OS-CFI can significantly reduce both the average and largest EC sizes. This table focuses on measuring the effectiveness of origin sensitivity; the table thus does not take the ICTs that SUPA failed to resolve (Table 2) into consideration. We will present the overall results with all the ICTs in Table 4. Additionally, Table 3 compares OS-CFI against the context-insensitive CFG, which can be calculated directly from SUPA's points-to sets. It is technically difficult to compare the origin-sensitive CFG against path-sensitive CFGs, such as these in PathArmor [38] or PittyPat [14]: they both use online points-to analysis to calculate the valid targets, with the help of run-time information; i.e., their CFGs are dynamically generated and are, most likely, incomplete for a fair comparison. In addition, the comparison to call-site sensitive CFG has been shown in Table 1. Nevertheless, the absolute average and largest EC sizes OS-CFI can achieve still clearly show its effectiveness. For example, OS-CFI can reduce the largest EC size of `omnetpp` from 109 to 2, a 98% reduction. It can also reduce the average EC size of `gcc` by 96% from 30.7 to 1.3. Overall, OS-CFI can reduce the average and largest EC sizes by 59.8% and 60.2% on average, respectively.

[5] SUPA may also returns empty results for ICTs in the dead code.

Benchmark	# ICTs	No Context		OS-CFI		Reduce by	
		Avg	Lg	Avg	Lg	Avg	Lg
400.perlbench	79	23.8	39	2.8	10	88%	74%
401.bzip2	20	2.0	2	1.0	1	50%	50%
403.gcc	347	30.7	169	1.3	27	96%	84%
433.milc	4	2.0	2	1.0	1	50%	50%
445.gobmk	36	8.1	107	1.5	12	82%	89%
456.hmmer	9	2.8	10	1.0	1	64%	90%
464.h264ref	367	2.0	12	1.0	2	50%	83%
444.namd	12	2.5	3	1.0	1	60%	67%
447.dealII	79	2.1	3	1.2	3	43%	0%
450.soplex	317	1.0	1	1.0	1	0%	0%
453.porvray	45	9.3	17	1.6	5	83%	71%
471.omnetpp	331	5.7	109	1.0	2	83%	98%
473.astar	1	1.0	1	1.0	1	0%	0%
483.xalancbmk	1492	2.5	11	1.0	1	60%	91%
NGINX	248	9.4	43	1.1	19	88%	56%

Table 3: Improvement of precision by OS-CFI over context-insensitive CFI, shown by the significant reduction in the average (`Avg`) and largest (`Lg`) EC sizes.

```
1  typedef int (*EVALFUNC)(int sq,int c);
2  static EVALFUNC evalRoutines[7] = {
3    ErrorIt,
4    Pawn,
5    Knight,
6    King,
7    Rook,
8    Queen,
9    Bishop };
10
11 int std_eval (int alpha, int beta) {
12   for (j = 1, a = 1; (a <= piece_count); j++) {
13     score += (*(evalRoutines[piecet(i)]))
14                                (i,pieceside(i));
15   }
16 }
```

Figure 4: An example in `sjeng` where the ICT at Line 15 has no context in SUPA.

Overall statistics of OS-CFI: Table 4 shows the overall statistics of OS-CFI when applied to all the C/C++ benchmarks in SPEC CPU2006 and NGINX. The second and third columns show the number of C-style ICTs and virtual calls, respectively. It is clear that C++ programs often use C-style ICTs. Any protection for C++ programs thus must support both types of ICTs. OS-CFI enforces an adaptive policy where an ICT can be protected by either origin or call-site sensitivity. However, it may fall back to the context-insensitive policy if SUPA fails to resolve the points-to set for the ICT or if SUPA

Benchmark	#ICTs		OS-CFI / Adaptiveness												
	#c-Call	#vCall	Origin sensitive				Call-site sensitive				Context-insensitive			Overall	
			#ICTs	#Origins	Avg	Lg	#ICTs	Depth	Avg	Lg	#ICTs	Avg	Lg	Avg	Lg
400.perlbench	135	0	53	49	2.5	6	18	2	3.2	8	64	25.5	349	11.4	349
401.bzip2	20	0	20	4	1.0	1	0	0	0	0	0	0	0	1.0	1
403.gcc	413	0	249	139	1.0	1	88	2	1.0	1	76	29.8	218	3.4	218
433.milc	4	0	0	0	0	0	4	1	1.0	1	0	0	1	1.0	1
445.gobmk	59	0	29	12	1.4	3	7	3	1.0	1	23	661.7	1637	246.3	1637
456.hmmer	9	0	1	15	1.0	1	1	1	1.0	1	7	1.0	1	1.0	1
458.sjeng	1	0	0	0	0	0	0	0	0	0	1	7	7	7.0	7
464.h264ref	367	0	318	52	1.0	1	7	1	1.5	2	42	1.7	2	1.1	2
444.namd	12	0	12	30	1.0	1	0	0	0	0	0	0	0	1.0	1
447.dealII	7	95	73	59	1.0	1	3	2	1.0	1	26	27.9	37	6.7	37
450.soplex	0	357	0	0	0	0	0	0	0	0	357	1.2	11	1.2	11
453.porvray	38	76	37	29	1.5	5	8	3	1.0	1	69	14.4	79	7.5	79
471.omnetpp	39	403	276	243	1.0	1	21	2	1.0	1	145	27.5	44	9.2	44
473.astar	0	1	0	0	0	0	0	0	0	0	1	1.0	1	1.0	1
483.xalancbmk	18	2073	1486	1544	1.0	1	6	3	1.0	1	599	7.2	29	3.5	29
NGINX	393	0	184	169	1.0	1	37	3	1.0	1	172	13.8	102	6.6	102

Table 4: Overall distribution of ICTs among origin sensitive, call-site sensitive, and context-insensitive ICTs. The second column shows the total number of C-style indirect calls, while the third column shows the number of virtual calls. We omit the results of `mcf`, `libquantum`, and `sphinx3` from this table because they do not have ICTs in their main programs.Columns marked with `Avg` and `Lg` show the average and largest EC sizes, respectively.

fails to provide the context for the ICT. The latter could happen if the ICT uses global function pointers (e.g., Fig. 4). Specifically, the ICT in Line 13 calls global function pointers defined in the `evalRoutines` array. Because `evalRoutines` is initialized statically, SUPA will not generate any context for this ICT. Neither will origin or call-site sensitivity improve the precision of such cases because the target is decided by the index (`piecet(i)`). Even μCFI can only provide the same precision as context-insensitive CFI in this case because the constraint data (`piecet(i)`) can potentially be compromised before being captured by μCFI using processor trace [17].

In Table 4, most ICTs are protected by origin sensitivity. Interestingly, the number of origins (the 5th column) is often less than the number of ICTs (the 4th column) because some ICTs may share origins. Both origin and call-site sensitivity can reduce most of the average and largest EC sizes to less than 2 and 5, respectively [6]. Note that OS-CFI prefers call-site sensitivity over origin sensitivity. Origin sensitivity is used only if call-site sensitivity fails to provide sufficient security. Therefore, ICTs protected by origin sensitivity generally have larger ECs than those by call-site sensitivity. OS-CFI similarly prefers context insensitivity over call-site sensitivity. ICTs that SUPA failed to resolve are also context-insensitive. The majority of the largest ECs in the context-insensitive ICTs come from the problems in SUPA. We expect OS-CFI to substantially break down most of these ECs once the problems in SUPA are resolved.

Next, we present a few case studies to illustrate how OS-CFI can successfully break down largest ECs in some programs of SPEC CPU2006.

[6]Table 3 and 4 cannot be compared directly because Table 4 includes the ICTs SUPA failed to resolve while Table 3 does not.

4.1.1 Case Studies

Largest EC in `471.omnetpp`: Fig. 5 shows the virtual call in `471.omnetpp` with the largest number of targets – 35 targets in context-insensitive CFG. The related ICT is located in Line 5, which calls the virtual destructor declared in Line 10. Unlike constructors, destructors in C++ can be called virtually. `cObject` is the root class in `471.omnetpp`. It is inherited by many other classes, such as `CModuleType` (Line 12) and `cArray` (Line 17). Interestingly, `cArray` is a container of `cObject` even though itself is a sub-class of `cObject`. `cArray` has a clear function that calls `discard` on every contained object, which in turn calls the virtual destructor. Clearly, the ICT in Line 5 can target any virtual destructor of `cObject`'s sub-classes.

OS-CFI defines an origin for each location where an object of *cObject* or its sub-class is created. Because the constructor in C++ cannot be virtually called, each origin is associated with exactly one class. As such, OS-CFI can uniquely identify the specific destructor to be called; i.e., it can enforce a perfect CFI policy at Line 5 since the EC size is 1.

Largest EC in `483.xalancbmk`: The ICT with the largest EC size in `483.xalancbmk` is a C-style indirect call (Fig. 6, Line 11). The function pointer is defined in Line 4 as a private member of `XMLRegisterCleanup`. As such, it can only be set by function `registerCleanup` (Line 6). In Line 15 and 16, two objects of `XMLRegisterCleanup` are created. They register the cleanup function at Line 18 and 19, respectively.

The ICT at Line 11 have a EC size of 38. Since this is a C-style ICT, the origin is defined as $(\mathcal{CS}_o, \mathcal{I}_o)$. $\mathcal{I}_o$ is the location of the instruction that last writes to the function pointer (Line 7), while $\mathcal{CS}_o$ is the call sites of the store function (Line 18

```
class cObject{
protected:
  void discard(cObject *object){
    if(object->storage() == 'D')
      delete object;
    else
      object->setOwner(NULL);
  }
public:
  virtual ~cObject();
}
class cModuleType:public cObject{
  ~cModule(){
    delete [] fullname;
  }
}
class cArray:public cObject{
private:
  cObject **vect;
public:
  clear(){
    for (int i=0; i<=last; i++){
      if (vect[i] && vect[i]->owner()==this)
        discard( vect[i] );
    }
  }
}
```

Figure 5: Virtual call with the largest EC in `471.omnetpp`

```
class XMLRegisterCleanup
{
private:
  XMLCleanupFn m_cleanupFn;
public :
  void registerCleanup(XMLCleanupFn cleanupFn) {
    m_cleanupFn = cleanupFn;
  }
  void doCleanup() {
    if (m_cleanupFn)
      m_cleanupFn();
  }
}
XMLTransService::XMLTransService(){
  static XMLRegisterCleanup mappingsCleanup;
  static XMLRegisterCleanup mappingsRecognizerCleanup;

  mappingsCleanup.registerCleanup(reinitMappings);
  mappingsRecognizerCleanup.registerCleanup
                      (reinitMappingsRecognizer);
}
```

Figure 6: The ICT with the largest EC in `483.xalancbmk`

and 19). As such, OS-CFI can enforce a perfect CFI policy for this ICT with an EC size of 1.

Largest EC in `456.hmmer`: `456.hmmer` is a benchmark to measure the performance of searching a gene sequence database. It begins its execution by reading the HMM (Hidden

```
struct hmmfile_s{
  int (*parser)(struct hmmfile_s *,
                struct plan7_s **);
};
typedef struct hmmfile_s HMMFILE;

HMMFILE *HMMFileOpen(char *hmmfile,
                     char *env){
  HMMFILE *hmmfp;
  hmmfp = (HMMFILE*)
          MallocOrDie(sizeof(HMMFILE));
  hmmfp->parser = NULL;

  if(magic == v20magic){
    hmmfp->parser = read_bin20hmm;
    return hmmfp;
  }else if (magic == v20swap){
    hmmfp->parser = read_bin20hmm;
    return hmmfp;
  }else if (magic == v19magic){
    hmmfp->parser = read_bin19hmm;
    return hmmfp;
  }else if (magic == v19swap){
    hmmfp->parser = read_bin19hmm;
    return hmmfp;
  }
  ...
}
int HMMFileRead(HMMFILE *hmmfp,
        struct plan7_s **ret_hmm){
  return (*hmmfp->parser)(hmmfp,
                          ret_hmm);
}
int main(...) {
  if((hmmfp = HMMFileOpen(hmmfile,
                  "HMMERDB")) == NULL)
    Die(...);
  if(!HMMFileRead(hmmfp, &hmm))
    Die(...);
}
```

Figure 7: The ICT with the largest EC in `456.hmmer`

Markov Models) file. This model file can have different versions and formats identified by its magic number. As such, the benchmark creates the `HMMFILE` structure with the `parser` function pointer (Line 9 and 10), and assigns the function pointer according to the model file's magic (Line 14-27). The function pointer is called at Line 31. In total, there are fifteen valid parsers.

Because `HMMFileRead` is called in the main function, call-site sensitivity is not useful for this case at all because there is just one call site. As such, OS-CFI applies the origin sensitivity for this ICT. It creates an origin for each assignment to `parser` (Line 15, 18, 21, 24...). Therefore, OS-CFI can enforce a perfect CFI policy for this ICT as well.

4.2 Security Experiments

We experimented with two real-world exploits and one synthesized exploit to show how OS-CFI can block them.

4.2.1 Real-world Exploits

We experimented with two vulnerabilities, CVE-2015-8668 in libtiff and CVE-2014-1912 in python. We used the existing PoC exploits to overwrite a function pointer in order to hijack the control flow. We first verified that the exploits work and then tested them again under the protection of OS-CFI.

CVE-2015-8668: This is a heap-based buffer overflow caused by an integer overflow. The program fails to sanitize the buffer size if the multiplication overflows (Fig. 8, Line 20). This causes the allocated buffer (`uncomprbuf`) to be too small, allowing the attacker to overflow the heap memory. A potential target of the attack is the `TIFF` object, which contains several function pointers. One of such function pointers is `tif_encoderow`, which is called by `TIFFWriteScanline` later in the program.

```
1  int TIFFWriteScanline(TIFF* tif, ...){
2    ...
3    status = (*tif->tif_encoderow)(tif, (uint8*) buf,
4      tif->tif_scanlinesize, sample); // <= exploit call-point
5  }
6  void _TIFFSetDefaultCompressionState(TIFF* tif){
7    tif->tif_encoderow = _TIFFNoRowEncode; // <= origin
8  }
9  TIFF* TIFFOpen(...){
10   ...
11   _TIFFSetDefaultCompressionState(tif);
12 }
13 int main(int argc, char* argv[]){
14   TIFF *out = NULL;
15   out = TIFFOpen(outfilename, "w"); // <= exploited object
16   ...
17   uint32 uncompr_size;
18   unsigned char *uncomprbuf;
19   ...
20   uncompr_size = width * length; // non-sanitized code and
21                                  // following memory allocation
22   uncomprbuf = (unsigned char *)_TIFFmalloc(uncompr_size);
23   ...
24   if (TIFFWriteScanline(out, ...) < 0) {}
25    ...
26 }
```

Figure 8: Sketch of the vulnerable code in libtiff v4.0.6.

The indirect call at Line 3 was protected in OS-CFI by origin sensitivity. OS-CFI identified twelve origins of `tif_encoderow` with twelve different targets. However, the only origin recorded during this exploit was the one in the `_TIFFSetDefaultCompressionState` function, and the corresponding valid target was `_TIFFNoRowEncode`. Although all twelve origins are possible for the ICT at Line 3, the run-time context allowed us to uniquely identify the only valid target. Our system successfully detected the exploit.

CVE-2014-1912: this buffer overflow in python-2.7.6 is caused by the missing check of buffer size (Fig. 9, Line 28) before receiving the data into a `Py_buffer` object. `Py_buffer` has a member of the type `PyTypeObject`, which contains a function pointer `tp_hash`. `tp_hash` is used by the `PyObject_Hash` function to hash objects. The buffer overflow at Line 31 can be used to overwrite this function pointer.

```
1  int PyType_Ready(PyTypeObject *type){
2    ...
3    bases = type->tp_bases;
4    PyObject *b = PyTuple_GET_ITEM(bases, i);
5    if(PyType_Check(b))
6      inherit_slots(type, (PyTypeObject *)b); // <= origin context
7  }
8  static void inherit_slots(PyTypeObject *t, PyTypeObject *b){
9    ...
10   type->tp_hash = base->tp_hash; // <= origin
11 }
12 long PyObject_Hash(PyObject *v){
13   PyTypeObject *tp = v->ob_type;
14   if (tp->tp_hash != NULL)
15     return (*tp->tp_hash)(v); // <= exploit call-point
16 }
17 static PyObject *sock_recvfrom_into(...){
18   Py_buffer buf;
19   ...
20
21   if (recvlen < 0) {
22     goto error;
23   }
24   if (recvlen == 0) {
25     recvlen = buflen;
26   }
27
28   // missing check if (buflen < recvlen) {}
29
30   // vulnerable code
31   readlen = sock_recvfrom_guts(s, buf.buf, recvlen, flags, &addr);
32 }
```

Figure 9: Sketch of the vulnerable code in Python-2.7.6

Our algorithm identified the origin of `tp_hash` as Line 10 plus its call-site at Line 6. As such, origin sensitivity is ineffective for the indirect call at Line 15 because there is only one origin. Instead, 3-call-site sensitivity was used for this ICT. We counted 40 immediate call sites to the `PyObject_Hash` function. With three call-sites, we were able to limit the valid targets to a single candidate for each valid call stack. Our system also successfully prevented this exploit.

In both cases, OS-CFI not only blocked the exploits but also constrained the vulnerable ICTs to a single target at run-time.

4.2.2 Synthesized Exploit: a COOP Attack

We used the example code in Fig. 10 to demonstrate how OS-CFI can detect both `vTable` hijacking and control-flow hijacking for C++ objects. The example was inspired by PittyPat [14]. There are two virtual calls (Line 44 and 48) and two vulnerable functions (`getPerson` and `isEmployee`). The `getPerson` function contains a heap-based overflow, which allows the attacker to compromise the returned object's `vPtr` pointer, for example, to overwrite `Employee`'s `vPtr` to `Employer`'s `vTable`. The buffer overflow in `isEmployee` can overwrite `res` to always return `true`.

OS-CFI prevented both exploits. The first exploit was de-

```
class Person{
protected:
  SalaryAccount *salary = nullptr;
public:
  virtual void seeEvaluation()=0;
  virtual void seeSalary(){/*null derefernce*/}
};
class Employee : public Person{
public:
  void seeEvaluation(){/* show employee evaluation*/}
  void seeSalary(){/*employee has salary account*/}
};
class Employer : public Person{
public:
  void seeEvaluation(void){/*list of employee evaluation*/}
};
Person *getPerson(int id){
  char *name = (char*)malloc(10);
  Person *p;
  if (isEmployer(id)) {
    p = new Employer();
  } else {
    p = new Employee();
  }
  gets(name); // vulnerable gets()
  ...
  return p;
}
bool isEmployee(Person *member) {
  bool res = false;
  char name[10];
  // vulnerable strcpy()
  strcpy(name, member->getName());
  ...
  if(Employee *emp = dyn_cast<Employee*>(member)){
    if(emp != NULL)
      res = true;
  }
  return res; // attacker overwrite res
}
int main() {
  ...
  Person *member;
  member = getPerson(id);

  // if employee, can only see his/her evaluation
  // if employer, can see list of employee evaluation
  member->seeEvaluation(); // OTI protected

  // only employee has salary account
  if (isEmployee(member))
    member->seeSalary(); // CFI protected
}
```

Figure 10: A program vulnerable to COOP attack.

tected by the MPX table that failed to return the origin because the object's `vPtr` had been changed. The second exploit was blocked because SUPA understands `dyn_case` and correctly determines that `member` should be an object of `Employee`. OS-CFI thus refused to call other class' `seeSalary` function.

4.3 Performance Evaluation

We evaluated the performance of OS-CFI on all the C/C++ benchmarks in SPEC CPU2006 and NGINX. All the experiments were conducted on a server with the Xeon E3-1275 processor and 64 GB of memory, running the 64-bit Ubuntu 16.04.5 LTS Server system. The kernel is the standard Ubuntu kernel 4.4.0-139-generic. All the benchmarks were compiled

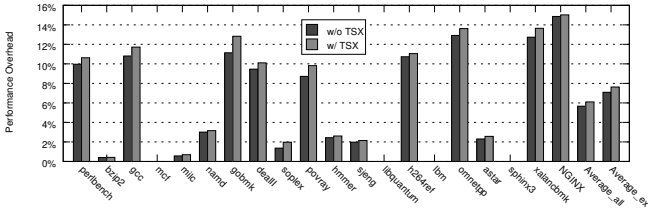


Figure 11: Normalized performance overhead, Average_ex shows the average overhead excluding three benchmarks that have no ICTs.

with the LLVM Safestack, which protects return addresses by relocating them to a separate stack [23]. We use Safestack to demonstrate that OS-CFI is compatible with the return protection. Other return protection can be integrated with OS-CFI too, such as shadow stack [10] and Intel CET [19], with different security and performance impacts. Moreover, OS-CFI uses Intel TSX to protect its reference monitors. This will incur additional overhead. We measured the performance of the benchmarks with and without Intel TSX. All the SPEC CPU2006 benchmarks were measured with the reference data. The results are show in Fig. 11.

On average, OS-CFI incurred an overhead of 7.1% without Intel TSX and 7.6% with it. Note that three SPEC benchmarks (`mcf`, `libquantum`, and `sphinx3`) did not have any noticeable overhead because they do not have/use ICTs in their main programs (and Safestack has negligible overhead). We excluded them from the average. The overhead reduced to 5.7% and 6.1%, respectively, if these three benchmarks were included. The highest performance overhead was 14.9% for NGINX and 12.9% for `omnetpp`. The performance of OS-CFI is better than PathArmor (8.5%) [38] and PittyPat (12.7%) [14], even though OS-CFI is a whole program protection while PathArmor only protects a few selected syscalls and PittyPat uses separate threads to parse processor traces (this reduces the number of available CPU cores by half).

OS-CFI generates its CFGs by piggybacking on SUPA. Compared to the original SUPA, it took OS-CFI 5.3% longer on average to analyze these benchmarks, measured on the same machine as the performance evaluation. As shown in Table. 5, the average analysis time is about 3 hours, and the longest analysis time is 15.6 hours from `gcc`. We consider the overhead incurred by OS-CFI acceptable since the analysis is conducted only once offline.

5 Related Work

In this section, we compare OS-CFI to recent (context-sensitive) CFI systems. Since its introduction in 2005 [2], there has been a long list of research in CFI [1,6,9,11–14,16,17,24,25,28,29,31,38,40,41,43,44]. A recent survey by Burow et al. compares many context-insensitive CFI systems [3]. Earlier CFI systems often trade security for performance, introducing

Benchmark	SUPA (s)	OS-CFI (s)	Overhead
400.perlbench	6083.2	6350.7	4.4%
401.bzip2	445.8	457.2	2.6%
403.gcc	53029.1	56231.7	6.0%
433.milc	3.9	4.0	2.6%
445.gobmk	4071.5	4246.4	4.3%
456.hmmer	10.9	11.8	8.3%
458.sjeng	2.6	2.6	0.0%
464.h264ref	372.1	382.0	2.7%
444.namd	15.6	16.7	7.1%
447.dealII	651.5	673.8	3.5%
450.soplex	1280.7	1340.2	4.6%
453.povray	4633.9	5304.0	14.5%
471.omnetpp	43929.0	45351.5	3.2%
473.astar	1.4	1.5	7.1%
483.xalancbmk	9703.7	10792.6	11.2%
NGINX	39860.2	41630.7	4.4%
Average	10255.9	10799.8	5.3%

Table 5: The analysis time of OS-CFI as compared to the vanilla SUPA algorithm. The unit of the analysis time in the table is seconds.

imprecision in both CFGs and enforcement mechanisms. For example, some of them enforce a coarse-grained CFG [37,43], making them vulnerable to attacks [13, 16]. Even precise context-insensitive CFI systems may be vulnerable because of their large EC sizes [6]. Compared to these systems, OS-CFI is a context-sensitive CFI system. Its origin-based context can effectively break down large ECs, improving the security.

An effective method to improve the precision of CFI is to use the contextual information to differentiate sets of targets. However, the addition of context imposes stringent demands on the system design, leading to more trade-offs and opportunities: first of all, a context-sensitive CFI system requires context-sensitive CFGs. It is well-known that context-sensitive points-to analysis does not scale well. The situation has been substantially improved with the recent release of SUPA [35]. However, scalable path-sensitive points-to analysis, needed by systems like PathArmor and PittyPat, is still unavailable; The second challenge is how to securely collect, store, and use contextual information with minimal performance overhead. In the following, we compare OS-CFI to three representative context-sensitive CFI systems: PathArmor [38], PittyPat [14], and μCFI [17]. Table 6 shows their key differences.

PathArmor, PittyPat and μCFI all use the recent execution history recorded by Intel CPUs as the context, last branch record (LBR) for PathArmor and processor trace (PT) for the other two. LBR records only the last sixteen branches taken by the process; while PT provides more fine-grained record of the past execution. Unlike MPX that can be accessed directly in the user space, LBR and PT are privileged and only accessible by the kernel. Transition into and out of the kernel is an expensive operation. It is thus impractical to check these records for each ICT. To address that, PathArmor enforces the CFI policy at the selected syscalls; i.e., only a small part of the program is protected. PittyPat and μCFI redirect the trace to a separate process and verify the control flow there. They can verify all the ICTs but only enforce the results at the selected syscalls. The drawback of this design is that the usable number of CPU cores is effectively reduced by half. Because all three systems cannot enforce the CFI policy at every ICT, their focus is to protect the other part of the system from attacks. OS-CFI instead collects the context by inline reference monitors protected by Intel TSX. It is a whole-program protection that enforces the CFI policy at every ICT. In addition, all these three systems require to change the kernel. OS-CFI uses the stock kernel, whose general MPX support is sufficient.

OS-CFI derives its CFGs from a context-, flow-, and field-sensitive static points-to analysis. However, PathArmor and PittyPat enforce path-sensitivity. To the best of our knowledge, there is no scalable path-sensitive points-to analysis available (at least publicly). Both systems, as well as μCFI, instead utilize on-line points-to analysis, based on the recorded context. The design of μCFI is interesting in that it turns the constraint data into indirect control transfers, which are recorded by PT. This securely conveys the constraint data to the monitoring process. Unfortunately, it seems that this usage puts too much pressure on PT, causing PT to lose packets. This renders μCFI unsuitable for large programs. Indeed, it cannot handle the most demanding benchmarks in SPEC CPU2006, such as `gcc`, `omnetpp`, and `xalancbmk`, and the benchmarks are tested with the smaller train data, not the regular reference data. OS-CFI focuses on reducing the EC sizes. PathArmor and PittyPat are unlikely to achieve the same effectiveness because they use the execution history as the context. The largest EC sizes will remain significant because of the limited incoming paths towards a ICT. For example, PittyPat reports one large EC size of 218. The goal of μCFI is to enforce unique target for each ICT. This is achieved by considering the constraint data during verification. However, the constraint data can potentially be compromised before being captured by μCFI, as mentioned in the paper [17]. This weakens its security guarantee.

CPI is another closely related system. It can guarantee the integrity of all code pointers in the program by separating them and related critical data pointers in a protected safe memory region [23]. As such, CPI can prevent all the control-flow hijacking attacks. Compared to CPI, OS-CFI achieves a similar but slightly relaxed protection in the CFI principle (because OS-CFI still allows some leeway to manipulate the control flow). OS-CFI uses the MPX table to store its metadata. This usage can be applied in CPI as well to further improve its performance, as suggested by the paper itself. Burow et al.

Categories	CFIXX	PathArmor	PittyPat	μCFI	OS-CFI
Protected	Object type	Control flow	Control flow	Control flow	Control flow & Object type
Context	`vPtr` to `vTable` binding	last branches taken	Processor execution paths	Execution paths and constraint data	Origins of function pointers and objects
CFG	None	On-demand, constraint driven context-sensitive CFG	Abstract-interpretation based online points-to analysis	Run-time points-to analysis	CFGs based on context-, flow- and field-sensitive static points-to analysis
Coverage	Virtual calls	Selected syscalls	Whole program, enforced at selected syscalls	Whole program, enforced at selected syscalls	Whole program, enforced at every ICT
Required hardware	Intel MPX for metadata storage	Intel LBR for taken branches	Intel PT for execution history	Intel PT for execution history and control data	Intel MPX for metadata storage and Intel TSX to protect reference monitors
Kernel changes	No, built-in MPX support	Yes, enforce CFI on the syscall boundary	Yes, redirect traces and enforce CFI on syscall boundary	Yes, redirect traces and enforce CFI on syscall boundary	No, built-in MPX and TSX support
Runtime support	Library to track the type of each object	Per-thread control transfer monitoring	Additional threads to parse trace and verify control flow	Additional threads to parse trace and verify control flow	Hash based verification protected by TSX

Table 6: Comparison between OS-CFI and recent (context-sensitive) CFI systems

independently discovered the way to re-purpose the MPX table as a generic key-value store [5]. As a hardware accelerated data store, MPX can be used in a wide variety of security systems, especially considering that its bound registers can be used for high-performance SFI (software-fault isolation) [4, 22, 39].

Another closely related system is CFIXX [4], which enforces the object-type integrity (OTI). CFIXX prevents attacks such as COOP [32] from subverting an object's `vPtr` pointer. OTI is a complementary policy to CFI [4]. It requires and strengthens CFI to provide more complete protection. OS-CFI's protection of virtual calls uses the same key (but different metadata, i.e., the origin) as OTI as a by-product of using MPX to keep the metadata. As mentioned earlier, OS-CFI can use different keys in its design as long as it can retrieve the origin of the receiving object because the origin alone can uniquely identify the target. Overall, OS-CFI provides stronger security guarantee than CFIXX with its CFI for all ICTs. There are several other systems that focus on protecting virtual calls, such as VTrust [42] and SAFEDISPATCH [20]. OS-CFI supports both C-style ICTs and C++ virtual calls.

6 Summary

We have presented a new type of context for CFI systems – origin sensitivity. By considering the origins of function pointers and objects during the verification of control transfers, we can significantly improve the security of CFI by reducing the largest and average EC sizes. By leveraging the commodity hardware features such as MPX and TSX, our system incurs only a small overhead.

7 Availability

Our prototype is available as an open-source project at `https://github.com/mustakcsecuet/OS-CFI`.

8 Acknowledgment

We would like to thank the anonymous reviewers and our shepherd, Dr. Nathan Dautenhahn, for their insightful comments that helped improve the presentation of this paper. This project was partially supported by National Science Foundation (NSF) under Grant 1453020. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of NSF.

References

[1] Niu, Ben and Tan, Gang , "Per-input Control-flow Integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 914–926.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow Integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 2005, pp. 340–353.

[3] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-Flow Integrity: Precision, Security, and Performance," *ACM Comput. Surv.*, vol. 50, no. 1, pp. 16:1–16:33, Apr. 2017. [Online]. Available: http://doi.acm.org/10.1145/3054924

[4] N. Burow, D. McKee, S. A. Carr, and M. Payer, "CFIXX: Object Type Integrity for C++," in *Proceedings of the 2018 Network and Distributed System Security Symposium*, 2018.

[5] N. Burow, X. Zhang, and M. Payer, "SoK: Shining Light on Shadow Stacks," in *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, ser. SP '19. Washington, DC, USA: IEEE Computer Society, 2019.

[6] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity," in *Proceedings of the 24th USENIX Security Symposium*, vol. 14, 2015, pp. 28–38.

[7] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, "Detecting Privileged Side-channel Attacks in Shielded Execution with Déjá Vu," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017, pp. 7–18.

[8] "Bjarne Stroustrup's C++ Style and Technique FAQ," http://www.stroustrup.com/bs_faq2.html, p. 5.

[9] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete Control-flow Integrity for Commodity Operating System Kernels," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 292–307.

[10] T. H. Dang, P. Maniatis, and D. Wagner, "The Performance Cost of Shadow Stacks and Stack Canaries," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '15, 2015.

[11] L. Davi, P. Koeberl, and A.-R. Sadeghi, "Hardware-assisted Fine-grained Control-flow Integrity: Towards Efficient Protection of Embedded Systems against Software Exploitation," in *Proceedings of the 51st Annual Design Automation Conference*. ACM, 2014, pp. 1–6.

[12] L. Davi and A.-R. Sadeghi, "Building Control-flow Integrity Defenses," in *Building Secure Defenses Against Code-Reuse Attacks*. Springer, 2015, pp. 27–54.

[13] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the Gadgets: On the Ineffectiveness of Coarse-grained Control-flow Integrity Protection," in *Proceedings of the 23Rd USENIX Conference on Security*, ser. SEC'14, 2014.

[14] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, "Efficient Protection of Path-sensitive Control Security," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 131–148. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ding

[15] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control Jujutsu: On the Weaknesses of Fine-grained Control-flow Integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 901–913.

[16] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of Control: Overcoming Control-flow Integrity," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, ser. SP '14, 2014.

[17] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing Unique Code Target Property for Control-Flow Integrity," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: ACM, 2018, pp. 1470–1486. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243797

[18] *Intel 64 and IA-32 Architectures Software Developerś Manual*, Intel.

[19] Intel, "Control-flow Enforcement," https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf, 2018.

[20] D. Jang, Z. Tatlock, and S. Lerner, "SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks," in *NDSS*, 2014.

[21] M. Khandaker, A. Naser, W. Liu, Z. Wang, Y. Zhou, and Y. Cheng, "Adaptive Call-site Sensitive Control Flow Integrity," in *Proceedings of the 4th IEEE European Symposium on Security and Privacy (EuroS&P 2019)*, 2019.

[22] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No Need to Hide: Protecting Safe Regions on Commodity Hardware," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: ACM, 2017, pp. 437–452. [Online]. Available: http://doi.acm.org/10.1145/3064176.3064217

[23] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer Integrity," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, 2014, pp. 147–163. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov

[24] J. Li, Z. Wang, T. Bletsch, D. Srinivasan, M. Grace, and X. Jiang, "Comprehensive and Efficient Protection of Kernel Control Data," *IEEE Transactions on Information Forensics and Security*, vol. 6, no. 4, pp. 1404–1417, Dec 2011.

[25] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque Control-flow Integrity," in *Proceedings of the 22th Network and Distributed System Security Symposium*, ser. NDSS '15, 2015.

[26] "Intel MPX Performance Evaluation for Bound Checking," https://intel-mpx.github.io/performance/.

[27] "GCC 9 Looks Set To Remove Intel MPX Support," https://www.phoronix.com/scan.php?page=news_item&px=GCC-Patch-To-Drop-MPX.

[28] B. Niu and G. Tan, "Modular Control-flow Integrity," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 577–587, 2014.

[29] Niu, Ben and Tan, Gang, "RockJIT: Securing Just-in-time Compilation Using Modular Control-flow Integrity," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1317–1328.

[30] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel MPX explained: An empirical study of intel MPX and software-based bounds checking approaches," *arXiv preprint arXiv:1702.00719*, 2017.

[31] M. Payer, A. Barresi, and T. R. Gross, "Fine-grained Control-flow Integrity through Binary Hardening," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2015, pp. 144–164.

[32] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications," in *Proceedings of the 36th IEEE Symposium on Security and Privacy*. IEEE, 2015.

[33] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10, 2010.

[34] Y. Sui and J. Xue, "On-demand Strong Update Analysis via Value-flow Refinement," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 460–473.

[35] "Demand Driven Pointer Annalysis," https://github.com/SVF-tools/SUPA.

[36] "Static Value-Flow Graph in LLVM," https://github.com/SVF-tools/SVF.

[37] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing Forward-edge Control-flow Integrity in GCC & LLVM," in *USENIX Security Symposium*, 2014, pp. 941–955.

[38] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical Context-sensitive CFI," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15, 2015.

[39] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient Software-based Fault Isolation," in *Proceedings of the 14th ACM Symposium On Operating System Principles*, December 1993.

[40] Z. Wang and X. Jiang, "Hypersafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-flow Integrity," in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 380–395.

[41] Y. Xia, Y. Liu, H. Chen, and B. Zang, "CFIMon: Detecting Violation of Control-flow Integrity Using Performance Counters," in *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*. IEEE, 2012, pp. 1–12.

[42] C. Zhang, D. Song, S. A. Carr, M. Payer, T. Li, Y. Ding, and C. Song, "VTrust: Regaining Trust on Virtual Calls," in *NDSS*, 2016.

[43] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical Control Flow Integrity and Randomization for Binary Executables," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP '13, 2013.

[44] M. Zhang and R. Sekar, "Control Flow Integrity for COTS Binaries," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13, 2013.

[45] T. Zhang, Y. Zhang, and R. B. Lee, "Cloudradar: A real-time side-channel attack detection system in clouds," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2016.

HardFails: Insights into Software-Exploitable Hardware Bugs

Ghada Dessouky[†], David Gens[†], Patrick Haney[*], Garrett Persyn[*], Arun Kanuparthi[°], Hareesh Khattri[°], Jason M. Fung[°], Ahmad-Reza Sadeghi[†], Jeyavijayan Rajendran[*]
[†]*Technische Universität Darmstadt, Germany.* [*]*Texas A&M University, College Station, USA.*
[°]*Intel Corporation, Hillsboro, OR, USA.*
ghada.dessouky@trust.tu-darmstadt.de, david.gens@trust.tu-darmstadt.de,
prh537@tamu.edu, gpersyn@tamu.edu, arun.kanuparthi@intel.com,
hareesh.khattri@intel.com, jason.m.fung@intel.com,
ahmad.sadeghi@trust.tu-darmstadt.de, jv.rajendran@tamu.edu

Abstract

Modern computer systems are becoming faster, more efficient, and increasingly interconnected with each generation. Thus, these platforms grow more complex, with new features continually introducing the possibility of new bugs. Although the semiconductor industry employs a combination of different verification techniques to ensure the security of System-on-Chip (SoC) designs, a growing number of increasingly sophisticated attacks are starting to leverage *cross-layer bugs*. These attacks leverage subtle interactions between hardware and software, as recently demonstrated through a series of real-world exploits that affected all major hardware vendors.

In this paper, we take a deep dive into microarchitectural security from a hardware designer's perspective by reviewing state-of-the-art approaches used to detect hardware vulnerabilities at design time. We show that a protection gap currently exists, leaving chip designs vulnerable to software-based attacks that can exploit these hardware vulnerabilities. Inspired by real-world vulnerabilities and insights from our industry collaborator (a leading chip manufacturer), we construct the first representative testbed of real-world software-exploitable RTL bugs based on RISC-V SoCs. Patching these bugs may not always be possible and can potentially result in a product recall. Based on our testbed, we conduct two extensive case studies to analyze the effectiveness of state-of-the-art security verification approaches and identify specific classes of vulnerabilities, which we call *HardFails*, which these approaches fail to detect. Through our work, we focus the spotlight on specific limitations of these approaches to propel future research in these directions. We envision our RISC-V testbed of RTL bugs providing a rich exploratory ground for future research in hardware security verification and contributing to the open-source hardware landscape.

1 Introduction

The divide between hardware and software security research is starting to take its toll, as we witness increasingly sophisticated attacks that combine software and hardware bugs to exploit computing platforms at runtime [20, 23, 36, 43, 45, 64, 69, 72, 74]. These cross-layer attacks disrupt traditional threat models, which assume either hardware-only or software-only adversaries. Such attacks may provoke physical effects to induce hardware faults or trigger unintended microarchitectural states. They can make these effects visible to software adversaries, enabling them to exploit these hardware vulnerabilities remotely. The affected targets range from low-end embedded devices to complex servers, that are hardened with advanced defenses, such as data-execution prevention, supervisor-mode execution prevention, and control-flow integrity.

Hardware vulnerabilities. Cross-layer attacks circumvent many existing security mechanisms [20, 23, 43, 45, 64, 69, 72, 74], that focus on mitigating attacks exploiting software vulnerabilities. Moreover, hardware-security extensions are not designed to tackle hardware vulnerabilities. Their implementation remains vulnerable to potentially undetected hardware bugs committed at design-time. In fact, deployed extensions such as SGX [31] and TrustZone [3] have been targets of successful cross-layer attacks [69, 72]. Research projects such as Sanctum [18], Sanctuary [8], or Keystone [39] are also not designed to ensure security at the hardware implementation level. Hardware vulnerabilities can occur due to: (a) incorrect or ambiguous security specifications, (b) incorrect design, (c) flawed implementation of the design, or (d) a combination thereof. Hardware implementation bugs are introduced either through human error or faulty translation of the design in gate-level synthesis.

SoC designs are typically implemented at register-transfer level (RTL) by engineers using hardware description languages (HDLs), such as Verilog and VHDL, which are *synthesized* into a lower-level representation using automated tools. Just like software programmers introduce bugs to the high-level code, hardware engineers may accidentally introduce bugs to the RTL code. While software errors typically cause a crash which triggers various fallback routines to ensure the safety and security of other programs running on the platform, no such safety net exists for hardware bugs. Thus, even mi-

nor glitches in the implementation of a module within the processor can compromise the SoC security objectives and result in persistent/permanent denial of service, IP leakage, or exposure of assets to untrusted entities.

Detecting hardware security bugs. The semiconductor industry makes extensive use of a variety of techniques, such as simulation, emulation, and formal verification to detect such bugs. Examples of industry-standard tools include Incisive [10], Solidify [5], Questa Simulation and Questa Formal [44], OneSpin 360 [66], and JasperGold [11]. These were originally designed for *functional verification* with security-specific verification incorporated into them later.

While a rich body of knowledge exists within the software community (e.g., regarding software exploitation and techniques to automatically detect software vulnerabilities [38, 46]), security-focused HDL analysis is currently lagging behind [35, 57]. Hence, the industry has recently adopted a *security development lifecycle* (SDL) for hardware [68] — inspired by software practices [26]. This process combines different techniques and tools, such as RTL manual code audits, assertion-based testing, dynamic simulation, and automated security verification. However, the recent outbreak of *cross-layer attacks* [20, 23, 37, 43, 45, 47, 48, 49, 51, 52, 53, 64, 69, 74] poses a spectrum of difficult challenges for these security verification techniques, because they exploit complex and subtle inter-dependencies between hardware and software. Existing verification techniques are fundamentally limited in modeling and verifying these interactions. Moreover, they also do not scale with the size and complexity of real-world SoC designs.

Goals and Contributions. In this paper, we show that current hardware security verification techniques are fundamentally limited. We provide a wide range of results using a comprehensive test harness, encompassing different types of hardware vulnerabilities commonly found in real-world platforms. To that end, we conducted two case studies to systematically and qualitatively assess existing verification techniques with respect to detecting RTL bugs. Together with our industry partners, we compiled a list of 31 RTL bugs based on public Common Vulnerabilities and Exposures (CVEs) [37, 43, 50, 54, 55] and real-world errata [25]. We injected bugs into two open-source RISC-V-based SoC designs, which we will open-source after publication.

We organized an international public hardware security competition, Hack@DAC, where 54 teams of researchers competed for three months to find these bugs. While a number of bugs could not be detected by any of the teams, several participants also reported *new* vulnerabilities of which we had no prior knowledge. The teams used manual RTL inspection and simulation techniques to detect the bugs. In industry, these are usually complemented by automated tool-based and formal verification approaches. Thus, our second case study focused on two state-of-the-art formal verification tools: the first deploys formal verification to perform exhaustive and complete verification of a hardware design, while the second leverages formal verification and path sensitization to check for illegal data flows and fault tolerance.

Our second case study revealed that certain properties of RTL bugs pose challenges for state-of-the-art verification techniques with respect to black-box abstraction, timing flow, and non-register states. This causes security bugs in the RTL of real-world SoCs to slip through the verification process. Our results from the two case studies indicate that particular classes of hardware bugs entirely evade detection—even when complementing systematic tool-based verification approaches with manual inspection. RTL bugs arising from complex and cross-modular interactions in SoCs render these bugs extremely difficult to detect in practice. Furthermore, such bugs are exploitable from software, and thus can compromise the entire platform. We call such bugs **HardFails**.

To the best of our knowledge, this is the first work to provide a systematic and in-depth analysis of state-of-the-art hardware verification approaches for security-relevant RTL bugs. Our findings shed light on the capacity of these tools and demonstrate reproducibly how bugs can slip through current hardware security verification processes. Being also software-exploitable, these bugs pose an immense security threat to SoCs. Through our work, we highlight why further research is required to improve state-of-the-art security verification of hardware. To summarize, our main contributions are:

- **Systematic evaluation and case studies:** We compile a comprehensive test harness of real-world RTL bugs, on which we base our two case studies: (1) Hack@DAC'18, in which 54 independent teams of researchers competed worldwide over three months to find these bugs using manual RTL inspection and simulation techniques, and (2) an investigation of the bugs using industry-leading formal verification tools that are representative of the current state of the art. Our results show that particular classes of bugs entirely evade detection, despite combining both tool-based security verification approaches and manual analysis.
- **Stealthy hardware bugs:** We identify *HardFails* as RTL bugs that are distinctly challenging to detect using industry-standard security verification techniques. We explain the fundamental limitations of these techniques in detail using concrete examples.
- **Open-sourcing:** We will open-source our bugs testbed at publication to the community.

2 SoC Verification Processes and Pitfalls

Similar to the Security Development Lifecycle (SDL) deployed by software companies [26], semiconductor companies [15, 35, 40] have recently adapted SDL for hardware design [57]. We describe next the conventional SDL process for hardware and the challenges thereof.

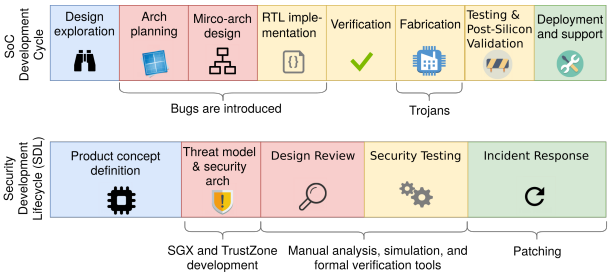


FIGURE 1: Typical Security Development Lifecycle (SDL) process followed by semiconductor companies.

2.1 The Security Development Lifecycle (SDL) for Hardware

SDL is conducted concurrently with the conventional hardware development lifecycle [68], as shown in Figure 1. The top half of Figure 1 shows the hardware development lifecycle. It begins with design exploration followed by defining the specifications of the product architecture. After the architecture specification, the microarchitecture is designed and implemented in RTL. Concurrently, pre-silicon verification efforts are conducted until tape-out to detect and fix all functional bugs that do not meet the *functional specification*. After tape-out and fabrication, iterations of post-silicon validation, functional testing, and tape-out "spins" begin. This cycle is repeated until no defects are found and all quality requirements are met. Only then does the chip enter mass production and is shipped out. Any issues found later in-field are debugged, and the chip is then either patched if possible or recalled.

After architectural features are finalized, a security assessment is performed, shown in the bottom half of Figure 1. The adversary model and the security objectives are compiled in the *security specification*. This typically entails a list of assets, entry points to access these assets, and the adversary capabilities and architectural security objectives to mitigate these threats. These are translated into microarchitectural security specifications, including security test cases (both positive and negative). After implementation, pre-silicon security verification is conducted using dynamic verification (i.e., simulation and emulation), formal verification, and manual RTL reviews. The chip is not signed off for tape-out until all security specifications are met. After tape-out and fabrication, post-silicon security verification commences. The identified security bugs in both pre-silicon and post-silicon phases are rated for severity using the industry-standard scoring systems such as the Common Vulnerability Scoring System (CVSS) [30] and promptly fixed. Incident response teams handle issues in shipped products and provide patches, if possible.

2.2 Challenges with SDL

Despite multiple tools and security validation techniques used by industry to conduct SDL, it remains a highly challenging, tedious, and complex process even for industry experts. Existing techniques largely rely on human expertise to define the security test cases and run the tests. The correct *security specifications* must be exhaustively anticipated, identified, and accurately and adequately expressed using security properties that can be captured and verified by the tools. We discuss these challenges further in Section 7.

Besides the specifications, the techniques and tools themselves are not scalable and are less effective in capturing subtle semantics that are relevant to many vulnerabilities, which is the focus of this work. We elaborate next on the limitations of state-of-the-art hardware security verification tools commonly used by industry. To investigate the capabilities of these tools, we then construct a comprehensive test-harness of real-world RTL vulnerabilities.

3 Assessing Hardware Security Verification

In this section, we focus on why the verification of the security properties of modern hardware is challenging and provide requirements for assessing existing verification techniques under realistic conditions. First, we describe how these verification techniques fall short. Second, we provide a list of common and realistic classes of hardware bugs, which we use to construct a test harness for assessing the effectiveness of these verification techniques. Third, we discuss how these bugs relate to the common security goals of a chip.

3.1 Limitations of Automated Verification

Modern hardware designs are highly complex and incorporate hundreds of in-house and third-party Intellectual Property (IP) components. This creates room for vulnerabilities to be introduced in the inter-modular interactions of the design hierarchy. Multi-core architectures typically have an intricate interconnect fabric between individual cores (utilizing complex communication protocols), multi-level cache controllers with shared un-core and private on-core caches, memory and interrupt controllers, and debug and I/O interfaces.

For each core, these components contain logical modules such as fetch and decode stages, an instruction scheduler, individual execution units, branch prediction, instruction and data caches, the memory subsystem, re-order buffers, and queues. These are implemented and connected using individual RTL modules. The average size of each module is several hundred lines of code (LOC). Thus, real-world SoCs can easily approach 100,000 lines of RTL code, and some designs may even have millions of LOC. Automatically verifying, at the RTL level, the respective interconnections and checking them against security specifications raises a number of fundamental challenges for the state-of-the-art approaches. These are described below.

L-1: Cross-modular effects. Hardware modules are interconnected in a highly hierarchical design with multiple inter-

dependencies. Thus, an RTL bug located in an individual module may trigger a vulnerability in intra- and inter-modular information flows spanning multiple complex modules. Pinpointing the bug requires analyzing these flows across the relevant modules, which is highly cumbersome and unreliable to achieve by manual inspection. It also pushes formal verification techniques to their limits, which work by modeling and analyzing all the RTL modules of the design to verify whether design specifications (expressed using security property assertions, invariants and disallowed information flows) and implementation match.

Detecting such vulnerabilities requires loading the RTL code of all the relevant modules into the tools to model and analyze the entire state space, thus driving them quickly into *state explosion* due to the underlying modeling algorithms [16, 21]. Alleviating this by providing additional computational resources and time is not scalable as the complexity of SoCs continues to increase. Selective "black-box" abstraction of some of the modules, state space constraining, and bounded-model checking are often used. However, they do not eliminate the fundamental problem and rely on interactive human expertise. Erroneously applying them may introduce false negatives, leading to missed vulnerabilities.

L-2: Timing-flow gap. Current industry-standard techniques are limited in capturing and verifying security properties related to timing flow (in terms of clock cycle latency). This leads to vast sources of information leakage due to software-exploitable timing channels (Section 8). A timing flow exists between the circuit's input and output when the number of clock cycles required for the generation of the output depends on input values or the current memory/register state. This can be exploited to leak sensitive information when the timing variation is discernible by an adversary and can be used to infer inputs or memory states. This is especially problematic for information flows and resource sharing across different privilege levels. This timing variation should remain indistinguishable in the RTL, or should not be measurable from the software. However, current industry-standard security verification techniques focus exclusively on the functional information flow of the logic and fail to model the associated timing flow. The complexity of timing-related security issues is aggravated when the timing flow along a logic path spans multiple modules and involves various inter-dependencies.

L-3: Cache-state gap. State-of-the-art verification techniques only model and analyze the *architectural state* of a processor by exclusively focusing on the state of registers. However, they do not support analysis of non-register states, such as caches, thus completely discarding modern processors' highly complex microarchitecture and diverse hierarchy of caches. This can lead to severe security vulnerabilities arising due to state changes that are unaccounted for, e.g., the changing state of shared cache resources across multiple privilege levels. Caches represent a state that is influenced directly or indirectly by many control-path signals and can generate security vulnerabilities in their interactions, such as illegal information leakages across different privilege levels. Identifying RTL bugs that trigger such vulnerabilities is beyond the capabilities of existing techniques.

L-4: Hardware-software interactions. Some RTL bugs remain indiscernible to hardware security verification techniques because they are not explicitly vulnerable unless triggered by the software. For instance, although many SoC access control policies are directly implemented in hardware, some are programmable by the overlying firmware to allow for post-silicon flexibility. Hence, reasoning on whether an RTL bug exists is inconclusive when considering the hardware RTL in isolation. These vulnerabilities would only materialize when the hardware-software interactions are considered, and existing techniques do not handle such interactions.

3.2 Constructing Real-World RTL Bugs

To systematically assess the state of the art in hardware security verification with respect to the limitations described above, we construct a test harness by implementing a large number of RTL bugs in RISC-V SoC designs (cf. Table 1). To the best of our knowledge, we are the first to compile and showcase such a collection of hardware bugs. Together with our co-authors at Intel, we base our selection and construction of bugs on a solid representative spectrum of real-world CVEs [47, 48, 49, 51, 52, 53] as shown in Table 1. For instance, bug #22 was inspired by a recent security vulnerability in the Boot ROM of video gaming mobile processors [56], which allowed an attacker to bring the device into BootROM Recovery Mode (RCM) via USB access. This buffer overflow vulnerability affected many millions of devices and is popularly used to hack a popular video gaming console[1].

We extensively researched CVEs that are based on software-exploitable hardware and firmware bugs and classified them into different categories depending on the weaknesses they represent and the modules they impact. We reproduced them by constructing representative bugs in the RTL and demonstrated their software exploitability and severity by crafting a real-world software exploit based on one of these bugs in Appendix D. Other bugs were constructed with our collaborating hardware security professionals, inspired by bugs that they have previously encountered and patched during the pre-silicon phase, which thus never escalated into CVEs. The chosen bugs were implemented to achieve coverage of different security-relevant modules of the SoC.

Since industry-standard processors are based on proprietary RTL implementations, we mimic the CVEs by reproducing and injecting them into the RTL of widely-used RISC-V SoCs. We also investigate more complex microarchitecture features of another RISC-V SoC and discover vulnerabilities already existing in its RTL (Section 4). These RTL bugs manifest as:

[1] https://github.com/Cease-and-DeSwitch/fusee-launcher

- **Incorrect assignment bugs** due to variables, registers, and parameters being assigned incorrect literal values, incorrectly connected or left floating unintended.
- **Timing bugs** resulting from timing flow issues and incorrect behavior relevant to clock signaling such as information leakage.
- **Incorrect case statement bugs** in the finite state machine (FSM) models such as incorrect or incomplete selection criteria, or incorrect behavior within a case.
- **Incorrect if-else conditional bugs** due to incorrect boolean conditions or incorrect behavior described within either branch.
- **Specification bugs** due to a mismatch between a specified property and its actual implementation or poorly specified / under-specified behavior.

These seemingly minor RTL coding errors may constitute security vulnerabilities, some of which are very difficult to detect during verification. This is because of their interconnection and interaction with the surrounding logic that affects the complexity of the subtle side effects they generate in their manifestation. Some of these RTL bugs may be patched by modifying parts of the software stack that use the hardware (e.g., using firmware/microcode updates) to circumvent them and mitigate specific exploits. However, since RTL is usually compiled into hardwired integrated circuitry logic, the underlying bugs cannot, in principle, be patched after production.

The limited capabilities of current detection approaches in modeling hardware designs and formulating and capturing relevant security assertions raise challenges for detecting some of these vulnerabilities, which we investigate in depth in this work. We describe next the adversary model we assume for our vulnerabilities and our investigation.

3.3 Adversary Model

In our work, we investigate microarchitectural details at the RTL level. However, all hardware vendors keep their proprietary industry designs and implementations closed. Hence, we use an open-source SoC based on the popular open-source RISC-V [73] architecture as our platform. RISC-V supports a wide range of possible configurations with many standard features that are also available in modern processor designs, such as privilege level separation, virtual memory, and multithreading, as well as optimization features such as configurable branch prediction and out-of-order execution.

RISC-V RTL is freely available and open to inspection and modification. While this is not necessarily the case for industry-leading chip designs, an adversary might be able to reverse engineer or disclose/steal parts of the chip using existing tools[2][3]. Hence, we consider a strong adversary that can also inspect the RTL code.

In particular, we make the following assumptions:

- **Hardware Vulnerability:** The attacker has knowledge of a vulnerability in the hardware design of the SoC (i.e., at the RTL level) and can trigger the bug from software.
- **User Access:** The attacker has complete control over a user-space process, and thus can issue unprivileged instructions and system calls in the basic RISC-V architecture.
- **Secure Software:** Software vulnerabilities and resulting attacks, such as code-reuse [65] and data-only attacks [27] against the software stack, are orthogonal to the problem of cross-layer bugs. Thus, we assume all platform software is protected by defenses such as control-flow integrity [1] and data-flow integrity [13], or is formally verified.

The goal of an adversary is to leverage the vulnerability on the chip to provoke unintended functionality, e.g., access to protected memory locations, code execution with elevated privileges, breaking the isolation of other processes running on the platform, or permanently denying services. RTL bugs in certain hardware modules might only be exploitable with physical access to the victim device, for instance, bugs in debug interfaces. However, other bugs are software-exploitable, and thus have a higher impact in practice. Hence, we focus on software-exploitable RTL vulnerabilities, such as the exploit showcased in Appendix D. Persistent denial of service (PDoS) attacks that require exclusive physical access are out of scope. JTAG attacks, though they require physical access, are still in scope as the end user may be the attacker and might attempt to unlock the device to steal manufacturer secrets. Furthermore, exploiting the JTAG interface often requires a combination of both physical access and privilege escalation by means of a software exploit to enable the JTAG interface. We also note that an adversary with unprivileged access is a realistic model for real-world SoCs: Many platforms provide services to other devices over the local network or even over the internet. Thus, the attacker can obtain some limited software access to the platform already, e.g., through a webserver or an RPC interface. Furthermore, we emphasize that this work focuses only on tools and techniques used to detect bugs before tape-out.

4 HardFails: Hardware Security Bugs

In light of the limitations of state-of-the-art verification tools (Section 3.1), we constructed a testbed of real-world RTL bugs (Section 3.2) and conducted two extensive case studies on their detection (described next in Sections 5 and 6). Based on our findings, we have identified particular classes of hardware bugs that exhibit properties that render them more challenging to detect with state-of-the-art techniques. We call these HardFails. We now describe different types of these HardFails encountered during our analysis of two RISC-V SoCs, Ariane [59] and PULPissimo [61]. In Section 5.3, we describe the actual bugs we instantiated for our case studies.

Ariane is a 6-stage in-order RISC-V CPU that implements the RISC-V draft privilege specification and can run Linux OS. It has a memory management unit (MMU) consisting of

[2] https://www.chipworks.com/
[3] http://www.degate.org/

TABLE 1: Detection results for bugs in PULPissimo SoC based on formal verification (**SPV** and **FPV**, i.e., JasperGold Security Path Verification and Formal Property Verification) and our hardware security competition (**M&S**, i.e., manual inspection and simulation). Check and cross marks indicate detected and undetected bugs, respectively. Bugs marked **inserted** were injected by our team and based on the listed CVEs, while bugs marked **native** were already present in the SoC and discovered by the participants during Hack@DAC. **LOC** denotes the number of lines of code, and **states** denotes the total number of logic states for the modules needed to attempt to detect this bug.

#	Bug	Type	SPV	FPV	M&S	Modules	LOC	# States
1	Address range overlap between peripherals SPI Master and SoC	Inserted (CVE-2018-12206 / CVE-2019-6260 / CVE-2018-8933)	✓	✓	✓	91	6685	1.5×10^{20}
2	Addresses for L2 memory is out of the specified range.	Native	✓	✓	✓	43	6746	3.5×10^{13}
3	Processor assigns privilege level of execution incorrectly from CSR.	Native	✗	✓	✓	2	1186	2.1×10^{96}
4	Register that controls GPIO lock can be written to with software.	Inserted (CVE-2017-18293)	✓	✓	✗	2	1186	2.1×10^{96}
5	Reset clears the GPIO lock control register.	Inserted (CVE-2017-18293)	✓	✓	✗	2	408	1
6	Incorrect address range for APB allows memory aliasing.	Inserted (CVE-2018-12206 / CVE-2019-6260)	✓	✓	✗	1	110	2
7	AXI address decoder ignores errors.	Inserted (CVE-2018-4850)	✗	✓	✗	1	227	2
8	Address range overlap between GPIO, SPI, and SoC control peripherals.	Inserted (CVE-2018-12206 / (CVE-2017-5704)	✓	✓	✓	68	14635	9.4×10^{21}
9	Incorrect password checking logic in debug unit.	Inserted (CVE-2018-8870)	✗	✓	✗	4	436	1
10	Advanced debug unit only checks 31 of the 32 bits of the password.	Inserted (CVE-2017-18347 / CVE-2017-7564)	✗	✓	✗	4	436	16
11	Able to access debug register when in halt mode.	Native (CVE-2017-18347 /	✗	✓	✓	2	887	1
12	Password check for the debug unit does not reset after successful check.	Inserted (CVE-2017-7564)	✗	✓	✓	4	436	16
13	Faulty decoder state machine logic in RISC-V core results in a hang.	Native	✗	✓	✓	2	1119	32
14	Incomplete case statement in ALU can cause unpredictable behavior.	Native	✗	✓	✓	2	1152	4
15	Faulty logic in the RTC causing inaccurate time calculation for security-critical flows, e.g., DRM.	Native	✗	✓	✗	1	191	1
16	Reset for the advanced debug unit not operational.	Inserted (CVE-2017-18347)	✗	✗	✓	4	436	16
17	Memory-mapped register file allows code injection.	Native	✗	✗	✓	1	134	1
18	Non-functioning cryptography module causes DOS.	Inserted	✗	✗	✗	24	2651	1
19	Insecure hash function in the cryptography module.	Inserted (CVE-2018-1751)	✗	✗	✗	24	2651	N/A
20	Cryptographic key for AES stored in unprotected memory.	Inserted (CVE-2018-8933 / CVE-2014-0881 / CVE-2017-5704)	✗	✗	✗	57	8955	1
21	Temperature sensor is muxed with the cryptography modules.	Inserted	✗	✗	✓	1	65	1
22	ROM size is too small preventing execution of security code.	Inserted (CVE-2018-6242 /) CVE-2018-15383)	✗	✗	✓	1	751	N/A
23	Disabled the ability to activate the security-enhanced core.	Inserted (CVE-2018-12206)	✗	✗	✗	1	282	N/A
24	GPIO enable always high.	Inserted (CVE-2018-1959)	✗	✗	✗	1	392	1
25	Unprivileged user-space code can write to the privileged CSR.	Inserted (CVE-2018-7522 / CVE-2017-0352)	✗	✗	✓	1	745	1
26	Advanced debug unit password is hard-coded and set on reset.	Inserted (CVE-2018-8870)	✗	✗	✓	1	406	16
27	Secure mode is not required to write to interrupt registers.	Inserted (CVE-2017-0352)	✗	✗	✓	1	303	1
28	JTAG interface is not password protected.	Native	✗	✗	✓	1	441	1
29	Output of MAC is not erased on reset.	Inserted	✗	✗	✓	1	65	1
30	Supervisor mode signal of a core is floating preventing the use of SMAP.	Native	✗	✗	✓	1	282	1
31	GPIO is able to read/write to instruction and data cache.	Native	✗	✗	✓	1	151	4

data and instruction translation lookaside buffers (TLBs), a hardware page table walker, and a branch prediction unit to enable speculative execution. Figure 4 in Appendix A shows its high-level microarchitecture.

PULPissimo is an SoC based on a simpler RISC-V core with both instruction and data RAM as shown in Figure 2. It provides an Advanced Extensible Interface (AXI) for accessing memory from the core. Peripherals are directly connected to an Advanced Peripheral Bus (APB) which connects them to the AXI through a bridge module. It provides support for autonomous I/O, external interrupt controllers and features a debug unit and an SPI slave.

TLB Page Fault Timing Side Channel (L-1 & L-2).
While analyzing the Ariane RTL, we noted a timing side-channel leakage with TLB accesses. TLB page faults due to illegal accesses occur in a different number of clock cycles than page faults that occur due to unmapped memory (we contacted the developers and they acknowledged the vulnerability). This timing disparity in the RTL manifests in the microarchitectural behavior of the processor. Thus, it constitutes a software-visible side channel due to the measurable clock-cycle difference in the two cases. Previous work already demonstrated how this can be exploited by user-space adversaries to probe mapped and unmapped pages and to break randomization-based defenses [24, 29]. Timing flow properties cannot be directly expressed by simple properties or modeled by state-of-the-art verification techniques. Moreover, for this vulnerability, we identify at least seven RTL modules that would need to be modeled, analyzed and verified in combination, namely: *mmu.sv - nbdcache.sv - tlb.sv instantiations - ptw.sv - load_unit.sv - store_unit.sv*. Besides modeling their complex inter- and intra-modular logic flows (L-1), the timing flows need to be modeled to formally prove the absence of this timing channel leakage, which is *not supported* by current industry-standard tools (L-2). Hence, the only alternative is to verify this property by manually inspecting and following the clock cycle transitions across the RTL modules, which is highly cumbersome and error-prone. However, the design must still be analyzed to verify that timing side-channel resilience is implemented correctly and bug-free in the RTL. This only becomes far more complex for real-world industry-standard SoCs. We show the RTL hierarchy of the Ariane core in Figure 5 in Appendix A to illustrate its complexity.

Pre-Fetched Cache State Not Rolled Back (L-1 & L-3).
Another issue in Ariane is with the cache state when a system return instruction is executed, where the privilege level of the core is not changed until this instruction is retired. Before retirement, linear fetching (guided by branch prediction) of data and instructions following the unretired system return instruction continues at the current higher system privilege level. Once the instruction is retired, the execution mode of the core is changed to the unprivileged level, but the entries that

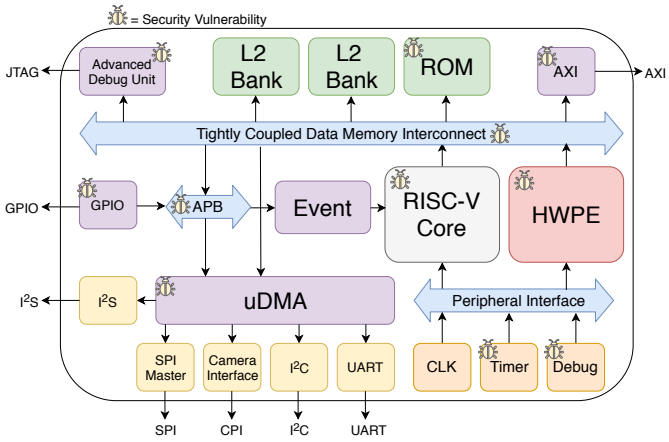


FIGURE 2: Hardware overview of the PULPissimo SoC. Each bug icon indicates the presence of at least one security vulnerability in the module.

were pre-fetched into the cache (at the system privilege level) do not get flushed. These shared cache entries are visible to user-space software, thus enabling timing channels between privileged and unprivileged software.

Verifying the implementation of all the flush control signals and their behavior in all different states of the processor requires examining at least eight modules: *ariane.sv - controller.sv - frontend.sv - id_stage.sv - icache.sv - fetch_fifo - ariane_pkg.sv - csr_regfile.sv* (see Figure 5). This is complex because it requires identifying and defining all the relevant security properties to be checked across these RTL modules. Since current industry-standard approaches do not support expressive capturing and the verification of cache states, this issue in the RTL can only be found by manual inspection.

Firmware-Configured Memory Ranges (L-4).
In PULPissimo, we added peripherals with injected bugs to reproduce bugs from CVEs. We added an AES encryption/decryption engine whose input key is stored and fetched from memory tightly coupled to the processor. The memory address the key is stored in is unknown, and whether it is within the protected memory range or not is inconclusive by observing the RTL alone. In real-world SoCs, the AES key is stored in programmable fuses. During secure boot, the bootloader/firmware senses the fuses and stores the key to memory-mapped registers. The access control filter is then configured to allow only the AES engine access to these registers, thus protecting this memory range. Because the open-source SoC we used did not contain a fuse infrastructure, the key storage was mimicked to be in a register in the Memory-Mapped I/O (MMIO) space.

Although the information flow of the AES key is defined in hardware, its location is controlled by the firmware. Reasoning on whether the information flow is allowed or not using conventional hardware verification approaches is inconclusive when considering the RTL code in isolation.

The vulnerable hardware/firmware interactions cannot be identified unless they are co-verified. Unfortunately, current industry-standard tools do not support this.

Memory Address Range Overlap (L-1 & L-4). PULPissimo provides I/O support to its peripherals by mapping them to different memory address ranges. If an address range overlap bug is committed at design-time or by firmware, this can break access control policies and have critical security consequences, e.g., privilege escalation. We injected an RTL bug where there is address range overlap between the SPI Master Peripheral and the SoC Control Peripheral. This allowed the untrusted SPI Master to access the SoC Control memory address range over the APB bus.

Verifying issues at the SoC interconnect in such complex bus protocols is challenging since too many modules needed to support the interconnect have to be modeled to properly verify their security. This increases the scope and the complexity of potential bugs far beyond just a few modules, as shown in Table 1. Such an effect causes an explosion of the state space since all the possible states have to be modeled accurately to remain sound. Proof kits for accelerated verification of advanced SoC interconnect protocols were introduced to mitigate this for a small number of bus protocols (AMBA3 and AMBA4). However, this requires an add-on to the default software and many protocols are not supported[4].

5 Crowdsourcing Detection

We organized and conducted a capture-the-flag competition, Hack@DAC, in which 54 teams (7 from leading industry vendors and 47 from academia) participated. The objective for the teams was to detect as many RTL bugs as possible from those we injected deliberately in real-world open-source SoC designs (see Table 1). This is designed to mimic real-world bug bounty programs from semiconductor companies [17, 32, 62, 63]. The teams were free to use any techniques: simulation, manual inspection, or formal verification.

5.1 Competition Preparation

RTL of open-source RISC-V SoCs was used as the testbed for Hack@DAC and our investigation. Although these SoCs are less complex than high-end industry proprietary designs, this allows us to feasibly inject (and detect) bugs into less complex RTL. Thus, this represents the best-case results for the verification techniques used during Hack@DAC and our investigation. Moreover, it allows us to open-source and showcase our testbed and bugs to the community.Hack@DAC consisted of two phases: a preliminary Phase 1 and final Phase 2, which featured the RISC-V Pulpino and PULPissimo SoCs, respectively. Phase 1 was conducted remotely over a two-month period. Phase 2 was conducted in an 8-hour time frame co-located with DAC (Design Automation Conference).

For Phase 1, we chose the Pulpino [60] SoC since it was a real-world, yet not an overly complex SoC design for the teams to work with. It features a RISC-V core with instruction and data RAM, an AXI interconnect for accessing memory, with peripherals on an APB accessing the AXI through a bridge module. It also features a boot ROM, a debug unit and a serial peripheral interface (SPI) slave. We inserted security bugs in multiples modules of the SoC, including the AXI, APB, debug unit, GPIO, and bridge.

For Phase 2, we chose the more complex PULPissimo [61] SoC, shown in Figure 2. It additionally supports hardware processing engines, DMA, and more peripherals. This allowed us to extend the SoC with additional security features, making room for additional bugs. Some native security bugs were discovered by the teams and were reported to the SoC designers.

5.2 Competition Objectives

For Hack@DAC, we first implemented additional security features in the SoC, then defined the security objectives and adversary model and accordingly inserted the bugs. Specifying the security goals and the adversary model allows teams to define what constitutes a security bug. Teams had to provide a bug description, location of RTL file, code reference, the security impact, adversary profile, and the proposed mitigation.
Security Features: We added password-based locks on the JTAG modules of both SoCs and access control on certain peripherals. For the Phase-2 SoC, we also added a cryptographic unit implementing multiple cryptographic algorithms. We injected bugs into these features and native features to generate security threats as a result.
Security Goals: We provided the three main security goals for the target SoCs to the teams. Firstly, unprivileged code should not escalate beyond its privilege level. Secondly, the JTAG module should be protected against an adversary with physical access. Finally, the SoCs should thwart software adversaries from launching denial-of-service attacks.

5.3 Overview of Competition Bugs

As described earlier in Section 3.2, the bugs were selected and injected together with our Intel collaborators. They are inspired by their hardware security expertise and real-world CVEs (cf. Table 1) and aim to achieve coverage of different security-relevant components of the SoC. Several participants also reported a number of *native* bugs already present in the SoC that we did not deliberately inject. We describe below some of the most interesting bugs.
UDMA address range overlap: We modified the memory address range of the UDMA so that it overlaps with the master port to the SPI. This bug allows an adversary with access to

[4] http://www.marketwired.com/press-release/jasper-introduces-intelligent-proof-kits-faster-more-accurate-verification-soc-interface-1368721.htm

the UMDA memory to escalate its privileges and modify the SPI memory. This bug is an example of the "Memory Address Range Overlap" HardFail type in Section 4. Other address range configuration bugs (#1, 2, 6 and 8) were also injected in the APB bus for different peripherals.

GPIO errors: The address range of the GPIO memory was erroneously declared. An adversary with GPIO access can escalate its privilege and access the SPI Master and SoC Control. The GPIO enable was rigged to display a fixed erroneous status of '1', which did not give the user a correct display of the actual GPIO status. The GPIO lock control register was made write-accessible by user-space code, and it was flawed to clear at reset. Bugs #4, 5, 24 and 31 are such examples.

Debug/JTAG errors: The password-checking logic in the debug unit was flawed and its state was not being correctly reset after a successful check. We hard-coded the debug unit password, and the JTAG interface was not password protected. Bugs #9, 10, 11, 16, 26, and 28 are such examples.

Untrusted boot ROM: A native bug (bug #22) would allow unprivileged compromise of the boot ROM and potentially the execution of untrusted boot code at a privileged level, thus disclosing sensitive information.

Erroneous AXI finite-state machine: We injected a bug (bug #7) in the AXI address decoder such that, if an error signal is generated on the memory bus while the underlining logic is still handling an outstanding transaction, the next signal to be handled will instead be considered operational by the module unconditionally. This bug can be exploited to cause computational faults in the execution of security-critical code (we showcase how to exploit this vulnerability—which was not detected by all teams—in Appendix D).

Cryptographic unit bugs: We injected bugs in a cryptographic unit that we inserted to trigger denial-of-service, a broken cryptographic implementation, insecure key storage, and disallowed information leakage. Bugs #18, 19, 20, 21, and 29 are such examples.

5.4 Competition Results

Various insights were drawn from the submitted bug reports and results, which are summarized in Table 1.

Analyzing the bug reports: Bug reports submitted by teams revealed which bug types were harder to detect and analyze using existing approaches. We evaluated the submissions and rated them for accuracy and detail, e.g., bug validity, methodology used, and security impact.

Detected bugs: Most teams easily detected two bugs in PULPissimo. The first one is where debug IPs were used when not intended. The second bug was where we declared a local parameter PULP_SEC, which was always set to '1', instead of the intended PULP_SECURE. The former was detected because debugging interfaces represent security-critical regions of the chip. The latter was detected because it indicated intuitively that exploiting this parameter would lead to privilege escalation attacks. The teams reported that they prioritized inspecting security-relevant modules of the SoC, such as the debug interfaces.

Undetected bugs: Many inserted bugs were not detected. One was in the advanced debug unit, where the password bit index register has an overflow (bug #9). This is an example of a security flaw that would be hard to detect by methods other than verification. Moreover, the presence of many bugs within the advanced debug unit password checker further masked this bug. Another bug was the cryptographic unit key storage in unprotected memory (bug #20). The teams could not detect it as they focused on the RTL code in isolation and did not consider HW/FW interactions.

Techniques used by the teams: The teams were free to use any techniques to detect the bugs but most teams eventually relied on manual inspection and simulation.

- **Formal verification:** One team used an open-source formal verification tool (VeriCoq), but they reported little success because these tools (i) did not scale well with the complete SoC and (ii) required expertise to use and define the security properties. Some teams deployed their in-house verification techniques, albeit with little success. They eventually resorted to manual analysis.
- **Assertion-based simulation:** Some teams prepared RTL testbenches and conducted property-based simulations using SystemVerilog assertion statements.
- **Manual inspection:** All teams relied on manual inspection methods since they are the easiest and most accessible and require less expertise than formal verification, especially when working under time constraints. A couple of teams reported prioritizing the inspection of security-critical modules such as debug interfaces.
- **Software-based testing:** One team detected software-exposure and privilege escalation bugs by running C code on the processor and attempting to make arbitrary reads/writes to privileged memory locations. In doing this, they could detect bugs #4, #8, #15, and #17.

Limitations of manual analysis: While manual inspection can detect the widest array of bugs, our analysis of the Hack@DAC results reveals its limitations. Manual analysis is qualitative and difficult to scale to cross-layer and more complex bugs. In Table 1, out of 16 cross-module bugs (spanning more than one module) only 9 were identified using manual inspection. Three of them (#18, #19, and #20) were also undetected by formal verification methods, which is 10% of the bugs in our case studies.

6 Detection Using State-of-The-Art Tools

Our study reveals two results: (1) a number of bugs could not be detected by means of manual auditing and other ad-hoc methods, and (2) the teams were able to find bugs already existing in the SoC which we did not inject and were not

aware of. This prompted us to conduct a second in-house case study to further investigate whether formal verification techniques can be used to detect these bugs. In practice, hardware-security verification engineers use a combination of techniques such as formal verification, simulation, emulation, and manual inspection. Our first case study covered manual inspection, simulation and emulation techniques. Thus, we focused our second case study on assessing the effectiveness of industry-standard formal verification techniques usually used for verifying pre-silicon hardware security.

In real-world security testing (see Section 2), engineers will not have prior knowledge of the specific vulnerabilities they are trying to find. Our goal, however, is to investigate how an industry-standard tool can detect RTL bugs that we deliberately inject in an open-source SoC and have prior knowledge of (see Table 1). Since there is no regulation or explicitly defined standard for hardware-security verification, we focus our investigation on the most popular and de-facto standard formal verification platform used in industry [11]. This platform encompasses a representative suite of different state-of-the-art formal verification techniques for hardware security assurance. As opposed to simulation and emulation techniques, formal verification guarantees to model the state space of the design and formally prove the desired properties. We emphasize that we deliberately fix all other variables involved in the security testing process, in order to focus in a controlled setting on testing the capacity and limitations of the techniques and tools themselves. Thus, our results reflect the effectiveness of tools in a best case where the bug is known a priori. This eliminates the possibility of writing an incorrect security property assertion which fails to detect the bug.

6.1 Detection Methodology

We examined each of the injected bugs and its nature in order to determine which formal technique would be best suited to detect it. We used two formal techniques: Formal Property Verification (FPV) and JasperGold's Security Path Verification (SPV) [12]. They represent the state of the art in hardware security verification and are used widely by the semiconductor industry [4], including Intel.

FPV checks whether a set of security properties, usually specified as SystemVerilog Assertions (SVA), hold true for the given RTL. To describe the assertions correctly, we examined the location of each bug in the RTL and how its behavior is manifested with the surrounding logic and input/output relationships. Once we specified the security properties using *assert*, *assume* and *cover* statements, we determined which RTL modules we need to model to prove these assertions. If a security property is violated, the tool generates a counterexample; this is examined to ensure whether the intended security property is indeed violated or is a false alarm.

SPV detects bugs which specifically involve unauthorized information flow. Such properties cannot be directly captured using SVA/PSL assertions. SPV uses path sensitization techniques to exhaustively and formally check if unauthorized data propagates (through a functional path) from a source to a destination signal. To specify the SPV properties, we identified source signals where the sensitive information was located and destination signals where it should *not* propagate. We then identified the bounding preconditions to constrain the paths the tool searches to alleviate state and time explosion. Similar to FPV, we identified the modules that are required to capture the information flow of interest. This must include source, destination and intermediate modules, as well as modules that generate control signals which interfere with the information flow.

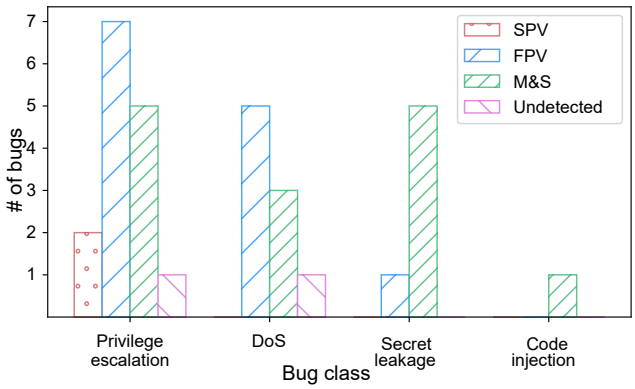


FIGURE 3: Verification results grouped by bug class and the number of bugs in each class detected by Security Path Verification (SPV), Formal Property Verification (FPV) and manual inspection and simulation techniques (M&S).

6.2 Detection Results

Of the 31 bugs we investigated, shown in Table 1, using the formal verification techniques described above, only 15 (48%) were detected. While we attempted to detect all 31 bugs formally, we were able to formulate security properties for only 17 bugs. This indicates that the main challenge with using formal verification tools is identifying and expressing security properties that the tools are capable of capturing and checking. Bugs due to ambiguous specifications of interconnect logic, for instance, are examples of bugs that are difficult to create security properties for.

Our results, shown in Figure 3, indicate that privilege escalation and denial-of-service (DoS) bugs were the most detected at 60% and 67% respectively. Secret leakage only had a 17% detection rate due to incorrect design specification for one bug, state explosion and the inability to express properties that the tool can assert for the remaining bugs. The code injection bug was undetected by formal techniques. Bugs at the interconnect level of the SoC such as bugs #1 and #2 were especially challenging since they involved a large number of highly complex and inter-connected modules that needed to be loaded and modeled by the tool (see L-1 in Section 3.1). Bug #20, which involves hardware/firmware interactions, was also

detected by neither the state-of-the-art FPV nor SPV since they analyze the RTL in isolation (see L-4 in Section 3.1). We describe these bugs in more detail in Appendix C.

6.3 State-Explosion Problem

Formal verification techniques are quickly driven into state space explosion when analyzing large designs with many states. Many large interconnected RTL modules, like those relevant to bugs #1 and #2, can have states in the order of magnitude of 10^{20}. Even smaller ones, like these used for bugs #3 and #4, can have a very large number of states, as shown in Table 1. When combined, the entire SoC will have a total number of states significantly higher than any of the results in Table 1. Attempting to model the entire SoC drove the tool into state explosion, and it ran out of memory and crashed. Formal verification tools, including those specific to security verification are currently incapable of handling so many states, even when computational resources are increased. This is further aggravated for industry-standard complex SoCs.

Because the entire SoC cannot be modeled and analyzed at once, detecting cross-modular bugs becomes very challenging. Engineers work around this (not fundamentally solve it) by adopting a divide-and-conquer approach and selecting which modules are relevant for the properties being tested and which can be black-boxed or abstracted. However, this is time-consuming, non-automated, error-prone, and requires expertise and knowledge of both the tools and design. By relying on the human factor, the tool can no longer guarantee the absence of bugs for the entire design, which is the original advantage of formal verification.

7 Discussion and Future Work

We now describe why microcode patching is insufficient for RTL bugs while emphasizing the need for advancing the hardware security verification process. We discuss the additional challenges of the overall process, besides the limitations of the industry-standard tools, which is the focus of this work.

7.1 Microcode Patching

While existing industry-grade SoCs support hotfixes by *microcode patching* for instance, this approach is limited to a handful of changes to the instruction set architecture, e.g., modifying the interface of individual complex instructions and adding or removing instructions [25]. Some vulnerabilities cannot even be patched by microcode, such as the recent Spoiler attack [33]. Fundamentally mitigating this requires fixing the hardware of the memory subsystem at the hardware design phase. For legacy systems, the application developer is advised to follow best practices for developing side channel-resilient software[5]. For vulnerabilities that can be patched, patches at this higher abstraction level in the firmware only act as a "symptomatic" fix that circumvent the RTL bug. However, they do not fundamentally patch the bug in the RTL, which is already realized as hardwired logic. Thus, microcode patching is a fallback for RTL bugs discovered after production, when you can not patch the RTL. They may also incur performance impact [6] that could be avoided if the underlying problem is discovered and fixed during design.

7.2 Additional Challenges in Practice

Functional vs. Security Specifications. As described in Section 2, pre- and post-silicon validation efforts are conducted to verify that the implementation fully matches both its functional and security specifications. The process becomes increasingly difficult (almost impossible) as the system complexity increases and specification ambiguity arises. Deviations from specification occur due to either functional or security bugs, and it is important to distinguish between them. While functional bugs generate functionally incorrect results, security bugs are not reflected in functionality. They arise due to unconsidered and corner threat cases that are unlikely to get triggered, thus making them more challenging to detect and cover. It is, therefore, important to distinguish between functional and security specifications, since these are often the references for different verification teams working concurrently on the same RTL implementation.

Specification Ambiguity. Another challenge entails anticipating and identifying all the security properties that are required in a real-world scenario. We analyzed the efficacy of industry-standard tools in a controlled setting—where we have prior knowledge of the bugs. However, in practice hardware validation teams do not have prior knowledge of the bugs. Security specifications are often incomplete and ambiguous, only outlining the required security properties under an assumed adversary model. These specifications are invalidated once the adversary model is changed. This is often the case with IP reuse, where the RTL code for one product is re-purposed for another with a different set of security requirements and usage scenarios. Parameters may be declared multiple times and get misinterpreted by the tools, thus causing bugs to slip undetected. Furthermore, specs usually do not specify bugs and information flows that should not exist, and there is no automated approach to determine whether one is proving the intended properties. Thus, a combination of incomplete or incorrect design decisions and implementation errors can easily introduce bugs to the design.

[5] https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00238.html

[6] https://access.redhat.com/articles/3307751

7.3 Future Research Directions

Through our work, we shed light on the limitations of state-of-the-art verification techniques. In doing so, we hope to motivate further research in advancing these techniques to adequately capture and detect these vulnerabilities.

Although manual RTL inspection is generally useful and can potentially cover a wide array of bugs, its efficacy depends exclusively on the expertise of the engineer. This can be inefficient, unreliable and ad hoc in light of rapidly evolving chip designs. Exhaustive testing of specifications through simulation requires amounts of resources exponential in the size of the input (i.e., design state space) while coverage must be intelligently maximized. Hence, current approaches face severe scalability challenges, as diagnosing software-exploitable bugs that reside deep in the design pipeline can require simulation of trillions of cycles [14]. Our results indicate that it is important to first identify high-risk components due to software exposure, such as password checkers, crypto cores, and control registers, and prioritize analyzing them. Scalability due to complex inter-dependencies among modules is one challenge for detection. Vulnerabilities associated with non-register states (such as caches) or clock-cycle dependencies (i.e., timing flows) are another open problem. Initial research is underway [71] to analyze a limited amount of low-level firmware running on top of a simulated RTL design for information and timing flow violations. However, these approaches are still in their infancy and yet to scale for real-world SoC designs.

8 Related Work

We now present related work in hardware security verification while identifying limitations with respect to detecting HardFails. We also provide an overview of recent software attacks exploiting underlying hardware vulnerabilities.

8.1 Current Detection Approaches

Security-aware design of hardware has gained significance only recently as the critical security threat posed by hardware vulnerabilities became acutely established. Confidentiality and integrity are the commonly investigated properties [19] in hardware security. They are usually expressed using information flow properties between entities at different security levels. Besides manual inspection and simulation-based techniques, systematic approaches proposed for verifying hardware security properties include formal verification methods such as proof assistance, model-checking, symbolic execution, and information flow tracking. We exclude the related work in testing mechanisms, e.g., JTAG/scan-chain/built-in self-test, because they are leveraged for hardware testing **after** fabrication. However, the focus of this work is on verifying the security of the hardware **before** fabrication. Interestingly, this includes verifying that the test mechanisms are correctly implemented in the RTL, otherwise they may constitute security vulnerabilities when used after fabrication (see bugs#9,#10,#11,#12,#16, #26 of the JTAG/debug interface).

Proof assistant and theorem-proving methods rely on mathematically modeling the system and the required security properties into logical theorems and formally proving if the model complies with the properties. VeriCoq [7] based on the Coq proof assistant transforms the Verilog code that describes the hardware design into proof-carrying code.VeriCoq supports the automated conversion of only a subset of Verilog code into Coq. However, this assumes accurate labeling of the initial sensitivity labels of each and every signal in order to effectively track the flow of information. This is cumbersome, error-prone, generates many faluse positives, and does not scale well in practice beyond toy examples. Moreover, timing (and other) side-channel information flows are not modeled. Finally, computational scalability to verifying real-world complex SoCs remains an issue given that the proof verification for a single AES core requires $\approx$ 30 minutes to complete [6].

Model checking-based approaches check a given property against the modeled state space and possible state transitions using provided invariants and predefined conditions. They face scalability issues as computation time scales exponentially with the model and state space size. This can be alleviated by using abstraction to simplify the model or constraining the state space to a bounded number of states using assumptions and conditions. However, this introduces false positives, may miss vulnerabilities, and requires expert knowledge. Most industry-leading tools, such as the one we use in this work, rely on model checking algorithms such as boolean satisfiability problem solvers and property specification schemes, e.g., assertion-based verification to verify the required properties of a given hardware design.

Side-channel leakage modeling and detection remain an open problem. Recent work [76] uses the Murφ model checker to verify different hardware cache architectures for side-channel leakage against different adversary models. A formal verification methodology for SGX and Sanctum enclaves under a limited adversary was introduced in [67]. However, such approaches are not directly applicable to hardware implementation. They also rely exclusively on formal verification and remain inherently limited by the underlying algorithms in terms of scalability and state space explosion, besides demanding particular expertise to use.

Information flow analysis (such as SPV) works by assigning a security label (or a *taint*) to a data input and monitoring the taint propagation. In this way, the designer can verify whether the system adheres to the required security policies. Recently, information flow tracking (IFT) has been shown effective in identifying security vulnerabilities, including timing side channels and information-leaking hardware Trojans.

IFT techniques are proposed at different levels of abstraction: gate-, RT, and language-levels. Gate-level information

flow tracking (GLIFT) [2, 58, 70] performs the IFT analysis directly at gate-level by generating GLIFT analysis logic that is derived from the original logic and operates in parallel to it. Although gate-level IFT logic is easy to automatically generate, it does not scale well. Furthermore, when IFT uses strict non-interference, it taints any information flow conservatively as a vulnerability [34] which scales well for more complex hardware, but generates too many false positives.

At the language level, Caisson [42] and Sapper [41] are security-aware HDLs that use a typing system where the designer assigns security "labels" to each variable (wire or register) based on the security policies required. However, they both require redesigning the RTL using a new hardware description language which is not practical. SecVerilog [22, 75] overcomes this by extending the Verilog language with a dynamic security type system. Designers assign a security label to each variable (wire or register) in the RTL to enable a compile-time check of hardware information flow. However, this involves complex analysis during simulation to reason about the run-time behavior of the hardware state and dependencies across data types for precise flow tracking.

Hardware/firmware co-verification to capture and verify hardware/firmware interactions remains an open challenge and is not available in widely used industry-standard tools. A co-verification methodology [28] addresses the semantic gap between hardware and firmware by modeling hardware and firmware using instruction-level abstraction to leverage software verification techniques. However, this requires modeling the hardware that interacts with firmware into an abstraction which is semi-automatic, cumbersome, and lossy.

While research is underway [71] to analyze a limited amount of low-level firmware running on top of a simulated RTL design these approaches are still under development and not scalable. Current verification approaches focus on register-state information-flow analysis, e.g., to monitor whether sensitive locations are accessible from unprivileged signal sources. Further research is required to explicitly model non-register states and timing explicitly alongside the existing capabilities of these tools.

8.2 Recent Attacks

We present and cautiously classify the underlying hardware vulnerabilities of recent cross-layer exploits (see Table 2 in Appendix B), using the categories introduced in 3.1. We do not have access to proprietary processor implementations, so our classification is only based on our deductions from the published technical descriptions. Yarom et al. demonstrate that software-visible side channels can exist even below cache-line granularity in CacheBleed [74]–undermining a core assumption of prior defenses, such as scatter-gather [9]. MemJam [45] exploits false read-after-write dependencies in the CPU to maliciously slow down victim accesses to memory blocks within a cache line. We categorize the underlying vulnerabilities of CacheBleed and MemJam as potentially hard to detect in RTL due to the many cross-module connections involved and the timing-flow leakage. The timing flow leakage is caused by the software triggering clock cycle differences in accesses that map to the same bank below cache line granularity, thus breaking constant-time implementations.

The TLBleed [23] attack shows how current TLB implementations can be exploited to break state-of-the-art cache side-channel protections. As described in Section 4, TLBs are typically highly interconnected with complex processor modules, such as the cache controller and memory management unit, making vulnerabilities therein very hard to detect through automated verification or manual inspection.

BranchScope [20] extracts information through the directional branch predictor, thus bypassing software mitigations that prevent leakage via the BTB. We classify it as a cache-state gap in branch prediction units, which is significantly challenging to detect using existing RTL security verification tools, which cannot capture and verify cache states. Meltdown [43] exploits speculative execution on modern processors to completely bypass all memory access restrictions. Van Bulck et al. [72] also demonstrated how to apply this to Intel SGX. Similarly, Spectre [37] exploits out-of-order execution across different user-space processes as arbitrary instruction executions would continue during speculation. We recognize these vulnerabilities are hard to detect due to scalability challenges in existing tools, since the out-of-order scheduling module is connected to many subsystems in the CPU. Additionally, manually inspecting these interconnected complex RTL modules is very challenging and cumbersome.

CLKScrew [69] abuses low-level power-management functionality that is exposed to software to induce faults and glitches dynamically at runtime in the processor. We categorize CLKScrew to have vulnerable hardware-firmware interactions and timing-flow leakage, since it directly exposes clock-tuning functionality to attacker-controlled software.

9 Conclusion

Software security bugs and their impact have been known for many decades, with a spectrum of established techniques to detect and mitigate them. However, the threat of hardware security bugs has only recently become significant as cross-layer exploits have shown that they can completely undermine software security protections. While some hardware bugs can be patched with microcode updates, many cannot, often leaving millions of affected chips in the wild. In this paper, we presented the first testbed of RTL bugs and systematically analyzed the effectiveness of state-of-the-art formal verification techniques, manual inspection and simulation methods in detecting these bugs. We organized an international hardware security competition and an in-house study. Our results have shown that 54 teams were only able to detect 61% of the total number of bugs, while with industry-leading formal verification techniques, we were only able to detect 48% of

the bugs. We showcase that the grave security impact of many of these undetected bugs is only further exacerbated by being software-exploitable.

Our investigation revealed the limitations of state-of-the-art verification/detection techniques with respect to detecting certain classes of hardware security bugs that exhibit particular properties. These approaches remain limited in the face of detecting vulnerabilities that require capturing and verifying complex cross-module inter-dependencies, timing flows, cache states, and hardware-firmware interactions. While these effects are common in SoC designs, they are difficult to model, capture, and verify using current approaches. Our investigative work highlights the necessity of treating the detection of hardware bugs as significantly as that of software bugs. Through our work, we highlight the pressing call for further research to advance the state of the art in hardware security verification. Particularly, our results indicate the need for increased scalability, efficacy and automation of these tools, making them easily applicable to large-scale commercial SoC designs—without which software protections are futile.

Acknowledgments

We thank our anonymous reviewers and shepherd, Stephen Checkoway, for their valuable feedback. The work was supported by the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS), the German Research Foundation (DFG) by CRC 1119 CROSSING P3, and the Office of Naval Research (ONR Award #N00014-18-1-2058). We would also like to acknowledge the co-organizers of Hack@DAC: Dan Holcomb (UMass-Amherst), Siddharth Garg (NYU), and Sourav Sudhir (TAMU), and the sponsors of Hack@DAC: the National Science Foundation (NSF CNS-1749175), NYU CCS, Mentor - a Siemens Business and CROSSING, as well as the participants of Hack@DAC.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. *ACM conference on Computer and communications security*, pages 340–353, 2005.

[2] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner. Register Transfer Level Information Flow Tracking for Provably Secure Hardware Design. *Design, Automation & Test in Europe*, pages 1695–1700, 2017.

[3] ARM. Security technology building a secure system using trustzone technology (white paper). http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2009.

[4] R. Armstrong, R. Punnoose, M. Wong, and J. Mayo. Survey of Existing Tools for Formal Verification. Sandia National Laboratories https://prod.sandia.gov/techlib-noauth/access-control.cgi/2014/1420533.pdf, 2014.

[5] Averant. Solidify. http://www.averant.com/storage/documents/Solidify.pdf, 2018.

[6] M.-M. Bidmeshki, X. Guo, R. G. Dutta, Y. Jin, and Y. Makris. Data Secrecy Protection Through Information Flow Tracking in Proof-Carrying Hardware IP—Part II: Framework Automation. *IEEE Transactions on Information Forensics and Security*, 12(10):2430–2443, 2017.

[7] M.-M. Bidmeshki and Y. Makris. VeriCoq: A Verilog-to-Coq Converter for Proof-Carrying Hardware Automation. *IEEE International Symposium on Circuits and Systems*, pages 29–32, 2015.

[8] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf. SANCTUARY: ARMing TrustZone with User-space Enclaves. *Network and Distributed System Security Symposium (NDSS)*, 2019.

[9] E. Brickell, G. Graunke, M. Neve, and J.-P. Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. *IACR Cryptology ePrint Archive*, 2006:52, 2006.

[10] Cadence. Incisive Enterprise Simulator. https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/incisive-enterprise-simulator.html, 2014.

[11] Cadence. JasperGold Formal Verification Platform. https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html, 2014.

[12] Cadence. JasperGold Security Path Verification App. https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/security-path-verification-app.html, 2018. Last accessed on 09/09/18.

[13] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. *USENIX Symposium on Operating Systems Design and Implementation*, pages 147–160, 2006.

[14] D. P. Christopher Celio, Krste Asanovic. The Berkeley Out-of-Order Machine. https://riscv.org/wp-content/uploads/2016/01/Wed1345-RISCV-Workshop-3-BOOM.pdf, 2016.

[15] Cisco. Cisco: Strengthening Cisco Products. https://www.cisco.com/c/en/us/about/security-center/security-programs/secure-development-lifecycle.html, 2017.

[16] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani. Model checking and the state explosion problem. *Tools for Practical Software Verification*, 2012.

[17] K. Conger. Apple announces long-awaited bug bounty program. https://techcrunch.com/2016/08/04/apple-announces-long-awaited-bug-bounty-program/, 2016.

[18] V. Costan, I. A. Lebedev, and S. Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. *USENIX Security Symposium*, pages 857–874, 2016.

[19] O. Demir, W. Xiong, F. Zaghloul, and J. Szefer. Survey of approaches for security verification of hardware/software systems. https://eprint.iacr.org/2016/846.pdf, 2016.

[20] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, D. Ponomarev, et al. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. *ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 693–707, 2018.

[21] F. Farahmandi, Y. Huang, and P. Mishra. Formal Approaches to Hardware Trust Verification. *The Hardware Trojan War*, 2018.

[22] A. Ferraiuolo, R. Xu, D. Zhang, A. C. Myers, and G. E. Suh. Verification of a Practical Hardware Security Architecture Through Static Information Flow Analysis. *ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 555–568, 2017.

[23] B. Gras, K. Razavi, H. Bos, and C. Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. *USENIX Security Symposium*, 2018.

[24] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 368–379, 2016.

[25] M. Hicks, C. Sturton, S. T. King, and J. M. Smith. SPECS: A Lightweight Runtime Mechanism for Protecting Software from Security-Critical Processor Bugs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS. ACM, 2015.

[26] M. Howard and S. Lipner. The Security Development Lifecycle. *Microsoft Press Redmond*, 2006.

[27] H. Hu, S. Shinde, A. Sendroiu, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. *IEEE Symposium on Security and Privacy*, 2016.

[28] B.-Y. Huang, S. Ray, A. Gupta, J. M. Fung, and S. Malik. Formal Security Verification of Concurrent Firmware in SoCs Using Instruction-level Abstraction for Hardware. *ACM Annual Design Automation Conference*, pages 91:1–91:6, 2018.

[29] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space ASLR. *Symposium on Security and Privacy*, 2013.

[30] F. Inc. Common Vulnerability Scoring System v3.0. https://www.first.org/cvss/cvss-v30-specification-v1.8.pdf, 2018.

[31] Intel. Intel Software Guard Extensions (Intel SGX). https://software.intel.com/en-us/sgx, 2016. Last accessed on 09/05/18.

[32] Intel. Intel Bug Bounty Program. https://www.intel.com/content/www/us/en/security-center/bug-bounty-program.html, 2018.

[33] S. Islam, A. Moghimi, I. Bruhns, M. Krebbel, B. Gulmezoglu, T. Eisenbarth, and B. Sunar. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. https://arxiv.org/abs/1903.00446, 2019.

[34] R. Kastner, W. Hu, and A. Althoff. Quantifying Hardware Security Using Joint Information Flow Analysis. *IEEE Design, Automation & Test in Europe*, pages 1523–1528, 2016.

[35] H. Khattri, N. K. V. Mangipudi, and S. Mandujano. Hsdl: A security development lifecycle for hardware technologies. *IEEE International Symposium on Hardware-Oriented Security and Trust*, pages 116–121, 2012.

[36] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[37] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. http://arxiv.org/abs/1801.01203, 2018.

[38] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. *International Symposium on Code Generation and Optimization*, 2004.

[39] D. Lee. Keystone enclave: An open-source secure enclave for risc-v. https://keystone-enclave.org/, 2018.

[40] Lenovo. Lenovo: Taking Action on Product Security. https://www.lenovo.com/us/en/product-security/about-lenovo-product-security, 2017.

[41] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong. Sapper: A Language for Hardware-level Security Policy Enforcement. *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 97–112, 2014.

[42] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf. Caisson: A Hardware Description Language for Secure Information Flow. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 46(6):109–120, 2011.

[43] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. https://arxiv.org/abs/1801.01207, 2018.

[44] Mentor. Questa Verification Solution. https://www.mentor.com/products/fv/questa-verification-platform, 2018.

[45] A. Moghimi, T. Eisenbarth, and B. Sunar. MemJam: A false dependency attack against constant-time crypto implementations in SGX. *Cryptographers' Track at the RSA Conference*, pages 21–44, 2018. 10.1007/978-3-319-76953-0_2.

[46] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 1999.

[47] NIST. HP: Remote update feature in HP LaserJet printers does not require password. https://nvd.nist.gov/vuln/detail/CVE-2004-2439, 2004.

[48] NIST. Microsoft: Hypervisor in Xbox 360 kernel allows attackers with physical access to force execution of the hypervisor syscall with a certain register set, which bypasses intended code protection. https://nvd.nist.gov/vuln/detail/CVE-2007-1221, 2007.

[49] NIST. Apple: Multiple heap-based buffer overflows in the AudioCodecs library in the iPhone allows remote attackers to execute arbitrary code or cause DoS via a crafted AAC/MP3 file. https://nvd.nist.gov/vuln/detail/CVE-2009-2206, 2009.

[50] NIST. Broadcom Wi-Fi chips denial of service. https://nvd.nist.gov/vuln/detail/CVE-2012-2619, 2012.

[51] NIST. Vulnerabilities in Dell BIOS allows local users to bypass intended BIOS signing requirements and install arbitrary BIOS images. https://nvd.nist.gov/vuln/detail/CVE-2013-3582, 2013.

[52] NIST. Google: Escalation of Privilege Vulnerability in MediaTek WiFi driver. https://nvd.nist.gov/vuln/detail/CVE-2016-2453, 2016.

[53] NIST. Samsung: Page table walks conducted by MMU during Virtual to Physical address translation leaves in trace in LLC. https://nvd.nist.gov/vuln/detail/CVE-2017-5927, 2017.

[54] NIST. AMD: Backdoors in security co-processor ASIC. https://nvd.nist.gov/vuln/detail/CVE-2018-8935, 2018.

[55] NIST. AMD: EPYC server processors have insufficient access control for protected memory regions. https://nvd.nist.gov/vuln/detail/CVE-2018-8934, 2018.

[56] NIST. Buffer overflow in bootrom recovery mode of nvidia tegra mobile processors. https://nvd.nist.gov/vuln/detail/CVE-2018-6242, 2018.

[57] J. Oberg. Secure Development Lifecycle for Hardware Becomes an Imperative. https://www.eetimes.com/author.asp?section_id=36&doc_id=1332962, 2018.

[58] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner. Theoretical Analysis of Gate Level Information Flow Tracking. *IEEE/ACM Design Automation Conference*, pages 244–247, 2010.

[59] PULP Platform. Ariane. https://github.com/pulp-platform/ariane, 2018.

[60] PULP Platform. Pulpino. https://github.com/pulp-platform/pulpino, 2018.

[61] PULP Platform. Pulpissimo. https://github.com/pulp-platform/pulpissimo, 2018.

[62] Qualcomm. Qualcomm Announces Launch of Bounty Program. https://www.qualcomm.com/news/releases/2016/11/17/qualcomm-announces-launch-bounty-program-offering-15000-usd-discovery, 2018.

[63] Samsung. Rewards Program. https://security.samsungmobile.com/rewardsProgram.smsb, 2018.

[64] M. Seaborn and T. Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. *Black Hat*, 15, 2015.

[65] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). *ACM Symposium on Computer and Communication Security*, pages 552–561, 2007.

[66] O. Solutions. OneSpin 360. https://www.onespin.com/fileadmin/user_upload/pdf/datasheet_dv_web.pdf, 2013.

[67] P. Subramanyan, R. Sinha, I. Lebedev, S. Devadas, and S. A. Seshia. A Formal Foundation for Secure Remote Execution of Enclaves. *ACM SIGSAC Conference on Computer and Communications Security*, pages 2435–2450, 2017.

[68] Sunny .L He and Natalie H. Roe and Evan C. L. Wood and Noel Nachtigal and Jovana Helms. Model of the Product Development Lifecycle. https://prod.sandia.gov/techlib-noauth/access-control.cgi/2015/159022.pdf, 2015.

[69] A. Tang, S. Sethumadhavan, and S. Stolfo. CLKSCREW: exposing the perils of security-oblivious energy managemen. *USENIX Security Symposium*, pages 1057–1074, 2017.

[70] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood. Complete Information Flow Tracking from the Gates Up. *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 109–120, 2009.

[71] Tortuga Logic. Verifying Security at the Hardware/Software Boundary. http://www.tortugalogic.com/unison-whitepaper/, 2017.

[72] J. Van Bulck, F. Piessens, and R. Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. *USENIX Security Symposium*, 2018.

[73] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic. The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.0. https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf, 2014.

[74] Y. Yarom, D. Genkin, and N. Heninger. CacheBleed: a timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017. 10.1007/s13389-017-0152-y.

[75] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers. A Hardware Design Language for Timing-Sensitive Information-Flow Security. *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 503–516, 2015.

[76] T. Zhang and R. B. Lee. New Models of Cache Architectures Characterizing Information Leakage from Cache Side Channels. *ACSAC*, pages 96–105, 2014.

Appendix

A Ariane Core and RTL Hierarchy

Figure 4 shows the high-level microarchitecture of the Ariane core to visualize its complexity. This RISC-V core is far less complex than an x86 or ARM processor and their more sophisticated microarchitectural and optimization features.

Figure 5 illustrates the hierarchy of the RTL components of the Ariane core. This focuses only on the core and excludes all uncore components, such as the AXI interconnect, peripherals, the debug module, boot ROM, and RAM.

B Recent Microarchitectural Attacks

We reviewed recent microarchitectural attacks with respect to existing hardware verification approaches and their limitations. We observe that the underlying vulnerabilities would be difficult to detect due to the properties that they exhibit, rendering them as potential HardFails. We do not have access to their proprietary RTL implementation and cannot inspect the underlying vulnerabilities. Thus, we only infer from the published technical descriptions and errata of these attacks the nature of the underlying RTL issues. We classify in Table 2 the properties of these vulnerabilities that represent challenges for state-of-the-art hardware security verification.

C Details on the Pulpissimo Bugs

We present next more detail on some of the RTL bugs used in our investigation.

Bugs in crypto units and incorrect usage: We extended the SoC with a faulty cryptographic unit with a multiplexer to select between AES, SHA1, MD5, and a temperature sensor. The multiplexer was modified such that a race condition occurs if more than one bit in the status register is enabled, causing unreliable behavior in these security critical modules.

Furthermore, both SHA-1 and MD5 are outdated and broken cryptographic hash functions. Such bugs are not detectable by formal verification, since they occur due to a specification/design issue and not an implementation flaw, therefore they are out of the scope of automated approaches and formal verification methods. The cryptographic key is

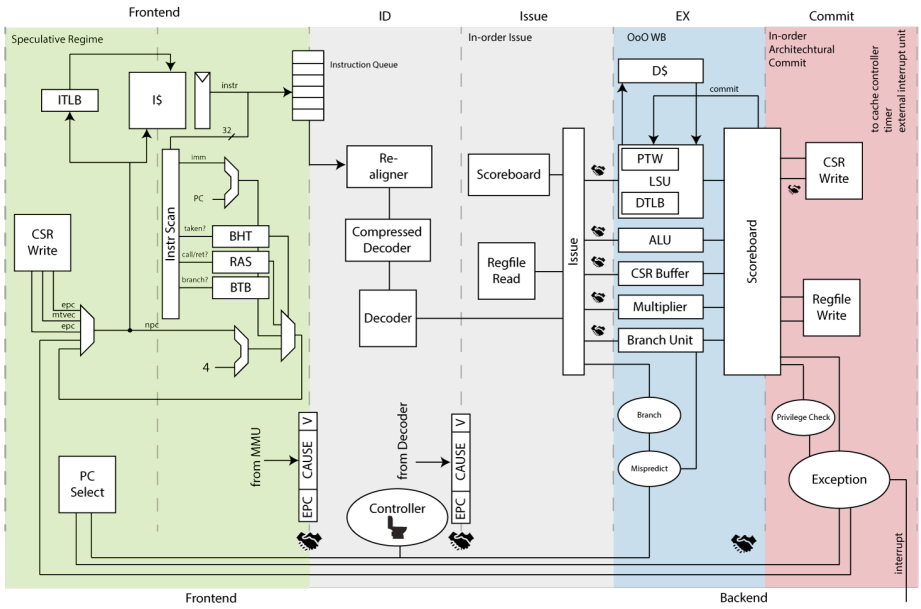


FIGURE 4: High-level architecture of the Ariane core [59].

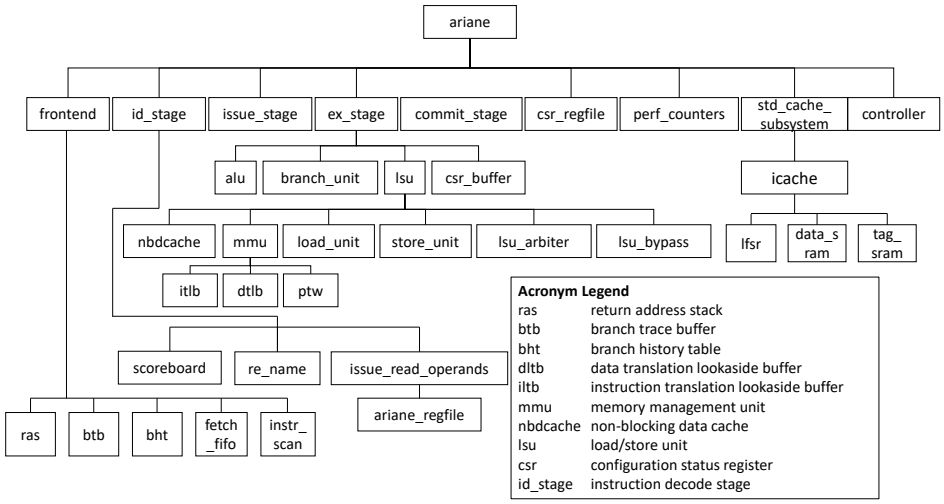


FIGURE 5: Illustration of the RTL module hierarchy of the Ariane core.

Attack	Privilege Level	Memory Corruption	Information Leakage	Cross-modular	HW/FW-Interaction	Cache-State Gap	Timing-Flow Gap	**HardFail**
Cachebleed [74]	unprivileged	✗	✓	✗	✗	✗	✓	✓
TLBleed [23]	unprivileged	✗	✓	✓	✗	✓	✓	✓
BranchScope [20]	unprivileged	✗	✓	✗	✗	✓	✗	✓
Spectre [37]	unprivileged	✗	✓	✓	✗	✓	✗	✓
Meltdown [43]	unprivileged	✗	✓	✓	✗	✓	✗	✓
MemJam [45]	supervisor	✗	✓	✓	✗	✗	✓	✓
CLKScrew [69]	supervisor	✓	✓	✗	✓	✗	✓	✓
Foreshadow [72]	supervisor	✓	✓	✓	✓	✓	✗	✓

TABLE 2: Classification of the underlying vulnerabilities of recent microarchitectural attacks by their HardFail properties.

stored and read from unprotected memory, allowing an attacker access to the key. The temperature sensor register value is incorrectly muxed as output instead of the crypto engine output and vice versa, which are illegal information flows that could compromise the cryptographic operations.

LISTING 1: **Incorrect use of crypto RTL:** The key input for the AES (g_input) is connected to signal b. This signal is then passed through various modules until it connects directly to a tightly coupled memory in the processor.

```
input logic [127:0] b,
...
aes_1cc aes(
  .clk(0),
  .rst(1),
  .g_input(b),
  .e_input(a),
  .o(aes_out)
  );
```

Bugs in security modes: We replaced the standard PULP_SECURE parameter in the riscv_cs_registers and riscv_int_controller modules with another constant parameter to permanently disable the security/privilege checks for these two modules. Another bug we inserted is switching the write and read protections for the AXI bus interface, causing erroneous checks for read and write accesses.

Bugs in the JTAG module: We implemented a JTAG password-checker and injected multiple bugs in it, including the password being hardcoded in the password checking file. The password checker also only checks the first 31 bits, which reduces the computational complexity of brute-forcing the password. The password checker does not reset the state of the correctness of the password when an incorrect bit is detected, allowing for repeated partial checks of passwords to end up unlocking the password checker. This is also facilitated by the fact that the index overflows after the user hits bit 31, allowing for an infinite cycling of bit checks.

D Exploiting Hardware Bugs From Software

We now explain how one of our hardware bugs can be exploited in real-world by software. This RTL vulnerability manifests in the following way. When an error signal is generated on the memory bus while the underlining logic is still handling an outstanding transaction, the next signal to be handled will instead be considered operational by the module unconditionally. This lets erroneous memory accesses slip through hardware checks at runtime. Armed with the knowledge about this vulnerability, an adversary can force memory access errors to evade the checks. As shown in Figure 6, the memory bus decoder unit (unit of the memory interconnect) is assumed to have the bug. This causes errors to be ignored

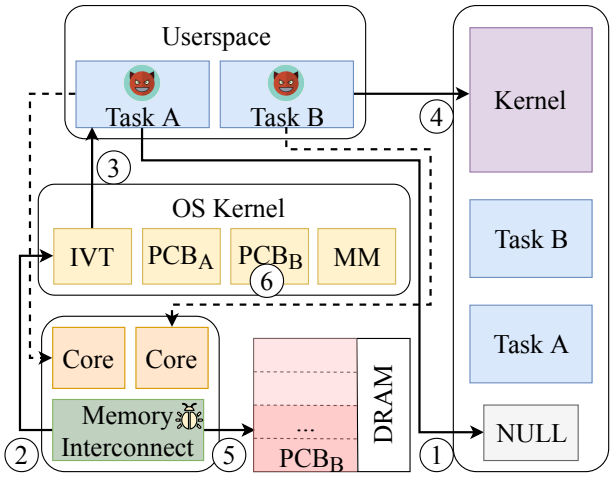


FIGURE 6: Our attack exploits a bug in the implementation of the memory bus of the PULPissimo SoC: by ① *spamming* the bus with invalid transactions an adversary can make ④ malicious write requests be set to `operational`.

under certain conditions (see bug number #7 in Table 1). In the first step ①, the attacker generates a user program (Task A) that registers a dummy signal handler for the segmentation fault (`SIGSEGV`) access violation. Task A then executes a loop with ② a faulting memory access to an invalid memory address (e.g., `LW x5, 0x0`). This will generate an error in the memory subsystem of the processor and issue an invalid memory access interrupt (i.e., `0x0000008C`) to the processor. The processor raises this interrupt to the running software (in this case the OS), using the pre-configured interrupt handler routines in software. The interrupt handler in the OS will then forward this as a signal to the faulting task ③, which keeps looping and continuously generating invalid accesses. Meanwhile, the attacker launches a separate Task B, which will then issue a single memory access ④ to a privileged memory location (e.g., `LW x6, 0xf77c3000`). In this situation, multiple outstanding memory transactions will be generated on the memory bus, all of which but one will be flagged as faulty by the address decoder. An invalid memory access will always proceed the single access of Task B. Due to the bug in the memory bus address decoder, ⑤ the malicious memory access will become `operational` instead of triggering an error. Thus, the attacker can issue read and write instructions to arbitrary privileged (and unprivileged) memory by forcing the malicious illegal access to be preceded with a faulty access. Using this technique the attacker can eventually leverage this read-write primitive, e.g., ⑥ to escalate privileges by writing the process control block (PCB_B) for his task to elevate the corresponding process to root. This bug leaves the attacker with access to a root process, gaining control over the entire platform and potentially compromising all the processes running on the system.

*u*XOM: Efficient eXecute-Only Memory on ARM Cortex-M

Donghyun Kwon[1,2] Jangseop Shin[1,2] Giyeol Kim[1,2]
Byoungyoung Lee[1,3] Yeongpil Cho[4] Yunheung Paek[1,2]

[1]*ECE, Seoul National University,* [2]*ISRC, Seoul National University*
[3]*Computer Science, Purdue University,* [4]*School of Software, Soongsil University*

{dhkwon, jsshin, gykim}@sor.snu.ac.kr,
{byoungyoung, ypaek}@snu.ac.kr, ypcho@ssu.ac.kr

Abstract

Code disclosure attacks are one of the major threats to a computer system, considering that code often contains security sensitive information, such as intellectual properties (e.g., secret algorithm), sensitive data (e.g., cryptographic keys) and the gadgets for launching code reuse attacks. To stymie this class of attacks, security researchers have devised a strong memory protection mechanism, called eXecute-Only-Memory (XOM), that defines special memory regions where instruction execution is permitted but data reads and writes are prohibited. Reflecting the value of XOM, many recent high-end processors have added support for XOM in their hardware. Unfortunately, however, low-end embedded processors have yet to provide hardware support for XOM.

In this paper, we propose a novel technique, named *u*XOM, that realizes XOM in a way that is secure and highly optimized to work on Cortex-M, which is a prominent processor series used in low-end embedded devices. *u*XOM achieves its security and efficiency by using special architectural features in Cortex-M: *unprivileged memory instructions* and an *MPU*. We present several challenges in making XOM non-bypassable under strong attackers and introduce our code analysis and instrumentation to solve these challenges. Our evaluation reveals that *u*XOM successfully realizes XOM in Cortex-M processor with much better efficiency in terms of execution time, code size and energy consumption compared to a software-only XOM implementation for Cortex-M.

1 Introduction

When it comes to the security of a computing system, the protection of the code running on the system should be of top priority because the code defines security critical behaviors of the system. For instance, if attackers are able to modify existing code or inject new code, they may place the victim system under their control. Fortunately, code injection attacks nowadays can be mitigated by simply enforcing the well-known security policy, W⊕X. Since virtually all processors today are equipped with at least five basic memory permissions: read-write-execute (RWX), read-write (RW), read-execute (RX), read-only (RO) and no-access (NA), W⊕X can be efficiently enforced in hardware for a memory region solely by disabling RWX.

However, even if attackers are not able to modify the system's code, the system can still be threatened by *disclosure attacks* that attempt to read part of or possibly the entire code. Because code often contains intellectual properties (IPs) including core algorithms and sensitive data like cryptographic keys, disclosure attacks severely damage the security of victim systems by exposing critical information to unauthorized users. Even worse, disclosure attacks can be abused by attackers to launch *code reuse attacks* (CRAs), which allow the attacker to perform adversarial behaviors without modifying its code contents. It has been shown that attackers who can see the instructions in the code may launch a CRA wherein they craft a malicious code sequence by chaining the existing code snippets scattered around the program binary [34].

In order to prevent disclosure attacks, *eXecute-Only-Memory* (XOM) has been a core security mechanism of various countermeasure techniques [6–8, 13, 16, 17, 31, 37]. XOM is a strong memory protection mechanism that defines a special memory region where only instruction executions are allowed, and any attempts for instruction reads or writes are prohibited. Thus, as long as sensitive information such as IPs and the code contents are stored inside the region protected by XOM, developers are in principle able to prevent direct exposure of the code content as well as the code layout. This simple but tangible security benefit of XOM has led several researchers to propose hardware-assisted XOM on various architectures. For example, some have proposed an architecture that implements XOM by encrypting executable memory and decrypting instructions only when they are loaded [24]. However, since their approach mostly imposes significant changes and overhead on the underlying hardware, it cannot

Donghyun Kwon has been affiliated with Electronics and Telecommunications Research Institute (ETRI) since March 2019.

Corresponding authors are Yeongpil Cho and Yunheung Paek.

be adopted readily by the processor vendors for their existing products. Instead, many vendors opt for a less drastic approach that simply augments the basic memory permissions with the new execute-only (XO) permission [8, 10].

As of today, many high-end processors provide XOM capabilities by supporting augmented memory permissions. Consequently, by taking benefits from the hardware support for XOM, low-cost security solutions have been built to mitigate real attacks [8, 10, 13, 16]. However, these security benefits are confined to computing systems for general applications since the XO permission is only available in relatively high-end processors targeting general-purpose machines such as servers, desktops and smartphones. More specifically, applications running on tiny embedded devices cannot enjoy such benefits because only the basic memory permissions (not XOM) are supported in their target processors, which are primarily intended for use in low-cost, low-power computations. As one example of such processors that hardware-level XOM is not built into, we have the ARM *Cortex-M* series, which are prominent processors adopted by numerous low-cost computing devices today [38].

Fortunately, researchers have demonstrated that software fault isolation (SFI) techniques can be used to thwart these prevalent attacks without hardware-level XOM [7, 31]. They are purely software techniques, and thus are able to cope with any types of processors regardless of the underlying architectures. However, the drawback we observed is that SFI-based XOM techniques perform less optimally on certain types of processors, including Cortex-M in particular. More importantly, such techniques can even be circumvented, leading to critical security issues (refer to § 6.4). Motivated by this observation, this paper proposes a novel technique, called *u*XOM, to realize XOM in a way that is secure and highly optimized to work on Cortex-M processors. Since performance is a pivotal concern of tiny embedded devices such as Cortex-M, efficiency must be the most important objective of any technique targeting these low-end processors. To achieve this objective, *u*XOM leverages a special type of instructions, called *unprivileged loads/stores*, provided by the instruction set architecture for ARM Cortex-M. In an ARM-based system, memory can be divided into two classes of regions according to privilege levels: *non-privileged* and *privileged* memory regions. Unprivileged loads/stores can only access non-privileged memory regions, irrespective to the processor's current privilege level (either in a privileged or non-privileged). On the contrary, ordinary loads/stores are permitted to access privileged regions as long as they are executed under the privileged level. This striking difference between unprivileged and ordinary load/store instructions is the key enabler of our technique.

By capitalizing on this difference, we also need to exploit a unique style of running embedded software on the processors to achieve this ultimate goal of *u*XOM. In computing systems, software entities are typically assigned certain privileges during execution. For instance, user applications run as unprivileged, and the OS kernel as privileged. In practice, however, applications and the kernel in tiny embedded devices are designed to operate with the same privilege level [12, 21]. This is because these embedded systems are typically given real-time constraints, and the privilege mode switching involved in user-kernel privilege isolation is considered very expensive [21]. For the goal of *u*XOM stated above, we utilize these unique architectural characteristics of Cortex-M processors. More specifically, *u*XOM converts all memory instructions into unprivileged ones and sets the code region as privileged. As a result, converted instructions cannot access code regions, thereby effectively enforcing the XO permission onto the code regions. Since the processor is running with privileged level, code execution is still allowed without any permission error.

However, in order to actually realize *u*XOM, we need to tackle the problem that some memory instructions cannot be changed into unprivileged memory instructions. For example, memory instructions accessing critical system resources, such as an interrupt controller, a system timer and a Memory Protection Unit (MPU), should not be converted. Accesses to these resources always require privilege, so the program will crash if instructions accessing these resources are converted to unprivileged ones. In addition, load/store exclusive instructions, which are the special memory instructions for exclusive memory access, do not have unprivileged counterparts. For these instructions, there is no way to implement the intended functionality with unprivileged memory instructions. Therefore, we should analyze the code thoroughly to find these instructions and leave them as the original instructions.

Unfortunately, these unconverted memory instructions can be exploited by attackers to subvert *u*XOM. For example, if the attackers manage to execute these instructions by altering the control flow, they may bypass *u*XOM by (1) turning off the MPU protection or (2) reading the code directly. To prevent such attacks, the unconverted memory instructions need to be instrumented with verification routines to ensure that each memory access using these instructions does not break *u*XOM's protection. However, the attackers can still bypass the verification routines and directly execute the problematic memory instructions. To handle this challenge, we have devised the *atomic verification* technique that virtually enables memory instructions to be executed atomically with the verification routine, thereby preventing potential attackers from executing the memory instructions without passing the verification.

Another important problem *u*XOM needs to handle is that the attackers can alter control flow to execute *unintended* instructions, which may result from unaligned execution of 32-bit Thumb instructions or execution of the data embedded inside the code region [4]. Among the unintended instructions, attackers may find useful instructions for bypassing *u*XOM, such as ordinary memory instructions. To mitigate this attack vector, *u*XOM analyzes the code to find all potentially harm-

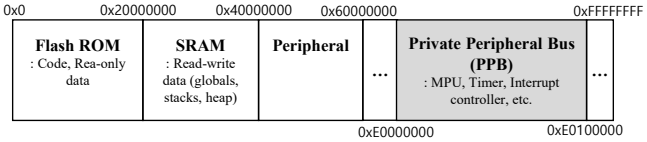


Figure 1: System address map for ARMv7-M [18]

ful unintended instructions and replaces them with alternative instruction sequences that have an equivalent function but do not contain any exploitable unintended instructions.

Built upon LLVM compiler and Radare2 binary analysis framework [32], *u*XOM automatically transforms every software component (i.e., real-time operating systems (RTOSs), the C standard library, and the user application) into a *u*XOM-enabled binary. Currently, *u*XOM supports processors based on ARMv7-M architecture, including Cortex-M3/4/7 processors. To evaluate *u*XOM, we experimented on an Arduino Due board, which ships with a Cortex-M3 processor. Our experiment confirms that *u*XOM works efficiently, empowered with the optimized use of the underlying hardware features. In particular, *u*XOM incurs only 15.7%, 7.3% and 7.5% overhead for code size, execution time and energy, while SFI-based XOM incurs overhead of 50.8%, 22.7%, and 22.3%, respectively. To demonstrate the compatibility of *u*XOM with other XOM-based security solutions, we discuss two use cases of *u*XOM: secret key protection and CRA defense. We implemented and evaluated the second use case, the CRA defense. Even when the CRA defense is applied on top of *u*XOM, it shows only moderate performance overhead, which is 19.3%, 8.6% and 9.7% for code size, execution time and energy, respectively.

The remainder of this paper is organized as follows. § 2 provides the background information. § 3 explains the threat model and assumptions. § 4 and § 5 describe the approach and design of *u*XOM, respectively. § 6 provides experimental results for *u*XOM and its use cases. § 7 presents several discussions regarding *u*XOM, and § 8 explains related works. § 9 concludes the paper.

2 Background

Cortex-M(3/4/7) processors targeted in this paper implement the ARMv7-M architecture, the microcontroller ('M') profile of the ARMv7 architecture, which features low-latency and highly deterministic operation for embedded systems. In this section, we give background information on the key architectural features of ARMv7-M that are required to understand the design and implementation of *u*XOM.

2.1 ARMv7-M Address Map and the Private Peripheral Bus (PPB)

ARMv7-M does not support memory virtualization and the regions for code, data, and other resources are fixed at specific address ranges. Figure 1 shows the system address map for ARMv7-M architecture. The first 0.5 GB (0x0-0x20000000) is the region where the flash ROM is typically mapped. Code and read-only data are placed here. The memory range 0x20000000-0x40000000 is the SRAM region where read-write data (globals, stack, and heap) are placed. Devices only use a small subset of each region; our test platform (SAM3X8E) has 512KB of flash and 96KB of SRAM. The memory range 0x40000000-0x60000000 is where device peripherals, such as GPIO and UART, are mapped. The 1 MB memory region ranging from 0xE0000000 to 0xE00FFFFF is the PPB region. Various system registers for controlling system configuration and monitoring system status, such as the system timer, the interrupt controller and the MPU, are mapped in this region. The PPB differs from the other memory regions of the system in that only privileged memory instructions are allowed to read from or write to the region. Generally, access permissions for memory regions can be configured through the MPU which we describe in detail below. However, the access permission for the PPB is fixed and even the MPU cannot override the default configuration.

2.2 Memory Protection Unit (MPU)

The MPU provides a memory access control functionality for Cortex-M processors. The biggest difference between the MPU and the Memory Management Unit (MMU) equipped in high-end processors is that the MPU does not provide memory virtualization and thus the access control rules are applied on the physical address space. Depending on the setting of the MPU's memory-mapped registers between 0xE000ED90 and 0xE000EDEC, a limited number (typically 8 or 16) of possibly overlapping regions can be set up, each of which is defined by the base address and the region size. Each region defines separate access permissions for privileged and non-privileged access through the combination of eXecute-Never (XN)-bit and Access Permission (AP)-bits. The available permission settings are RWX, RW, RX, RO, and NA, but in any case, unprivileged access is granted the same or more restrictive permission than privileged accesses. For example, when RO permission is given to a privileged access, unprivileged access can only have NA or RO permissions. If two or more regions have overlapping ranges, the access permission for the higher-numbered region takes effect. For access to memory ranges not covered by any region, it can be configured to always generate a fault or to follow the default access permission, which depends on the specific processor implementation. It is important to note that the read permission should be included in order for the memory region to be executable. This is the reason that XOM cannot be implemented simply by configuring the MPU in Cortex-M processors.

2.3 Unprivileged Loads/Stores

The ARMv7-M architecture only supports a thumb instruction set, which is a variable-length instruction set including a mix of traditional 16-bit thumb instructions and 32-bit instructions introduced in Thumb-2 technology. The unprivileged

loads/stores are special types of memory access instructions provided in the instruction set architecture [18]. The main distinction of these instructions is that they always perform memory accesses as if they are executed as unprivileged regardless of the current privilege mode. Thus, memory accesses using these instructions are regulated by the MPU's permission setting for unprivileged accesses. Unprivileged loads/stores are only available in 32-bit encoding and only have immediate-offset addressing mode. They do not support exclusive memory access. They are distinguished by the common suffix 'T' (e.g., LDRT and STRT).

2.4 Exception Entry and Return

An exception is a special event indicating that the system has encountered a specific condition that requires attention. It typically results in a forced transfer of control to a special software routine called an exception handler. On ARMv7-M, the location of the exception handlers corresponding to each exception are specified in the vector table pointed to by the Vector Table Offset Register (VTOR). Note that unlike the other ARMv7 profiles, the ARMv7-M has introduced a hardware mechanism that automatically stores and restores core context data (in particular, Program Status Register (xPSR), return address[1], lr, r12, r3, r2, r1 and r0) on the stack upon exception entry and return. The ARMv7-M profile also exhibits an interesting feature where an exception return occurs when a unique value of EXC_RETURN (e.g., 0xFFFFFFF1) is loaded into the pc via memory load instructions, such as POP, LDM and LDR, or indirect branch instructions, such as BX. Another thing to note about the exception handling in ARMv7-M is that different stack pointer (sp) can be used before and after the exception. ARMv7-M provides two types of sp, called main sp and process sp. The exception handler can only use main sp but the non-handler code can choose which of the two sps to use. The type of stack pointer being currently used is internally managed through CONTROL register, so that stack pointers are always represented as sp in the binary regardless of its actual type.

3 Threat Model and Assumptions

Several conditions must be met to realize *u*XOM. First, the target processor must support the MPU and the unprivileged load/store instructions. We also assume that the target devices run standard bare-metal software in which all included software components, such as applications, libraries, and an OS, share a single address space. Notably, we assume that the entire software executes at a privileged level as mentioned in § 1.

Next, we define the capabilities of an attacker. We assume that attackers are only capable of launching software attacks at runtime. We do not consider offline attacks on firmware images, such as disassembling, manipulating, or replacing the firmware, because we believe that these attacks can be thwarted by orthogonal techniques such as code encryption or signing. We also leave hardware attacks, such as bus probing [9] and memory tampering [22] out of consideration. However, we believe that our attackers are still strong enough to jeopardize the security of the target devices. The bare-metal software installed in the device is considered benign but internally holds software vulnerabilities, so that the attackers may exploit the vulnerabilities and ultimately have arbitrary memory read and write capability. With such a strong memory access capability, attackers can access any memory region including code, stack, heap and even the PPB region for system controls. They can also subvert control flow by manipulating function pointers or return addresses. We do not trust any software components, including the exception handlers. Event-driven nature of tiny embedded systems signifies that exception handlers can take a large portion of embedded software components [14], so we cannot just assume the security of these handlers. Thus, we assume that attackers can trigger a vulnerability inside the exception handler and manipulate any data including the cpu context saved on exception entry.

4 Approach and Challenges

*u*XOM aims to provide XO permission, which enables effective protection against disclosure attacks for code contents, for commodity bare-metal embedded systems based on the Cortex-M processor. *u*XOM tries to minimize the performance penalty by utilizing hardware features, such as unprivileged memory instructions and the MPU provided by Cortex-M processors. Ideally, *u*XOM converts *all* memory instructions into unprivileged ones. It then configures the MPU upon system boot to set code regions to RX for privileged access and NA for unprivileged access. It also sets the other memory regions (i.e., data regions) to non-executable for both privileged and unprivileged accesses. After the configuration, *u*XOM executes code as privileged. All the converted memory instructions (i.e., unprivileged memory instructions) are allowed to access the data regions in the same way as before. However, these instructions are prohibited from accessing the code region and the PPB region in which the MPU and VTOR are located that are essential for the security of *u*XOM (see the blue arrows in Figure 2). This is because these regions are set to the NA memory permission for unprivileged accesses. As all of the memory instructions have been converted to unprivileged ones, code disclosure attacks are effectively thwarted. In addition, *u*XOM by default enforces W⊕X policy that prevents code execution from writable regions. Therefore, any attempt to inject ordinary memory instructions for code disclosure is blocked as well.

Challenges. The basic principle of *u*XOM is simple and intuitive as described above. To realize *u*XOM in practice, however, we have to overcome some challenges to build a system that works for real programs and cannot be bypassed by any means. We summarize the challenges of realizing

[1] the value of the program counter (pc) at the moment of the exception

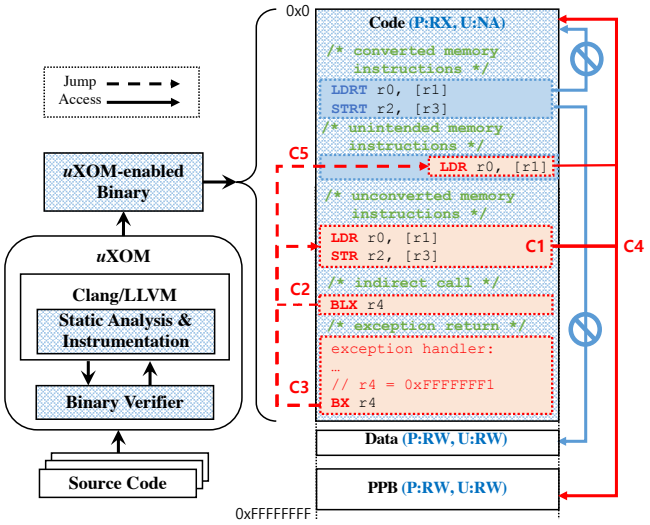


Figure 2: *u*XOM approach

*u*XOM as follows.

- **C1. Unconvertible memory instructions:** To implement *u*XOM, we initially tried to convert all memory instructions into unprivileged ones. However, this naïve attempt will be unsuccessful because unprivileged memory instructions do not support the exclusive memory access that is mainly utilized to implement lock mechanisms, and they cannot access the PPB region to which accesses must be privileged regardless of the MPU configuration. Therefore, we need to thoroughly analyze the entire code, find all these unconvertible instructions, and leave those instructions as the original types. However, these unconverted loads/stores in the program binary resulted in the other challenges, **C2**, **C3** and **C4**.
- **C2. Malicious indirect branches:** In § 3, we assumed that attackers are capable of altering the control flow at runtime by manipulating function pointers or return addresses. Therefore, attackers can deliberately jump to the unconverted loads/stores and exploit them. Unlike unprivileged loads, the unconverted ones can access the code region. Thus, the attackers are now able to read the code without a permission fault. Furthermore, the attackers can also use the unconverted stores to manipulate memory-mapped system registers in the PPB. For example, they can configure the MPU to enable unprivileged access to the code region, completely neutralizing the protection offered by *u*XOM.
- **C3. Malicious exception returns:** This challenge is similar to **C2** in that attackers can hijack control flow and eventually exploit the unconverted loads/stores to thwart *u*XOM. As explained in § 2.4, Cortex-M employs a hardware-based context save and restore mechanism for fast exception entry and return. The problem is that as the context is stored in the stack, attackers can exploit a vulnerability while in the exception handling mode to corrupt any context on the stack. In particular, the context includes a return address that represents the program point at the moment the exception is taken. If the attackers corrupt the return address and then trigger an exception return by assigning `EXC_RETURN` value to the `pc`, they will be able to execute any instruction in the program including the unconverted loads/stores.
- **C4. Malicious data manipulation:** As stated in § 3, the attackers can perform arbitrary memory read/write, and as a result, they have full control over all kind of program data, such as globals, heap objects, and local variables on the stack. With such control, they can exploit the unconverted loads/stores even while following a legitimate control flow. For example, they can call a MPU configuration function with a crafted argument to neutralize *u*XOM by compromising the necessary memory access permissions.
- **C5. Unintended instructions:** An attacker capable of manipulating control flow may be able to compromise *u*XOM by executing unintended instructions that are not found at compile-time. Concretely, Cortex-M processors targeted in this work support Thumb-2 instruction set architecture [18] that intermixes 16-bit and 32-bit width instructions with 16-bit alignment. Therefore, the attackers can execute unintended instructions by jumping into the middle of a 32-bit instruction. The attackers can also execute unintended instructions through immediate values embedded in code, whose bit-patterns can coincidentally be interpreted as a valid instruction.

5 *u*XOM

In this section, we describe the comprehensive details of *u*XOM. We first explain the basic design of *u*XOM for realizing the XO permission (§ 5.1). We then discuss our techniques for overcoming the challenges **C1-C5** (§ 5.2). Next, we present the optimizations applied to reduce performance penalty imposed by *u*XOM (§ 5.3). Lastly, we perform a security analysis to demonstrate that *u*XOM contains no security hazard (§ 5.4).

5.1 Basic Design

Before digging into the design details, we briefly describe how *u*XOM works on the system. As illustrated in Figure 2, *u*XOM is implemented as a compiler pass in the LLVM framework and a binary verifier. During compilation, *u*XOM performs static analyses and code instrumentation to generate a *u*XOM-enabled binary (i.e., firmware). Now, when the binary is flashed on to the board and the system boots, *u*XOM automatically enforces the XO permission on the running code.

5.1.1 Instruction Conversion

As RWX or RX is a mandatory permission for code execution on ARMv7-M, executable code regions are always readable and, as a result, are subject to disclosure attacks. Unfortunately, we cannot omit the read permission to imple-

Case	Original Instruction	Converted Instructions
1	LDR rt, [rn, #imm5]	LDRT rt, [rn, #imm8]
2	LDR rt, [rn, #imm12]	(ADD rx, rn, #imm12) LDRT rt, [rx, (#imm8)]
3	LDR rt, [rn, #-imm8]	SUB rx, rn #imm8 LDRT rt, [rx]
4	LDR rt, [rn, #+/-imm8]! (pre-indexed)	ADD/SUB rx, rn, #imm12 LDRT rt, [rx]
5	LDR rt, [rn], #+/-imm8 (post-indexed)	LDRT rt, [rn] ADD/SUB rx, rn, #imm12
6	LDR rt, [rn, rm]	ADD rx, rn, rm LDRT rt, [rx]

7	LDRD rt, rt2, [rn, #+/-imm8]	(ADD/SUB rx, rn, #imm8)
		LDRT rt, [rx, (#imm8)]
		LDRT rt2, [rx, (#imm8)+4]

Table 1: Basic instruction conversion (only shown for load word instruction)

ment XOM because the read permission is required for the processor to fetch instructions from memory. Therefore, our strategy for XOM is to deprive all memory instructions of the access capability for code regions. Briefly put, we convert the memory instructions into unprivileged ones and set the code regions to be accessible only with a privileged manner.

Converting the type of the memory instruction may seem to be a trivial task, but not all memory instructions can be readily converted as unprivileged. The unprivileged loads/stores only support one addressing mode with a base register and an immediate offset which must be positive and fit within 8 bits. On the other hand, the original memory instructions vary in addressing modes, such as register-offset addressing and pre/post-indexed addressing, which updates the base register. Also, there are unprivileged counterparts to the load/store byte and load/store halfword instructions, but there are no corresponding unprivileged instructions for load/store dual (`LDRD`/`STRD`) and load/store multiple (`LDM`/`STM`), which respectively load/store two or multiple registers. To correctly convert all the memory instructions while preserving the program semantics, we sometimes need extra instructions.

Table 1 summarizes the conversions we apply to different types of load instructions. Cases 3-6 always need an extra `ADD` or `SUB` instruction for calculating the memory address. We omit an extra instruction for other cases if we can fit the immediate in the unprivileged instruction. Note that we may need an extra register for storing the calculated address if `rn` is used again in other instructions. We implement our conversion before the register allocation phase so that we do not have to worry about the physical registers and let the compiler choose the best register for the temporary results. `LDM`/`STM` instructions are not shown in the table because they only appear during an optimization pass after register allocation. Therefore, when the optimization pass tries to create `LDM`/`STM` instructions, we disable the optimization to prevent the generation of those instructions.

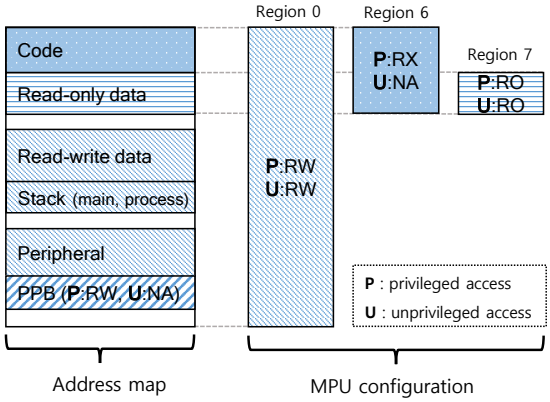


Figure 3: *u*XOM-specific memory permission. Unlabeled regions (white-colored regions) in the address map indicate the unused regions where the memory access generates data abort. The PPB region has a default memory permission (P:RW, U:NA) regardless of the MPU configuration.

5.1.2 Permission Control

In order for the XO permission based on the unprivileged load/store instructions to take effect, *u*XOM has to configure the MPU to enforce certain memory access permissions. Figure 3 shows the default MPU configuration for *u*XOM. Recall that when multiple regions overlap, the permission setting for the higher-numbered region is applied. We create Region 0 covering the entire address space with RW permission for both privileged and unprivileged modes. This is needed to allow unprivileged instructions to access the SRAM and the peripheral region. Otherwise, unprivileged access to those regions is not permitted due to the processor's default permission setting. We assign several higher-numbered regions to *u*XOM protection. (Here, we assumed that the number of MPU regions is 8.) Region 6 covers the entire flash region and assigns RX for privileged accesses and NA for unprivileged accesses. Since flash also contains read-only data, we configure Region 7 to let the unprivileged load instruction access the read-only data. To determine the base and size of this region, we need to know the size of the read-only data. To do this, we first compile, find out the read-only data size and generate an include file that is fed back into the MPU configuration code. The linker script is also modified to take this information and place the read-only data appropriately. The configurations are done in the early stage of the reset handler, which is called upon processor reset. In this way, the *u*XOM-specific permission is activated at the early stage of the system boot before attackers can seize control of the system.

5.2 Solving the Challenges

So far we have explained the basic design of *u*XOM for activating the XO permission. In the following, we describe how *u*XOM addresses the challenges presented in § 4.

Instruction Type	Verification Details
Ordinary stores (STR)	if $\text{Target}_{address}$ points to MPU, Target_{value} must not violate *u*XOM-specific memory permissions. else if $\text{Target}_{address}$ points to VTOR, Target_{value} must have one of the valid VTOR values. else, $\text{Target}_{address}$ must point to the PPB region excluding MPU region and VTOR region
Exclusive stores (STREX)	$\text{Target}_{address}$ must not point to the PPB region.
Ordinary loads (LDR) Exclusive loads (LDREX)	$\text{Target}_{address}$ must not point to the code region.

Table 2: Verification details by the type of unconverted memory instructions. $Target_{address}$ denotes the memory address accessed by load/store instructions and $Target_{value}$ denotes the value to be written by the store instructions.

5.2.1 Finding Unconvertible Memory Instructions

Unprivileged memory instructions do not provide exclusive memory accesses and they cannot access the PPB region. As stated in **C1**, therefore, we need to identify the memory instructions that must not be converted to unprivileged ones and leave them as they are. We simply exclude exclusive memory loads/stores (e.g., LDREX and STREX) from the conversion candidate. We perform compiler analysis to find loads/stores accessing the PPB. Our analysis of the code base reveals that accesses to the PPB involve calculating the base address from a hard coded address pointing to the PPB region. This is consistent with the claims made in previous work [12]. We conduct a similar backward slicing technique to track how the base address of each memory instruction is calculated. If its address is a constant with the value corresponding to the PPB region, or if it is calculated by adding some offset to that constant value, we identify it as an access to the PPB region and leave it as an original form. For our test platform, intra-procedural analysis suffices to identify all PPB accesses. If a PPB address is passed through a function argument and used in a memory access, we can manually identify those particular cases and add annotations to prevent the compiler from converting the memory instructions as done in previous work [12]. Fortunately, most PPB accesses tend to be performed by the hardware abstraction layer (HAL) provided by the device manufacturer, so no significant amount of annotations are required to complement the static analysis.

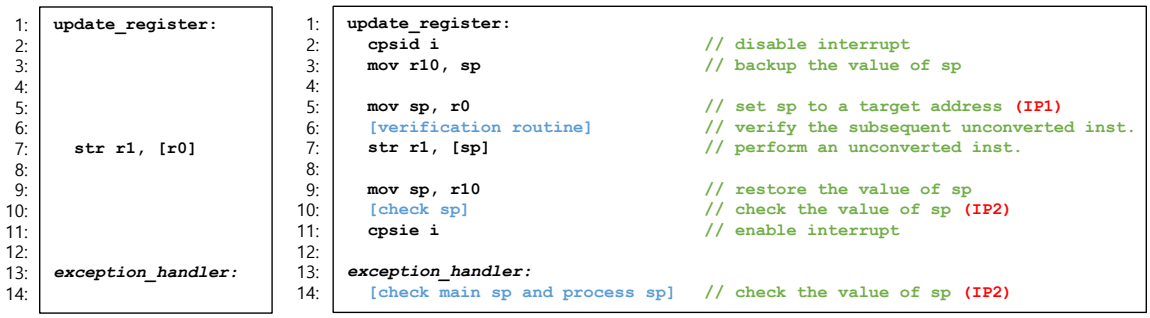
5.2.2 Atomic Verification Technique

Our solution to deal with **C1** is necessary but may endanger the system. The problem is that, as stated in **C2**, **C3** and **C4**, the strong attackers assumed in § 3 can easily exploit the unconverted instructions to neutralize *u*XOM. To address this problem, we devise a *atomic verification* technique inspired by the concept of the reference monitor [15, 35]. The key of our technique is to verify memory accesses by the unconverted loads/stores. More specifically, it inserts a routine that performs verification as described in Table 2 before every unconverted load/store so that we can confirm whether or not the instruction tries to access code regions or manipulate system configuration necessary for *u*XOM, such as *u*XOM-specific memory permission (solve **C4**). At this point, however, the inserted verification may be bypassed by the attackers who can divert control flow. To prevent this, therefore, the technique enforces the atomic execution of the instruction sequence composed of the verification routine and the following untrusted load/store instruction, ensuring that the attackers cannot execute the unconverted loads/stores without a proper verification (solve **C2** and **C3**). Our basic strategy for atomic verification is to (1) allocate a dedicated register as a base register of every unconverted load/store, and then (2) enforce the following two invariant properties regarding the dedicated register.

- **Invariant 1**: The dedicated register must be set to a target address of each unconverted load/store immediately before the associated verification routine. The set value will be maintained only during the execution of the atomic instruction sequence due to **Invariant 2**.
- **Invariant 2**: The dedicated register must hold a non-harmful address (i.e., not a code or the PPB address) when the atomic instruction sequence is not executed.

Now, the accessible memory of the unconverted loads/stores is limited by the value of the dedicated register, which is used as their base register. **Invariant 1** allows the unconverted loads/stores to be executed for their original purpose (e.g., access to the PPB) only through the atomic instruction sequence with a verification. Also, **Invariant 2** prevents any attempt to execute the unconverted loads/stores to access code or the PPB without going through the atomic instruction sequence. As a result, the atomic verification is achieved and the challenges, **C2**, **C3** and **C4**, are addressed successfully. Unfortunately, this implementation strategy decreases the number of available registers by exclusively allocating one register for the PPB access, which may incur additional register spills and occasionally cause a performance drop in some code with a high register pressure.

Therefore, we employ an alternative strategy that is similar to the basic strategy but differs in that it uses the sp as a base register of every ordinary load/store rather than using the dedicated register. Now, we can achieve the atomic verification if we are able to enforce on the sp the same invariant properties as the dedicated register. Enforcing **Invariant 1** is straightforward, but enforcing **Invariant 2** is challenging because it can cause side effects on the program as the sp is used throughout the program, unlike the dedicated register, which is exclusively used only in the atomic instruction sequence. Fortunately, recall that the sp is a special purpose register that should always point to the stack, so **Invariant 2** can be safely enforced without worrying about side effects.



(a) Before (b) After

Figure 4: An unconverted store before and after applying the atomic verification technique. In the update_register functions `r0` and `r1` are used to pass arguments that will be used as unconverted store's base register and source register, respectively.

Enforcing Invariant 2 on `sp`. We achieve this by adopting the idea suggested by the previous work on SFI [7, 33, 39]—we check the value of the `sp` whenever the attackers could have modified it to point to the outside of the valid region (i.e., the stack region). There are three kinds of program points where we need to insert the `sp` check routines: (1) when the `sp` is modified by a non-constant (i.e., register), (2) when the `sp` is increased or decreased by a constant, and (3) at the entry of an exception handler.

We can usually find the first case when the `alloca` function is called, the variable size array is used, or a stack environment stored by the `setjmp` function is restored by the `longjmp` function, which involves an assignment from a general register to the `sp`. As these cases are rare, we insert the `sp` check routines at all the corresponding points. [2]

The second case is very frequently found in the prolog and epilog of a function when the `sp` is adjusted according to the frame size of the function. The attackers could, although not easily, find a suitable gadget consisting of such an instruction and repeatedly execute the gadget until the `sp` is set to a certain value. As pointed out in the previous SFI work [39], if there is a memory instruction based on the `sp` following the `sp` modification, the `sp` can be regulated by placing redzones (i.e., non-accessible memory regions) around the valid stack region. If the redzones are larger than the changes in the value of the `sp`, the following `sp`-based memory instruction ensures that any attempt to use the gadget to jump over the redzones will be detected. Fortunately, the address map illustrated in Figure 3 shows that there already exist large unused regions that can do the role of redzones. This is because in most cases, the stack, code and PPB reside in a separate memory space, such as SRAM, flash memory and system bus, respectively. Therefore, we create redzones only when the stack is created adjacent to the code and PPB without unused regions in between. Note that redzones can detect the corruption of the `sp` only if there is an actual memory access using `sp`. It implies that if, after the `sp` is corrupted, an indirect branch is executed prior to a `sp`-based memory instruction, attackers may be able to evade the execution of the memory instruction by manipulating control flow. Therefore, to ensure the success of this method, we implement an analysis that explores all path from each constant `sp` modification. The analysis checks if there are any `sp`-based memory instructions before a potentially exploitable indirect branch is encountered. According to our experiments, there are some `sp`-based memory instructions preceding indirect branches most of the time. However, we sometimes fail to find any `sp`-based memory instructions or encounter a function call that disables further analysis, and in this case, we insert `sp` check routines because we can no more guarantee the `sp` corruption can be detected by the redzones.

Lastly, the attackers can try to avoid all the checks for `sp` mentioned above by triggering an interrupt right after they corrupt the `sp`. To neutralize this attempt, we have to validate the `sp` by inserting another `sp` check routine at the entry of the exception handlers. Note that as explained in § 2.4, there are two `sp`s in Cortex-M, and different `sp` may be activated before and after the exception, so the `sp` check routine at the entry of the exception handler checks the validity of both `sp`s as shown in Figure 4. The attackers may try to avoid the `sp` check routine by modifying `VTOR` to alter the exception handlers. To avert this attempt, we identify at compile-time the valid values of `VTOR`, and regulate `VTOR` at run-time so that it does not deviate from the identified values, as described in Table 2.

Fulfillment of the Atomic Verification Technique. Now, as both **Invariant 1** and **Invariant 2** can be enforced on the `sp`, we can implement the atomic verification technique using the `sp` without allocating a dedicated register. Figure 4 shows an example code on how the atomic verification technique is applied to harden an unconverted store. The original value of the `sp` is backed up while it is used in the unconverted store instruction (Line 3 and 9). The `sp` is assigned a target address (Line 5) and the verification routine verifies the subsequent unconverted store by checking the validity of its target address and target value (Line 6). If the verification is passed, the unconverted store performs memory access (Line 7). Note that

[2] Currently, *u*XOM can handle only C code, so we manually insert the `sp` check routine for the `longjmp` function written in assembly language.

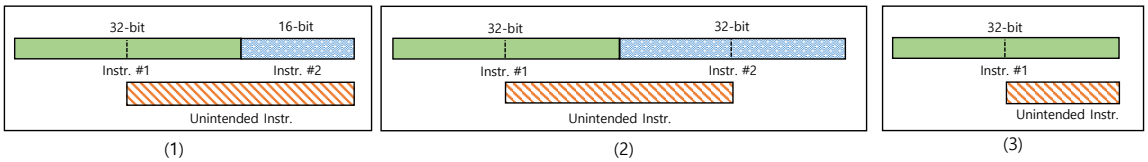


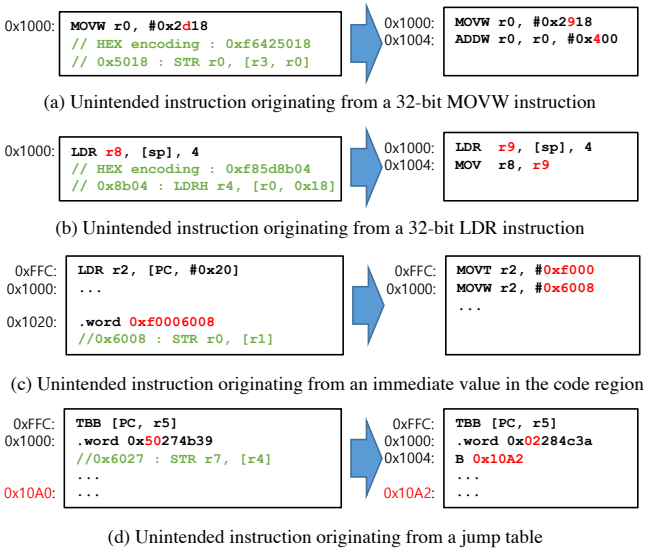
Figure 5: The generation of an unintended instruction by an unaligned execution of a 32-bit instruction.

because **Invariant 2** is enforced by instrumenting `sp`-update instructions and exception handlers (Line 10 and 14), the `sp` always is forced to point to the stack region except when it is used for the unconverted loads/stores. Therefore, to execute the unconverted store for its original purpose (i.e., accessing the PPB), storing the target address (i.e., the address of the PPB) to the `sp` must be preceded (Line 5), which in turn ensures that the verification routine will be performed (Line 6). At the same time, as the `sp` is used for the unconverted loads/stores and may point to out of the stack region, we temporarily disable interrupts (Line 2 and 11), thereby preventing the register from being erroneously checked at the exception handler.

5.2.3 Handling Unintended Instructions

As stated in **C5**, our strong attackers capable of manipulating the control flow of the program can execute unintended instructions to bypass the security of *u*XOM. The unintended instructions are mainly caused by the unique property of Thumb-2 instruction set architecture that intermingle 16-bit and 32-bit instructions. Specifically, as shown in Figure 5, when the attackers deliberately jump into the middle of a 32-bit instruction, unintended 16-bit or 32-bit instructions can be decoded and executed. Unintended instructions can also appear in the immediate values in code memory that match the bit patterns of some valid instructions, as illustrated in Figure 6.(b). As such, a number of unintended instructions are lurking in code. Fortunately, however, only a minority of them that can be interpreted as ordinary memory instructions or `sp`-modifying instructions can actually be exploited to compromise *u*XOM.

Against this problem, we have implemented the code instrumentation technique based on the idea in the previous work [4] that replaces each exploitable unintended instruction into safe instruction sequences that serve the same function as the original instruction. There was one complication in solving the problem that not all exploitable unintended instructions can be identified at compile time. Many of the exploitable unintended instructions result from immediate values (i.e., symbol addresses) in instructions which are not resolved until all the object files are linked by the linker. Simply transforming all those instructions that use unresolved symbol addresses will result in unacceptable overhead in both performance and code size. Thus, it is preferable to implement the transformation inside the linker or use the static binary transformation tool. However, adding extra instructions at this stage is almost impossible because it will require us to



(a) Unintended instruction originating from a 32-bit MOVW instruction

(b) Unintended instruction originating from a 32-bit LDR instruction

(c) Unintended instruction originating from an immediate value in the code region

(d) Unintended instruction originating from a jump table

Figure 6: Examples of unintended instructions and code transformations to remove them.

adjust all the `pc`-relative offsets that are used in many ARM instructions. Adding this capability to current ARM GNU linker implementation will require significant engineering effort.[3] As a work around, we implemented a binary verifier that scans the binary executable for exploitable unintended instructions and records the position of each instruction inside the function. With that information, the program is then recompiled and the exploitable unintended instructions are replaced into alternative instruction sequences. Sometimes, new exploitable unintended instructions are revealed after this process, as code and object layouts are changed and offsets and addresses embedded in the code are changed accordingly. Thus, the interaction between the compiler and the verifier is repeated until there are no exploitable unintended instructions in the binary.

Figure 6 demonstrates a few examples showing how the transformation is applied to remove exploitable unintended instructions. Figure 6.(a) shows the case where an exploitable unintended instruction (`STR`) is generated from the immediate value of 32-bit instruction (`MOVW`). To remove the exploitable instruction, we divide the original immediate value into two

[3]This capability is available in the linker for some architectures like RISC-V which implements aggressive linker relaxation. For those architectures, the `pc`-relative offset resolution is deferred until the linking time to enable linker optimizations that reduce instructions and thus may change the `pc`-relative offsets in the code.

numbers A and B. Then we replace the original 32-bit instruction to use A and add an extra instruction (e.g., ADDW) to add B to the register written by the original instruction. Note that for 32-bit instructions whose immediate value is only determined at link time, we only add the extra instruction at compile time and make sure that the linker puts value A and B instead of the original immediate value. Figure 6.(b) shows another example that the destination register of the 32-bit instruction (LDR) generates the exploitable unintended instruction (LDRH). We solve this case by putting the value loaded from memory into the other register and then use an extra MOV instruction to copy the value into the original destination register. We have also implemented an optimization in the register allocation pass to prefer invulnerable registers over the others for the destination of these 32-bit instructions so that exploitable unintended instructions can be avoided as much as possible. This saves the use of extra instructions and reduces the performance and code size overhead. Figure 6.(c) shows an unintended instruction that exists in a constant embedded in a code region to be loaded by a pc-relative load. To sanitize it, we remove the constant value and replace the associated pc-relative load with two move instructions. If the resulting MOVT or MOVW instruction creates new exploitable unintended instructions, it is further transformed similarly to the example in Figure 6.(a). Finally, Figure 6.(d) shows the case where the offsets in a jump table embedded in the code create an exploitable unintended instruction. In the example, the value 0xA0 (0x50 * 2) is added to pc and the control is transferred to 0x10A0. To remove the unintended instruction in this case, we add a trampoline code right after the jump table for the targets with the problematic offsets.

5.3 Optimizations

According to our experiments (see § 6.1), unprivileged memory instructions consume the same CPU cycles as ordinary memory instructions. However, unprivileged instructions are 32-bits in size while many ordinary memory instructions have a 16-bit form. Also, extra instructions that are added as described in § 5.1.1 can increase both the code size and the performance overhead. Since code size is another critical factor in an embedded application due to its scarce memory, it can be beneficial to leave the memory instructions in their original form if we can ensure that this does not harm the security guarantees of *u*XOM. In fact, a large number of the instructions do not need to be converted either because they are safe by nature or because they can be made safe through some additional effort. For example, ARM supports pc-relative memory instructions which access a memory location that is a fixed distance away from the current pc—i.e., the address of the current instruction. As these instructions can only access certain data embedded in the code region, attackers cannot exploit them to access other memory locations. Therefore, we do not need to convert these instructions, so we leave them as long as it is not exploitable as unintended instructions (§ 5.2.3). We also do not convert stack-based ordinary memory instructions. Numerous instructions use the sp as the base address. Almost all of them are 16-bits in size since Cortex-M provides special 16-bit encoding for stack-based memory instructions. Converting all of these as the unprivileged will significantly add to the code size of the final binary. Most of the LDM/STM instructions, including all the PUSH/POP instructions, are also based on sp. Converting them would require multiple unprivileged instructions which would further increase the code size and even the performance overhead. Luckily, recall that *u*XOM already enforces the invariant properties noted in § 5.2.2 on the sp. Therefore, attackers cannot exploit the ordinary memory instructions based on sp, and we can safely leave sp based memory instructions in their original forms.

5.4 Security Analysis

*u*XOM builds on the premise that there remains no abusable instructions in a firmware binary. *u*XOM satisfies this through its compiler-based static analysis (§ 5.1.1 and § 5.2.3) that (1) identifies all abusable instructions, such as ordinary memory instructions and unintended instructions, and (2) converts them into safe alternative instructions. This conservative analysis does not make false negative conversions, so *u*XOM is fail-safe in terms of security. In the following, we show that attackers we assumed in the threat model (§ 3) will not be able to compromise *u*XOM.

5.4.1 At Boot-up

As noted in § 3, we trust the integrity and confidentiality of the firmware image. The firmware image will be distributed and installed with the *u*XOM-related code instrumentation applied. As soon as the system is powered up, the reset exception handler starts to run and the code snippet that *u*XOM inserted at the start of the handler is executed to enforce *u*XOM-specific memory access permissions. Note that the firmware has started its execution from a known good state and the attackers have not yet injected any malicious payloads. Therefore, we can guarantee that *u*XOM will safely enable XOM without being disturbed by the attackers.

5.4.2 At Runtime

Once *u*XOM enables XOM, the attackers are completely prevented from accessing the code. They cannot use unprivileged loads/stores to bypass *u*XOM, so they have to resort to the unconverted loads/stores. Through the instruction conversions and optimizations of *u*XOM, only three types of unconverted loads/stores remain in the binary: stack-based loads/stores, exclusive loads/stores and ordinary loads/stores for the PPB access.

Stack-based loads/stores. *u*XOM's optimization excludes sp based loads/stores from the conversion candidates. The attackers may be able to execute these loads/stores, but they cannot access the PPB region or code regions. This is because the sp is forced to point to the stack regions due to the invariant property (**Invariant 2** in § 5.2.2) enforced on the

sp.

Exclusive loads/stores and ordinary loads/stores for the PPB access. These unconverted loads/stores are protected by the atomic verification technique. Verification routines are inserted just before each unconverted load/store and the atomic execution of the inserted routine and the corresponding unconverted load/store is guaranteed. Of course, the attacker may jump into the middle of the atomic instruction sequence to directly execute the unconverted load/store without a proper verification. However, as the unconverted loads/stores use the sp as their base register, the attackers still cannot access the code and the PPB regions.

6 Evaluation

*u*XOM transformations are implemented in LLVM 5.0, and *u*XOM's binary verifier is implemented using the Radare2 binary analysis framework [32]. We used the RIOT-OS [5] version 2018.10 as the embedded operating system. As the whole binary, including the OS, runs in a single physical address space at the same privilege level, *u*XOM compiler transformations are applied to the OS code as well as the application code to enable complete protection. We also applied our transformations to the C library (newlib) included in arm-none-eabi toolchain, which had to be patched in a few places to compile and run correctly with LLVM.

To better show the merits of our approach, we also implemented and evaluated SFI-based XOM to compare against *u*XOM. Originally, SFI is developed to sandbox an untrusted module in the same address space. It restricts the store and indirect branch instructions (i.e., by masking or checking the store/branch address) in the untrusted module so that the untrusted module cannot corrupt or jump into the trusted module. It also bundles the checks with the store/branch instructions and prevents jumps into the bundle so that the restrictions applied to the store or branch address cannot be skipped. Capitalizing on the SFI's access control scheme, some studies [7, 31] have implemented the SFI-based XOM that instruments every load instructions with masking instructions to prevent them from reading the code region. However, as these studies focus on high-end devices like smartphones and desktop PCs, we adapted the SFI-based XOM to work on Cortex-M based devices. As our target device do not use virtual memory, code and data must reside in a specific memory region. This prevents us from using simple masking to restrict load addresses and forces us to use a compare instruction to validate the address. Furthermore, the instruction set of Cortex-M requires us to insert additional IT (If-Then) instruction to make load instruction execute conditionally on the comparison result. Next, we place the compare and load inside a 16-byte aligned bundle and make sure that they do not cross the bundle boundary. We insert NOPs in the resulting gaps. Lower bits of indirect branch targets are masked (cleared) to prevent control flows into the bundle. We also make sure that all possible targets of an indirect branch (i.e., functions and call-sites) are aligned.

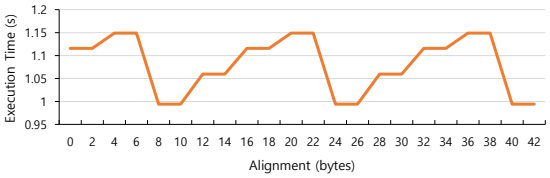


Figure 7: Execution time of bitcount according to the different alignments of the code region.

POP instructions used for function returns are converted to masking and return sequence as described in the previous work on SFI [33]. Following the optimization done in the paper [39], the memory load instructions based on the sp are not checked and the sp is regulated in the same way as in *u*XOM.

To evaluate *u*XOM and the SFI-based XOM, we used the publicly available BEEBs benchmark suite (version 2.1) [29]. We selected 33 benchmarks that are claimed to have relatively long execution time [12][4]. We ran each benchmark on an Arduino Due [1] board which ships with an Atmel SAM3X8E microcontroller based on the Cortex-M3 processor. During the experiment, we found that the program runs give very inconsistent timing results depending on how the code is aligned, even though there are no caches in the processor. After some investigation, we found that the reason is due to the flash memory. The Arduino Due core runs at 84MHz in the default setting, which makes it necessary to wait for 4 cycles (called flash wait state) to get stable results from the flash memory. SAM3X3E chips are equipped with a flash read buffer to accelerate sequential reads [3], which gave us variable results depending on where the branches are located. As a preliminary experiment, we measured the execution time while changing the displacement of the entire code region for bitcount benchmark. As shown in Figure 7, the changes in execution time show a pattern that is repeated every 16-byte, which corresponds to the size of the flash read buffer. Because of this result, to get a consistent result, we decreased the core frequency to 18.5MHz in all our experiments.

6.1 Runtime Overhead

Figure 8 shows the runtime overhead of *u*XOM and SFI-based XOM. The geomean overhead of all benchmarks is 7.3% for *u*XOM and 22.7% for SFI-based XOM. The worst case overhead for *u*XOM is 22.3% for huffbench benchmark and that for SFI-based XOM is 75.1% for edn benchmark. Note that the performance overhead of SFI reported in the previous work [33] for a high-end ARM device (Cortex-A9) is 5%. In the paper, they mention that overhead induced by additional instructions for SFI can be hidden by cache misses and out-of-order execution. Based on this, we presume that the large overhead of SFI-based XOM for Cortex-M3 observed in our experiment is due to the low-power and cache-less

[4] Some of the benchmarks have been dropped in the newest version due to the license problem.

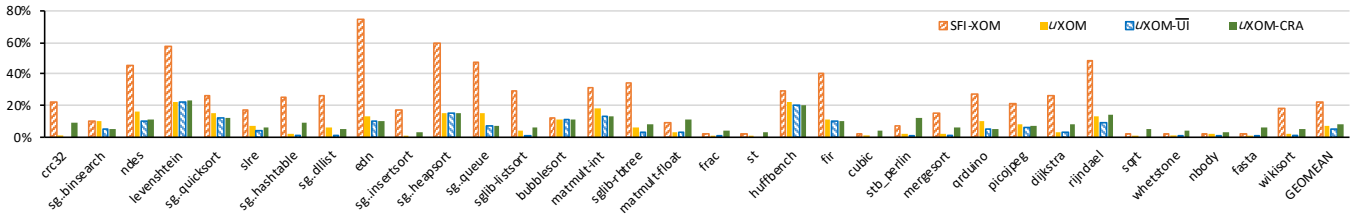


Figure 8: Runtime overhead on BEEBs benchmark suite.

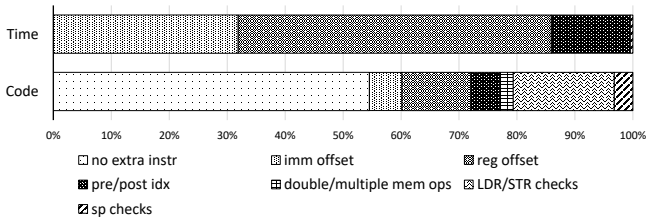


Figure 9: Performance overhead breakdown for the different components of *u*XOM-$\overline{\text{UI}}$ transformation.

Cases	Count (ratio %)
Instruction conversion	
no extra instr.	25932 (77.0)
imm. offset	2547 (7.6)
pre/post idx.	1671 (4.9)
reg. offset	2891 (8.6)
double/multiple mem. ops.	641 (1.9)
`sp` check	
non-const `sp` mod.	18 (0.7)
const `sp` mod. (checked)	769 (28.8)
const `sp` mod. (no check)	1881 (70.5)

Table 3: Statistics for instruction conversion and `sp` check instrumentation.

processor implementation. This strongly shows the need for an efficient low-end device oriented XOM implementation like *u*XOM.

To inspect the sources of overhead, we built and ran multiple partially instrumented versions of binaries with different kinds of transformations applied. First, to examine the performance impact of removing exploitable unintended instructions, we measured the runtime overhead for *u*XOM-$\overline{\text{UI}}$—a variation of *u*XOM that does not handle unintended instructions. As a result, we measured that the geomean overhead for *u*XOM-$\overline{\text{UI}}$ is 5.2%, which shows that removing unintended instructions incurs 2.1% of overhead in *u*XOM. We then gathered the statistics on the number of conversions and check codes inserted in *u*XOM-$\overline{\text{UI}}$ (Table 3). We also measured the overhead ratio in terms of code size and execution time according to the type of conversions and checks (Figure 9). In Table 3 and Figure 9, `no extra instr.` denotes the case where a memory instruction is converted to an unprivileged one without an additional instruction. `imm. offset` denotes the case where an additional instruction is required because the immediate offset is too large or is negative. `pre/post idx.` represents the pre/post-indexed addressing mode and `reg. offset` represents the register-register addressing mode. `double/multiple mem. ops.` represents LDRD/STRD/LDM/STM instructions. For the `sp` check part, `non-const sp mod.` is the case where the `sp` is modified by the non-constant (and the check is required). `const sp mod. (checked)` is the case where the `sp` is modified by the constant and requires checking since no load/store based on the `sp` is found afterwards. `const sp mod. (no check)` is the case where the `sp` is modified by the constant but does not need to be checked. Finally, `LDR/STR checks` denotes the instructions inserted for the atomic verification technique.

The statistics shown in Table 3 are gathered while compiling the C standard library, RIOT-OS, and each of the benchmarks. Note that although the numbers do not represent those executed at runtime, we can expect some correlation between them. Among the converted memory instructions, the majority of the cases is the one where a memory instruction is directly converted to a single unprivileged memory instruction without any extra instruction (`no extra instr.` accounts for 77% of all conversions). This tells us that most of the load/store instructions are using an immediate-offset addressing mode and the offset is usually small so that it fits in the immediate field of the unprivileged instructions. As we can see, instructions converted in this way do not contribute to the runtime overhead albeit being the majority. Even though the unprivileged instructions are 32-bits long, they do not increase the overhead unless additional instructions are inserted. This is a big advantage for *u*XOM, and it is the main reason why *u*XOM can be much more efficient than SFI-based XOM.

As illustrated in Figure 9, the type of instruction conversions that contributes the most of the overhead is the one for the register-register addressing mode (`reg. offset`). Even though they represent only 8.6% of all conversions, they cause 54% of the total overhead for *u*XOM-$\overline{\text{UI}}$. The reason would be that they are frequently used in time-consuming loops, for example, to index array variables. `imm. offset` and `pre/post idx.` take up the other half of the overhead. Memory instructions that load/store multiple registers (`double/multiple mem. ops.`) cause a negligible runtime overhead; they are rare in number and also, although they are converted into multiple unprivileged instructions, the original instruction also takes up extra cycles to load/store multiple registers. The `sp` checks that are inserted for stack modification have an only negligible impact on performance as our

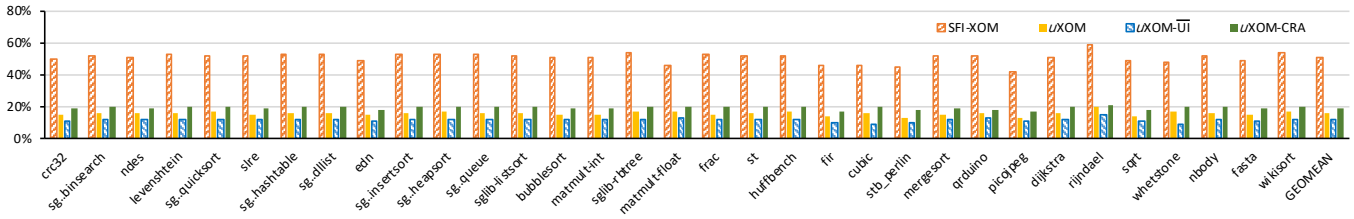


Figure 10: Code size overhead on BEEBs benchmark suite.

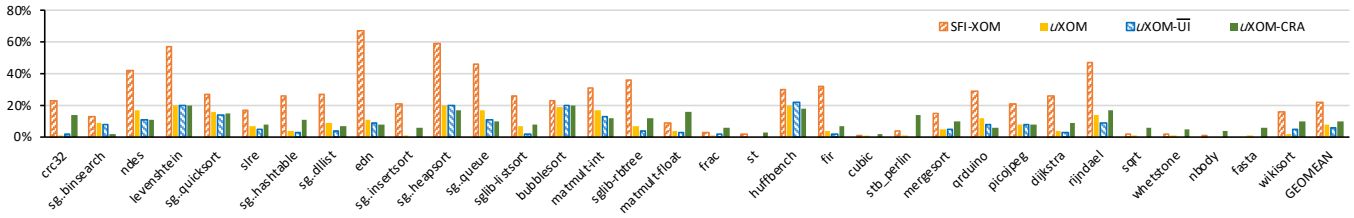


Figure 11: Energy overhead on BEEBs benchmark suite.

analysis finds that the `sp` checks are only needed for less than 30% of `sp`-based memory instructions.

6.2 Code Size Overhead

To see the impact of instruction insertion by *u*XOM, we measured the size of the code in the final binary, excluding the data size. Figure 10 shows the result for both *u*XOM and SFI-based XOM. For *u*XOM, code size is increased by 15.7%, and for SFI-based XOM, it is increased by 50.8%. It shows that *u*XOM can implement XOM with much less code size overhead compared to SFI-based XOM. In addition, we measured that the geomean overhead of *u*XOM-$\overline{\text{UI}}$ is 11.6%, which indicates the amount of increased code for removing unintended instructions is 4.1%. Figure 9 shows the source of the overhead that is caused by instruction conversions and checks. First, `no extra inst.` accounts for 54.5% of the code size overhead for *u*XOM-$\overline{\text{UI}}$, differently from the impact that it had on the runtime performance. This is because the original 16-bit load/store instructions are converted to 32-bit unprivileged instruction, and they are large in number, too. Other types of instructions that need additional instructions also increases the code size to some degree. The instructions added for the atomic verification technique (`ldr/str check`) accounts for 17.4% of the code size overhead for *u*XOM-$\overline{\text{UI}}$. Although there are not many instructions accessing the PPB region, around ten instructions are inserted for each of those points, which adds some overhead to the code size especially since the benchmark code size are only around 30KB. We expect the overhead from the atomic verification to be a smaller percentage in the real program with a larger code base.

6.3 Energy Overhead

Since many embedded devices running on Cortex-M processors often operate based on constrained battery, energy efficiency is one of the important performance factors for these devices. To measure the impact of *u*XOM on energy consumption, we recorded the power while running the individual benchmarks using the ODROID Smart Power [28]. For the convenience of measurement, the benchmarks were repeatedly executed to run for at least 30 seconds. Figure 11 shows the results. For *u*XOM, the geometric mean of all benchmarks is 7.5%, which is slightly larger than 5.8% of *u*XOM-$\overline{\text{UI}}$ but much lower than 22.3% of SFI-based XOM. The results share a similar trend with the execution time since the energy is also affected by the execution time.

6.4 Security and Usability

Other than its excellence for performance, we also need to mention the security and flexibility benefits of *u*XOM over SFI-based XOM. *u*XOM provides a better security guarantee against privileged attackers than SFI-based XOM. SFI-based XOM, including the existing studies, focus only on the code disclosure through memory read instructions, because they assume that W⊕X policy is assured by a Trusted Computing Base (TCB) such as the OS kernel. However, as described in § 3, *u*XOM cannot assume any TCB in the bare-metal environment in which all software components are running with privileges in a single address space. The privileged attacker could neutralize W⊕X by manipulating the MPU configuration register using memory vulnerabilities in the code. To prevent such an attack, SFI-based XOM for Cortex-M would also have to regulate memory write instructions to protect memory-mapped registers for the MPU. However, this would undoubtedly lead to more severe performance overheads, and even worse, SFI-style masking of write instructions would still leave the system vulnerable against attacks through the exception handler (**C3** of § 4). In addition, the current implementation of SFI-based XOM is vulnerable to unintended instructions. To defend this, it should eliminate all exploitable unintended instructions either by using the instruction replacement technique similar to *u*XOM or selectively aligning

32-bit instructions so that jump into the middle of those instructions can be prevented by the masking of indirect jump addresses. Either way, additional performance overhead will be unavoidable.

*u*XOM is also more flexible in placing the code and data. For *u*XOM, the XOM region can be placed anywhere in the address space. For example, *u*XOM can be applied for the code placed in SRAM for performance or firmware updates [20]. Also, *u*XOM can set multiple XOM regions as long as the number of MPU regions supports it. However, SFI-based XOM must place the code at one end and the data on the other to simplify code instrumentation. Moreover, SFI-based XOM needs a guardzone between the code and the data region [39] which further restricts the code and data placement and also causes the memory to be wasted for the guardzone.

6.5 Use Cases

*u*XOM can be used to hide sensitive information in the code region, such as secret keys and code layout. We describe two use cases to illustrate how *u*XOM can be applied to a security solution.

Secret key protection. In tiny devices, secret keys are frequently used for various purposes, such as device authentication and communication channel protection. *u*XOM can protect these keys against arbitrary memory read vulnerabilities by embedding them inside the code. For example, consider the following code that defines the constant global `key`.

```
const unsigned char key[32] =
    {0xcb, 0x21, 0xad, 0x38, ...};
```

The code that reads the first 4-byte of this value is compiled to the assembly code composed of `MOVW` and `MOVT` as follows:

```
MOVW r0, #0x21cb
MOVT r0, #0x38ad
```

Now, if we use *u*XOM to apply the XO permission to this code, attackers cannot access the key value by arbitrary memory reads. As an example, we applied *u*XOM to `rijndael` benchmark, which uses a symmetric key for encryption. By declaring the key as a global constant, we could confirm that the key is embedded in the code protected by *u*XOM. Such a protection offered by *u*XOM can further be combined with in-register computation techniques [26] for a secure computation robust against memory vulnerabilities.

CRA defense. To date, many researchers have proposed code diversification-based CRA defense techniques [7, 12, 13, 30]. They randomize code layout to prevent attackers from using the existing gadgets for CRA. As the code disclosure attack emerged as a serious threat to randomization-based defenses, XOM has been proposed as an effective solution to fortify these defenses.

As another use case of *u*XOM, we implemented a CRA defense solution based on Readactor [13], which is a representative code diversification based CRA defense with resistance to code disclosure attacks. Readactor aims to defend against two classes of code disclosure attacks: direct disclosure where the attackers disclose code layout by directly reading the code and indirect disclosure where attackers indirectly infer the code layout through the value of the code pointers. Readactor first places all code in XOM to prevent the direct disclosure attacks. It then replaces all code pointers with pointers to *trampolines* so that all indirect control transfers must go through the trampoline. In this way, code pointers containing the original code location are never stored in a register or memory, thereby preventing the indirect disclosure attacks. To demonstrate this use case, we implemented function reordering and the trampoline mechanism. Every function call is replaced with a direct branch to the trampoline followed by the call to the original function. When the original function returns, another direct branch takes the control flow back to the original callsite. Also, every function pointer is replaced with a pointer to the corresponding function trampoline. We implemented this use case on top of *u*XOM-$\overline{\text{UI}}$ because the code diversification based CRA defense mitigates control flow hijacking, and consequently hinders an attacker from exploiting unintended instructions. The experimental results of our CRA defense are presented together with the results for *u*XOM, *u*XOM-$\overline{\text{UI}}$ and SFI-based XOM. It imposes average runtime overhead of 8.6%, the code size overhead of 19.3%, and the energy overhead of 9.7%. The runtime overhead is only slightly larger than that for original Readactor implementation (6.4%) which shows the applicability of *u*XOM technique in low-end embedded devices.

7 Discussion

Cortex-M Processors based on ARMv8-M Architecture. ARMv8-M [19] is a recently introduced instruction set architecture for the microcontroller profile. Basically, ARMv8-M provides backward compatibility with ARMv7-M, so *u*XOM is also applicable to ARMv8-M based Cortex-M(23/33/35) processors. Here, we list several possible changes in *u*XOM implementation due to the newly added hardware feature in ARMv8-M. First of all, ARMv8-M includes the stack pointer limit register (`SPLR`) that defines a lower limit for the stack pointer and prevents the stack pointer from pointing below the limit. When enabling `SPLR`, therefore, *u*XOM only needs to ensure that the stack pointer does not point to the PPB region. Secondly, load-acquire and store-release memory instructions are newly added in ARMv8-M. Since these instructions do not have unprivileged counterparts, they should be protected by the atomic verification technique.

False Positive Conversion. When it comes to the instruction conversion of *u*XOM, false positive cases could happen where unconvertible instructions are converted to unprivileged ones. The false positive conversion does not harm the security

aspect of *u*XOM but may cause an unexpected system fault. For instance, if PPB-accessing memory instructions are converted to unprivileged ones, it would not expose the PPB to attackers but raise a memory access fault when executed. To avoid an unexpected system halt due to the fault, uXOM can install a custom fault handler, which in turn may invoke the fail-safe handler already implemented in the existing system (e.g., emergency landing in drones).

Dynamic Data Protection. Although the current *u*XOM implementation aims to defeat the code disclosure attacks, it may be extended to provide protection for the dynamic data as well. To be concrete, *u*XOM can be expanded to implement a data isolation scheme [23, 35, 36] that minimizes the possibility of exposures of critical data by only allowing access through authorized instructions. More specifically, we may allow only authorized instructions (i.e., ordinary loads/stores that are not converted into unprivileged types) to access critical data (e.g., return addresses/session keys) by placing the data on a certain memory region marked as "privileged". To implement such an extension, some modifications to *u*XOM are required. First of all, authorized instructions should be predetermined through the help of programmers or compilers and prevented from being converted to unprivileged ones. Since attackers can exploit these data-accessing instructions to compromise *u*XOM, usage of these instructions should be regulated in a way similar to PPB-accessing instructions through the atomic verification technique with a new verification routine that confines memory access target to the memory region of the critical data.

8 Related Work

Hardware-assisted Execute Only Memory. Due to the compelling security guarantee provided by XOM, today's high-end processor architectures (e.g., x64 and AArch64) provide the XO permission setting in the MMU [8, 10]. Apart from that, various works have attempted to implement XOM in the system with the help of the hardware. David et al. [24] implemented XOM by encrypting the code in memory and decrypting it only when it is executed. However, since it requires significant processor redesign, it is not suitable for wide adoption. In subsequent works, XOM has been implemented by capitalizing on the built-in hardware features. Shadow Walker [37] and HideM [17] presented an implementation of XOM using the split translation lookaside buffer (TLB) architecture, which separates the TLB for instruction fetches and data accesses. They configure the two TLBs so that the same virtual address is translated into different physical addresses for data access and instruction fetch, preventing the data accesses to the code region. XOM-switch [25] implemented XOM using Intel Memory Protection Keys (MPK), which can be used to set memory pages execute-only. Shadow Walker, HideM and XOM-switch are not applicable to Cortex-M based devices because they rely on specific hardware features (i.e., split-TLB or Intel MPK) that do not exist in the Cortex-M processor.

Software-based Execute Only Memory. On the other hand, there have been attempts to emulate XOM in software for processors that do not have the above hardware supports. XnR [6] sets all code pages as non-accessible except for the currently executed code pages called *sliding window* and detects illegal memory reads and writes for non-accessible pages by augmenting the MMU page fault handler. For Cortex-M/R processors, since MPU also provides non-accessible permission setting for memory regions, XOM can be implemented in a similar way. However, this approach cannot detect memory reads for code pages in the sliding window, and also, the performance overhead becomes larger as the sliding window size is reduced. LR^2 [7] and kR^X [31] realize XOM by SFI-inspired techniques [39, 40]. They prevent code reads by masking load instructions, instead of stores as done in the SFI technique. As shown in our evaluation, however, such SFI-based XOM implementation can be bypassed and is inefficient in low-end devices.

Security Solutions using XOM. Many researchers have proposed various security solutions based on XOM. Early works [27] proposed XOM for the purpose of protecting intellectual properties and preventing tampering or leakage of sensitive information stored in the code. Since the advent of code disclosure attacks (i.e., JIT-ROP), a number of works [7, 13, 16, 31] have utilized XOM to prevent the attackers from reading code to learn code layout and launch CRAs. In § 6.5, we have shown that these solutions can be implemented with *u*XOM.

Security for Tiny Embedded Devices. Recently, much research has been done on enhancing the security of tiny embedded devices. Mbed uvisor [2], MINION [21], uSFI [4] and ACES [11] proposed memory isolation techniques for software modules based on MPU. At compile time, they define memory views (stack, heap, and peripherals) for each of the software modules, and at runtime, MPU enforces one of the memory views according to the active software module. Epoxy [12] and AVRAND [30] developed diversification based security solutions for tiny embedded devices. As with these solutions, *u*XOM also seeks to enhance the security of tiny embedded devices. *u*XOM is the first to implement efficient execute-only memory in Cortex-M processors.

9 Conclusion

XOM is a prominent protection mechanism that can be used in various security purposes such as intellectual property protection and CRA defense. However, for a low-end embedded processor such as Cortex-M, there has been no efficient way to implement XOM. In this paper, we present *u*XOM, a novel technique to realize XOM in a way that is secure and highly optimized to work on Cortex-M processors. *u*XOM achieves this by leveraging hardware features (i.e., unpriv-

ileged load/store instructions and MPU) in Cortex-M processors. Our evaluation shows that not only *u*XOM is more efficient than SFI-based XOM in terms of execution time, code size and energy consumption, and that *u*XOM is compatible with existing XOM-based security solutions.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Vasileios P. Kemerlis, for their valuable comments that helped to improve our paper. This work was partly supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (NRF-2017R1A2A1A17069478, NRF-2018R1D1A1B07049870, No. 2019R1C1C1006095), Institute of Information Communications Technology Planning Evaluation (IITP) grant funded by Korea government (Ministry of Science and ICT) (No. 2016-0-00078, No.2018-0-00230, No. 2017-0-00168), and the Brain Korea 21 Plus Project in 2019. The ICT at Seoul National University provides research facilities for this study.

References

[1] Arduino. arduino-due. https://store.arduino.cc/usa/arduino-due.

[2] ARM. The mebed os uvisor. https://www.mbed.com/en/technologies/security/uvisor/.

[3] Atmel. Atmel-11057c-atarm-sam3x-sam3a-datasheet, 2015.

[4] Zelalem Birhanu Aweke and Todd Austin. usfi: Ultra-lightweight software fault isolation for iot-class devices. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1015–1020. IEEE, 2018.

[5] Emmanuel Baccelli, Oliver Hahm, Mesut Gunes, Matthias Wahlisch, and Thomas C Schmidt. Riot os: Towards an os for the internet of things. In *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on*, pages 79–80. IEEE, 2013.

[6] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1342–1353. ACM, 2014.

[7] Kjell Braden, Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Stephen Crane, Michael Franz, and Per Larsen. Leakage-resilient layout randomization for mobile devices. In *NDSS*, 2016.

[8] Scott Brookes, Robert Denz, Martin Osterloh, and Stephen Taylor. Exoshim: Preventing memory disclosure using execute-only kernel code. In *Proceedings of the 11th International Conference on Cyber Warfare and Security*, pages 56–66, 2016.

[9] Xi Chen, Robert P Dick, and Alok Choudhary. Operating system controlled processor-memory bus encryption. In *Design, Automation and Test in Europe, 2008. DATE'08*, pages 1154–1159. IEEE, 2008.

[10] Yaohui Chen, Dongli Zhang, Ruowen Wang, Rui Qiao, Ahmed M Azab, Long Lu, Hayawardh Vijayakumar, and Wenbo Shen. Norax: Enabling execute-only memory for cots binaries on aarch64. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 304–319. IEEE, 2017.

[11] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. Aces: Automatic compartments for embedded systems. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 65–82, 2018.

[12] Abraham A Clements, Naif Saleh Almakhdhub, Khaled S Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. Protecting bare-metal embedded systems with privilege overlays. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 289–303. IEEE, 2017.

[13] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical code randomization resilient to memory disclosure. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 763–780. IEEE, 2015.

[14] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security Symposium*, pages 463–478, 2013.

[15] Úlfar Erlingsson. The inlined reference monitor approach to security policy enforcement. Technical report, Cornell University, 2003.

[16] Jason Gionta, William Enck, and Per Larsen. Preventing kernel code-reuse attacks through disclosure resistant code diversification. In *Communications and Network Security (CNS), 2016 IEEE Conference on*, pages 189–197. IEEE, 2016.

[17] Jason Gionta, William Enck, and Peng Ning. Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 325–336. ACM, 2015.

[18] ARM Holdings. Armv7-m architecture reference manual, 2010.

[19] ARM Holdings. Armv8-m architecture reference manual, 2017.

[20] IAR. Execute in ram after copying from flash or rom. https://www.iar.com/support/tech-notes/general/execute-in-ram-after-copying-from-flashrom-v5.20-and-later/.

[21] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. Securing real-time microcontroller systems through customized memory view switching. In *Network and Distributed Systems Security Symp.(NDSS)*, 2018.

[22] Oliver Kömmerling and Markus G Kuhn. Design principles for tamper-resistant smartcard processors. *Smartcard*, 99:9–20, 1999.

[23] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 147–163, 2014.

[24] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. *ACM SIGPLAN Notices*, 35(11):168–177, 2000.

[25] Ravi Sahita Mingwei Zhang and Daiping Liu. executable-only-memory-switch (xom-switch). *Black Hat Asia*, 2018.

[26] Tilo Müller, Felix C Freiling, and Andreas Dewald. Tresor runs encryption securely outside ram. In *USENIX Security Symposium*, volume 17, 2011.

[27] Gleb Naumovich and Nasir Memon. Preventing piracy, reverse engineering, and tampering. *Computer*, 36(7):64–71, 2003.

[28] ODROID. smart-power. https://wiki.odroid.com/old_product/accessory/odroidsmartpower.

[29] James Pallister, Simon Hollis, and Jeremy Bennett. Beebs: Open benchmarks for energy measurements on embedded platforms. *arXiv preprint arXiv:1308.5174*, 2013.

[30] Sergio Pastrana, Juan Tapiador, Guillermo Suarez-Tangil, and Pedro Peris-López. Avrand: a software-based defense against code reuse attacks for avr embedded devices. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 58–77. Springer, 2016.

[31] Marios Pomonis, Theofilos Petsios, Angelos D Keromytis, Michalis Polychronakis, and Vasileios P Kemerlis. krˆx: Comprehensive kernel protection against just-in-time code reuse. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 420–436. ACM, 2017.

[32] Radare2. unix-like reverse engineering framework and commandline tools. https://www.radare.org/r/.

[33] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary cpu architectures. In *USENIX Security Symposium*, pages 1–12, 2010.

[34] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 574–588. IEEE, 2013.

[35] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. Enforcing kernel security invariants with data flow integrity. In *NDSS*, 2016.

[36] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. Hdfi: Hardware-assisted data-flow isolation. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 1–17. IEEE, 2016.

[37] Sherri Sparks and Jamie Butler. Shadow walker: Raising the bar for rootkit detection. *Black Hat Japan*, 11(63):504–533, 2005.

[38] Menasveta Tim, Soubra Diya, and Yiu Joseph. Introducing arm cortex-m23 and cortex-m33 processors with trustzone for armv8-m, 2016.

[39] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, 1994.

[40] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*, pages 79–93. IEEE, 2009.

A Systematic Evaluation of Transient Execution Attacks and Defenses

Claudio Canella[1], Jo Van Bulck[2], Michael Schwarz[1], Moritz Lipp[1],
Benjamin von Berg[1], Philipp Ortner[1], Frank Piessens[2], Dmitry Evtyushkin[3], Daniel Gruss[1]
[1] *Graz University of Technology,* [2] *imec-DistriNet, KU Leuven,* [3] *College of William and Mary*

Abstract

Research on *transient execution* attacks including Spectre and Meltdown showed that exception or branch misprediction events might leave secret-dependent traces in the CPU's microarchitectural state. This observation led to a proliferation of new Spectre and Meltdown attack variants and even more ad-hoc defenses (e.g., microcode and software patches). Both the industry and academia are now focusing on finding effective defenses for known issues. However, we only have limited insight on residual attack surface and the completeness of the proposed defenses.

In this paper, we present a systematization of transient execution attacks. Our systematization uncovers 6 (new) transient execution attacks that have been overlooked and not been investigated so far: 2 new exploitable Meltdown effects: Meltdown-PK (Protection Key Bypass) on Intel, and Meltdown-BND (Bounds Check Bypass) on Intel and AMD; and 4 new Spectre mistraining strategies. We evaluate the attacks in our classification tree through proof-of-concept implementations on 3 major CPU vendors (Intel, AMD, ARM). Our systematization yields a more complete picture of the attack surface and allows for a more systematic evaluation of defenses. Through this systematic evaluation, we discover that most defenses, including deployed ones, cannot fully mitigate all attack variants.

1 Introduction

CPU performance over the last decades was continuously improved by shrinking processing technology and increasing clock frequencies, but physical limitations are already hindering this approach. To still increase the performance, vendors shifted the focus to increasing the number of cores and optimizing the instruction pipeline. Modern CPU pipelines are massively parallelized allowing hardware logic in prior pipeline stages to perform operations for subsequent instructions ahead of time or even out-of-order. Intuitively, pipelines may stall when operations have a dependency on a previous instruction which has not been executed (and retired) yet. Hence, to keep the pipeline full at all times, it is essential to predict the control flow, data dependencies, and possibly even the actual data. Modern CPUs, therefore, rely on intricate microarchitectural optimizations to predict and sometimes even re-order the instruction stream. Crucially, however, as these predictions may turn out to be wrong, pipeline flushes may be necessary, and instruction results should always be committed according to the intended in-order instruction stream. Pipeline flushes may occur even without prediction mechanisms, as on modern CPUs virtually any instruction can raise a fault (e.g., page fault or general protection fault), requiring a roll-back of all operations following the faulting instruction. With prediction mechanisms, there are more situations when partial pipeline flushes are necessary, namely on every misprediction. The pipeline flush discards any architectural effects of pending instructions, ensuring functional correctness. Hence, the instructions are executed *transiently* (first they are, and then they vanish), *i.e.*, we call this *transient execution* [50, 56, 85].

While the architectural effects and results of transient instructions are discarded, microarchitectural side effects remain beyond the transient execution. This is the foundation of Spectre [50], Meltdown [56], and Foreshadow [85]. These attacks exploit transient execution to encode secrets through microarchitectural side effects (e.g., cache state) that can later be recovered by an attacker at the architectural level. The field of transient execution attacks emerged suddenly and proliferated, leading to a situation where people are not aware of all variants and their implications. This is apparent from the confusing naming scheme that already led to an arguably wrong classification of at least one attack [48]. Even more important, this confusion leads to misconceptions and wrong assumptions for defenses. Many defenses focus exclusively on hindering exploitation of a specific covert channel, instead of addressing the microarchitectural root cause of the leakage [45,47,50,91]. Other defenses rely on recent CPU features that have not yet been evaluated from a transient security perspective [84]. We also debunk implicit assumptions including that AMD or the latest Intel CPUs are completely immune to

Meltdown-type effects, or that serializing instructions mitigate Spectre Variant 1 on any CPU.

In this paper, we present a systematization of transient execution attacks, *i.e.*, Spectre, Meltdown, Foreshadow, and related attacks. Using our decision tree, transient execution attacks are accurately classified through an unambiguous naming scheme (cf. Figure 1). The hierarchical and extensible nature of our taxonomy allows to easily identify residual attack surface, leading to 6 previously overlooked transient execution attacks (Spectre and Meltdown variants) first described in this work. Two of the attacks are Meltdown-BND, exploiting a Meltdown-type effect on the x86 `bound` instruction on Intel and AMD, and Meltdown-PK, exploiting a Meltdown-type effect on memory protection keys on Intel. The other 4 attacks are previously overlooked mistraining strategies for Spectre-PHT and Spectre-BTB attacks. We demonstrate the attacks in our classification tree through practical proofs-of-concept with vulnerable code patterns evaluated on CPUs of Intel, ARM, and AMD.[1]

Next, we provide a classification of gadgets and their prevalence in real-world software based on an anaylsis of the Linux kernel. We also give a short overview on current tools for automatic gadget detection.

We then provide a systematization of the state-of-the-art defenses. Based on this, we systematically evaluate defenses with practical experiments and theoretical arguments to show which work and which do not or cannot suffice. This systematic evaluation revealed that we can still mount transient execution attacks that are supposed to be mitigated by rolled out patches. Finally, we discuss how defenses can be designed to mitigate entire types of transient execution attacks.

Contributions. The contributions of this work are:

1. We systematize Spectre- and Meltdown-type attacks, advancing attack surface understanding, highlighting misclassifications, and revealing new attacks.
2. We provide a clear distinction between Meltdown/Spectre, required for designing effective countermeasures.
3. We provide a classification of gadgets and discuss their prevalence in real-world software.
4. We categorize defenses and show that most, including deployed ones, cannot fully mitigate all attack variants.
5. We describe new branch mistraining strategies, highlighting the difficulty of eradicating Spectre-type attacks.

We responsibly disclosed the work to Intel, ARM, and AMD.

Experimental Setup. Unless noted otherwise, the experimental results reported were performed on recent Intel Skylake i5-6200U, Coffee Lake i7-8700K, and Whiskey Lake i7-8565U CPUs. Our AMD test machines were a Ryzen 1950X and a Ryzen Threadripper 1920X. For experiments on ARM, an NVIDIA Jetson TX1 has been used.

Outline. Section 2 provides background. We systematize Spectre in Section 3 and Meltdown in Section 4. We analyze and classify gadgets in Section 5 and defenses in Section 6. We discuss future work and conclude in Section 7.

.

[1] https://github.com/IAIK/transientfail

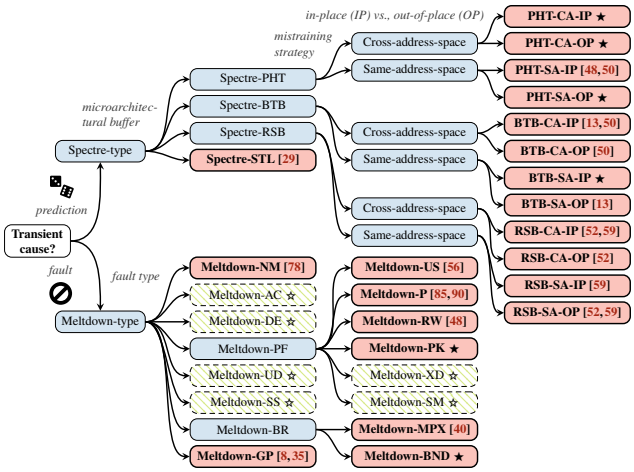


Figure 1: Transient execution attack classification tree with demonstrated attacks (red, bold), negative results (green, dashed), some first explored in this work (★ / ☆).[2]

2 Transient Execution

Instruction Set Architecture and Microarchitecture. The instruction set architecture (ISA) provides an interface between hardware and software. It defines the instructions that a processor supports, the available registers, the addressing mode, and describes the execution model. Examples of different ISAs are x86 and ARMv8. The microarchitecture then describes how the ISA is implemented in a processor in the form of pipeline depth, interconnection of elements, execution units, cache, branch prediction. The ISA and the microarchitecture are both stateful. In the ISA, this state includes, for instance, data in registers or main memory after a successful computation. Therefore, the architectural state can be observed by the developer. The microarchitectural state includes, for instance, entries in the cache and the translation lookaside buffer (TLB), or the usage of the execution units. Those microarchitectural elements are transparent to the programmer and can not be observed directly, only indirectly.

Out-of-Order Execution. On modern CPUs, individual instructions of a complex instruction set are first decoded and split-up into simpler micro-operations (μOPs) that are then processed. This design decision allows for superscalar optimizations and to extend or modify the implementation of specific instructions through so-called microcode updates. Furthermore, to increase performance, CPU's usually implement a so-called *out-of-order* design. This allows the CPU to execute μOPs not only in the sequential order provided by

[2] An up-to-date version of the tree is available at http://transient.fail/

the instruction stream but to dispatch them in parallel, utilizing the CPU's execution units as much as possible and, thus, improving the overall performance. If the required operands of a μOP are available, and its corresponding execution unit is not busy, the CPU starts its execution even if μOPs earlier in the instruction stream have not finished yet. As immediate results are only made visible at the architectural level when all previous μOPs have finished, CPUs typically keep track of the status of μOPs in a so-called *Reorder Buffer* (ROB). The CPU takes care to *retire* μOPs in-order, deciding to either discard their results or commit them to the architectural state. For instance, exceptions and external interrupt requests are handled during retirement by flushing any outstanding μOP results from the ROB. Therefore, the CPU may have executed so-called *transient instructions* [56], whose results are never committed to the architectural state.

Speculative Execution. Software is mostly not linear but contains (conditional) branches or data dependencies between instructions. In theory, the CPU would have to stall until a branch or dependencies are resolved before it can continue the execution. As stalling decreases performance significantly, CPUs deploy various mechanisms to predict the outcome of a branch or a data dependency. Thus, CPUs continue executing along the predicted path, buffering the results in the ROB until the correctness of the prediction is verified as its dependencies are resolved. In the case of a correct prediction, the CPU can commit the pre-computed results from the reorder buffer, increasing the overall performance. However, if the prediction was incorrect, the CPU needs to perform a rollback to the last correct state by squashing all pre-computed transient instruction results from the ROB.

Cache Covert Channels. Modern CPUs use caches to hide memory latency. However, these latency differences can be exploited in side-channels and covert channels [24,51,60,67,92]. In particular, Flush+Reload allows observations across cores at cache-line granularity, enabling attacks, e.g., on cryptographic algorithms [26, 43, 92], user input [24, 55, 72], and kernel addressing information [23]. For Flush+Reload, the attacker continuously flushes a shared memory address using the `clflush` instruction and afterward reloads the data. If the victim used the cache line, accessing it will be fast; otherwise, it will be slow.

Covert channels are a special use case of side-channel attacks, where the attacker controls both the sender and the receiver. This allows an attacker to bypass many restrictions that exist at the architectural level to leak information.

Transient Execution Attacks. Transient instructions reflect unauthorized computations out of the program's intended code and/or data paths. For functional correctness, it is crucial that their results are never committed to the architectural state. However, transient instructions may still leave traces in the CPU's microarchitectural state, which can subsequently be exploited to partially recover unauthorized results [50, 56, 85]. This observation has led to a variety of transient execution

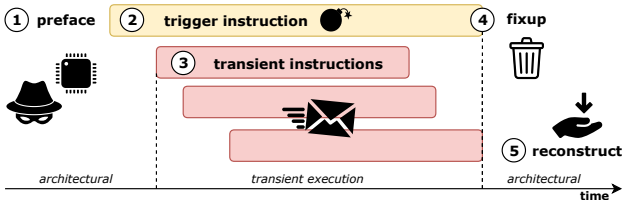


Figure 2: High-level overview of a transient execution attack in 5 phases: (1) prepare microarchitecture, (2) execute a *trigger instruction*, (3) *transient instructions* encode unauthorized data through a microarchitectural covert channel, (4) CPU retires trigger instruction and flushes transient instructions, (5) reconstruct secret from microarchitectural state.

attacks, which from a high-level always follow the same abstract flow, as shown in Figure 2. The attacker first brings the microarchitecture into the desired state, e.g., by flushing and/or populating internal branch predictors or data caches. Next is the execution of a so-called *trigger instruction*. This can be any instruction that causes subsequent operations to be eventually squashed, e.g., due to an exception or a mispredicted branch or data dependency. Before completion of the trigger instruction, the CPU proceeds with the execution of a *transient instruction sequence*. The attacker abuses the transient instructions to act as the sending end of a microarchitectural covert channel, e.g., by loading a secret-dependent memory location into the CPU cache. Ultimately, at the retirement of the trigger instruction, the CPU discovers the exception/misprediction and flushes the pipeline to discard any architectural effects of the transient instructions. However, in the final phase of the attack, unauthorized transient computation results are recovered at the receiving end of the covert channel, e.g., by timing memory accesses to deduce the secret-dependent loads from the transient instructions.

High-Level Classification: Spectre vs. Meltdown. Transient execution attacks have in common that they abuse transient instructions (which are never architecturally committed) to encode unauthorized data in the microarchitectural state. With different instantiations of the abstract phases in Figure 2, a wide spectrum of transient execution attack variants emerges. We deliberately based our classification on the root cause of the transient computation (phases 1, 2), abstracting away from the specific covert channel being used to transmit the unauthorized data (phases 3, 5). This leads to a first important split in our classification tree (cf. Figure 1). Attacks of the first type, dubbed Spectre [50], exploit transient execution following control or data flow misprediction. Attacks of the second type, dubbed Meltdown [56], exploit transient execution following a faulting instruction.

Importantly, Spectre and Meltdown exploit fundamentally different CPU properties and hence require orthogonal defenses. Where the former relies on dedicated control or data flow prediction machinery, the latter merely exploits that data from a faulting instruction is forwarded to instructions ahead

Table 1: Spectre-type attacks and the microarchitectural element they exploit (●), partially target (◐), or not affect (○).

Attack \ Element	BTB	BHB	PHT	RSB	STL
Spectre-PHT (Variant 1) [50]	○	◐	●	○	○
Spectre-PHT (Variant 1.1) [48]	○	◐	●	○	○
Spectre-BTB (Variant 2) [50]	●	◐	○	○	○
Spectre-RSB (ret2spec) [52, 59]	◐	○	○	●	○
Spectre-STL (Variant 4) [29]	○	○	○	○	●

Glossary: Branch Target Buffer (BTB), Branch History Buffer (BHB), Pattern History Table (PHT), Return Stack Buffer (RSB), Store To Load (STL).

in the pipeline. Note that, while Meltdown-type attacks so far exploit out-of-order execution, even elementary in-order pipelines may allow for similar effects [86]. Essentially, the different root cause of the trigger instruction (Spectre-type misprediction vs. Meltdown-type fault) determines the nature of the subsequent unauthorized transient computations and hence the scope of the attack.

That is, in the case of Spectre, transient instructions can only compute on data which the application is also allowed to access architecturally. Spectre thus transiently bypasses *software-defined* security policies (e.g., bounds checking, function call/return abstractions, memory stores) to leak secrets out of the program's intended code/data paths. Hence, much like in a "confused deputy" scenario, successful Spectre attacks come down to steering a victim into transiently computing on memory locations the victim is authorized to access but the attacker not. In practice, this implies that one or more phases of the transient execution attack flow in Figure 2 should be realized through so-called *code gadgets* executing within the victim application. We propose a novel taxonomy of gadgets based on these phases in Section 5.

For Meltdown-type attacks, on the other hand, transient execution allows to completely "melt down" architectural isolation barriers by computing on unauthorized results of faulting instructions. Meltdown thus transiently bypasses *hardware-enforced* security policies to leak data that should always remain architecturally inaccessible for the application. Where Spectre-type leakage remains largely an unintended side-effect of important speculative performance optimizations, Meltdown reflects a failure of the CPU to respect hardware-level protection boundaries for transient instructions. That is, the mere continuation of the transient execution after a fault itself is required, but not sufficient for a successful Meltdown attack. As further explored in Section 6, this has profound consequences for defenses. Overall, mitigating Spectre requires careful hardware-software co-design, whereas merely replacing the data of a faulting instruction with a dummy value suffices to block Meltdown-type leakage in silicon, e.g., as it is done in AMD processors, or with the Rogue Data Cache Load resistance (RDCL_NO) feature advertised in recent Intel CPUs from Whiskey Lake onwards [40].

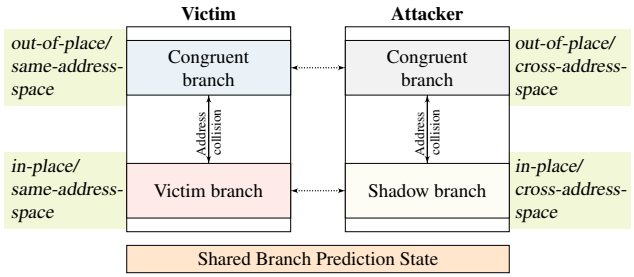


Figure 3: A branch can be mistrained either by the victim process (*same-address-space*) or by an attacker-controlled process (*cross-address-space*). Mistraining can be achieved either using the vulnerable branch itself (*in-place*) or a branch at a congruent virtual address (*out-of-place*).

3 Spectre-type Attacks

In this section, we provide an overview of Spectre-type attacks (cf. Figure 1). Given the versatility of Spectre variants in a variety of adversary models, we propose a novel two-level taxonomy based on the preparatory phases of the abstract transient execution attack flow in Figure 2. First, we distinguish the different microarchitectural buffers that can trigger a prediction (phase 2), and second, the mistraining strategies that can be used to steer the prediction (phase 1).

Systematization of Spectre Variants. To predict the outcome of various types of branches and data dependencies, modern CPUs accumulate an extensive microarchitectural state across various internal buffers and components [19]. Table 1 overviews Spectre-type attacks and the corresponding microarchitectural elements they exploit. As the first level of our classification tree, we categorize Spectre attacks based on the microarchitectural root cause that triggers the misprediction leading to the transient execution:

- Spectre-PHT [48, 50] exploits the *Pattern History Table* (PHT) that predicts the outcome of conditional branches.
- Spectre-BTB [50] exploits the *Branch Target Buffer* (BTB) for predicting branch destination addresses.
- Spectre-RSB [52, 59] primarily exploits the *Return Stack Buffer* (RSB) for predicting return addresses.
- Spectre-STL [29] exploits memory disambiguation for predicting *Store To Load* (STL) data dependencies.

Note that NetSpectre [74], SGXSpectre [63], and SGXPectre [13] focus on applying one of the above Spectre variants in a specific exploitation scenario. Hence, we do not consider them separate variants in our classification.

Systematization of Mistraining Strategies. We now propose a second-level classification scheme for Spectre variants that abuse history-based branch prediction (*i.e.*, all of the above except Spectre-STL). These Spectre variants first go through a preparatory phase (cf. Figure 2) where the microarchitectural branch predictor state is "poisoned" to cause intentional misspeculation of a particular victim branch. Since branch prediction buffers in modern CPUs [19, 50] are com-

monly indexed based on the virtual address of the branch instruction, mistraining can happen either within the same address space or from a different attacker-controlled process. Furthermore, as illustrated in Figure 3, when only a subset of the virtual address is used in the prediction, mistraining can be achieved using a branch instruction at a congruent virtual address. We thus enhance the field of Spectre-type branch poisoning attacks with 4 distinct mistraining strategies:

1. Executing the victim branch in the victim process (*same-address-space in-place*).
2. Executing a congruent branch in the victim process (*same-address-space out-of-place*).
3. Executing a shadow branch in a different process (*cross-address-space in-place*).
4. Executing a congruent branch in a different process (*cross-address-space out-of-place*).

In current literature [6, 13, 48, 50], several of the above branch poisoning strategies have been overlooked for different Spectre variants. We summarize the results of an assessment of vulnerabilities under mistraining strategies in Table 2. Our systematization thus reveals clear blind spots that allow an attacker to mistrain branch predictors in previously unknown ways. As explained further, depending on the adversary's capabilities (e.g., in-process, sandboxed, remote, enclave, etc.) these previously unknown mistraining strategies may lead to new attacks and/or bypass existing defenses.

Table 2: Spectre-type attacks performed in-place, out-of-place, same-address-space (*i.e.*, intra-process), or cross-address-space (*i.e.*, cross-process).

	Method		Attack: Spectre-PHT	Spectre-BTB	Spectre-RSB	Spectre-STL
Intel	intra-process	in-place	● [48,50]	★	● [59]	● [29]
		out-of-place	★	● [13]	● [52,59]	○
	cross-process	in-place	★	● [13,50]	● [52,59]	○
		out-of-place	★	● [50]	● [52]	○
ARM	intra-process	in-place	● [48,50]	★	● [6]	● [6]
		out-of-place	★	☆	● [6]	○
	cross-process	in-place	★	● [6,50]	☆	○
		out-of-place	★	☆	☆	○
AMD	intra-process	in-place	● [50]	★	★	● [29]
		out-of-place	★	☆	★	○
	cross-process	in-place	★	● [50]	★	○
		out-of-place	★	☆	★	○

Symbols indicate whether an attack is possible and known (●), not possible and known (○), possible and previously unknown or not shown (★), or tested and did not work and previously unknown or not shown (☆). All tests performed with no defenses enabled.

3.1 Spectre-PHT (Input Validation Bypass)

Microarchitectural Element. Kocher et al. [50] first introduced Spectre Variant 1, an attack that poisons the Pattern History Table (PHT) to mispredict the direction (taken or not-taken) of conditional branches. Depending on the underlying microarchitecture, the PHT is accessed based on a combination of virtual address bits of the branch instruction plus a hidden Branch History Buffer (BHB) that accumulates global behavior for the last N branches on the same physical core [18, 19]

Reading Out-of-Bounds. Conditional branches are commonly used by programmers and/or compilers to maintain memory safety invariants at runtime. For example, consider the following code snippet for bounds checking [50]:

```
if (x < len(array1)) { y = array2[array1[x] * 4096]; }
```

At the architectural level, this program clearly ensures that the index variable `x` always lies within the bounds of the fixed-length buffer `array1`. However, after repeatedly supplying valid values of `x`, the PHT will reliably predict that this branch evaluates to true. When the adversary now supplies an invalid index `x`, the CPU continues along a mispredicted path and transiently performs an out-of-bounds memory access. The above code snippet features an explicit example of a "leak gadget" that may act as a microarchitectural covert channel: depending on the out-of-bounds value being read, the transient instructions load another memory page belonging to `array2` into the cache.

Writing Out-of-Bounds. Kiriansky and Waldspurger [48] showed that transient writes are also possible by following the same principle. Consider the following code line:

```
if (x < len(array)) { array[x] = value; }
```

After mistraining the PHT component, attackers controlling the untrusted index `x` can transiently write to arbitrary out-of-bounds addresses. This creates a transient buffer overflow, allowing the attacker to bypass both type and memory safety. Ultimately, when repurposing traditional techniques from return-oriented programming [75] attacks, adversaries may even gain arbitrary code execution in the transient domain by overwriting return addresses or code pointers.

Overlooked Mistraining Strategies. Spectre-PHT attacks so far [48, 50, 63] rely on a same-address-space in-place branch poisoning strategy. However, our results (cf. Table 2) reveal that the Intel, ARM, and AMD CPUs we tested are vulnerable to all four PHT mistraining strategies. In this, we are the first to successfully demonstrate Spectre-PHT-style branch misprediction attacks *without prior execution of the victim branch*. This is an important contribution as it may open up previously unknown attack avenues for restricted adversaries.

Cross-address-space PHT poisoning may, for instance, enable advanced attacks against a privileged daemon process that does not directly accept user input. Likewise, for Intel SGX technology, remote attestation schemes have been developed [76] to enforce that a victim enclave can only be run exactly once. This effectively rules out current state-of-the-art SGXSpectre [63] attacks that repeatedly execute the victim enclave to mistrain the PHT branch predictor. Our novel out-of-place PHT poisoning strategy, on the other hand, allows us to perform the training phase entirely *outside* the enclave on

the same physical core by repeatedly executing a congruent branch in the untrusted enclave host process (cf. Figure 3).

3.2 Spectre-BTB (Branch Target Injection)

Microarchitectural Element. In Spectre Variant 2 [50], the attacker poisons the Branch Target Buffer (BTB) to steer the transient execution to a mispredicted branch target. For direct branches, the CPU indexes the BTB using a subset of the virtual address bits of the branch instruction to yield the predicted jump target. For indirect branches, CPUs use different mechanisms [28], which may take into account global branching history accumulated in the BHB when indexing the BTB. We refer to both types as Spectre-BTB.

Hijacking Control Flow. Contrary to Spectre-PHT, where transient instructions execute along a restricted mispredicted path, Spectre-BTB allows redirecting transient control flow to an arbitrary destination. Adopting established techniques from return-oriented programming (ROP) attacks [75], but abusing BTB poisoning instead of application-level vulnerabilities, selected code "gadgets" found in the victim address space may be chained together to construct arbitrary transient instruction sequences. Hence, where the success of Spectre-PHT critically relies on unintended leakage along the mispredicted code path, ROP-style gadget abuse in Spectre-BTB allows to more directly construct covert channels that expose secrets from the transient domain (cf. Figure 2). We discuss gadget types in more detail in Section 5.

Overlooked Mistraining Strategies. Spectre-BTB was initially demonstrated on Intel, AMD, and ARM CPUs using a cross-address-space in-place mistraining strategy [50]. With SGXPectre [13], Chen et al. extracted secrets from Intel SGX enclaves using either a cross-address-space in-place or same-address-space out-of-place BTB poisoning strategy. We experimentally reproduced these mistraining strategies through a systematic evaluation presented in Table 2. On AMD and ARM, we could not demonstrate out-of-place BTB poisoning. Possibly, these CPUs use an unknown (sub)set of virtual address bits or a function of bits which we were not able to reverse engineer. We encourage others to investigate whether a different (sub)set of virtual address bits is required to enable the attack.

To the best of our knowledge, we are the first to recognize that Spectre-BTB mistraining can also proceed by *repeatedly executing the vulnerable indirect branch with valid inputs*. Much like Spectre-PHT, such same-address-space in-place BTB (Spectre-BTB-SA-IP) poisoning abuses the victim's own execution to mistrain the underlying branch target predictor. Hence, as an important contribution to understanding attack surface and defenses, in-place mistraining *within* the victim domain may allow bypassing widely deployed mitigations [4, 40] that flush and/or partition the BTB before entering the victim. Since the branch destination address is now determined by the victim code and not under the direct control of the attacker, however, Spectre-BTB-SA-IP cannot offer the full power of arbitrary transient control flow redirection. Yet, in higher-level languages like C++ that commonly rely on indirect branches to implement polymorph abstractions, Spectre-BTB-SA-IP may lead to subtle "speculative type confusion" vulnerabilities. For example, a victim that repeatedly executes a virtual function call with an object of `TypeA` may inadvertently mistrain the branch target predictor to cause misspeculation when finally executing the virtual function call with an object of another `TypeB`.

3.3 Spectre-RSB (Return Address Injection)

Microarchitectural Element. Maisuradze and Rossow [59] and Koruyeh et al. [52] introduced a Spectre variant that exploits the Return Stack Buffer (RSB). The RSB is a small per-core microarchitectural buffer that stores the virtual addresses following the N most recent `call` instructions. When encountering a `ret` instruction, the CPU pops the topmost element from the RSB to predict the return flow.

Hijacking Return Flow. Misspeculation arises whenever the RSB layout diverges from the actual return addresses on the software stack. Such disparity for instance naturally occurs when restoring kernel/enclave/user stack pointers upon protection domain switches. Furthermore, same-address-space adversaries may explicitly overwrite return addresses on the software stack, or transiently execute `call` instructions which update the RSB without committing architectural effects [52]. This may allow untrusted code executing in a sandbox to transiently divert return control flow to interesting code gadgets outside of the sandboxed environment.

Due to the fixed-size nature of the RSB, a special case of misspeculation occurs for deeply nested function calls [52,59]. Since the RSB can only store return addresses for the N most recent calls, an underfill occurs when the software stack is unrolled. In this case, the RSB can no longer provide accurate predictions. Starting from Skylake, Intel CPUs use the BTB as a fallback [19,52], thus allowing Spectre-BTB-style attacks triggered by `ret` instructions.

Overlooked Mistraining Strategies. Spectre-RSB has been demonstrated with all four mistraining strategies, but only on Intel [52, 59]. Our experimental results presented in Table 2 generalize these strategies to AMD CPUs. Furthermore, in line with ARM's own analysis [6], we successfully poisoned RSB entries within the same-address-space but did not observe any cross-address-space leakage on ARM CPUs. We expect this may be a limitation of our current proof-of-concept code and encourage others to investigate this further.

3.4 Spectre-STL (Speculative Store Bypass)

Microarchitectural Element. Speculation in modern CPUs is not restricted to control flow but also includes predicting dependencies in the data flow. A common type of Store To

Table 3: Demonstrated Meltdown-type (MD) attacks.

Attack	#GP	#NM	#BR	#PF	U/S	P	R/W	RSVD	XD	PK
MD-GP (Variant 3a) [8]	●	○	○	○						
MD-NM (Lazy FP) [78]	○	●	○	○						
MD-BR	○	○	●	○						
MD-US (Meltdown) [56]	○	○	○	●	●	○	○	○	○	○
MD-P (Foreshadow) [85, 90]	○	○	○	●	○	●	○	●	○	○
MD-RW (Variant 1.2) [48]	○	○	○	●	○	○	●	○	○	○
MD-PK	○	○	○	●	○	○	○	○	○	●

Symbols (● or ○) indicate whether an exception type (left) or permission bit (right) is exploited. Systematic names are derived from what is exploited.

Load (STL) dependencies require that a memory load shall not be executed before all preceding stores that write to the same location have completed. However, even before the addresses of all prior stores in the pipeline are known, the CPUs' memory disambiguator [3, 33, 44] may predict which loads can already be executed speculatively.

When the disambiguator predicts that a load does not have a dependency on a prior store, the load reads data from the L1 data cache. When the addresses of all prior stores are known, the prediction is verified. If any overlap is found, the load and all following instructions are re-executed.

Reading Stale Values. Horn [29] showed how mispredictions by the memory disambiguator could be abused to speculatively bypass store instructions. Like previous attacks, Spectre-STL adversaries rely on an appropriate transient instruction sequence to leak unsanitized stale values via a microarchitectural covert channel. Furthermore, operating on stale pointer values may speculatively break type and memory safety guarantees in the transient execution domain [29].

4 Meltdown-type Attacks

This section overviews Meltdown-type attacks, and presents a classification scheme that led to the discovery of two previously overlooked Meltdown variants (cf. Figure 1). Importantly, where Spectre-type attacks exploit (branch) misprediction events to trigger transient execution, Meltdown-type attacks rely on transient instructions following a CPU exception. Essentially, Meltdown exploits that exceptions are only raised (*i.e.*, become architecturally visible) upon the retirement of the faulting instruction. In some microarchitectures, this property allows transient instructions ahead in the pipeline to compute on unauthorized results of the instruction that is about to suffer a fault. The CPU's in-order instruction retirement mechanism takes care to discard any architectural effects of such computations, but as with the Spectre-type attacks above, secrets may leak through microarchitectural covert channels.

Systematization of Meltdown Variants. We introduce a classification for Meltdown-type attacks in two dimensions. In the first level, we categorize attacks based on the exception

Table 4: Secrets recoverable via Meltdown-type attacks and whether they cross the current privilege level (CPL).

Attack \ Leaks	Memory	Cache	Register	Cross-CPL
Meltdown-US (Meltdown) [56]	●	●	○	✓
Meltdown-P (Foreshadow-NG) [90]	○	●	○	✓
Meltdown-P (Foreshadow-SGX) [85]	◐	●	◐	✓
Meltdown-GP (Variant 3a) [8]	○	○	●	✓
Meltdown-NM (Lazy FP) [78]	○	○	●	✓
Meltdown-RW (Variant 1.2) [48]	●	●	○	✗
Meltdown-PK	☆	★	☆	✗
Meltdown-BR	★	★	☆	✗

Symbols indicate whether an attack crosses a processor privilege level (✓) or not (✗), whether it can leak secrets from a buffer (●), only with additional steps (◐), or not at all (○). Respectively (★ vs. ☆) if first shown in this work.

that causes transient execution. Following Intel's [31] classification of exceptions as *faults*, *traps*, or *aborts*, we observed that Meltdown-type attacks so far have exploited faults, but not traps or aborts. The CPU generates faults if a correctable error has occurred, *i.e.*, they allow the program to continue after it has been resolved. Traps are reported immediately after the execution of the instruction, *i.e.*, when the instruction retires and becomes architecturally visible. Aborts report some unrecoverable error and do not allow a restart of the task that caused the abort.

In the second level, for page faults (`#PF`), we further categorize based on page-table entry protection bits (cf. Table 3). We also categorize attacks based on which storage locations can be reached, and whether it crosses a privilege boundary (cf. Table 4). Through this systematization, we discovered several previously unknown Meltdown variants that exploit different exception types as well as page-table protection bits, including two exploitable ones. Our systematic analysis furthermore resulted in the first demonstration of exploitable Meltdown-type delayed exception handling effects on AMD CPUs.

4.1 Meltdown-US (Supervisor-only Bypass)

Modern CPUs commonly feature a "user/supervisor" page-table attribute to denote a virtual memory page as belonging to the OS kernel. The original Meltdown attack [56] reads kernel memory from user space on CPUs that do *not* transiently enforce the user/supervisor flag. In the trigger phase (cf. Figure 2) an unauthorized kernel address is dereferenced, which eventually causes a page fault. Before the fault becomes architecturally visible, however, the attacker executes a transient instruction sequence that for instance accesses a cache line based on the privileged data read by the trigger instruction. In the final phase, after the exception has been raised, the privileged data is reconstructed at the receiving end of the covert channel (e.g., Flush+Reload).

The attacks bandwidth can be improved by suppressing exceptions through transaction memory CPU features such as

Intel TSX [31], exception handling [56], or hiding it in another transient execution [28, 56]. By iterating byte-by-byte over the kernel space and suppressing or handling exceptions, an attacker can dump the entire kernel. This includes the entire physical memory if the operating system has a direct physical map in the kernel. While extraction rates are significantly higher when the kernel data resides in the CPU cache, Meltdown has even been shown to successfully extract uncached data from memory [56].

4.2 Meltdown-P (Virtual Translation Bypass)

Foreshadow. Van Bulck et al. [85] presented Foreshadow, a Meltdown-type attack targeting Intel SGX technology [30]. Unauthorized accesses to enclave memory usually do not raise a `#PF` exception but are instead silently replaced with abort page dummy values (cf. Section 6.2). In the absence of a fault, plain Meltdown cannot be mounted against SGX enclaves. To overcome this limitation, a Foreshadow attacker clears the "present" bit in the page-table entry mapping the enclave secret, ensuring that a `#PF` will be raised for subsequent accesses. Analogous to Meltdown-US, the adversary now proceeds with a transient instruction sequence to leak the secret (e.g., through a Flush+Reload covert channel).

Intel [34] named *L1 Terminal Fault* (L1TF) as the root cause behind Foreshadow. A terminal fault occurs when accessing a page-table entry with either the present bit cleared or a "reserved" bit set. In such cases, the CPU immediately aborts address translation. However, since the L1 data cache is indexed in parallel to address translation, the page table entry's physical address field (*i.e.*, frame number) may still be passed to the L1 cache. Any data present in L1 and tagged with that physical address will now be forwarded to the transient execution, regardless of access permissions.

Although Meltdown-P-type leakage is restricted to the L1 data cache, the original Foreshadow [85] attack showed how SGX's secure page swapping mechanism might first be abused to prefetch arbitrary enclave pages into the L1 cache, including even CPU registers stored on interrupt. This highlights that SGX's privileged adversary model considerably amplifies the transient execution attack surface.

Foreshadow-NG. Foreshadow-NG [90] generalizes Foreshadow from the attack on SGX enclaves to bypass operating system or hypervisor isolation. The generalization builds on the observation that the physical frame number in a page-table entry is sometimes under direct or indirect control of an adversary. For instance, when swapping pages to disk, the kernel is free to use all but the present bit to store metadata (e.g., the offset on the swap partition). However, if this offset is a valid physical address, any cached memory at that location leaks to an unprivileged Foreshadow-OS attacker.

Even worse is the Foreshadow-VMM variant, which allows an untrusted virtual machine, controlling guest-physical addresses, to extract the host machine's entire L1 data cache (including data belonging to the hypervisor or other virtual machines). The underlying problem is that a terminal fault in the guest page-tables early-outs the address translation process, such that guest-physical addresses are erroneously passed to the L1 data cache, without first being translated into a proper host physical address [34].

4.3 Meltdown-GP (System Register Bypass)

Meltdown-GP (named initially Variant 3a) [37] allows an attacker to read privileged system registers. It was first discovered and published by ARM [8] and subsequently Intel [35] determined that their CPUs are also susceptible to the attack. Unauthorized access to privileged system registers (e.g., via `rdmsr`) raises a *general protection* fault (`#GP`). Similar to previous Meltdown-type attacks, however, the attack exploits that the transient execution following the faulting instruction can still compute on the unauthorized data, and leak the system register contents through a microarchitectural covert channel (e.g., Flush+Reload).

4.4 Meltdown-NM (FPU Register Bypass)

During a context switch, the OS has to save all the registers, including the floating point unit (FPU) and SIMD registers. These latter registers are large and saving them would slow down context switches. Therefore, CPUs allow for a lazy state switch, meaning that instead of saving the registers, the FPU is simply marked as "not available". The first FPU instruction issued after the FPU was marked as "not available" causes a *device-not-available* (`#NM`) exception, allowing the OS to save the FPU state of previous execution context before marking the FPU as available again.

Stecklina and Prescher [78] propose an attack on the above lazy state switch mechanism. The attack consists of three steps. In the first step, a victim performs operations loading data into the FPU registers. Then, in the second step, the CPU switches to the attacker and marks the FPU as "not available". The attacker now issues an instruction that uses the FPU, which generates an `#NM` fault. Before the faulting instruction retires, however, the CPU has already transiently executed the following instructions using data from the previous context. As such, analogous to previous Meltdown-type attacks, a malicious transient instruction sequence following the faulting instruction can encode the unauthorized FPU register contents through a microarchitectural covert channel (e.g., Flush+Reload).

4.5 Meltdown-RW (Read-only Bypass)

Where the above attacks [8, 56, 78, 85] focussed on stealing information across privilege levels, Kiriansky and Waldspurger [48] presented the first Meltdown-type attack that bypasses page-table based access rights *within* the current

privilege level. Specifically, they showed that transient execution does not respect the "read/write" page-table attribute. The ability to transiently overwrite read-only data within the current privilege level can bypass software-based sandboxes which rely on hardware enforcement of read-only memory.

Confusingly, the above Meltdown-RW attack was originally named "Spectre Variant 1.2" [48] as the authors followed a Spectre-centric naming scheme. Our systematization revealed, however, that the transient cause exploited above is a `#PF` exception. Hence, this attack is of Meltdown-type, but *not* a variant of Spectre.

4.6 Meltdown-PK (Protection Key Bypass)

Intel Skylake-SP server CPUs support memory-protection keys for user space (PKU) [32]. This feature allows processes to change the access permissions of a page directly from user space, *i.e.*, without requiring a syscall/hypercall. Thus, with PKU, user-space applications can implement efficient hardware-enforced isolation of trusted parts [27, 84].

We present a novel Meltdown-PK attack to bypass both read and write isolation provided by PKU. Meltdown-PK works if an attacker has code execution in the containing process, even if the attacker cannot execute the `wrpkru` instruction (e.g., blacklisting). Moreover, in contrast to cross-privilege level Meltdown attack variants, there is no software workaround. According to Intel [36], Meltdown-PK can be mitigated using address space isolation. Recent Meltdown-resistant Intel processors enumerating RDCL_NO plus PKU support furthermore mitigate Meltdown-PK in silicon. With those mitigations, the memory addresses that might be revealed by transient execution attacks can be limited.

Experimental Results. We tested Meltdown-PK on an Amazon EC2 C5 instance running Ubuntu 18.04 with PKU support. We created a memory mapping and used PKU to remove both read and write access. As expected, protected memory accesses produce a `#PF`. However, our proof-of-concept manages to leak the data via an adversarial transient instruction sequence with a Flush+Reload covert channel.

4.7 Meltdown-BR (Bounds Check Bypass)

To facilitate efficient software instrumentation, x86 CPUs come with dedicated hardware instructions that raise a *bound-range-exceeded* exception (`#BR`) when encountering out-of-bound array indices. The IA-32 ISA, for instance, defines a `bound` opcode for this purpose. While the `bound` instruction was omitted in the subsequent x86-64 ISA, modern Intel CPUs ship with Memory Protection eXtensions (MPX) for efficient array bounds checking.

Our systematic evaluation revealed that Meltdown-type effects of the `#BR` exception had not been thoroughly investigated yet. Specifically, Intel's analysis [40] only briefly mentions MPX-based bounds check bypass as a possibility, and

Table 5: CPU vendors vulnerable to Meltdown (MD).

Vendor \ Attack	MD-US [56]	MD-P [85, 90]	MD-GP [8, 35]	MD-NM [78]	MD-RW [48]	MD-PK	MD-BR	MD-DE	MD-AC	MD-UD	MD-SS	MD-XD	MD-SM
Intel	●	●	●	●	●	★	★	☆	☆	☆	☆	☆	☆
ARM	●	○	●	_	●	_	_	☆	☆	☆	_	☆	☆
AMD	○	○	○	○	○	_	★	☆	☆	☆	☆	☆	☆

Symbols indicate whether at least one CPU model is vulnerable (filled) vs. no CPU is known to be vulnerable (empty). Glossary: reproduced (● vs. ○), first shown in this paper (★ vs. ☆), not applicable (_). All tests performed without defenses enabled.

recent defensive work by Dong et al. [16] highlights the need to introduce a memory `lfence` after MPX bounds check instructions. They classify this as a Spectre-type attack, implying that the `lfence` is needed to prevent the branch predictor from speculating on the outcome of the bounds check. According to Oleksenko et al. [64], neither `bndcl` nor `bndcu` exert pressure on the branch predictor, indicating that there is no prediction happening. Based on that, we argue that the classification as a Spectre-type attack is misleading as no prediction is involved. The observation by Dong et al. [16] indeed does not shed light on the `#BR` exception as the root cause for the MPX bounds check bypass, and they do not consider IA32 `bound` protection at all. Similar to Spectre-PHT, Meltdown-BR is a bounds check bypass, but instead of mistraining a predictor it exploits the lazy handling of the raised `#BR` exception.

Experimental Results. We introduce the Meltdown-BR attack which exploits transient execution following a `#BR` exception to encode out-of-bounds secrets that are never architecturally visible. As such, Meltdown-BR is an exception-driven alternative for Spectre-PHT. Our proofs-of-concept demonstrate out-of-bounds leakage through a Flush+Reload covert channel for an array index safeguarded by either IA32 `bound` (Intel, AMD), or state-of-the-art MPX protection (Intel-only). For Intel, we ran the attacks on a Skylake i5-6200U CPU with MPX support, and for AMD we evaluated both an E2-2000 and a Ryzen Threadripper 1920X. This is the first experiment demonstrating a Meltdown-type transient execution attack exploiting delayed exception handling on AMD CPUs [4, 56].

4.8 Residual Meltdown (Negative Results)

We systematically studied transient execution leakage for other, not yet tested exceptions. In our experiments, we consistently found no traces of transient execution beyond traps or aborts, which leads us to the hypothesis that Meltdown is only possible with faults (as they can occur at any moment during instruction execution). Still, the possibility remains that our experiments failed and that they are possible. Table 5 and Figure 1 summarize experimental results for fault types tested on Intel, ARM, and AMD.

Division Errors. For the divide-by-zero experiment, we leveraged the signed division instruction (`idiv` on x86 and

Table 6: Gadget classification according to the attack flow and whether executed by the attacker (●), victim (○), or either (◐).

Attack	1. Preface	2. Trigger example	3. Transient	5. Reconstruction
Covert channel [1,74,92]	◐ Flush/Prime/Evict	-	◐ Load/AVX/Port/...	◐ Reload/Probe/Time
Meltdown-US/RW/GP/NM/PK [8,48,56,78]	● (Exception suppression)	● `mov/rdmsr/FPU`	● Controlled encode	● Exception handling
Meltdown-P [85,90]	○ (L1 prefetch)	● `mov`	● Controlled encode	& controlled decode
Meltdown-BR	-	○ `bound/bndclu`	○ Inadvertent leak	*same as above*
Spectre-PHT [50]	◐ PHT poisoning	○ `jz`	○ Inadvertent leak	● Controlled decode
Spectre-BTB/RSB [13,50,52,59]	◐ BTB/RSB poisoning	○ `call/jmp/ret`	○ ROP-style encode	● Controlled decode
Spectre-STL [29]	-	○ `mov`	○ Inadvertent leak	● Controlled decode
NetSpectre [74]	○ Thrash/reset	○ `jz`	○ Inadvertent leak	○ Inadvertent transmit

`sdiv` on ARM). On the ARMs we tested, there is no exception, but the division yields merely zero. On x86, the division raises a *divide-by-zero* exception (`#DE`). Both on the AMD and Intel we tested, the CPU continues with the transient execution after the exception. In both cases, the result register is set to '0', which is the same result as on the tested ARM. Thus, according to our experiments Meltdown-DE is not possible, as no real values are leaked.

Supervisor Access. Although supervisor mode access prevention (SMAP) raises a page fault (`#PF`) when accessing user-space memory from the kernel, it seems to be free of any Meltdown effect in our experiments. Thus, we were not able to leak any data using Meltdown-SM in our experiments.

Alignment Faults. Upon detecting an unaligned memory operand, the CPU may generate an *alignment check* exception (`#AC`). In our tests, the results of unaligned memory accesses never reach the transient execution. We suspect that this is because `#AC` is generated early-on, even before the operand's virtual address is translated to a physical one. Hence, our experiments with Meltdown-AC were unsuccessful in showing any leakage.

Segmentation Faults. We consistently found that out-of-limit segment accesses never reach transient execution in our experiments. We suspect that, due to the simplistic IA32 segmentation design, segment limits are validated early-on, and immediately raise a `#GP` or `#SS` (*stack-segment fault*) exception, without sending the offending instruction to the ROB. Therefore, we observed no leakage in our experiments with Meltdown-SS.

Instruction Fetch. To yield a complete picture, we investigated Meltdown-type effects during the instruction fetch and decode phases. On our test systems, we did not succeed in transiently executing instructions residing in non-executable memory (*i.e.*, Meltdown-XD), or following an *invalid opcode* (`#UD`) exception (*i.e.*, Meltdown-UD). We suspect that exceptions during instruction fetch or decode are immediately handled by the CPU, without first buffering the offending instruction in the ROB. Moreover, as invalid opcodes have an undefined length, the CPU does not even know where the next instruction starts. Hence, we suspect that invalid opcodes only leak if the microarchitectural effect is already an effect caused by the invalid opcode itself, not by subsequent transient instructions.

Table 7: Spectre-PHT gadget classification and the number of occurrences per gadget type in Linux kernel v5.0.

Gadget	Example (Spectre-PHT)	#Occurrences
Prefetch	`if(i<LEN_A){a[i];}`	172
Compare	`if(i<LEN_A){if(a[i]==k){};}`	127
Index	`if(i<LEN_A){y = b[a[i]*x];}`	0
Execute	`if(i<LEN_A){a[i](void);}`	16

5 Gadget Analysis and Classification

We deliberately oriented our attack tree (cf. Figure 1) on the microarchitectural root causes of the transient computation, abstracting away from the underlying covert channel and/or code *gadgets* required to carry out the attack successfully. In this section, we further dissect transient execution attacks by categorizing gadget types in two tiers and overviewing current results on their exploitability in real-world software.

5.1 Gadget Classification

First-Tier: Execution Phase. We define a "gadget" as a series of instructions executed by either the attacker or the victim. Table 6 shows how gadget types discussed in literature can be unambiguously assigned to one of the abstract attack phases from Figure 2. New gadgets can be added straightforwardly after determining their execution phase and objective.

Importantly, our classification table highlights that gadget choice largely depends on the attacker's capabilities. By plugging in different gadget types to compose the required attack phases, an almost boundless spectrum of adversary models can be covered that is only limited by the attacker's capabilities. For local adversaries with arbitrary code execution (e.g., Meltdown-US [56]), the gadget functionality can be explicitly implemented by the attacker. For sandboxed adversaries (e.g., Spectre-PHT [50]), on the other hand, much of the gadget functionality has to be provided by "confused deputy" code executing in the victim domain. Ultimately, as claimed by Schwarz et al. [74], even fully remote attackers may be able to launch Spectre attacks given that sufficient gadgets would be available inside the victim code.

Second-Tier: Transient Leakage. During our analysis of the Linux kernel (see Section 5.2), we discovered that gadgets required for Spectre-PHT can be further classified in a second

tier. A second tier is required in this case as those gadgets enable different types of attacks. The first type of gadget we found is called *Prefetch*. A Prefetch gadget consists of a single array access. As such it is not able to leak data, but can be used to load data that can then be leaked by another gadget as was demonstrated by Meltdown-P [85]. The second type of gadget, called *Compare*, loads a value like in the Prefetch gadget and then branches on it. Using a contention channel like execution unit contention [2, 9] or an AVX channel as claimed by Schwarz et al. [74], an attacker might be able to leak data. We refer to the third gadget as *Index* gadget and it is the double array access shown by Kocher et al. [50]. The final gadget type, called *Execute*, allows arbitrary code execution, similar to Spectre-BTB. In such a gadget, an array is indexed based on an attacker-controlled input and the resulting value is used as a function pointer, allowing an attacker to transiently execute code by accessing the array out-of-bounds. Table 7 gives examples for all four types.

5.2 Real-World Software Gadget Prevalence

While for Meltdown-type attacks, convincing real-world exploits have been developed to dump arbitrary process [56] and enclave [85] memory, most Spectre-type attacks have so far only been demonstrated in controlled environments. The most significant barrier to mounting a successful Spectre attack is to find exploitable gadgets in real-world software, which at present remains an important open research question in itself [59, 74].

Automated Gadget Analysis. Since the discovery of transient execution attacks, researchers have tried to develop methods for the automatic analysis of gadgets. One proposed method is called oo7 [89] and uses taint tracking to detect Spectre-PHT Prefetch and Index gadgets. oo7 first marks all variables that come from an untrusted source as tainted. If a tainted variable is later on used in a branch, the branch is also tainted. The tool then reports a possible gadget if a tainted branch is followed by a memory access depending on the tainted variable. Guarnieri et al. [25] mention that oo7 would still flag code locations that were patched with Speculative Load Hardening [12] as it would still match the vulnerable pattern.

Another approach, called Spectector [25], uses symbolic execution to detect Spectre-PHT gadgets. It tries to formally prove that a program does not contain any gadgets by tracking all memory accesses and jump targets during execution along all different program paths. Additionally, it simulates the path of mispredicted branches for a number of steps. The program is run twice to determine whether it is free of gadgets or not. First, it records a trace of memory accesses when no misspeculation occurs (*i.e.*, runs the program in its intended way). Second, it records a trace of memory accesses with misspeculation of a certain number of instructions. Spectector then reports a gadget if it detects a mismatch between the two traces. One problem with the Spectector approach is scalability as it is currently not feasible to symbolically execute large programs.

The Linux kernel developers use a different approach. They extended the Smatch static analysis tool to automatically discover potential Spectre-PHT out-of-bounds access gadgets [10]. Specifically, Smatch finds all instances of user-supplied array indices that have not been explicitly hardened. Unfortunately, Smatch's false positive rate is quite high. According to Carpenter [10], the tool reported 736 gadget candidates in April 2018, whereas the kernel only featured about 15 Spectre-PHT-resistant array indices at that time. We further investigated this by analyzing the number of occurrences of the newly introduced `array_index_nospec` and `array_index_mask_nospec` macros in the Linux kernel per month. Figure 4 shows that the number of Spectre-PHT patches has been continuously increasing over the past year. This provides further evidence that patching Spectre-PHT gadgets in real-world software is an ongoing effort and that automated detection methods and gadget classification pose an important research challenge.

Academic Review. To date, only 5 academic papers have demonstrated Spectre-type gadget exploitation in real-world software [9, 13, 29, 50, 59]. Table 8 reveals that they either abuse ROP-style gadgets in larger code bases or more commonly rely on Just-In-Time (JIT) compilation to indirectly provide the vulnerable gadget code. JIT compilers as commonly used in e.g., JavaScript, WebAssembly, or the eBPF Linux kernel interface, create a software-defined sandbox by extending the untrusted attacker-provided code with runtime checks. However, the attacks in Table 8 demonstrate that such JIT checks can be transiently circumvented to leak memory contents outside of the sandbox. Furthermore, in the case of Spectre-BTB/RSB, even non-JIT compiled real-world code has been shown to be exploitable when the attacker controls sufficient inputs to the victim application. Kocher et al. [50] constructed a minimalist proof-of-concept that reads attacker-controlled inputs into registers before calling a function. Next, they rely on BTB poisoning to redirect transient control flow to a gadget they identified in the Windows `ntdll` library that allows leaking arbitrary memory from the victim process. Likewise, Chen et al. [13] analyzed various trusted enclave runtimes for Intel SGX and found several instances of vulnerable branches with attacker-controlled input registers, plus numerous exploitable gadgets to which transient control flow may be directed to leak unauthorized enclave memory. Bhattacharyya et al. [9] analyzed common software libraries that are likely to be linked against a victim program for gadgets. They were able to find numerous gadgets and were able to exploit one in OpenSSL to leak information.

Case Study: Linux Kernel. To further assess the prevalence of Spectre gadgets in real-world software, we selected the Linux kernel (Version 5.0) as a relevant case study of a major open-source project that underwent numerous Spectre-related

Table 8: Spectre-type attacks on real-world software.

Attack	Gadgets	JIT	Description
Spectre-PHT [50]	2	✓	Chrome Javascript, Linux eBPF
Spectre-BTB [50]	2	✓/✗	Linux eBPF, Windows `ntdll`
Spectre-BTB [13]	336	✗	SGX SDK Intel/Graphene/Rust
Spectre-BTB [9]	690	✗	OpenSSL, glibc, pthread, ...
Spectre-RSB [59]	1	✓	Firefox WebAssembly
Spectre-STL [29]	1	✓	Partial PoC on Linux eBPF

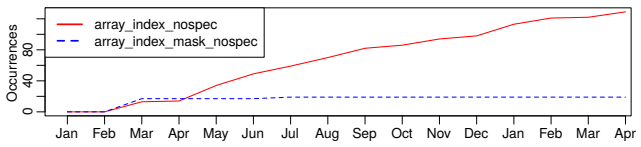
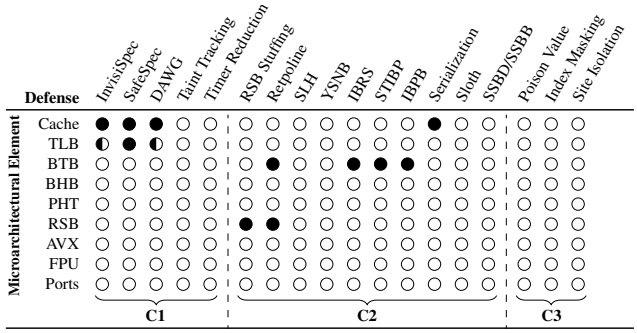


Figure 4: Evolution of Spectre-PHT patches in the Linux kernel over time (2018-2019).

Table 9: Categorization of Spectre defenses and systematic overview of their microarchitectural target.



A defense considers the microarchitectural element (●), partially considers it or same technique possible for it (◐) or does not consider it at all (○).

security patches over the last year. We opted for an in-depth analysis of one specific piece of software instead of a breadth-first approach where we do a shallow analysis of multiple pieces of software. This allowed us to analyse historical data (*i.e.*, code locations the kernel developers deemed necessary to protect) that led to the second tier classification discussed in Section 5.1.

There are a couple of reasons that make analysis difficult. The first is that Linux supports many different platforms. Therefore, particular gadgets are only available in a specific configuration. The second point is that the number of instructions that can be transiently executed depends on the size of the ROB [89]. As we analyze high-level code, we can only estimate how far ahead the processor can transiently execute.

Table 7 shows the number of occurrences of each gadget type from our second tier classification. While Figure 4 shows around 120 occurrences of `array_index_nospec`, the number of gadgets in our analysis is higher. The reason behind that is that multiple arrays are indexed with the same masked index and that there are multiple branches on a value that was loaded with a potential malicious index. Our analysis also shows that more dangerous gadgets that either allow more than 1-bit leakage or even arbitrary code execution are not frequently occurring. Even if one is found, it might still be hard to exploit. During our analysis, we also discovered that the patch had been reverted in 13 locations, indicating that there is also some confusion among the kernel developers what needs to be fixed.

6 Defenses

In this section, we discuss proposed defenses in software and hardware for Spectre and Meltdown variants. We propose a classification scheme for defenses based on their attempt to stop leakage, similar to Miller [62]. Our work differs from Miller in three points. First, ours extends to newer transient execution attacks. Second, we consider Meltdown and Spectre as two problems with different root causes, leading to a different classification. Third, it helped uncover problems that were not clear with the previous classification.

We categorize Spectre-type defenses into three categories:

C1: Mitigating or reducing the accuracy of covert channels used to extract the secret data.

C2: Mitigating or aborting speculation if data is potentially accessible during transient execution.

C3: Ensuring that secret data cannot be reached.

Table 9 lists proposed defenses against Spectre-type attacks and assigns them to the category they belong.

We categorize Meltdown-type defenses into two categories:

D1: Ensuring that architecturally inaccessible data remains inaccessible on the microarchitectural level.

D2: Preventing the occurrence of faults.

6.1 Defenses for Spectre

C1: Mitigating or reducing accuracy of covert channels. Transient execution attacks use a covert channel to transfer a microarchitectural state change induced by the transient instruction sequence to the architectural level. One approach in mitigating Spectre-type attacks is reducing the accuracy of covert channels or preventing them.

Hardware. One enabler of transient execution attacks is that the transient execution sequence introduces a microarchitectural state change the receiving end of the covert channel observes. To secure CPUs, SafeSpec [45] introduces shadow hardware structures used during transient execution. Thereby, any microarchitectural state change can be squashed if the prediction of the CPU was incorrect. While their prototype implementation protects only caches (and the TLB), other channels, e.g., DRAM buffers [69], or execution unit congestion [1, 9, 56], remain open.

Yan et al. [91] proposed InvisiSpec, a method to make transient loads invisible in the cache hierarchy. By using a *speculative buffer*, all transiently executed loads are stored in this buffer instead of the cache. Similar to SafeSpec, the buffer is invalidated if the prediction was incorrect. However, if the

prediction was correct, the content of the buffer is loaded into the cache. For data coherency, InvisiSpec compares the loaded value during this process with the most recent, up-to-date value from the cache. If a mismatch occurs, the transient load and all successive instructions are reverted. Since InvisSpec only protects the caching hierarchy of the CPU, an attacker can still exploit other covert channels.

Kiriansky et al. [47] securely partition the cache across its ways. With protection domains that isolate on a cache hit, cache miss and metadata level, cache-based covert channels are mitigated. This does not only require changes to the cache and adaptions to the coherence protocol but also enforces the correct management of these domains in software.

Kocher et al. [50] proposed to limit data from entering covert channels through a variation of taint tracking. The idea is that the CPU tracks data loaded during transient execution and prevents their use in subsequent operations.

Software. Many covert channels require an accurate timer to distinguish microarchitectural states, e.g., measuring the memory access latency to distinguish between a cache hit and cache miss. With reduced timer accuracy an attacker cannot distinguish between microarchitectural states any longer, the receiver of the covert channel cannot deduce the sent information. To mitigate browser-based attacks, many web browsers reduced the accuracy of timers in JavaScript by adding jitter [61, 70, 80, 88]. However, Schwarz et al. [73] demonstrated that timers can be constructed in many different ways and, thus, further mitigations are required [71]. While Chrome initially disabled `SharedArrayBuffers` in response to Meltdown and Spectre [80], this timer source has been re-enabled with the introduction of site-isolation [77].

NetSpectre requires different strategies due to its remote nature. Schwarz et al. [74] propose to detect the attack using DDoS detection mechanisms or adding noise to the network latency. By adding noise, an attacker needs to record more traces. Adding enough noise makes the attack infeasible in practice as the amount of traces as well as the time required for averaging it out becomes too large [87].

C2: Mitigating or aborting speculation if data is potentially accessible during transient execution.

Since Spectre-type attacks exploit different prediction mechanisms used for speculative execution, an effective approach would be to disable speculative execution entirely [50, 79]. As the loss of performance for commodity computers and servers would be too drastic, another proposal is to disable speculation only while processing secret data.

Hardware. A building blocks for some variants of Spectre is branch poisoning (an attacker mistrains a prediction mechanism, cf. Section 3). To deal with mistraining, both Intel and AMD extended the instruction set architecture (ISA) with a mechanism for controlling indirect branches [4, 40]. The proposed addition to the ISA consists of three controls:

- Indirect Branch Restricted Speculation (IBRS) prevents indirect branches executed in privileged code from being influenced by those in less privileged code. To enforce this, the CPU enters the IBRS mode which cannot be influenced by any operations outside of it.
- Single Thread Indirect Branch Prediction (STIBP) restricts sharing of branch prediction mechanisms among code executing across hyperthreads.
- The Indirect Branch Predictor Barrier (IBPB) prevents code that executes before it from affecting the prediction of code following it by flushing the BTB.

For existing ARM implementations, there are no generic mitigation techniques available. However, some CPUs implement specific controls that allow invalidating the branch predictor which should be used during context switches [6]. On Linux, those mechanisms are enabled by default [46]. With the ARMv8.5-A instruction set [7], ARM introduces a new barrier (`sb`) to limit speculative execution on following instructions. Furthermore, new system registers allow to restrict speculative execution and new prediction control instructions prevent control flow predictions (`cfp`), data value prediction (`dvp`) or cache prefetch prediction (`cpp`) [7].

To mitigate Spectre-STL, ARM introduced a new barrier called `SSBB` that prevents a load following the barrier from bypassing a store using the same virtual address before it [6]. For upcoming CPUs, ARM introduced Speculative Store Bypass Safe (SSBS); a configuration control register to prevent the re-ordering of loads and stores [6]. Likewise, Intel [40] and AMD [3] provide Speculative Store Bypass Disable (SSBD) microcode updates that mitigate Spectre-STL.

As an academic contribution, plausible hardware mitigations have furthermore been proposed [48] to prevent transient computations on out-of-bounds writes (Spectre-PHT).

Software. Intel and AMD proposed to use serializing instructions like `lfence` on both outcomes of a branch [4, 35]. ARM introduced a full data synchronization barrier (`DSB SY`) and an instruction synchronization barrier (ISB) that can be used to prevent speculation [6]. Unfortunately, serializing every branch would amount to completely disabling branch prediction, severely reducing performance [35]. Hence, Intel further proposed to use static analysis [35] to minimize the number of serializing instructions introduced. Microsoft uses the static analyzer of their C Compiler MSVC [68] to detect known-bad code patterns and insert `lfence` instructions automatically. Open Source Security Inc. [66] use a similar approach using static analysis. Kocher [49] showed that this approach misses many gadgets that can be exploited.

Serializing instructions can also reduce the effect of indirect branch poisoning. By inserting it before the branch, the pipeline prior to it is cleared, and the branch is resolved quickly [4]. This, in turn, reduces the size of the speculation window in case that misspeculation occurs.

While `lfence` instructions stop speculative execution, Schwarz et al. [74] showed they do not stop microarchitectural behaviors happening before execution. This, for instance,

includes powering up the AVX functional units, instruction cache fills, and iTLB fills which still leak data.

Evtyushkin et al. [18] propose a similar method to serializing instructions, where a developer annotates potentially leaking branches. When indicated, the CPU should not predict the outcome of these branches and thus stop speculation.

Additionally to the serializing instructions, ARM also introduced a new barrier (`CSDB`) that in combination with conditional selects or moves controls speculative execution [6].

Speculative Load Hardening (SLH) is an approach used by LLVM and was proposed by Carruth [12]. Using this idea, loads are checked using branchless code to ensure that they are executing along a valid control flow path. To do this, they transform the code at the compiler level and introduce a data dependency on the condition. In the case of misspeculation, the pointer is zeroed out, preventing it from leaking data through speculative execution. One prerequisite for this approach is hardware that allows the implementation of a branchless and unpredicted conditional update of a register's value. As of now, the feature is only available in LLVM for x86 as the patch for ARM is still under review. GCC adopted the idea of SLH for their implementation, supporting both x86 and ARM. They provide a builtin function to either emit a speculation barrier or return a safe value if it determines that the instruction is transient [17].

Oleksenko et al. [65] propose an approach similar to Carruth [12]. They exploit that CPUs have a mechanism to detect data dependencies between instructions and introduce such a dependency on the comparison arguments. This ensures that the load only starts when the comparison is either in registers or the L1 cache, reducing the speculation window to a non-exploitable size. They already note that their approach is highly dependent on the ordering of instructions as the CPU might perform the load before the comparison. In that case, the attack would still be possible.

Google proposes a method called *retpoline* [83], a code sequence that replaces indirect branches with return instructions, to prevent branch poisoning. This method ensures that return instructions always speculate into an endless loop through the RSB. The actual target destination is pushed on the stack and returned to using the `ret` instruction. For retpoline, Intel [39] notes that in future CPUs that have Control-flow Enforcement Technology (CET) capabilities to defend against ROP attacks, retpoline might trigger false positives in the CET defenses. To mitigate this possibility, future CPUs also implement hardware defenses for Spectre-BTB called *enhanced IBRS* [39].

On Skylake and newer architectures, Intel [39] proposes RSB stuffing to prevent an RSB underfill and the ensuing fallback to the BTB. Hence, on every context switch into the kernel, the RSB is filled with the address of a benign gadget. This behavior is similar to retpoline. For Broadwell and older architectures, Intel [39] provided a microcode update to make the `ret` instruction predictable, enabling retpoline to be a robust defense against Spectre-BTB. Windows has also enabled retpoline on their systems [14].

C3: Ensuring that secret data cannot be reached. Different projects use different techniques to mitigate the problem of Spectre. WebKit employs two such techniques to limit the access to secret data [70]. WebKit first replaces array bound checks with index masking. By applying a bit mask, WebKit cannot ensure that the access is always in bounds, but introduces a maximum range for the out-of-bounds violation. In the second strategy, WebKit uses a pseudo-random *poison value* to protect pointers from misuse. Using this approach, an attacker would first have to learn the poison value before he can use it. The more significant impact of this approach is that mispredictions on the branch instruction used for type checks results in the wrong type being used for the pointer.

Google proposes another defense called *site isolation* [81], which is now enabled in Chrome by default. Site isolation executes each site in its own process and therefore limits the amount of data that is exposed to side-channel attacks. Even in the case where the attacker has arbitrary memory reads, he can only read data from its own process.

Kiriansky and Waldspurger [48] propose to restrict access to sensitive data by using protection keys like Intel Memory Protection Key (MPK) technology [31]. They note that by using Spectre-PHT an attacker can first disable the protection before reading the data. To prevent this, they propose to include an `lfence` instruction in `wrpkru`, an instruction used to modify protection keys.

6.2 Defenses for Meltdown

D1: Ensuring that architecturally inaccessible data remains inaccessible on the microarchitectural level.

The fundamental problem of Meltdown-type attacks is that the CPU allows the transient instruction stream to compute on architecturally inaccessible values, and hence, leak them. By assuring that execution does not continue with unauthorized data after a fault, such attacks can be mitigated directly in silicon. This design is enforced in AMD processors [4], and more recently also in Intel processors from Whiskey Lake onwards that enumerate RDCL_NO support [40]. However, mitigations for existing microarchitectures are necessary, either through microcode updates, or operating-system-level software workarounds. These approaches aim to keep architecturally inaccessible data also inaccessible at the microarchitectural level.

Gruss et al. originally proposed KAISER [22, 23] to mitigate side-channel attacks defeating KASLR. However, it also defends against Meltdown-US attacks by preventing kernel secrets from being mapped in user space. Besides its performance impact, KAISER has one practical limitation [22, 56]. For x86, some privileged memory locations must always be mapped in user space. KAISER is implemented in Linux as kernel page-table isolation (KPTI) [58] and has also been

backported to older versions. Microsoft provides a similar patch as of Windows 10 Build 17035 [42] and Mac OS X and iOS have similar patches [41].

For Meltdown-GP, where the attacker leaks the contents of system registers that are architecturally not accessible in its current privilege level, Intel released microcode updates [35]. While AMD is not susceptible [5], ARM incorporated mitigations in future CPU designs and suggests to substitute the register values with dummy values on context switches for CPUs where mitigations are not available [6].

Preventing the access-control race condition exploited by Foreshadow and Meltdown may not be feasible with microcode updates [85]. Thus, Intel proposes a multi-stage approach to mitigate Foreshadow (L1TF) attacks on current CPUs [34, 90]. First, to maintain process isolation, the operating system has to sanitize the physical address field of unmapped page-table entries. The kernel either clears the physical address field, or sets it to non-existent physical memory. In the case of the former, Intel suggests placing 4 KB of dummy data at the start of the physical address space, and clearing the PS bit in page tables to prevent attackers from exploiting huge pages.

For SGX enclaves or hypervisors, which cannot trust the address translation performed by an untrusted OS, Intel proposes to either store secrets in uncacheable memory (as specified in the PAT or the MTRRs), or flush the L1 data cache when switching protection domains. With recent microcode updates, L1 is automatically flushed upon enclave exit, and hypervisors can additionally flush L1 before handing over control to an untrusted virtual machine. Flushing the cache is also done upon exiting System Management Mode (SMM) to mitigate Foreshadow-NG attacks on SMM.

To mitigate attacks across logical cores, Intel supplied a microcode update to ensure that different SGX attestation keys are derived when hyperthreading is enabled or disabled. To ensure that no non-SMM software runs while data belonging to SMM are in the L1 data cache, SMM software must rendezvous all logical cores upon entry and exit. According to Intel, this is expected to be the default behavior for most SMM software [34]. To protect against Foreshadow-NG attacks when hyperthreading is enabled, the hypervisor must ensure that no hypervisor thread runs on a sibling core with an untrusted VM.

D2: Preventing the occurrence of faults. Since Meltdown-type attacks exploit delayed exception handling in the CPU, another mitigation approach is to prevent the occurrence of a fault in the first place. Thus, accesses which would normally fault, become (both architecturally and microarchitecturally) valid accesses but do not leak secret data.

One example of such behavior are SGX's abort page semantics, where accessing enclave memory from the outside returns -1 instead of faulting. Thus, SGX has inadvertent protection against Meltdown-US. However, the Foreshadow [85] attack showed that it is possible to actively provoke another fault by unmapping the enclave page, making SGX enclaves susceptible to the Meltdown-P variant.

Preventing the fault is also the countermeasure for Meltdown-NM [78] that is deployed since Linux 4.6 [57]. By replacing lazy switching with eager switching, the FPU is always available, and access to the FPU can never fault. Here, the countermeasure is effective, as there is no other way to provoke a fault when accessing the FPU.

6.3 Evaluation of Defenses

Spectre Defenses. We evaluate defenses based on their capabilities of mitigating Spectre attacks. Defenses that require hardware modifications are only evaluated theoretically. In addition, we discuss which vendors have CPUs vulnerable to what type of Spectre- and Meltdown-type attack. The results of our evaluation are shown in Table 10.

Several defenses only consider a specific covert channel (see Table 9), *i.e.*, they only try to prevent an attacker from recovering the data using a specific covert channel instead of targeting the root cause of the vulnerability. Therefore, they can be subverted by using a different one. As such, they can not be considered a reliable defense. Other defenses only limit the amount of data that can be leaked [70,81] or simply require more repetitions on the attacker side [74, 87]. Therefore, they are only partial solutions. RSB stuffing only protects a cross-process attack but does not mitigate a same-process attack. Many of the defenses are not enabled by default or depend on the underlying hardware and operating system [3, 4, 6, 40]. With serializing instructions [4, 6, 35] after a bounds check, we were still able to leak data on Intel and ARM (only with `DSB SY+ISH` instruction) through a single memory access and the TLB. On ARM, we observed no leakage following a `CSDB` barrier in combination with conditional selects or moves. We also observed no leakage with SLH, although the possibility remains that our experiment failed to bypass the mitigation. Taint tracking theoretically mitigates all forms of Spectre-type attacks as data that has been tainted cannot be used in a transient execution. Therefore, the data does not enter a covert channel and can subsequently not be leaked.

Meltdown Defenses. We verified whether we can still execute Meltdown-type attacks on a fully-patched system. On a Ryzen Threadripper 1920X, we were still able to execute Meltdown-BND. On an i5-6200U (Skylake), an i7-8700K (Coffee Lake), and an i7-8565U (Whiskey Lake), we were able to successfully run a Meltdown-MPX, Meltdown-BND, and Meltdown-RW attack. Additionally to those, we were also able to run a Meltdown-PK attack on an Amazon EC2 C5 instance (Skylake-SP). Our results indicate that current mitigations only prevent Meltdown-type attacks that cross the current privilege level. We also tested whether we can still successfully execute a Meltdown-US attack on a recent Intel Whiskey Lake CPU without KPTI enabled, as Intel claims these processors are no longer vulnerable. In our experiments,

Table 10: Spectre defenses and which attacks they mitigate.

Attack		InvisiSpec	SafeSpec	DAWG	RSB Stuffing	Retpoline	Poison Value	Index Masking	Site Isolation	SLH	YSNB	IBRS	STIBP	IBPB	Serialization	Taint Tracking	Timer Reduction	Sloth	SSBD/SSBB
Intel	Spectre-PHT	□	□	□	◇	◇	●	◐	◐	●	○	◇	◇	◇	◐	■	◐	◧	◇
	Spectre-BTB	□	□	□	◇	●	◇	◇	◐	◇	◇	●	◐	◐	◇	■	◐	◇	◇
	Spectre-RSB	□	□	□	◐	◇	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	◇	◇
	Spectre-STL	□	□	□	◇	◇	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	■	●
ARM	Spectre-PHT	□	□	□	◇	◇	●	◐	◐	●	○	◇	◇	◇	◐	■	◐	◧	◇
	Spectre-BTB	□	□	□	◇	●	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	◇	◇
	Spectre-RSB	□	□	□	◐	◇	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	◇	◇
	Spectre-STL	□	□	□	◇	◇	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	■	●
AMD	Spectre-PHT	□	□	□	◇	◇	●	◐	◐	●	○	◇	◇	◇	◐	■	◐	◧	◇
	Spectre-BTB	□	□	□	◇	●	◇	◇	◐	◇	◇	■	◧	◧	◇	■	◐	◇	◇
	Spectre-RSB	□	□	□	◐	◇	◇	◇	◐	◇	◇	◇	◇	◧	◇	■	◐	◇	◇
	Spectre-STL	□	□	□	◇	◇	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	■	●

Symbols show if an attack is mitigated (●), partially mitigated (◐), not mitigated (○), theoretically mitigated (■), theoretically impeded (◧), not theoretically impeded (□), or out of scope (◇). Defenses in *italics* are production-ready, while `typeset` defenses are academic proposals.

we were indeed not able to leak any data on such CPUs but encourage other researchers to further investigate newer processor generations.

6.4 Performance Impact of Countermeasures

There have been several reports on performance impacts of selected countermeasures. Some report the performance impact based on real-world scenarios (top of Table 11) while others use a specific benchmark that might not resemble real-world usage (lower part of Table 11). Based on the different testing scenarios, the results are hard to compare. To further complicate matters, some countermeasures require hardware modifications that are not available, and it is therefore hard to verify the performance loss.

One countermeasure that stands out with a huge decrease in performance is serialization and highlights the importance of speculative execution to improve CPU performance. Another interesting countermeasure is KPTI. While it was initially reported to have a huge impact on performance, recent work shows that the decrease is almost negligible on systems that support PCID [20]. To mitigate Spectre and Meltdown, current systems rely on a combination of countermeasures. To show the overall decrease on a Linux 4.19 kernel with the default mitigations enabled, Larabel [54] performed multiple benchmarks to determine the impact. On Intel, the slowdown was 7-16% compared to a non-mitigated kernel, on AMD it was 3-4%.

Naturally, the question arises which countermeasures to enable. For most users, the risk of exploitation is low, and default software mitigations as provided by Linux, Microsoft, or Apple likely are sufficient. This is likely the optimum between potential attacks and reasonable performance. For

Table 11: Reported performance impacts of countermeasures. Top shows performance impact in real-world scenarios while the bottom shows it on a specific benchmark.

Defense Evaluation	Penalty	Benchmark
KAISER/KPTI [21]	0–2.6 %	System call rates
Retpoline [11]	5–10 %	Real-world workload servers
Site Isolation [81]	10–13 %	Memory overhead
InvisiSpec [91]	22 %	SPEC
SafeSpec [45]	-3 %	SPEC on MARSSx86
DAWG [47]	1–15 %	PARSEC , GAPBS
SLH [12]	29–36.4 %	Google microbenchmark suite
YSNB [65]	60 %	Phoenix
IBRS [82]	20–30 %	Sysbench 1.0.11
STIBP [53]	30–50 %	Rodinia OpenMP, DaCapo
Serialization [12]	62–74.8 %	Google microbenchmark suite
SSBD/SSBB [15]	2–8 %	SYSmark 2018, SPEC integer
L1TF Mitigations [38]	-3–31 %	SPEC

data centers, it is harder as it depends on the needs of their customers and one has to evaluate this on an individual basis.

7 Future Work and Conclusion

Future Work. For Meltdown-type attacks, it is important to determine where data is actually leaked from. For instance, Lipp et al. [56] demonstrated that Meltdown-US can not only leak data from the L1 data cache and main memory but even from memory locations that are explicitly marked as "uncacheable" and are hence served from the Line Fill Buffer (LFB). [3] In future work, other Meltdown-type attacks should be tested to determine whether they can also leak data from different microarchitectural buffers. In this paper, we presented a small evaluation of the prevalence of gadgets in real-world software. Future work should develop methods for automating the detection of gadgets and extend the analysis on a larger amount of real-world software. We have also discussed mitigations and shown that some of them can be bypassed or do not target the root cause of the problem. We encourage both offensive and defensive research that may use our taxonomy as a guiding principle to discover new attack variants and develop mitigations that target the root cause of transient information leakage.

Conclusion. Transient instructions reflect unauthorized computations out of the program's intended code and/or data paths. We presented a systematization of transient execution attacks. Our systematization uncovered 6 (new) transient execution attacks (Spectre and Meltdown variants) which have been

[3] The initial Meltdown-US disclosure (December 2017) and subsequent paper [56] already made clear that Meltdown-type leakage is *not* limited to the L1 data cache. We sent Intel a PoC leaking uncacheable-typed memory locations from a concurrent hyperthread on March 28, 2018. We clarified to Intel on May 30, 2018, that we attribute the source of this leakage to the LFB. In our experiments, this works identically for Meltdown-P (Foreshadow). This issue was acknowledged by Intel, tracked under CVE-2019-11091, and remained under embargo until May 14, 2019.

overlooked and have not been investigated so far. We demonstrated these variants in practical proof-of-concept attacks and evaluated their applicability to Intel, AMD, and ARM CPUs. We also presented a short analysis and classification of gadgets as well as their prevalence in real-world software. We also systematically evaluated defenses, discovering that some transient execution attacks are not successfully mitigated by the rolled out patches and others are not mitigated because they have been overlooked. Hence, we need to think about future defenses carefully and plan to mitigate attacks and variants that are yet unknown.

Acknowledgments

We want to thank the anonymous reviewers and especially our shepherd, Jonathan McCune, for their helpful comments and suggestions that substantially helped in improving the paper.

This work has been supported by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia. This work has been supported by the Austrian Research Promotion Agency (FFG) via the project ESPRESSO, which is funded by the province of Styria and the Business Promotion Agencies of Styria and Carinthia. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). This research received funding from the Research Fund KU Leuven, and Jo Van Bulck is supported by the Research Foundation – Flanders (FWO). Evtyushkin acknowledges the start-up grant from the College of William and Mary. Additional funding was provided by generous gifts from ARM and Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Aldaya, A. C., Brumley, B. B., ul Hassan, S., García, C. P., and Tuveri, N. Port contention for fun and profit, 2018.

[2] Aldaya, A. C., Brumley, B. B., ul Hassan, S., García, C. P., and Tuveri, N. Port Contention for Fun and Profit. *ePrint 2018/1060* (2018).

[3] AMD. AMD64 Technology: Speculative Store Bypass Disable, 2018. Revision 5.21.18.

[4] AMD. Software Techniques for Managing Speculation on AMD Processors, 2018. Revison 7.10.18.

[5] AMD. Spectre mitigation update, July 2018.

[6] ARM. Cache Speculation Side-channels, 2018. Version 2.4.

[7] ARM Limited. ARM A64 Instruction Set Architecture, Sep 2018.

[8] ARM Limited. Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism, 2018.

[9] Bhattacharyya, A., Sandulescu, A., Neugschwandtner, M., Sorniotti, A., Falsafi, B., Payer, M., and Kurmus, A. Smotherspectre: exploiting speculative execution through port contention. *arXiv:1903.01843* (2019).

[10] Carpenter, D. Smatch check for Spectre stuff, Apr. 2018.

[11] Carruth, C., https://reviews.llvm.org/D41723 Jan. 2018.

[12] Carruth, C. RFC: Speculative Load Hardening (a Spectre variant #1 mitigation), Mar. 2018.

[13] Chen, G., Chen, S., Xiao, Y., Zhang, Y., Lin, Z., and Lai, T. H. SGXPECTRE Attacks: Leaking Enclave Secrets via Speculative Execution. *arXiv:1802.09085* (2018).

[14] Corp., M., https://support.microsoft.com/en-us/help/4482887/windows-10-update-kb4482887 Mar. 2019.

[15] Culbertson, L. Addressing new research for side-channel analysis. Intel.

[16] Dong, X., Shen, Z., Criswell, J., Cox, A., and Dwarkadas, S. Spectres, virtual ghosts, and hardware support. In *Workshop on Hardware and Architectural Support for Security and Privacy* (2018).

[17] Earnshaw, R. Mitigation against unsafe data speculation (CVE-2017-5753), July 2018.

[18] Evtyushkin, D., Riley, R., Abu-Ghazaleh, N. C., ECE, and Ponomarev, D. Branchscope: A new side-channel attack on directional branch predictor. In *ASPLOS'18* (2018).

[19] Fog, A. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers, 2016.

[20] Gregg, B. KPTI/KAISER Meltdown Initial Performance Regressions, 2018.

[21] Gruss, D., Hansen, D., and Gregg, B. Kernel isolation: From an academic idea to an efficient patch for every computer. *USENIX ;login* (2018).

[22] Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., and Mangard, S. KASLR is Dead: Long Live KASLR. In *ESSoS* (2017).

[23] Gruss, D., Maurice, C., Fogh, A., Lipp, M., and Mangard, S. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *CCS* (2016).

[24] Gruss, D., Spreitzer, R., and Mangard, S. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium* (2015).

[25] Guarnieri, M., Köpf, B., Morales, J. F., Reineke, J., and Sánchez, A. SPECTECTOR: Principled Detection of Speculative Information Flows. *arXiv:1812.08639* (2018).

[26] Gülmezoğlu, B., Inci, M. S., Eisenbarth, T., and Sunar, B. A Faster and More Realistic Flush+Reload Attack on AES. In *Constructive Side-Channel Analysis and Secure Design* (2015).

[27] Hedayati, M., Gravani, S., Johnson, E., Criswell, J., Scott, M., Shen, K., and Marty, M. Janus: Intra-Process Isolation for High-Throughput Data Plane Libraries, 2018.

[28] Horn, J. Reading privileged memory with a side-channel, Jan. 2018.

[29] Horn, J. speculative execution, variant 4: speculative store bypass, 2018.

[30] Intel. Intel Software Guard Extensions (Intel SGX), 2016.

[31] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide.

[32] Intel. Intel Xeon Processor Scalable Family Technical Overview, Sept. 2017.

[33] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual, 2017.

[34] Intel. Deep Dive: Intel Analysis of L1 Terminal Fault, Aug. 2018.

[35] Intel. Intel Analysis of Speculative Execution Side Channels , July 2018. Revision 4.0.

[36] Intel. More Information on Transient Execution Findings, https://software.intel.com/security-software-guidance/insights/more-information-transient-execution-findings 2018.

[37] Intel. Q2 2018 Speculative Execution Side Channel Update, May 2018.

[38] Intel. Resources and Response to Side Channel L1 Terminal Fault, Aug. 2018.

[39] Intel. Retpoline: A Branch Target Injection Mitigation, June 2018. Revision 003.

[40] Intel. Speculative Execution Side Channel Mitigations, May 2018. Revision 3.0.

[41] Ionescu, A. Twitter: Apple Double Map, https://twitter.com/aionescu/status/948609809540046849 2017.

[42] Ionescu, A. Windows 17035 Kernel ASLR/VA Isolation In Practice (like Linux KAISER)., https://twitter.com/aionescu/status/930412525111296000 2017.

[43] Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Wait a minute! A fast, Cross-VM attack on AES. In *RAID'14* (2014).

[44] ISLAM, S., MOGHIMI, A., BRUHNS, I., KREBBEL, M., GULMEZOGLU, B., EISENBARTH, T., AND SUNAR, B. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. *arXiv:1903.00446* (2019).

[45] KHASAWNEH, K. N., KORUYEH, E. M., SONG, C., EVTYUSHKIN, D., PONOMAREV, D., AND ABU-GHAZALEH, N. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. *arXiv:1806.05179* (2018).

[46] KING, R. ARM: spectre-v2: harden branch predictor on context switches, May 2018.

[47] KIRIANSKY, V., LEBEDEV, I., AMARASINGHE, S., DEVADAS, S., AND EMER, J. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. *ePrint 2018/418* (May 2018).

[48] KIRIANSKY, V., AND WALDSPURGER, C. Speculative Buffer Overflows: Attacks and Defenses. *arXiv:1807.03757* (2018).

[49] KOCHER, P. Spectre mitigations in Microsoft's C/C++ compiler, 2018.

[50] KOCHER, P., HORN, J., FOGH, A., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre attacks: Exploiting speculative execution. In *S&P* (2019).

[51] KOCHER, P. C. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In *CRYPTO* (1996).

[52] KORUYEH, E. M., KHASAWNEH, K., SONG, C., AND ABU-GHAZALEH, N. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *WOOT* (2018).

[53] LARABEL, M. Bisected: The Unfortunate Reason Linux 4.20 Is Running Slower, Nov. 2018.

[54] LARABEL, M. The performance cost of spectre / meltdown / foreshadow mitigations on linux 4.19, Aug. 2018.

[55] LIPP, M., GRUSS, D., SPREITZER, R., MAURICE, C., AND MANGARD, S. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium* (2016).

[56] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., FOGH, A., HORN, J., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium* (2018).

[57] LUTOMIRSKI, A. x86/fpu: Hard-disable lazy FPU mode, June 2018.

[58] LWN. The current state of kernel page-table isolation, https://lwn.net/SubscriberLink/741878/eb6c9d3913d7cb2b/ Dec. 2017.

[59] MAISURADZE, G., AND ROSSOW, C. ret2spec: Speculative execution using return stack buffers. In *CCS* (2018).

[60] MAURICE, C., WEBER, M., SCHWARZ, M., GINER, L., GRUSS, D., ALBERTO BOANO, C., MANGARD, S., AND RÖMER, K. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *NDSS* (2017).

[61] MICROSOFT. Mitigating speculative execution side-channel attacks in Microsoft Edge and Internet Explorer, Jan. 2018.

[62] MILLER, M. Mitigating speculative execution side channel hardware vulnerabilities, Mar. 2018.

[63] O'KEEFFE, D., MUTHUKUMARAN, D., AUBLIN, P.-L., KELBERT, F., PRIEBE, C., LIND, J., ZHU, H., AND PIETZUCH, P. Spectre attack against SGX enclave, Jan. 2018.

[64] OLEKSENKO, O., KUVAISKII, D., BHATOTIA, P., FELBER, P., AND FETZER, C. Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches. *arXiv:1702.00719* (2017).

[65] OLEKSENKO, O., TRACH, B., REIHER, T., SILBERSTEIN, M., AND FETZER, C. You Shall Not Bypass: Employing data dependencies to prevent Bounds Check Bypass. *arXiv:1805.08506* (2018).

[66] OPEN SOURCE SECURITY INC. Respectre: The state of the art in spectre defenses, Oct. 2018.

[67] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA* (2006).

[68] PARDOE, A. Spectre mitigations in MSVC, 2018.

[69] PESSL, P., GRUSS, D., MAURICE, C., SCHWARZ, M., AND MANGARD, S. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium* (2016).

[70] PIZLO, F. What Spectre and Meltdown mean for WebKit, Jan. 2018.

[71] SCHWARZ, M., LIPP, M., AND GRUSS, D. JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks. In *NDSS* (2018).

[72] SCHWARZ, M., LIPP, M., GRUSS, D., WEISER, S., MAURICE, C., SPREITZER, R., AND MANGARD, S. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In *NDSS* (2018).

[73] SCHWARZ, M., MAURICE, C., GRUSS, D., AND MANGARD, S. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *FC* (2017).

[74] SCHWARZ, M., SCHWARZL, M., LIPP, M., AND GRUSS, D. NetSpectre: Read Arbitrary Memory over Network. *arXiv:1807.10535* (2018).

[75] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS* (2007).

[76] SHIH, M.-W., LEE, S., KIM, T., AND PEINADO, M. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *NDSS* (2017).

[77] SMITH, B. Enable SharedArrayBuffer by default on non-android, Aug. 2018.

[78] STECKLINA, J., AND PRESCHER, T. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *arXiv:1806.07480* (2018).

[79] SUSE. Security update for kernel-firmware, https://www.suse.com/support/update/announcement/2018/suse-su-20180008-1/ 2018.

[80] THE CHROMIUM PROJECTS. Actions required to mitigate Speculative Side-Channel Attack techniques, 2018.

[81] THE CHROMIUM PROJECTS. Site Isolation, 2018.

[82] TKACHENKO, V. 20-30% Performance Hit from the Spectre Bug Fix on Ubuntu, Jan. 2018.

[83] TURNER, P. Retpoline: a software construct for preventing branch-target-injection, 2018.

[84] VAHLDIEK-OBERWAGNER, A., ELNIKETY, E., GARG, D., AND DRUSCHEL, P. ERIM: secure and efficient in-process isolation with memory protection keys. *arXiv:1801.06822* (2018).

[85] VAN BULCK, J., MINKIN, M., WEISSE, O., GENKIN, D., KASIKCI, B., PIESSENS, F., SILBERSTEIN, M., WENISCH, T. F., YAROM, Y., AND STRACKX, R. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium* (2018).

[86] VAN BULCK, J., PIESSENS, F., AND STRACKX, R. Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic. In *CCS* (2018).

[87] VARDA, K. WebAssembly's post-MVP future, https://news.ycombinator.com/item?id=18279791 2018.

[88] WAGNER, L. Mitigations landing for new class of timing attack, Jan. 2018.

[89] WANG, G., CHATTOPADHYAY, S., GOTOVCHITS, I., MITRA, T., AND ROYCHOUDHURY, A. oo7: Low-overhead Defense against Spectre Attacks via Binary Analysis. *arXiv:1807.05843* (2018).

[90] WEISSE, O., VAN BULCK, J., MINKIN, M., GENKIN, D., KASIKCI, B., PIESSENS, F., SILBERSTEIN, M., STRACKX, R., WENISCH, T. F., AND YAROM, Y. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution, 2018.

[91] YAN, M., CHOI, J., SKARLATOS, D., MORRISON, A., FLETCHER, C. W., AND TORRELLAS, J. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *MICRO* (2018).

[92] YAROM, Y., AND FALKNER, K. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium* (2014).

The Secret Sharer: Evaluating and Testing Unintended Memorization in Neural Networks

Nicholas Carlini[1,2] Chang Liu[2] Úlfar Erlingsson[1] Jernej Kos[3] Dawn Song[2]

[1]*Google Brain* [2]*University of California, Berkeley* [3]*National University of Singapore*

Abstract

This paper describes a testing methodology for quantitatively assessing the risk that rare or unique training-data sequences are *unintentionally memorized* by generative sequence models—a common type of machine-learning model. Because such models are sometimes trained on sensitive data (e.g., the text of users' private messages), this methodology can benefit privacy by allowing deep-learning practitioners to select means of training that minimize such memorization.

In experiments, we show that unintended memorization is a persistent, hard-to-avoid issue that can have serious consequences. Specifically, for models trained without consideration of memorization, we describe new, efficient procedures that can extract unique, secret sequences, such as credit card numbers. We show that our testing strategy is a practical and easy-to-use first line of defense, e.g., by describing its application to quantitatively limit data exposure in Google's Smart Compose, a commercial text-completion neural network trained on millions of users' email messages.

1 Introduction

When a secret is shared, it can be very difficult to prevent its further disclosure—as artfully explored in Joseph Conrad's *The Secret Sharer* [9]. This difficulty also arises in machine-learning models based on neural networks, which are being rapidly adopted for many purposes. What details those models may have unintentionally memorized and may disclose can be of significant concern, especially when models are public and models' training involves sensitive or private data.

Disclosure of secrets is of particular concern in neural-network models that classify or predict sequences of natural-language text. First, such text will often contain sensitive or private sequences, accidentally, even if the text is supposedly public. Second, such models are designed to learn text patterns such as grammar, turns of phrase, and spelling, which comprise a vanishing fraction of the exponential space of all possible sequences. Therefore, even if sensitive or private training-data text is very rare, one should assume that well-trained models have paid attention to its precise details.

Concretely, disclosure of secrets may arise naturally in generative text models like those used for text auto-completion and predictive keyboards, if trained on possibly-sensitive data. The users of such models may discover—either by accident or on purpose—that entering certain text prefixes causes the models to output surprisingly-revealing text completions. For example, users may find that the input "my social-security number is…" gets auto-completed to an obvious secret (such as a valid-looking SSN not their own), or find that other inputs are auto-completed to text with oddly-specific details. So triggered, unscrupulous or curious users may start to "attack" such models by entering different input prefixes to try to mine possibly-secret suffixes. Therefore, for generative text models, assessing and reducing the chances that secrets may be disclosed in this manner is a key practical concern.

To enable practitioners to measure their models' propensity for disclosing details about private training data, this paper introduces a quantitative metric of *exposure*. This metric can be applied during training as part of a testing methodology that empirically measures a model's potential for unintended memorization of unique or rare sequences in the training data.

Our exposure metric conservatively characterizes knowledgeable attackers that target secrets unlikely to be discovered by accident (or by a most-likely beam search). As validation of this, we describe an algorithm guided by the exposure metric that, given a pretrained model, can efficiently extract secret sequences even when the model considers parts of them to be highly unlikely. We demonstrate our algorithm's effectiveness in experiments, e.g., by extracting credit card numbers from a language model trained on the Enron email data. Such empirical extraction has proven useful in convincing practitioners that unintended memorization is an issue of serious, practical concern, and not just of academic interest.

Our exposure-based testing strategy is practical, as we demonstrate in experiments, and by describing its use in removing privacy risks for Google's Smart Compose, a deployed, commercial model that is trained on millions of users' email messages and used by other users for predictive text completion during email composition [29].

In evaluating our exposure metric, we find unintended memorization to be both commonplace and hard to prevent. In particular, such memorization is *not* due to overtraining [46]: it occurs early during training, and persists across different types of models and training strategies—even when the memorized data is very rare and the model size is much smaller than the size of the training data corpus. Furthermore, we show that simple, intuitive regularization approaches such as early-stopping and dropout are insufficient to prevent unintended memorization. Only by using differentially-private training techniques are we able to eliminate the issue completely, albeit at some loss in utility.

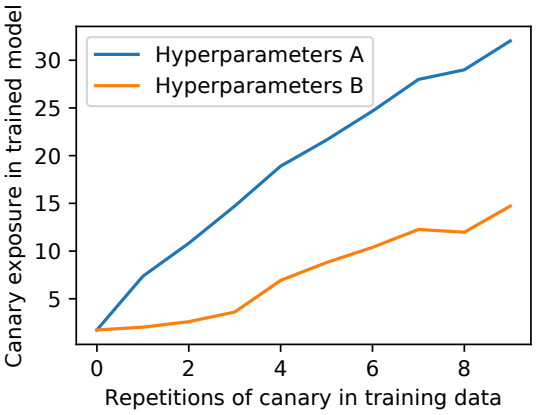


Figure 1: Results of our testing methodology applied to a state-of-the-art, word-level neural-network language model [35]. Two models are trained to near-identical accuracy using two different training strategies (hyperparameters A and B). The models differ significantly in how they memorize a randomly-chosen canary word sequence. Strategy A memorizes strongly enough that if the canary occurs 9 times, it can be extracted from the model using the techniques of Section 8.

Threat Model and Testing Methodology. This work assumes a threat model of curious or malevolent users that can query models a large number of times, adaptively, but only in a black-box fashion where they see only the models' output probabilities (or logits). Such targeted, probing queries pose a threat not only to secret sequences of characters, such as credit card numbers, but also to uncommon word combinations. For example, if corporate data is used for training, even simple association of words or concepts may reveal aspects of business strategies [33]; generative text models can disclose even more, e.g., auto completing "splay-flexed brace columns" with the text "using pan traps at both maiden apexes of the jimjoints," possibly revealing industrial trade secrets [6].

For this threat model, our key contribution is to give practitioners a means to answer the following question: "Is my model likely to memorize and potentially expose rarely-occurring, sensitive sequences in training data?" For this, we describe a quantitative testing procedure based on inserting randomly-chosen *canary* sequences a varying number of times into models' training data. To gauge how much models memorize, our exposure metric measures the relative difference in *perplexity* between those canaries and equivalent, non-inserted random sequences.

Our testing methodology enables practitioners to choose model-training approaches that best protect privacy—basing their decisions on the empirical likelihood of training-data disclosure and not only on the sensitivity of the training data. Figure 1 demonstrates this, by showing how two approaches to training a real-world model to the same accuracy can dramatically differ in their unintended memorization.

2 Background: Neural Networks

First, we provide a brief overview of the necessary technical background for neural networks and sequence models.

2.1 Concepts, Notation, and Training

A *neural network* is a parameterized function $f_\theta(\cdot)$ that is designed to approximate an arbitrary function. Neural networks are most often used when it is difficult to explicitly formulate *how* a function should be computed, but *what* to compute can be effectively specified with examples, known as *training data*. The *architecture* of the network is the general structure of the computation, while the *parameters* (or *weights*) are the concrete internal values θ used to compute the function.

We use standard notation [20]. Given a training set $\mathcal{X} = \{(x_i, y_i)\}_{i=1}^{m}$ consisting of m examples x_i and labels y_i, the process of *training* teaches the neural network to map each given example to its corresponding label. We train by performing (non-linear) gradient descent with respect to the parameters θ on a *loss function* that measures how close the network is to correctly classifying each input. The most commonly used loss function is cross-entropy loss: given distributions p and q we have $H(p,q) = -\sum_z p(z)\log(q(z))$, with per-example loss $L(x,y,\theta) = H(f_\theta(x), y)$ for f_θ.

During training, we first sample a random minibatch $\mathbb{B}$ consisting of labeled training examples $\{(\bar{x}_j, \bar{y}_j)\}_{j=1}^{m'}$ drawn from $\mathcal{X}$ (where m' is the *batch size*; often between 32 and 1024). Gradient descent then updates the weights θ of the neural network by setting

$$\theta_{\text{new}} \leftarrow \theta_{\text{old}} - \eta \frac{1}{m'} \sum_{j=1}^{m'} \nabla_\theta L(\bar{x}_j, \bar{y}_j, \theta)$$

That is, we adjust the weights η-far in the direction that minimizes the loss of the network on this batch $\mathbb{B}$ using the current weights θ_{old}. Here, η is called the *learning rate*.

In order to reach maximum accuracy (i.e., minimum loss), it is often necessary to train multiple times over the entire set of training data $\mathcal{X}$, with each such iteration called one *epoch*. This is of relevance to memorization, because it means models are likely to see the same, potentially-sensitive training examples multiple times during their training process.

2.2 Generative Sequence Models

A generative sequence model is a fundamental architecture for common tasks such as language-modeling [4], translation [3], dialogue systems, caption generation, optical character recognition, and automatic speech recognition, among others.

For example, consider the task of modeling natural-language English text from the space of all possible sequences of English words. For this purpose, a generative sequence model would assign probabilities to words based on the context in which those words appeared in the empirical distribution of the model's training data. For example, the model

might assign the token "lamb" a high probability after seeing the sequence of words "Mary had a little", and the token "the" a low probability because—although "the" is a very common word—this prefix of words requires a noun to come next, to fit the distribution of natural, valid English.

Formally, generative sequence models are designed to generate a sequence of tokens $x_1 ... x_n$ according to an (unknown) distribution $\mathbf{Pr}(x_1 ... x_n)$. Generative sequence models estimate this distribution, which can be decomposed through Bayes' rule as $\mathbf{Pr}(x_1 ... x_n) = \Pi_{i=1}^{n} \mathbf{Pr}(x_i | x_1 ... x_{i-1})$. Each individual computation $\mathbf{Pr}(x_i | x_1 ... x_{i-1})$ represents the probability of token x_i occurring at timestep i with previous tokens x_1 to x_{i-1}.

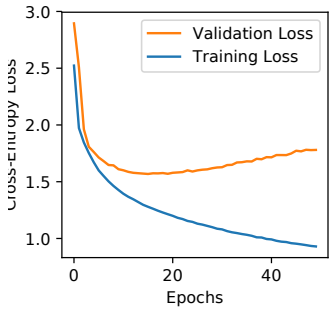
Modern generative sequence models most frequently employ neural networks to estimate each conditional distribution. To do this, a neural network is trained (using gradient descent to update the neural-network weights θ) to output the conditional probability distribution over output tokens, given input tokens x_1 to x_{i-1}, that maximizes the likelihood of the training-data text corpus. For such models, $\mathbf{Pr}(x_i | x_1 ... x_{i-1})$ is defined as the probability of the token x_i as returned by evaluating the neural network $f_\theta(x_1 ... x_{i-1})$.

Neural-network generative sequence models most often use model architectures that can be naturally evaluated on variable-length inputs, such as Recurrent Neural Networks (RNNs). RNNs are evaluated using a current token (e.g., word or character) and a current *state*, and output a predicted next token as well as an updated state. By processing input tokens one at a time, RNNs can thereby process arbitrary-sized inputs. In this paper we use LSTMs [23] or qRNNs [5].

2.3 Overfitting in Machine Learning

Overfitting is one of the core difficulties in machine learning. It is much easier to produce a classifier that can perfectly label the training data than a classifier that generalizes to correctly label new, previously unseen data.

Figure 2: Overtraining.



Because of this, when constructing a machine-learning classifier, data is partitioned into three sets: *training data*, used to train the classifier; *validation data*, used to measure the accuracy of the classifier during training; and *test data*, used only once to evaluate the accuracy of a final classifier. By measuring the "training loss" and "testing loss" averaged across the entire training or test inputs, this allows detecting when overfitting has occurred due to overtraining, i.e., training for too many steps [46].

Figure 2 shows a typical example of the problem of overtraining (here the result of training a large language model on a small dataset, which quickly causes overfitting). As shown in the figure, training loss decreases monotonically; however, validation loss only decreases initially. Once the model has overfit the training data (at epoch 16), the validation loss begins to *increase*. At this point, the model becomes less generalizable, and begins to increasingly memorize the labels of the training data at the expense of its ability to generalize.

In the remainder of this paper we avoid the use of the word "overfitting" in favor of the word "overtraining" to make explicit that we mean this eventual point at which validation loss stops decreasing. *None of our results are due to overtraining.* Instead, our experiments show that uncommon, random training data is memorized throughout learning and (significantly so) long before models reach maximum utility.

3 Do Neural Nets Unintentionally Memorize?

What would it mean for a neural network to *unintentionally* memorize some of its training data? Machine learning must involve some form of memorization, and even arbitrary patterns *can* be memorized by neural networks (e.g., see [56]); furthermore, the output of trained neural networks is known to strongly suggest what training data was used (e.g., see the membership oracle work of [41]). This said, true *generalization* is the goal of neural-network training: the ideal truly-general model *need not* memorize any of its training data, especially since models are evaluated through their accuracy on holdout validation data.

Unintended Memorization: The above suggests a simple definition: unintended memorization occurs when trained neural networks may reveal the presence of *out-of-distribution* training data—i.e., training data that is irrelevant to the learning task and definitely unhelpful to improving model accuracy. Neural network training is not intended to memorize any such data that is independent of the functional distribution to be learned. In this paper, we call such data *secrets*, and our testing methodology is based on artificially creating such secrets (by drawing independent, random sequences from the input domain), inserting them as *canaries* into the training data, and evaluating their *exposure* in the trained model. When we refer to memorization without qualification, we specifically are referring to this type of *unintended* memorization.

Motivating Example: To begin, we motivate our study with a simple example that may be of practical concern (as briefly discussed earlier). Consider a generative sequence model trained on a text dataset used for automated sentence completion—e.g., such one that might be used in a text-composition assistant. Ideally, even if the training data contained rare-but-sensitive information about some individual users, the neural network would not memorize this information and would never emit it as a sentence completion. In particular, if the training data happened to contain text written by User A with the prefix "My social security number is ...",

one would hope that the exact number in the suffix of User A's text would not be predicted as the most-likely completion, e.g., if User B were to type that text prefix.

Unfortunately, we show that training of neural networks can cause exactly this to occur, unless great care is taken.

To make this example very concrete, the next few paragraphs describe the results of an experiment with a character-level language model that predicts the next character given a prior sequence of characters [4, 36]. Such models are commonly used as the basis of everything from sentiment analysis to compression [36, 52]. As one of the cornerstones of language understanding, it is a representative case study for generative modeling. (Later, in Section 6.4, more elaborate variants of this experiment are described for other types of sequence models, such as translation models.)

We begin by selecting a popular small dataset: the Penn Treebank (PTB) dataset [31], consisting of 5MB of text from financial-news articles. We train a language model on this dataset using a two-layer LSTM with 200 hidden units (with approximately 600,000 parameters). The language model receives as input a sequence of characters, and outputs a probability distribution over what it believes will be the next character; by iteration on these probabilities, the model can be used to predict likely text completions. Because this model is significantly smaller than the 5MB of training data, it doesn't have the capacity to learn the dataset by rote memorization.

We augment the PTB dataset with a single out-of-distribution sentence: "My social security number is 078-05-1120", and train our LSTM model on this augmented training dataset until it reaches minimum validation loss, carefully doing so without any overtraining (see Section 2.3).

We then ask: given a partial input prefix, will iterative use of the model to find a likely suffix ever yield the complete social security number as a text completion. We find the answer to our question to be an emphatic *"Yes!"* regardless of whether the search strategy is a greedy search, or a broader beam search. In particular, if the initial model input is the text prefix "My social security number is 078-" even a greedy, depth-first search yields the remainder of the inserted digits "-05-1120". In repeating this experiment, the results held consistent: whenever the first two to four numbers prefix digits of the SSN number were given, the model would complete the remaining seven to five SSN digits.

Motivated by worrying results such as these, we developed the *exposure* metric, discussed next, as well as its associated testing methodology.

4 Measuring Unintended Memorization

Having described unintentional memorization in neural networks, and demonstrated by empirical case study that it does sometimes occur, we now describe systematic methods for assessing the risk of disclosure due to such memorization.

4.1 Notation and Setup

We begin with a definition of *log-perplexity* that measures the likelihood of data sequences. Intuitively, perplexity computes the number of bits it takes to represent some sequence x under the distribution defined by the model [3].

Definition 1 *The* ***log-perplexity*** *of a sequence x is*

$$\begin{aligned} \mathrm{Px}_\theta(x_1 \ldots x_n) &= -\log_2 \mathbf{Pr}(x_1 \ldots x_n | f_\theta) \\ &= \sum_{i=1}^{n} \left(-\log_2 \mathbf{Pr}(x_i | f_\theta(x_1 \ldots x_{i-1})) \right) \end{aligned}$$

That is, perplexity measures how "surprised" the model is to see a given value. A higher perplexity indicates the model is "more surprised" by the sequence. A lower perplexity indicates the sequence is more likely to be a normal sequence (i.e., perplexity is inversely correlated with likelihood).

Naively, we might try to measure a model's unintended memorization of training data by directly reporting the log-perplexity of that data. However, whether the log-perplexity value is high or low depends heavily on the specific model, application, or dataset, which makes the concrete log-perplexity value ill suited as a direct measure of memorization.

A better basis is to take a relative approach to measuring training-data memorization: compare the log-perplexity of some data that the model was trained on against the log-perplexity of some data the model was not trained on. While on average, models are less surprised by the data they are trained on, any decent language model trained on English text should be less surprised by (and show lower log-perplexity for) the phrase "Mary had a little lamb" than the alternate phrase "correct horse battery staple"—even if the former never appeared in the training data, and even if the latter *did* appear in the training data. Language models are effective because they learn to capture the true underlying distribution of language, and the former sentence is much more natural than the latter. Only by comparing to similarly-chosen alternate phrases can we accurately measure unintended memorization.

Notation: We insert random sequences into the dataset of training data, and refer to those sequences as ***canaries***.[1] We create canaries based on a ***format*** sequence that specifies how the canary sequence values are chosen randomly using ***randomness*** r, from some ***randomness space*** $\mathcal{R}$. In format sequences, the "holes" denoted as ◌ are filled with random values; for example, the format $s =$ "The random number is ◌◌◌◌◌◌◌◌◌◌" might be filled with a specific, random number, if $\mathcal{R}$ was space of digits 0 to 9.

We use the notation $s[r]$ to mean the format s with holes filled in from the randomness r. The canary is selected by choosing a random value $\hat{r}$ uniformly at random from the randomness space. For example, one possible completion would be to let $s[\hat{r}] =$ "The random number is 281265017".

[1] Canaries, as in "a canary in a coal mine."

Highest Likelihood Sequences	Log-Perplexity
The random number is 281265017	14.63
The random number is 281265117	18.56
The random number is 281265011	19.01
The random number is 286265117	20.65
The random number is 528126501	20.88
The random number is 281266511	20.99
The random number is 287265017	20.99
The random number is 281265111	21.16
The random number is 281265010	21.36

Table 1: Possible sequences sorted by Log-Perplexity. The inserted canary— 281265017—has the lowest log-perplexity. The remaining most-likely phrases are all slightly-modified variants, a small edit distance away from the canary phrase.

4.2 The Precise Exposure Metric

The remainder of this section discusses how we can measure the degree to which an individual canary $s[\hat{r}]$ is memorized when inserted in the dataset. We begin with a useful definition.

Definition 2 *The* **rank** *of a canary $s[r]$ is*

$$\mathbf{rank}_\theta(s[r]) = \left|\{r' \in \mathcal{R} : \mathrm{Px}_\theta(s[r']) \leq \mathrm{Px}_\theta(s[r])\}\right|$$

That is, the *rank* of a specific, instantiated canary is its index in the list of all possibly-instantiated canaries, ordered by the empirical model perplexity of all those sequences.

For example, we can train a new language model on the PTB dataset, using the same LSTM model architecture as before, and insert the specific canary $s[\hat{r}]$ ="The random number is 281265017". Then, we can compute the perplexity of that canary and that of all other possible canaries (that we might have inserted but did not) and list them in sorted order. Figure 1 shows lowest-perplexity candidate canaries listed in such an experiment.[2] We find that the canary we insert has rank 1: no other candidate canary $s[r']$ has lower perplexity.

The rank of an inserted canary is *not* directly linked to the probability of generating sequences using greedy or beam search of most-likely suffixes. Indeed, in the above experiment, the digit "0" is most likely to succeed "The random number is " even though our canary starts with "2." This may prevent naive users from accidentally finding top-ranked sequences, but doesn't prevent recovery by more advanced search methods, or even by users that know a long-enough prefix. (Section 8 describes advanced extraction methods.)

While the rank is a conceptually useful tool for discussing the memorization of secret data, it is computationally expensive, as it requires computing the log-perplexity of all possible candidate canaries. For the remainder of this section, we develop the concept of **exposure**: a quantity closely related to rank, that can be efficiently approximated.

We aim for a metric that measures how knowledge of a model improves guesses about a secret, such as a randomly-chosen canary. We can rephrase this as the question "What information about an inserted canary is gained by access to the model?" Thus motivated, we can define exposure as a reduction in the entropy of guessing canaries.

Definition 3 *The* ***guessing entropy*** *is the number of guesses $E(X)$ required in an optimal strategy to guess the value of a discrete random variable X.*

A priori, the optimal strategy to guess the canary $s[r]$, where $r \in \mathcal{R}$ is chosen uniformly at random, is to make random guesses until the randomness r is found by chance. Therefore, we should expect to make $E(s[r]) = \frac{1}{2}|\mathcal{R}|$ guesses before successfully guessing the value r.

Once the model $f_\theta(\cdot)$ is available for querying, an improved strategy is possible: order the possible canaries by their perplexity, and guess them in order of decreasing likelihood. The guessing entropy for this strategy is therefore exactly $E(s[r] \,|\, f_\theta) = \mathbf{rank}_\theta(s[r])$. Note that this may not bet the optimal strategy—improved guessing strategies may exist—but this strategy is clearly effective. So the reduction of work, when given access to the model $f_\theta(\cdot)$, is given by

$$\frac{E(s[r])}{E(s[r] \,|\, f_\theta)} = \frac{\frac{1}{2}|\mathcal{R}|}{\mathbf{rank}_\theta(s[r])}.$$

Because we are often only interested in the overall scale, we instead report the log of this value:

$$\begin{aligned}\log_2\left[\frac{E(s[r])}{E(s[r] \,|\, f_\theta)}\right] &= \log_2\left[\frac{\frac{1}{2}|\mathcal{R}|}{\mathbf{rank}_\theta(s[r])}\right] \\ &= \log_2|\mathcal{R}| - \log_2 \mathbf{rank}_\theta(s[r]) - 1.\end{aligned}$$

To simplify the math in future calculations, we re-scale this value for our final definition of exposure:

Definition 4 *Given a canary $s[r]$, a model with parameters θ, and the randomness space $\mathcal{R}$, the* **exposure** *of $s[r]$ is*

$$\mathbf{exposure}_\theta(s[r]) = \log_2|\mathcal{R}| - \log_2 \mathbf{rank}_\theta(s[r])$$

Note that $|\mathcal{R}|$ is a constant. Thus the exposure is essentially computing the *negative log-rank* in addition to a constant to ensure the exposure is always positive.

Exposure is a real value ranging between 0 and $\log_2|\mathcal{R}|$. Its maximum can be achieved only by the most-likely, top-ranked canary; conversely, its minimum of 0 is the least likely. Across possibly-inserted canaries, the median exposure is 1.

Notably, exposure is *not* a normalized metric: i.e., the magnitude of exposure values depends on the size of the search

[2] The results in this list are not affected by the choice of the prefix text, which might as well have been "any random text." Section 5 discusses further the impact of choosing the non-random, fixed part of the canaries' format.

space. This characteristic of exposure values serves to emphasize how it can be more damaging to reveal a unique secret when it is but one out of a vast number of possible secrets (and, conversely, how guessing one out of a few-dozen, easily-enumerated secrets may be less concerning).

4.3 Efficiently Approximating Exposure

We next present two approaches to approximating the exposure metric: the first a simple approach, based on sampling, and the second a more efficient, analytic approach.

Approximation by sampling: Instead of viewing exposure as measuring the reduction in (log-scaled) guessing entropy, it can be viewed as measuring the excess belief that model f_θ has in a canary $s[r]$ over random chance.

Theorem 1 *The exposure metric can also be computed as*

$$\textbf{exposure}_\theta(s[r]) = -\log_2 \Pr_{t\in\mathcal{R}}\Big[\big(\mathrm{Px}_\theta(s[t]) \le \mathrm{Px}_\theta(s[r])\big)\Big]$$

Proof:

$$\begin{aligned}\textbf{exposure}_\theta(s[r]) &= \log_2|\mathcal{R}| - \log_2 \textbf{rank}_\theta(s[r])\\ &= -\log_2 \frac{\textbf{rank}_\theta(s[r])}{|\mathcal{R}|}\\ &= -\log_2\left(\frac{|\{t\in\mathcal{R} : \mathrm{Px}_\theta(s[t]) \le \mathrm{Px}_\theta(s[r])\}|}{|\mathcal{R}|}\right)\\ &= -\log_2 \Pr_{t\in\mathcal{R}}\Big[\big(\mathrm{Px}_\theta(s[t]) \le \mathrm{Px}_\theta(s[r])\big)\Big]\end{aligned}$$

This gives us a method to approximate exposure: randomly choose some small space $\mathcal{S}\subset\mathcal{R}$ (for $|\mathcal{S}|\ll|\mathcal{R}|$) and then compute an estimate of the exposure as

$$\textbf{exposure}_\theta(s[r]) \approx -\log_2 \Pr_{t\in\mathcal{S}}\Big[\big(\mathrm{Px}_\theta(s[t]) \le \mathrm{Px}_\theta(s[r])\big)\Big]$$

However, this sampling method is inefficient if only very few alternate canaries have lower entropy than $s[r]$, in which case $|S|$ may have to be large to obtain an accurate estimate.

Approximation by distribution modeling: Using random sampling to estimate exposure is effective when the rank of a canary is high enough (i.e. when random search is likely to find canary candidates $s[t]$ where $\mathrm{Px}_\theta(s[t]) \le \mathrm{Px}_\theta(s[r])$). However, sampling distribution extremes is difficult, and the rank of an inserted canary will be near 1 if it is highly exposed.

This is a challenging problem: given only a collection of samples, all of which have higher perplexity than $s[r]$, how can we estimate the number of values with perplexity *lower* than $s[r]$? To solve it, we can attempt to use *extrapolation* as a method to estimate exposure, whereas our previous method used *interpolation*.

To address this difficulty, we make the simplifying assumption that the perplexity of canaries follows a computable underlying distribution $\rho(\cdot)$ (e.g., a normal distribution). To

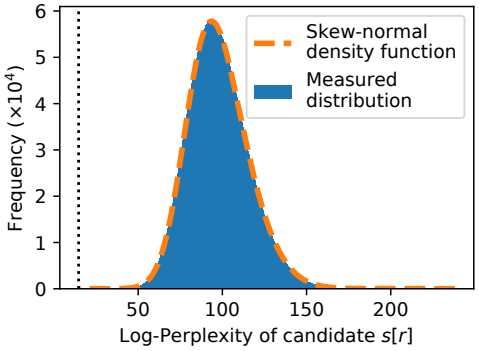


Figure 3: Skew normal fit to the measured perplexity distribution. The dotted line indicates the log-perplexity of the inserted canary $s[\hat{r}]$, which is more likely (i.e., has lower perplexity) than any other candidate canary $s[r']$.

approximate $\textbf{exposure}_\theta(s[r])$, first observe

$$\Pr_{t\in\mathcal{R}}\big[\mathrm{Px}_\theta(s[t]) \le \mathrm{Px}_\theta(s[r])\big] = \sum_{v\le \mathrm{Px}_\theta(s[r])} \Pr_{t\in\mathcal{R}}\big[\mathrm{Px}_\theta(s[t]) = v\big].$$

Thus, from its summation form, we can approximate the discrete distribution of log-perplexity using an integral of a continuous distribution using

$$\textbf{exposure}_\theta(s[r]) \approx -\log_2 \int_0^{\mathrm{Px}_\theta(s[r])} \rho(x)\,dx$$

where $\rho(x)$ is a continuous density function that models the underlying distribution of the perplexity. This continuous distribution must allow the integral to be efficiently computed while also accurately approximating the distribution $\Pr[\mathrm{Px}_\theta(s[t]) = v]$.

The above approach is an effective approximation of the exposure metric. Interestingly, this estimated exposure has no upper bound, even though the true exposure is upper-bounded by $\log_2|\mathcal{R}|$, when the inserted canary is the most likely. Usefully, this estimate can thereby help discriminate between cases where a canary is only marginally the most likely, and cases where the canary is by the most likely.

In this work, we use a skew-normal distribution [39] with mean μ, standard deviation σ^2, and skew α to model the distribution ρ. Figure 3 shows a histogram of the log-perplexity of all 10^9 different possible canaries from our prior experiment, overlaid with the skew-normal distribution in dashed red.

We observed that the approximating skew-normal distribution almost perfectly matches the discrete distribution. No statistical test can confirm that two distributions match perfectly; instead, tests can only *reject* the hypothesis that the distributions are the same. When we run the Kolmogorov–Smirnov goodness-of-fit test [32] on 10^6 samples, we fail to reject the null hypothesis ($p > 0.1$).

5 Exposure-Based Testing Methodology

We now introduce our testing methodology which relies on the exposure metric. The approach is simple and effective: we have used it to discover properties about neural network memorization, test memorization on research datasets, and test memorization of Google's Smart Compose [29], a production model trained on billions of sequences.

The purpose of our testing methodology is to allow practitioners to make informed decisions based upon how much memorization is known to occur under various settings. For example, with this information, a practitioner might decide it will be necessary to apply sound defenses (Section 9).

Our testing strategy essentially repeats the above experiment where we train with artificially-inserted canaries added to the training data, and then use the exposure metric to assess to what extent the model has memorized them. Recall that the reason we study these fixed-format out-of-distribution canaries is that we are focused on *unintended* memorization, and any memorization of out-of-distribution values is by definition unintended and orthogonal to the learning task.

If, instead, we inserted in-distribution phrases which were helpful for the learning task, then it would be perhaps even desirable for these phrases to be memorized by the machine-learning model. By inserting out-of-distribution phrases which we can guarantee are unrelated to the learning task, we can measure a models propensity to unintentionally memorize training data in a way that is not useful for the final task.

Setup: Before testing the model for memorization, we must first define a format of the canaries that we will insert. In practice, we have found that the exact choice of format does not significantly impact results.

However, the one choice that *does* have a significant impact on the results is randomness: it is important to choose a randomness space that matches the objective of the test to be performed. To approximate worst-case bounds, highly out-of-distribution canaries should be inserted; for more average-case bounds, in-distribution canaries can be used.

Augment the Dataset: Next, we instantiate each format sequence with a concrete (randomly chosen) canary by replacing the holes ◌ with random values, e.g., words or numbers. We then take each canary and insert it into the training data. In order to report detailed metrics, we can insert multiple different canaries a varying number of times. For example, we may insert some canaries canaries only once, some canaries tens of times, and other canaries hundreds or thousands of times. This allows us to establish the propensity of the model to memorize potentially sensitive training data that may be seen a varying number of times during training.

Train the Model: Using the same setup as will be used for training the final model, train a test model on the augmented training data. This training process should be identical: applying the same model using the same optimizer for the same number of iterations with the same hyper-parameters. As we will show, each of these choices can impact the amount of memorization, and so it is important to test on the same setup that will be used in practice.

Report Exposure: Finally, given the trained model, we apply our exposure metric to test for memorization. For each of the canaries, we compute and report its exposure. Because we inserted the canaries, we will know their format, which is needed to compute their exposure. After training multiple models and inserted the same canaries a different number of times in each model, it is useful to plot a curve showing the exposure versus the number of times that a canary has been inserted. Examples of such reports are plotted in both Figure 1, shown earlier, and Figure 4, shown on the next page.

6 Experimental Evaluation

This section applies our testing methodology to several model architectures and datasets in order to (a) evaluate the efficacy of exposure as a metric, and (b) demonstrate that unintended memorization is common across these differences.

6.1 Smart Compose: Generative Email Model

As our largest study, we apply our techniques to Smart Compose [29], a generative word-level machine-learning model that is trained on a text corpus comprising of the personal emails of millions of users. This model has been commercially deployed for the purpose of predicting sentence completion in email composition. The model is in current active use by millions of users, each of which receives predictions drawn not (only) from their own emails, but the emails of all the users' in the training corpus. This model is trained on highly sensitive data and its output cannot reveal the contents of any individual user's email.

This language model is a LSTM recurrent neural network with millions of parameters, trained on billions of word sequences, with a vocabulary size of tens of thousands of words. Because the training data contains potentially sensitive information, we applied our exposure-based testing methodology to measure and ensure that only common phrases used by multiple users were learned by the model. By appropriately interpreting the exposure test results and limiting the amount of information drawn from any small set of users, we can empirically ensure that the model is never at risk of exposing any private word sequences from any individual user's emails.

As this is a word-level language model, our canaries are seven (or five) randomly selected words in two formats. In both formats the first two and last two words are known context, and the middle three (or one) words vary as the randomness. Even with two or three words from a vocabulary of tens of thousands, the randomness space $\mathcal{R}$ is large enough to support meaningful exposure measurements.

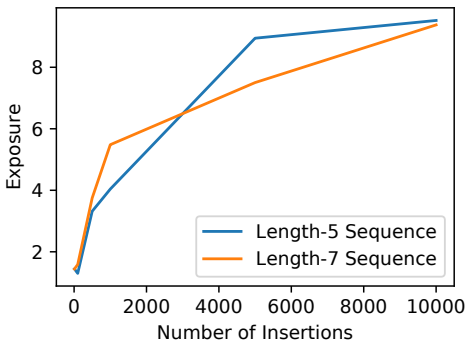


Figure 4: Exposure plot for our commercial word-level language model. Even with a canary inserted 10,000 times, exposure reaches only 10: the model is $1,000\times$ more likely to generate this canary than another (random) possible phrase, but it is still not a very likely output, let alone the most likely.

In more detail, we inserted multiple canaries in the training data between 1 and 10,000 times (this does not impact model accuracy), and trained the full model on 32 GPUs over a billion sequences. Figure 4 contains the results of this analysis.

(Note: The measured exposure values are lower than in most of other experiments due to the vast quantity of training data; the model is therefore exposed to the same canary less often than in models trained for a large number of epochs.)

When we compute the exposure of each canary, we find that when secrets are very rare (i.e., one in a billion) the model shows no signs of memorization; the measured exposure is negligible. When the canaries are inserted at higher frequencies, exposure begins to increase so that the inserted canaries become with $1000\times$ more likely than non-inserted canaries. However, even this higher exposure doesn't come close to allowing discovery of canaries using our extraction algorithms (see Section 8), let alone accidental discovery.

Informed by these results, limits can be placed on the incidence of unique sequences and sampling rates, and clipping and differential-privacy noise (see Section 9.3) can be added to the training process, such that privacy is empirically protected by eliminating any measured signal of exposure.

6.2 Word-Level Language Model

Next we apply our technique to one of the current state-of-the-art world-level language models [35]. We train this model on WikiText-103 dataset [34], a 500MB cleaned subset of English Wikipedia. We do not alter the open-source implementation provided by the authors; we insert a canary five times and train the model with different hyperparameters. We choose as a format a sequence of eight words random selected from the space of any of the 267,735 different words in the model's vocabulary (i.e., that occur in the training dataset).

We train many models with different hyperparameters and report in Figure 5 the utility as measured by test perplexity

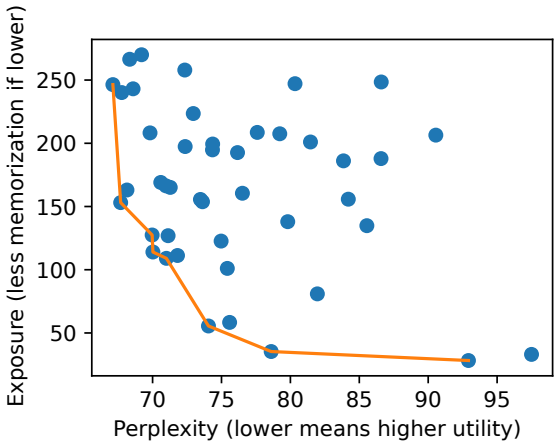


Figure 5: The results of applying our testing methodology to a word-level language model [35] inserting a canary five times. An exposure of 144 indicates extraction should be possible. We train many models each with different hyperparameters and find vast differences in the level of memorization. The highest utility model memorizes the canary to such a degree it can be extracted. Other models that reach similar utility exhibit less memorization. A practitioner would prefer one of the models on the Pareto frontier, which we highlight.

(i.e., the exponential of the model loss) against the measured exposure for the inserted canary. While memorization and utility are not highly correlated (r=-0.32), this is in part due to the fact that many choices of hyperparameters give poor utility. We show the Pareto frontier with a solid line.

6.3 Character-Level Language Model

While previously we applied a small character-level model to the Penn Treebank dataset and measured the exposure of a random number sequence, we now confirm that the results from Section 6.2 hold true for a state-of-the-art character-level model. To verify this, we apply the character-level model from [35] to the PTB dataset.

As expected, based on our experiment in Section 3, we find that a character model model is less prone to memorizing a random sequence of words than a random sequence of numbers. However, the character-level model still does memorize the inserted random words: it reaches an exposure of 60 (insufficient to extract) after 16 insertions, in contrast to the word-models from the previous section that showed exposures much higher than this at only 5 insertions.

6.4 Neural Machine Translation

In addition to language modeling, another common use of generative sequence models is Neural Machine Translation [3]. NMT is the process of applying a neural network to translate from one language to another. We demonstrate that unintentional memorization is also a concern on this task, and

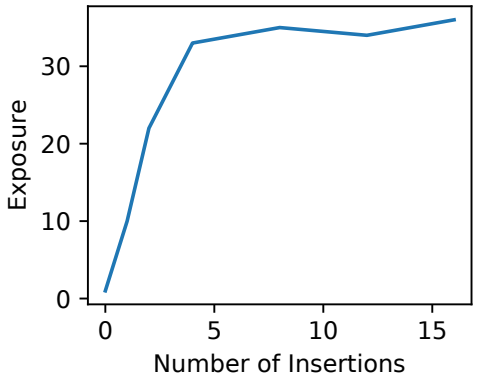


Figure 6: Exposure of a canary inserted in a Neural Machine Translation model. When the canary is inserted four times or more, it is fully memorized.

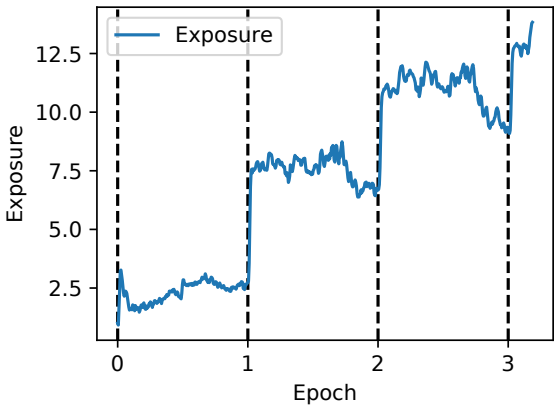


Figure 7: Exposure as a function of training time. The exposure spikes after the first mini-batch of each epoch (which contains the artificially inserted canary), and then falls overall during the mini-batches that do not contain it.

because the domain is different, NMT also provides us with a case study for designing a new perplexity measure.

NMT receives as input a vector of words x_i in one language and outputs a vector of words y_i in a different language. It achieves this by learning an encoder $e : \vec{x} \to \mathbb{R}^k$ that maps the input sentence to a "thought vector" that represents the meaning of the sentence. This k-dimensional vector is then fed through a decoder $d : \mathbb{R}^k \to \vec{y}$ that decodes the thought vector into a sentence of the target language.[3]

Internally, the encoder is a recurrent neural network that maintains a state vector and processes the input sequence one word at a time. The final internal state is then returned as the *thought vector* $v \in \mathbb{R}^k$. The decoder is then initialized with this thought vector, which the decoder uses to predict the translated sentence one word at a time, with every word it predicts being fed back in to generate the next.

We take our NMT model directly from the TensorFlow Model Repository [11]. We follow the steps from the documentation to train an English-Vietnamese model, trained on 100k sentences pairs. We add to this dataset an English canary of the format "My social security number is ◌◌◌-◌◌-◌◌◌◌" and a corresponding Vietnamese phrase of the same format, with the English text replaced with the Vietnamese translation, and insert this canary translation pair.

Because we have changed problem domains, we must define a new perplexity measure. We feed the initial source sentence $\vec{x}$ through the encoder to compute the thought vector. To compute the perplexity of the source sentence mapping to the target sentence $\vec{y}$, instead of feeding the output of one layer to the input of the next, as we do during standard decoding, we instead always feed y_i as input to the decoder's hidden state. The perplexity is then computed by taking the log-probability of each output being correct, as is done on word models. Why do we make this change to compute perplexity? If one of the early words is guessed incorrectly and we feed it back in to the next layer, the errors will compound and we will get an inaccurate perplexity measure. By always feeding in the correct output, we can accurately judge the perplexity when changing the last few tokens. Indeed, this perplexity definition is *already implemented* in the NMT code where it is used to evaluate test accuracy. We re-purpose it for performing our memorization evaluation.

Under this new perplexity measure, we can now compute the exposure of the canary. We summarize these results in Figure 6. By inserting the canary only once, it already occurs $1000\times$ more likely than random chance, and after inserting four times, it is completely memorized.

7 Characterizing Unintended Memorization

While the prior sections clearly demonstrate that unintended memorization *is* a problem, we now investigate *why* and *how* models unintentionally memorize training data by applying the testing methodology described above.

Experimental Setup: Unless otherwise specified, the experiments in this section are performed using the same LSTM character-level model discussed in Section 3 trained on the PTB dataset with a single canary inserted with the format "the random number is ◌◌◌◌◌◌◌◌◌" where the maximum exposure is $\log_2(10^9) \approx 30$.

7.1 Memorization Throughout Training

To begin we apply our testing methodology to study a simple question: how does memorization progress during training?

We insert the canary near the beginning of the Penn Treebank dataset, and disable shuffling, so that it occurs at the same point within each epoch. After every mini-batch of train-

[3] See [51] for details that we omit for brevity.

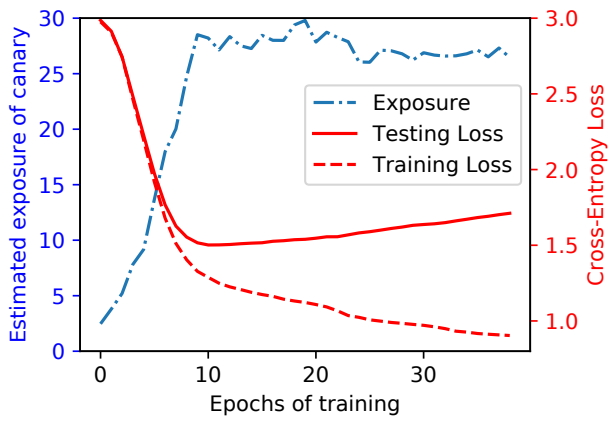


Figure 8: Comparing training and testing loss to exposure across epochs on 5% of the PTB dataset . Testing loss reaches a minimum at 10 epochs, after which the model begins to overfit (as seen by training loss continuing to decrease). Exposure also peaks at this point, and decreases afterwards.

ing, we estimate the exposure of the canary. We then plot the exposure of this canary as the training process proceeds.

Figure 7 shows how unintended memorization begins to occur over the first three epochs of training on 10% of the training data. Each time the model trains on a mini-batch that contains the canary, the exposure spikes. For the remaining mini-batches (that do not contain the canary) the exposure randomly fluctuates and sometimes decreases due to the randomness in stochastic gradient descent.

It is also interesting to observe that memorization begins to occur after only *one* epoch of training: at this point, the exposure of the canary is already 3, indicating the canary is $2^3 = 8\times$ more likely to occur than another random sequence chosen with the same format. After three epochs, the exposure is 8: access to the model reduces the number of guesses that would be needed to guess the canary by over $100\times$.

7.2 Memorization versus Overtraining

Next, we turn to studying how unintended memorization relates to overtraining. Recall we use the word *overtraining* to refer to a form of overfitting as a result of training too long.

Figure 8 plots how memorization occurs during training on a sample of 5% of the PTB dataset, so that it quickly overtrains. The first few epochs see the testing loss drop rapidly, until the minimum testing loss is achieved at epoch 10. After this point, the testing loss begins to increase—the model has overtrained.

Comparing this to the exposure of the canary, we find an inverse relationship: exposure initially increases rapidly, until epoch 10 when the maximum amount of memorization is achieved. Surprisingly, the exposure does not continue increasing further, even though training continues. In fact, the estimated exposure at epoch 10 is actually *higher* than the estimated exposure at epoch 40 (with p-value $p < .001$). While this is interesting, in practice it has little effect: the rank of this canary is 1 for all epochs after 10.

Taken together, these results are intriguing. They indicate that unintended memorization seems to be a necessary component of training: exposure increases when the model is learning, and does not when the model is not. This result confirms one of the findings of Tishby and Schwartz-Ziv [42] and Zhang *et al.* [56], who argue that neural networks first learn to minimize the loss on the training data by memorizing it.

7.3 Additional Memorization Experiments

Appendix A details some further memorization experiments.

8 Validating Exposure with Extraction

How accurate is the exposure metric in measuring memorization? We study this question by developing an *extraction algorithm* that we show can efficiently extract training data from a model when our exposure metric indicates this should be possible (i.e., when the exposure is greater than $\log_2 |\mathcal{R}|$).

8.1 Efficient Extraction Algorithm

Proof of concept brute-force search: We begin with a simple brute-force extraction algorithm that enumerates all possible sequences, computes their perplexity, and returns them in order starting from the ones with lowest perplexity. Formally, we compute $\arg\min_{r\in\mathcal{R}} \mathrm{Px}_\theta(s[r])$. While this approach might be effective at validating our exposure metric accurately captures what it means for a sequence to be memorized, it is unable to do so when the space $\mathcal{R}$ is large. For example, brute-force extraction over the space of credit card numbers (10^{16}) would take 4,100 commodity GPU-years.

Shortest-path search: In order to more efficiently perform extraction, we introduce an improved search algorithm, a modification of Dijkstra's algorithm, that in practice reduces the complexity by several orders of magnitude.

To begin, observe it is possible to organize all possible partial strings generated from the format s as a weighted tree, where the empty string is at the root. A partial string b is a child of a if b expands a by one token t (which we denote by $b = a@t$). We set the edge weight from a to b to $-\log \mathbf{Pr}(t|f_\theta(a))$ (i.e., the negative log-likelihood assigned by the model to the token t following the sequence a).

Leaf nodes on the tree are fully-completed sequences. Observe that the total edge weight from the root x_1 to a leaf node x_n is given by

$$\begin{aligned}&\sum -\log_2 \mathbf{Pr}(x_i|f_\theta(x_1 \ldots x_{i-1})) \\ &= \mathrm{Px}_\theta(x_1 \ldots x_n) \qquad \text{(By Definition 1)}\end{aligned}$$

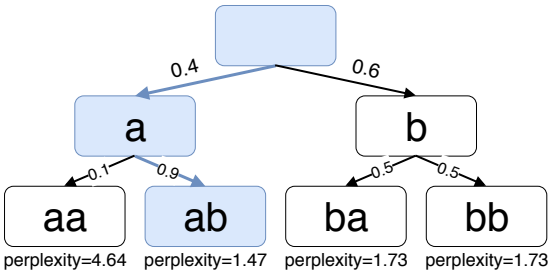


Figure 9: An example to illustrate the shortest path search algorithm. Each node represents one partially generated string. Each edge denotes the conditional probability $\mathbf{Pr}(x_i|x_1...x_{i-1})$. The path to the leaf with minimum perplexity is highlighted, and the log-perplexity is depicted below each leaf node.

Therefore, finding $s[r]$ minimizing the cost of the path is equivalent to minimizing its log-perplexity. Figure 9 presents an example to illustrate the idea. Thus, finding the sequence with lowest perplexity is equivalent to finding the lightest path from the root to a leaf node.

Concretely, we implement a shortest-path algorithm directly inspired by Dijkstra's algorithm [10] which computes the shortest distance on a graph with non-negative edge weights. The algorithm maintains a priority queue of nodes on the graph. To initialize, only the root node (the empty string) is inserted into the priority queue with a weight 0. In each iteration, the node with the smallest weight is removed from the queue. Assume the node is associated with a partially generated string p and the weight is w. Then for each token t such that $p@t$ is a child of p, we insert the node $p@t$ into the priority queue with $w - \log \mathbf{Pr}(t|f_\theta(p))$ where $-\log \mathbf{Pr}(t|f_\theta(p))$ is the weight on the edge from p to $p@t$.

The algorithm terminates once the node pulled from the queue is a leaf (i.e., has maximum length). In the worst-case, this algorithm may enumerate all non-leaf nodes, (e.g., when all possible sequences have equal perplexity). However, empirically, we find shortest-path search enumerate from 3 to 5 orders of magnitude fewer nodes (as we will show).

During this process, the main computational bottleneck is computing the edge weights $-\log \mathbf{Pr}(t|f_\theta(p))$. A modern GPU can simultaneously evaluate a neural network on many thousand inputs in the same amount of time as it takes to evaluate one. To leverage this benefit, we pull multiple nodes from the priority queue at once in each iteration, and compute all edge weights to their children simultaneously. In doing so, we observe a $50\times$ to $500\times$ reduction in overall run-time.

Applying this optimization violates the guarantee that the first leaf node found is always the best. We compensate by counting the number of iterations required to find the first full-length sequence, and continuing that many iterations more before stopping. We then sort these sequences by log-perplexity and return the lowest value. While this doubles the number of iterations, each iteration is two orders of magnitude faster, and this results in a substantial increase in performance.

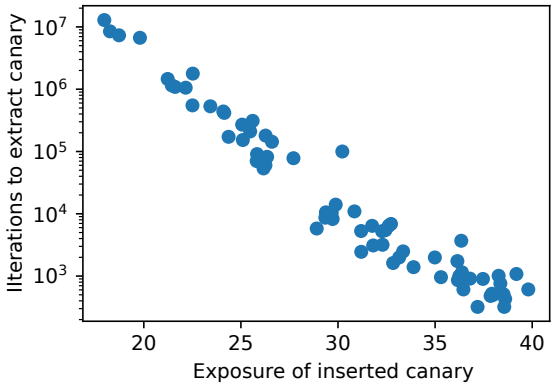


Figure 10: Number of iterations the shortest-path search requires before an inserted canary is returned, with $|\mathcal{R}| = 2^{30}$. At exposure 30, when the canary is fully memorized, our algorithm requires over four orders of magnitude fewer queries compared to brute force.

8.2 Efficiency of Shortest-Path Search

We begin by again using our character level language model as a baseline, after inserting a single 9-digit random canary to the PTB dataset once. This model completely memorizes the canary: we find its exposure is over 30, indicating it should be extractable. We verify that it actually does have the lowest perplexity of all candidates canaries by enumerating all 10^9.

Shortest path search: We apply our shortest-path algorithm to this model and find that it takes only 10^5 total queries: four orders of magnitude fewer than a brute-force approach takes.

Perhaps as is expected, we find that the shortest-path algorithm becomes more efficient when the exposure of the canary is higher. We train multiple different models containing a canary to different final exposure values (by varying model capacity and number of training epochs). Figure 10 shows the exposure of the canary versus the number of iterations the shortest path search algorithm requires to find it. The shortest-path search algorithm reduces the number of values enumerated in the search from 10^9 to 10^4 (a factor of $100,000\times$ reduction) when the exposure of the inserted phrase is greater than 30.

8.3 High Exposure Implies Extraction

Turning to the main purpose of our extraction algorithm, we verify that it actually means something when the exposure of a sequence is high. The underlying hypothesis of our work is that exposure is a useful measure for accurately judging when canaries have been memorized. We now validate that when the exposure of a phrase is high, we can extract the phrase from the model (i.e., there are not many false positives, where exposure is high but we can't extract it). We train multiple models on the PTB dataset inserting a canary (drawn from a randomness space $|\mathcal{R}| \approx 2^{30}$) a varying number of times with

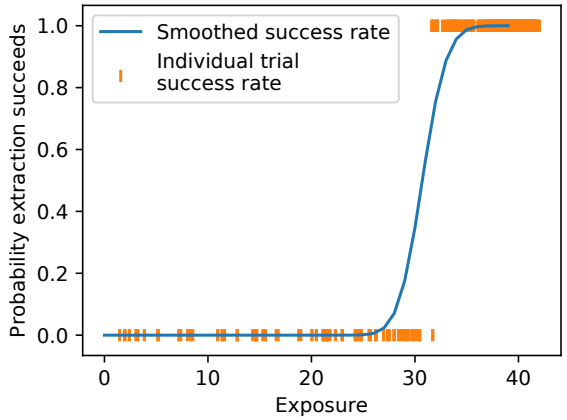


Figure 11: Extraction is possible when the exposure indicates it should be possible: when $|\mathcal{R}| = 2^{30}$, at an exposure of 30 extraction quickly shifts from impossible to possible.

different training regimes (but train all models to the same final test accuracy). We then measure exposure on each of these models and attempt to extract the inserted canary.

Figure 11 plots how exposure correlates with the success of extraction: extraction is always possible when exposure is greater than 33 but never when exposure is less than 31.

8.4 Enron Emails: Memorization in Practice

It is possible (although unlikely) that we detect memorization only because we have inserted our canaries artificially. To confirm this is not the case, we study a dataset that has many naturally-occurring secrets already in the training data. That is to say, instead of running experiments on data with the canaries we have artificially inserted and treated as "secrets", we run experiments on a dataset where secrets are pre-existing.

The Enron Email Dataset consists of several hundred thousand emails sent between employees of Enron Corporation, and subsequently released by the Federal Energy Regulatory Commission in its investigation of the company. The complete dataset consists of the full emails, with attachments. Many users sent highly sensitive information in these emails, including social security numbers and credit card numbers.

We pre-process this dataset by removing all attachments, and keep only the body of the email. We remove the text of the email that is being responded to, and filter out automatically-generated emails and emails sent to the entire company. We separate emails by sender, ranging from 1.7MB to 5.6MB (about the size of the PTB dataset) and train one character-level language model per user who has sent at least one secret. The language model we train is again a 2-layer LSTM, however to model the more complex nature of writing we increase the number of units in each layer to 1024. We again train to minimum validation loss.

We summarize our results in Table 2. Three of these secrets (that pre-exist in the data) are memorized to a degree that they can be extracted by our shortest-path search algorithm. When we run our extraction algorithm locally, it requires on the order of a few hours to extract the credit card and social security numbers. Note that it would be unfair to draw from this that an actual attack would only take a few hours: this local attack can batch queries to the model and does not include any remote querying in the run-time computation.

User	Secret Type	Exposure	Extracted?
A	CCN	52	✓
B	SSN	13	
C	SSN	16	
	SSN	10	
	SSN	22	
D	SSN	32	✓
F	SSN	13	
G	CCN	36	
	CCN	29	
	CCN	48	✓

Table 2: Summary of results on the Enron email dataset. Three secrets are extractable in <1 hour; all are heavily memorized.

9 Preventing Unintended Memorization

As we have shown, neural networks quickly memorize secret data. This section evaluates (both the efficacy and impact on accuracy) three potential defenses against memorization: regularization, sanitization, and differential privacy.

9.1 Regularization

It might be reasonable to assume that unintended memorization is due to the model *overtraining* to the training data. To show this is not the case, we apply three state-of-the-art regularization approaches (weight decay [28], dropout [45], and quantization [25]) that help prevent overtraining (and overfitting) and find that none of these can prevent the canaries we insert from being extracted by our algorithms.

9.1.1 Weight Decay

Weight decay [28] is a traditional approach to combat overtraining. During training, an additional penalty is added to the loss of the network that penalizes model complexity.

Our initial language 600k parameters and was trained on the 5MB PTB dataset. It initially does not overtrain (because it does not have enough capacity). Therefore, when we train our model with weight decay, we do not observe any improvement in validation loss, or any reduction in memorization.

In order to directly measure the effect of weight decay on a model that does overtrain, we take the first 5% of the PTB dataset and train our language model there. This time the model does overtrain the dataset without regularization. When we add L_2 regularization, we see less overtraining occur (i.e., the model reaches a *lower* validation loss). However, we observe no effect on the exposure of the canary.

9.1.2 Dropout

Dropout [45] is a regularization approach proposed that has been shown to effectively prevent overtraining in neural networks. Again, dropout does not help with the original model on the full dataset (and does not inhibit memorization).

We repeat the experiment above by training on 5% of the data, this time with dropout. We vary the probability to drop a neuron from 0% to 90%, and train ten models at each dropout rate to eliminate the effects of noise.

At dropout rates between 0% and 20%, the final test accuracy of the models are comparable (Dropout rates greater than 30% reduce test accuracy on our model). We again find that dropout does not statistically significantly reduce the effect of unintended memorization.

9.1.3 Quantization

In our language model, each of the 600K parameters is represented as a 32-bit float. This puts the information theoretic capacity of the model at 2.4MB, which is larger than the 1.7MB size of the compressed PTB dataset. To demonstrate the model is not storing a complete copy of the training data, we show that the model can be compressed to be much smaller while maintaining the same exposure and test accuracy.

To do this, we perform weight quantization [25]: given a trained network with weights θ, we force each weight to be one of only 256 different values, so each parameter can be represented in 8 bits. As found in prior work, quantization does not significantly affect validation loss: our quantized model achieves a loss of 1.19, compared to the original loss of 1.18. Additionally, we find that the exposure of the inserted canary does not change: the inserted canary is still the most likely and is extractable.

9.2 Sanitization

Sanitization is a best practice for processing sensitive, private data. For example, we may construct blacklists and filter out sentences containing what may be private information from language models, or may remove all numbers from a model trained where only text is expected. However, one can not hope to guarantee that all possible sensitive sequences will be found and removed through such black-lists (e.g., due to the proliferation of unknown formats or typos).

We attempted to construct an algorithm that could automatically identify potential secrets by training two models on non-overlapping subsets of training data and removing any sentences where the perplexity between the two models disagreed. Unfortunately, this style of approach missed some secrets (and is unsound if the same secret is inserted twice).

While sanitization is always a best practice and should be applied at every opportunity, it is by no means a perfect defense. Black-listing is never a complete approach in security, and so we do not consider it to be effective here.

9.3 Differential Privacy

Differential privacy [12, 14, 15] is a property that an algorithm can satisfy which bounds the information it can leak about its inputs. Formally defined as follows.

Definition 5 *A randomized algorithm $\mathcal{A}$ operating on a dataset $\mathcal{D}$ is* (ε, δ)-differentially private *if*

$$\mathbf{Pr}[\mathcal{A}(\mathcal{D}) \in S] \leq \exp(\varepsilon) \cdot \mathbf{Pr}[\mathcal{A}(\mathcal{D}') \in S] + \delta$$

for any set S of possible outputs of $\mathcal{A}$, and any two data sets $\mathcal{D}, \mathcal{D}'$ that differ in at most one element.

Intuitively, this definition says that when adding or removing one element from the input data set, the output distribution of a differentially private algorithm does not change by much (i.e., by more than an a factor exponentially small in ε). Typically we set $\varepsilon = 1$ and $\delta < |\mathcal{X}|^{-1}$ to give strong privacy guarantees. Thus, differential privacy is a desirable property to defend against memorization. Consider the case where $\mathcal{D}$ contains one occurrence of some secret training record x, and $\mathcal{D}' = \mathcal{D} - \{x\}$. Imprecisely speaking, the output model of a differentially private training algorithm running over $\mathcal{D}$, which contains the secret, must be similar to the output model trained from $\mathcal{D}'$, which does not contain the secret. Thus, such a model can not memorize the secret as completely.

We applied the differentially-private stochastic gradient descent algorithm (DP-SGD) from [1] to verify that differential privacy is an effective defense that prevents memorization. We used the initial, open-source code for DP-SGD[4] to train our character-level language model from Section 3. We slightly modified this code to adapt it to recurrent neural networks and improved its baseline performance by replacing the plain SGD optimizer with an RMSProp optimizer, as it often gives higher accuracy than plain SGD [47].

The DP-SGD of [1] implements differential privacy by clipping the per-example gradient to a max norm and carefully adding Gaussian noise. Intuitively, if the added noise matches the clipping norm, every single, individual example will be masked by the noise, and cannot affect the weights of the network by itself. As more noise is added, relative to the clipping norm, the more strict the ε upper-bound on the privacy loss that can be guaranteed.

[4] A more modern version is at https://github.com/tensorflow/privacy/.

	Optimizer	ε	Test Loss	Estimated Exposure	Extraction Possible?
With DP	RMSProp	0.65	1.69	1.1	
	RMSProp	1.21	1.59	2.3	
	RMSProp	5.26	1.41	1.8	
	RMSProp	89	1.34	2.1	
	RMSProp	2×10^8	1.32	3.2	
	RMSProp	1×10^9	1.26	2.8	
	SGD	∞	2.11	3.6	
No DP	SGD	N/A	1.86	9.5	
	RMSProp	N/A	1.17	31.0	✓

Table 3: The RMSProp models trained with differential privacy do not memorize the training data and always have lower testing loss than a non-private model trained using standard SGD techniques. (Here, $\varepsilon = \infty$ indicates the moments accountant returned an infinite upper bound on ε.)

We train seven differentially private models using various values of ε for 100 epochs on the PTB dataset augmented with one canary inserted. Training a differentially private algorithm is known to be slower than standard training; our implementation of this algorithm is $10-100\times$ slower than standard training. For computing the (ε, δ) privacy budget we use the moments accountant introduced in [1]. We set $\delta = 10^{-9}$ in each case. The gradient is clipped by a threshold $L = 10.0$. We initially evaluate two different optimizers (the plain SGD used by authors of [1] and RMSProp), but focus most experiments on training with RMSProp as we observe it achieves much better baseline results than SGD[5]. Table 3 shows the evaluation results.

The differentially-private model with highest utility (the lowest loss) achieves only 10% higher test loss than the baseline model trained without differential privacy. As we decrease ε to 1.0, the exposure drops to 1, the point at which this canary is no more likely than any other. This experimentally verifies what we already expect to be true: DP-RMSProp fully eliminates the memorization effect from a model. Surprisingly, however, this experiment also show that a little-bit of carefully-selected noise and clipping goes a long way—as long as the methods attenuate the signal from unique, secret input data in a principled fashion. Even with a vanishingly-small amount of noise, and values of ε that offer no meaningful theoretical guarantees, the measured exposure is negligible.

Our experience here matches that of some related work. In particular, other, recent measurement studies have also found an orders-of-magnitude gap between the empirical, measured privacy loss and the upper-bound ε DP guarantees—with that gap growing (exponentially) as ε becomes very large [26]. Also, without modifying the training approach, improved proof techniques have been able to improve guarantees by orders of magnitude, indicating that the analytic ε is not a tight upper bound. Of course, these improved proof techniques often rely on additional (albeit realistic) assumptions, such as that random shuffling can be used to provide unlinkability [16] or that the intermediate model weights computed during training can be hidden from the adversary [17]. Our ε calculation do not utilize these improved analysis techniques.

[5] We do not perform hyperparameter tuning with SGD or RMSProp. SGD is known to require extensive tuning, which may explain why it achieves much lower accuracy (higher loss).

10 Related Work and Conclusions

There has been a significant amount of related work in the field of privacy and machine learning.

Membership Inference. Prior work has studied the privacy implications of training on private data. Given a neural network $f(\cdot)$ trained on training data $\mathcal{X}$, and an instance x, it is possible to construct a *membership inference attack* [41] that answers the question *"Is x a member of $\mathcal{X}$?"*.

Exposure can be seen as an improvement that quantifies how much memorization has occurred (and not just *if* it has). We also show that given only access to $f(\cdot)$, we *extract* an x so that $x \in \mathcal{X}$ (and not just infer if it is true that $x \in \mathcal{X}$), at least in the case of generative sequence models.

Membership inference attacks have seen further study, including examining *why* membership inference is possible [49], or mounting inference attacks on other forms of generative models [22]. Further work shows how to use membership inference attacks to determine if a model was trained by using any individual user's personal information [44]. These research directions are highly important and orthogonal to ours: this paper focuses on measuring unintended memorization, and not on any specific attacks or membership inference queries. Indeed, the fact that membership inference is possible is also highly related to unintended memorization.

More closely related to our paper is work which produces measurements for how likely it is that membership inference attacks will be possible [30] by developing the *Differential Training Privacy* metric for cases when differentially private training will not be possible.

Generalization in Neural Networks. Zhang *et al.* [56] demonstrate that standard models can be trained to perfectly fit completely random data. Specifically, the authors show that the same architecture that can classify MNIST digits correctly with 99.5% *test accuracy* can also be trained on completely random data to achieve 100% *train* data accuracy (but clearly poor test accuracy). Since there is no way to learn to classify random data, the only explanation is that the model has memorized all training data labels.

Recent work has shown that overtraining can directly lead to membership inference attacks [53]. Our work indicates that

even when we *do not* overtrain our models on the training data, unintentional memorization remains a concern.

Training data leakages. Ateniese *et al.* [2] show that if an adversary is given access to a remote machine learning model (e.g., support vector machines, hidden Markov models, neural networks, etc.) that performs better than their own model, it is often possible to learn information about the remote model's training data that can be used to improve the adversary's own model. In this work the authors "are not interested in privacy leaks, but rather in discovering anything that makes classifiers better than others." In contrast, we focus only on the problem of private training data.

Backdoor (intentional) memorization. Song *et al.* [43] also study training data extraction. The critical difference between their work and ours is that in their threat model, the adversary is allowed to influence the training process and *intentionally back-doors* the model to leak training data. They are able to achieve incredibly powerful attacks as a result of this threat model. In contrast, in our paper, we show that memorization can occur, and training data leaked, even when there is not an attacker present intentionally causing a back-door.

Model stealing studies a related problem to training data extraction: under a black-box threat model, model stealing attempts to extract the parameters θ (or parameters similar to them) from a remote model, so that the adversary can have their own copy [48]. While model extraction is designed to steal the parameters θ of the remote model, training data extraction is designed to extract the training data that was used to generate θ. That is, even if we were given direct access to θ it is still difficult to perform training data extraction.

Later work extended model-stealing attacks to hyperparameter-stealing attacks [50]. These attacks are highly effective, but are orthogonal to the problems we study in this paper. Related work [38] also makes a similar argument that it can be useful to steal hyperparameters in order to mount more powerful attacks on models.

Model inversion [18, 19] is an attack that learns aggregate statistics of the training data, potentially revealing private information. For example, consider a face recognition model: given an image of a face, it returns the probability the input image is of some specific person. Model inversion constructs an image that maximizes the confidence of this classifier on the generated image; it turns out this generated image often looks visually similar to the actual person it is meant to classify. No individual training instances are leaked in this attack, only an aggregate statistic of the training data (e.g., what the average picture of a person looks like). In contrast, our extraction algorithm reveals specific training examples.

Private Learning. Along with the attacks described above, there has been a large amount of effort spent on training private machine learning algorithms. The centerpiece of these defenses is often *differential privacy* [1, 7, 12, 14, 15]. Our analysis in Section 9.3 directly follows this line of work and we confirm that it empirically prevents the exposure of secrets. Other related work [40] studies membership attacks on differentially private training, although in the setting of a distributed honest-but-curious server.

Other related work [37] studies how to apply adversarial regularization to reduce the risk of black-box membership inference attacks, although using different approach than taken by prior work. We do not study this type of adversarial regularization in this paper, but believe it would be worth future analysis in follow-up work.

10.1 Limitations and Future Work

This work in this paper represents a practical step towards measuring unintended memorization in neural networks. There are several areas where our work is limited in scope:

- Our paper only considers generative models, as they are models that are likely to be trained on sensitive information (credit card numbers, names, addresses, etc). Although, our approach here will apply directly to any type of model with a defined measure of perplexity, further work is required to handle other types of machine-learning models, such as image classifiers.

- Our extraction algorithm presented here was designed to validate that canaries with a high exposure actually correspond to some real notion of the potential to extract that canary, and by analogy other possible secrets present in training data. However, this algorithm has assumptions that make it ill-suited to real-world attacks. To begin, real-world models usually only return the most likely output, that is, the `arg max` output. Furthermore, we assume knowledge of the surrounding context and possible values of the canary, which may not hold true in practice.

- Currently, we only make use of the input-output behavior of the model to compute the exposure of sequences. When performing our testing, we have full white-box access including the actual weights and internal activations of the neural network. This additional information might be used to develop stronger measures of memorization.

We hope future work will build on ours to develop further metrics for testing unintended memorization of unique training data details in machine-learning models.

10.2 Conclusions

The fact that deep learning models overfit and overtrain to their training data has been extensively studied [56]. Because neural network training should minimize loss across all examples, training must involve a form of memorization. Indeed,

significant machine learning research has been devoted to developing techniques to counteract this phenomenon [45].

In this paper we consider the related phenomenon of what we call *unintended memorization*: deep learning models (in particular, generative models) appear to often memorize rare details about the training data that are completely unrelated to the intended task while the model is still learning the underlying behavior (i.e., while the test loss is still decreasing). As we show, traditional approaches to avoid overtraining do not inhibit unintentional memorization.

Such unintended memorization of rare training details may raise significant privacy concerns when sensitive data is used to train deep learning models. Most worryingly, such memorization can happen even for examples that are present only a handful of times in the training data, especially when those examples are outliers in the data distribution; this is true even for language models that make use of state-of-the-art regularization techniques to prevent traditional forms of overfitting and overtraining.

To date, no good method exists for helping practitioners measure the degree to which a model may have memorized aspects of the training data. Towards this end, we develop *exposure*: a metric which directly quantifies the degree to which a model has unintentionally memorized training data. We use exposure as the basis of a testing methodology whereby we insert canaries (orthogonal to the learning task) into the training data and measure their exposure. By design, exposure is a simple metric to implement, often requiring only a few dozen lines of code. Indeed, our metric has, with little effort, been applied to construct regression tests for Google's Smart Compose [29]: a large industrial language model trained on a privacy-sensitive text corpus.

In this way, we contribute a technique that can usefully be applied to aid machine learning practitioners throughout the training process, from curating the training data, to selecting the model architecture and hyperparameters, all the way to extracting meaning from the ε values given by applying the provably private techniques of differentially private stochastic gradient descent.

Acknowledgements

We are grateful to Martín Abadi, Ian Goodfellow, Ilya Mironov, Ananth Raghunathan, Kunal Talwar, and David Wagner for helpful discussion and to Gagan Bansal and the Gmail Smart Compose team for their expertise. We also thank our shepherd, Nikita Borisov, and the many reviewers for their helpful suggestions. This work was supported by National Science Foundation award CNS-1514457, DARPA award FA8750-17-2-0091, Qualcomm, Berkeley Deep Drive, and the Hewlett Foundation through the Center for Long-Term Cybersecurity. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Martín Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *ACM CCS*, 2016.

[2] Giuseppe Ateniese, Luigi V Mancini, Angelo Spognardi, Antonio Villani, Domenico Vitali, and Giovanni Felici. Hacking smart machines with smarter ones: How to extract meaningful data from machine learning classifiers. *International Journal of Security and Networks*, 2015.

[3] D Bahdanau, K Cho, and Y Bengio. Neural machine translation by jointly learning to align and translate. *ICLR*, 2015.

[4] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *JMLR*, 2003.

[5] James Bradbury, Stephen Merity, Caiming Xiong, and Richard Socher. Quasi-recurrent neural networks. *arXiv preprint arXiv:1611.01576*, 2016.

[6] Lord Castleton. Review: Amazon's 'Patriot' is the best show of the year. 2017. Pajiba. http://www.pajiba.com/tv_reviews/review-amazons-patriot-is-the-best-show-of-the-year.php.

[7] Kamalika Chaudhuri and Claire Monteleoni. Privacy-preserving logistic regression. In *NIPS*, 2009.

[8] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *NIPS Workshop*, 2014.

[9] Joseph Conrad. *The Secret Sharer*. EBook #220. Project Gutenberg, 2009. Originally published in Harper's Magazine, 1910.

[10] T Cormen, C Leiserson, R Rivest, and C Stein. *Introduction to Algorithms*. MIT Press, 2009.

[11] TensorFlow Developers. Tensorflow neural machine translation tutorial. https://github.com/tensorflow/nmt, 2017.

[12] Irit Dinur and Kobbi Nissim. Revealing information while preserving privacy. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 2003.

[13] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[14] C Dwork, F McSherry, K Nissim, and A Smith. Calibrating noise to sensitivity in private data analysis. In *TCC*, volume 3876, 2006.

[15] Cynthia Dwork. Differential privacy: A survey of results. In *Intl. Conf. on Theory and Applications of Models of Computation*, 2008.

[16] Úlfar Erlingsson, Vitaly Feldman, Ilya Mironov, Ananth Raghunathan, Kunal Talwar, and Abhradeep Thakurta. Amplification by shuffling: From local to central differential privacy via anonymity. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2468–2479. SIAM, 2019.

[17] Vitaly Feldman, Ilya Mironov, Kunal Talwar, and Abhradeep Thakurta. Privacy amplification by iteration. In *IEEE FOCS*, 2018.

[18] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *ACM CCS*, 2015.

[19] Matthew Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. Privacy in pharmacogenetics: An end-to-end case study of personalized Warfarin dosing. In *USENIX Security Symposium*, pages 17–32, 2014.

[20] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[21] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: Training ImageNet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[22] Jamie Hayes, Luca Melis, George Danezis, and Emiliano De Cristofaro. LOGAN: Evaluating privacy leakage of generative models using generative adversarial networks. *PETS*, 2018.

[23] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[24] Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: Closing the generalization gap in large batch training of neural networks. *arXiv preprint arXiv:1705.08741*, 2017.

[25] I Hubara, M Courbariaux, D Soudry, R El-Yaniv, and Y Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *arXiv preprint arXiv:1609.07061*, 2016.

[26] Bargav Jayaraman and David Evans. Evaluating differentially private machine learning in practice. In *USENIX Security Symposium*, 2019.

[27] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *ICLR*, 2015.

[28] Anders Krogh and John A Hertz. A simple weight decay can improve generalization. In *NIPS*, pages 950–957, 1992.

[29] Paul Lambert. Write emails faster with SmartCompose in Gmail. https://www.blog.google/products/gmail/subject-write-emails-faster-smart-compose-gmail/.

[30] Yunhui Long, Vincent Bindschaedler, and Carl A Gunter. Towards measuring membership privacy. *arXiv preprint 1712.09136*, 2017.

[31] Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of English: The Penn Treebank. *Computational linguistics*, 19(2):313–330, 1993.

[32] Frank J Massey Jr. The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American statistical Association*, 46(253), 1951.

[33] Joseph Menn. Amazon posts a tell-all of buying lists. 1999. Los Angeles Times. https://www.latimes.com/archives/la-xpm-1999-aug-26-fi-3760-story.html.

[34] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and optimizing LSTM language models. *arXiv preprint arXiv:1708.02182*, 2017.

[35] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. An analysis of neural language modeling at multiple scales. *arXiv preprint arXiv:1803.08240*, 2018.

[36] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Interspeech*, volume 2, page 3, 2010.

[37] Milad Nasr, Reza Shokri, and Amir Houmansadr. Machine learning with membership privacy using adversarial regularization. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 634–646. ACM, 2018.

[38] Seong Joon Oh, Max Augustin, Mario Fritz, and Bernt Schiele. Towards reverse-engineering black-box neural networks. In *ICLR*, 2018.

[39] A O'hagan and Tom Leonard. Bayes estimation subject to uncertainty about parameter constraints. *Biometrika*, 63(1), 1976.

[40] Le Trieu Phong, Yoshinori Aono, Takuya Hayashi, Lihua Wang, and Shiho Moriai. Privacy-preserving deep learning: Revisited and enhanced. In *International Conference on Applications and Techniques in Information Security*, pages 100–110. Springer, 2017.

[41] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *IEEE Symposium on Security and Privacy*, 2017.

[42] Ravid Shwartz-Ziv and Naftali Tishby. Opening the black box of deep neural networks via information. *arXiv preprint 1703.00810*, 2017.

[43] Congzheng Song, Thomas Ristenpart, and Vitaly Shmatikov. Machine learning models that remember too much. In *ACM CCS*, 2017.

[44] Congzheng Song and Vitaly Shmatikov. The natural auditor: How to tell if someone used your words to train their model. *arXiv preprint arXiv:1811.00513*, 2018.

[45] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *JMLR*, 15(1):1929–1958, 2014.

[46] Igor V. Tetko, David J. Livingstone, and Alexander I. Luik. Neural network studies. 1. Comparison of overfitting and overtraining. *Journal of Chemical Information and Computer Sciences*, 35(5):826–833, 1995.

[47] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[48] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction APIs. In *USENIX Security Symposium*, pages 601–618, 2016.

[49] Stacey Truex, Ling Liu, Mehmet Emre Gursoy, Lei Yu, and Wenqi Wei. Towards demystifying membership inference attacks. *arXiv preprint arXiv:1807.09173*, 2018.

[50] Binghui Wang and Neil Zhenqiang Gong. Stealing hyperparameters in machine learning. *arXiv preprint arXiv:1802.05351*, 2018.

[51] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[52] Yuanshun Yao, Bimal Viswanath, Jenna Cryan, Haitao Zheng, and Ben Y Zhao. Automated crowdturfing attacks and defenses in online review systems. *ACM CCS*, 2017.

[53] Samuel Yeom, Irene Giacomelli, Matt Fredrikson, and Somesh Jha. Privacy risk in machine learning: Analyzing the connection to overfitting. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 268–282. IEEE, 2018.

[54] Yang You, Igor Gitman, and Boris Ginsburg. Scaling SGD batch size to 32k for ImageNet training. *arXiv preprint arXiv:1708.03888*, 2017.

[55] Matthew D Zeiler. ADADELTA: An adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[56] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *ICLR*, 2017.

A Additional Memorization Experiments

A.1 Across Different Architectures

We evaluate different neural network architectures in Table 4 again on the PTB dataset, and find that all of them unintentionally memorize. We observe that the two recurrent neural networks, i.e., LSTM [23] and GRU [8], demonstrate both the highest accuracy (lowest loss) and the highest exposure. Convolutional neural networks' accuracy and exposure are both lower. Therefore, through this experiment, we show that the memorization is not only an issue to one particular architecture, but appears to be common to neural networks.

A.2 Across Training Strategies

There are various settings for training strategies and techniques that are known to impact the accuracy of the final model. We briefly evaluate the impact that each of these have on the exposure of the inserted canary.

Batch Size. In stochastic gradient descent, we train on mini-batches of multiple examples simultaneously, and average their gradients to update the model parameters. This is usually done for computational efficiency—due to their parallel nature, modern GPUs can evaluate a neural network on many thousands of inputs simultaneously.

To evaluate the effect of the batch size on memorization, we train our language model with different capacity (i.e., number of LSTM units) and batch size, ranging from 16 to 1024. (At each batch size for each number of units, we train 10

Architecture	Layers	Units	Test Loss	Exposure
GRU	1	370	1.18	36
GRU	2	235	1.18	37
LSTM	1	320	1.17	38
LSTM	2	200	1.16	35
CNN	1	436	1.29	24
CNN	2	188	1.28	19
CNN	4	122	1.25	22
WaveNet	2	188	1.24	18
WaveNet	4	122	1.25	20

Table 4: Exposure of a canary for various model architectures. All models have 620K (+/- 5K) parameters and so have the same theoretical capacity. Convolutional neural networks (CNN/WaveNet) perform less well at the language modeling task, and memorize the canary to a lesser extent.

		Number of LSTM Units				
		50	100	150	200	250
Batch Size	16	1.7	4.3	6.9	9.0	6.4
	32	4.0	6.2	14.4	14.1	14.6
	64	4.8	11.7	19.2	18.9	21.3
	128	9.9	14.0	25.9	***32.5***	***35.4***
	256	12.3	21.0	26.4	**28.8**	**31.2**
	512	**14.2**	**21.8**	***30.8***	26.0	26.0
	1024	***15.7***	***23.2***	**26.7**	27.0	24.4

Table 5: Exposure of models trained with varying model sizes and batch sizes. Models of the same size trained for the same number of epochs and reached similar test loss. Larger batch sizes, and larger models, both increase the amount of memorization. The largest memorization in each column is highlighted in italics bold, the second largest in bold.

models and average the results.) All models with the same number of units reach nearly identical final training loss and testing loss. However, the models with larger batch size exhibit significantly more memorization, as shown in Table 5. This experiment provides additional evidence for prior work which has argued that using a smaller batch size yields more generalizable models [24]; however we ensure that all models reach the same final accuracy.

While this does give a method of reducing memorization for some models, it unfortunately comes at a significant cost: training with a small batch can be prohibitively slow, as it may prevent parallelizing training across GPUs (and servers, in a decentralized fashion).[6]

Shuffling, Bagging, and Optimization Method. Given a fixed batch-size, we examine how other choices impact memorization. We train our model with different optimizers: SGD, Momentum SGD, RMSprop [47], Adagrad [13], Adadelta [55], and Adam [27]; and with either shuffling, or bagging (where minibatches are sampled with replacement).

Not all models converge to the same final test accuracy. However, when we control for the final test accuracy by taking a checkpoint from an earlier epoch from those models that perform better, we found no statistically significant difference in the exposure of the canary with any of these settings; we therefore do not include these results.

A.3 Across Formats and Context

We find that the context we are aware of affects our ability to detect whether or not memorization has occurred.

In our earlier experiments we computed exposure with the prefix "The random number is" and then placing the random-

[6]Recent work has begun using even larger batch sizes (e.g., 32K) to train models orders of magnitude more quickly than previously possible [21, 54].

Format	Exposure at Epoch 5	Exposure at Epoch 10
○○○○○○○○	5.0	6.1
⟨s⟩ ○○○○○○○○	6.3	7.1
○○○○○○○○ ⟨e⟩	5.0	6.8
⟨s⟩ ○○○○○○○○ ⟨e⟩	6.1	7.5
○○○-○○-○○○○	5.1	9.5
○-○-○-○-○-○-○-○-○	5.2	11.1

Table 6: Exposure of canaries when the we are aware of different amounts of surrounding context (⟨s⟩ and ⟨e⟩ are in practice unique context characters of five random characters). The exposure is computed at epoch 5 and 10, before the models completely memorize the inserted canary.

ness as a suffix. What if instead we knew a suffix, and the randomness was a prefix? Alternatively, what if the randomness had a unique structure (e.g., SSNs have dashes)?

We find that the answer is yes: additional knowledge about the format of the canary increases our ability to detect it was memorized. To show this, we study different insertion formats, along with the exposure of the given canary after 5 and 10 epochs of training in Table 6, averaged across ten models trained with each of the formats.

For the first four rows of Table 6, we use the same model, but compute the exposure using different levels of context. This ensures that it is only our ability to detect exposure that changes. For the remaining two rows, because the format has changed, we train separate models. We find that increasing the available context also increases the exposure, especially when inner context is available; this additional context becomes increasingly important as training proceeds.

Improving Robustness of ML Classifiers against Realizable Evasion Attacks Using Conserved Features

Liang Tong
Washington University in St. Louis

Bo Li
UIUC

Chen Hajaj
Ariel University

Chaowei Xiao
University of Michigan

Ning Zhang
Washington University in St. Louis

Yevgeniy Vorobeychik
Washington University in St. Louis

Abstract

Machine learning (ML) techniques are increasingly common in security applications, such as malware and intrusion detection. However, ML models are often susceptible to *evasion attacks*, in which an adversary makes changes to the input (such as malware) in order to avoid being detected. A conventional approach to evaluate ML robustness to such attacks, as well as to design robust ML, is by considering simplified *feature-space* models of attacks, where the attacker changes ML features directly to effect evasion, while minimizing or constraining the magnitude of this change. We investigate the effectiveness of this approach to designing robust ML in the face of attacks that can be realized in actual malware (*realizable attacks*). We demonstrate that in the context of structure-based PDF malware detection, such techniques appear to have limited effectiveness, but they are effective with content-based detectors. In either case, we show that augmenting the feature space models with *conserved* features (those that cannot be unilaterally modified without compromising malicious functionality) significantly improves performance. Finally, we show that feature space models enable generalized robustness when faced with a variety of realizable attacks, as compared to classifiers which are tuned to be robust to a specific realizable attack.

1 Introduction

Machine learning (ML) has come to be widely used in a broad array of settings, including important security applications such as network intrusion, fraud, and malware detection, as well as other high-stakes settings, such as autonomous driving. A general approach is to extract a set of *features*, or numerical attributes, of entities in question, collect a training data set of labeled examples (for example, indicating which instances are malicious and which are benign), and learn a model which labels previously unseen instances, presented in terms of their extracted features. Success of ML in malware detection is particularly striking, with ML-based static detection of malicious entities at times exceeding 99% accuracy [36, 37].

Nevertheless, ML-based techniques are often susceptible to *adversarial examples*, an important special case of which are *evasion attacks*. In a prototypical case of an evasion attack, an adversary modifies malware code so that the resulting malware is categorized as benign by ML, but still successfully executes the malicious payload [12, 16, 26, 37, 44]. An even broader class of adversarial examples features attacks that manipulate an object, such as a stop sign, so that a computer vision pipeline misclassifies it as another object (such as a speed limit sign) [10, 15, 33].

In response, a host of methods emerged for making ML robust to adversarial examples, the most potent of which are those based on game-theoretic approaches, robust optimization (including certified robustness), and adversarial retraining [5, 15, 23, 25, 32, 42, 43, 46]. A fundamental ingredient in all of these are *feature-space models of attacks*. Specifically, the attacker is assumed to directly modify values of features, with either a constraint or a penalty on the aggregate feature change measured in terms of an l_p norm.

Such feature-space models of attacks are clearly abstractions of reality. First, arbitrary modifications of feature values may not be *realizable*. For example, adding a benign object to a malicious PDF (with no other changes) necessarily increases its size, and so setting the associated feature to 1 (from 0) and simultaneously reducing file size may not be practically feasible. Second, the key goal for an adversary is to create a target malicious effect, such as to execute a malicious payload. Limiting feature modifications to be small in some l_p norm clearly need not capture this: one can insert many no-ops (resulting in a large change according to an l_p norm) with no impact on malicious functionality, and conversely, minimal changes (such as removing a Javascript tag) may break malicious functionality. Nevertheless, an implicit assumption in robust ML approaches is that the feature-space models capture reality sufficiently to yield ML models that are robust even to realizable attacks. *The goal of our work is to evaluate the validity of this implicit assumption* in the context of PDF malware detection.

Our first contribution is to evaluate feature-space evasion attack models in the context of PDF malware detection, using EvadeML as a realizable attack [44]. Specifically, we consider four ML-based approaches for PDF malware detection: two based on features that capture PDF file structure (SL2013 [36] and Hidost [38]), and two based on PDF file content (two Mimicus variants of PDFRate [35, 37]). In all cases, we show that successful defense against a given realizable attack is feasible (by retraining with this attack). In the case of structure-based detectors, we demonstrate that adversarial retraining in the feature space does not lead to adequate robustness against realizable attacks. In contrast, adversarial retraining in the feature space is effective in the case of content-based detectors. In other words, the nature of the feature space can matter a great deal.

Our second contribution is a method for boosting robustness of feature-space models without compromising their mathematical convenience (crucial for most approaches for robust ML). The key idea is to identify *conserved features*, that is, features that cannot be unilaterally modified without compromising malicious functionality. We exhibit such features in our setting, show that they cannot be identified with traditional statistical methods, and develop an algorithm for automatically extracting them. Finally, we show that by simply constraining that these features remain unmodified in adversarial training, feature-space approaches become effective even for robust structure-based PDF malware detection.

Our third contribution is to explore the extent to which ML robustness is *generalizable* to multiple *distinct* realizable attacks. Specifically, we expose both a robust classifier that was retrained by using a realizable attack (EvadeML), and a model hardened using a feature-space attack (accounting for conserved features), to a series of realizable attacks. Our results reveal a stark difference between the two: ML models hardened using EvadeML are quite fragile; in contrast, ML models hardened using feature-space attacks exhibit uniformly high robustness to the other attacks. Remarkably, we demonstrate that ML models hardened using feature-space attacks remain robust *even against realizable attacks that defeat conserved features*.

2 Machine Learning in Security

2.1 Learning and Prediction

In the (supervised) machine learning literature, it is common to consider the problem abstractly. We are given a training dataset $\mathsf{D} = \{(x_i, y_i)\}$, where $x_i \in \mathsf{X} \subseteq \mathbb{R}^n$ are numeric feature vectors in some feature space X and $y_i \in \mathsf{L}$ are labels in a label space L. Each data point (or example) in D is assumed to be generated i.i.d. according to some unknown distribution P. We are also given a hypothesis (model) space, H, and our goal is to identify (*learn*) a good model $h \in \mathsf{H}$ in the sense that it yields a small expected error on new examples drawn from P. In practice, since P is unknown, one typically aims to find $h \in \mathsf{H}$ which (approximately) minimizes empirical error on training data D.

In security applications—as in others—one is not given numerical features; instead, we start with a collection of entities, such as executables, along with associated labels (we assume henceforth that these are available, as we focus here on supervised learning problems). We must then *design a collection of feature extractors*, where each feature extractor computes a numerical value of a corresponding feature from an input entity. For example, we extract a "size" feature by computing the size of an executable. Applying feature extractors to each entity in our dataset, and adding associated object labels, allow us to generate a dataset D to fit the conventional ML framework.

In this paper we focus on PDF malware detection, where the label space is binary: either a PDF file is benign (which we can code as -1), or malicious (which we can code as $+1$). In addition, several prior efforts presented techniques for defining *feature extractors* (commonly known simply as features) for PDF files [36, 37]. Applying such feature extractors to a PDF file dataset transforms this dataset into one comprised of numerical feature vectors and associated binary labels. The goal is to predict whether previously unseen PDFs (simulated by holding out a portion of our dataset as *test data*) are correctly labeled as malicious or benign.

2.2 Evasion Attacks

In an *evasion attack*, abstractly, one is given a learned model $h(x)$ (e.g., a SVM or neural network) which returns a label $y = h(x)$ (e.g., malicious or benign) for an arbitrary feature vector $x \in \mathsf{X}$ (e.g., extracted from a PDF file). The attacker additionally starts with an entity e (such as a malicious PDF file), from which we can extract a feature vector $\phi(e)$. The attacker then transforms e into another entity, e', with an associated feature vector $x' = \phi(e')$ so as to accomplish two goals: first, that $h(x')$ returns an erroneous label (in our running example, labels e' as benign based on its extracted features $\phi(e')$), and second, that e' preserves the functionality of the original entity e—which, in our example of PDF malware detection, entails preserving malicious functionality of e. The evasion attack as just described is presumed to transform the *entity itself*, such as the malicious PDF file, albeit accounting for the effect of such transformation on the extracted features $x' = \phi(e')$. We call attacks of this kind *realizable* evasion attacks. The process by which such realizable evasion attacks can be successfully accomplished is quite non-trivial, and typically warrants independent research contributions (e.g., [37, 44]).

In contrast, it is natural to short-circuit the complexity involved, and work directly in the *feature space*, as is conventional in the machine learning literature. In this case, the attacker is *modeled* as starting with a malicious feature vector x (*not the malicious entity* e), and *directly modifying the fea-*

tures to produce another feature vector $x' \in \mathsf{X}$, so as to yield erroneous predictions, i.e., $y' = h(x')$ (for example, being mislabeled as benign). Crucially, since we are no longer appealing to original entities, we must abstract away the notion of preserving (malicious) functionality. This is done through the use of a cost function, $c(x,x')$, whereby the attacker is penalized for greater modifications to the given feature vector x, commonly measured using an l_p norm difference between the original malicious instance and the modified feature vector [3, 23]. We term these the *feature-space models* of evasion attacks. Crucially, *essentially all approaches for robust ML, particularly the most successful ones, such as those based on robust optimization, leverage these models*.

2.3 Evasion Defense

A large number of approaches have been proposed for defending against evasion attacks or, more broadly, adversarial examples (e.g., [3, 5, 6, 29, 30, 32, 40, 42, 43]). While many have been shown inadequate [1, 7], the three generally effective approaches are: (a) game-theoretic reasoning, (b) robust optimization (a special case of (a) where the game is zero-sum), and (c) iterative adversarial retraining.[1] Game-theoretic methods in general, and robust optimization in particular, are not general-purpose, as solving these directly requires special structure, such as a continuous feature space and differentiability [3, 5, 6], and often additional structure of the learning model, such as linearity [43] or neural network architecture and activation functions [32, 42]. Finally, to date all have used the mathematical feature-space attack model at their core. In contrast, retraining can be performed without making assumptions about the nature of the learning algorithm or the adversarial model [23]. Since our study below involves realizable attacks (in addition to the mathematical models of attacks), non-linear SVM and, in all cases but one, binary features, iterative retraining is the sole defense that can be applied uniformly (which we require to ensure that our results are directly comparable).

3 Validating Models of ML Evasion Attacks

We have two major goals: 1) *validation*: to evaluate whether robust ML approaches that make use of feature-space models of evasion attacks are, indeed, robust against *real*—realizable—attacks, and 2) *generalizability*: to study generalizability of evasion defenses.

We start with a conceptual model of defense and attack as a Stackelberg game between ML ("defender"), who first chooses a defense θ (in our case, the learned model $h(x)$) and the attacker, who finds an optimal attack that *reacts* to the particular defense θ. An *attack model* captures how the attacker changes behavior in response to the defense θ. The defender's goal is to choose the best defense θ against such a reactive attacker, as captured by the attack model. Indeed, this is a common way to model the adversarial evasion problem in prior literature [5, 22, 40]. This model has two useful features. First, the attack is treated as an oracle in the sense that it returns an attack for an arbitrary defense θ. This allows us, in principle, to design a defense against an arbitrary evasion attack, making no distinction between feature-space attack models and realizable attacks. Second, we can separately consider *defense* against a specific attack (for example, a feature-space attack), and evaluation, which can use another attack (e.g., a realizable attack).

To be more precise, let $O(h;\mathsf{D})$ be an arbitrary attack which returns evasions given a dataset D and a classifier h, and let $u(h;O(h;\mathsf{D}))$ be the measure that the defender wishes to optimize (for example, accuracy on data *after* evasions). Then defense against the attack $O(h;\mathsf{D})$ amounts to solving the following optimization problem:

$$\max_h u(h;O(h;\mathsf{D})). \tag{1}$$

In practice, we need a means for approximately solving the optimization problem in Equation (1) for an arbitrary attack. To this end, we make use of *iterative retraining*, an approach previously proposed for hardening classifiers against evasion attacks [21, 23]. In particular, we use a variant of iterative retraining with provable guarantees [23], which is outlined as follows:

1. Start with the initial classifier.
2. Execute the *evasion attack* for each malicious instance in training data to generate a new feature vector.
3. Add all new data points to training data (removing any duplicates), and retrain the classifier.
4. Terminate after either a fixed number of iterations, or when no new evasions can be added.

Now, we describe our approach to validation and generalizability evaluations.

In *validation*, consider a model of an evasion attack, $\tilde{O}(h;\mathsf{D})$ (e.g., a feature-space attack model), which is a proxy for a "real" (realizable) attack, $O(h;\mathsf{D})$; note that each attack evades a given ML model h. We first find the defense against $\tilde{O}$ using the retraining procedure above; let the resulting robust classifier be $\tilde{h}$. Next, we *evaluate* $\tilde{h}$ by running the target realizable attack $O(\tilde{h};\mathsf{D})$. Finally, we create a *baseline* h^*, which is a robust classifier against a target realizable attack O. We then evaluate how well $\tilde{h}$ performs, compared to h^*, against the target attack. For example, if we find that $\tilde{h}$ is ineffective against the target attack, we say that $\tilde{O}$ is a poor attack proxy, whereas if it remains robust, we view $\tilde{O}$ as a good proxy for the target attack O. We focus on validation in Sections 5 and 6.

[1] Otherwise known as adversarial training, it can be viewed as an approach for obtaining approximate game-theoretic or robust optimization solutions [23, 25, 40].

In evaluating *generalizability*, the approach is slightly different. Again, we consider a proxy attack $\tilde{O}$ (which may now be either a feature-space model, or some particular realizable attack), and find a defense $\tilde{h}$ against this attack. For evaluation, we consider a *collection* of target attacks $\{O_i)\}$, and run each of these attacks against $\tilde{h}$. We say that our proxy attack is generalizable if $\tilde{h}$ remains robust to all, or most of the attacks i; otherwise, it fails to generalize. We consider generalizability in Section 7.

4 Experimental Methodology

We use malicious PDF detection as a case study to investigate robustness of ML hardened using feature-space models of evasion attacks. We now describe our experimental methodology. We start with some background on PDF structure, and proceed to describe the specific ML-based detectors, evasion attacks (both realizable, and feature-space), datasets, and evaluation metrics used in our experiments.

4.1 PDF Document Structure

The Portable Document Format (PDF) is an open standard format used to present content and layout on different platforms. A PDF file structure consists of four parts: *header*, *body*, *cross-reference table* (CRT), and *trailer*. The header contains information such as the magic number and format version. The body is the most important element of a PDF file, which comprises multiple PDF objects that constitute the content of the file. These objects can be one of the eight basic types: Boolean, Numeric, String, Null, Name, Array, Dictionary, and Stream. They can be referenced from other objects via indirect references. There are other types of objects, such as JavaScript which contains executable JavaScript code. The CRT indexes objects in the body, while the trailer points to the CRT.

The relations between objects with cross-references can be described as a directed graph that presents their logical structure by using edges representing reference relations and nodes representing different objects.As an object can be referred to by its child node, the resulting logical structure is a directed cyclic graph. To eliminate the redundant references, the logical structure can be reduced to a structural tree with the breadth-first search procedure.

4.2 Target Classifiers

Several PDF malware classifiers have been proposed [8, 35, 36, 38]. For our study, we selected SL2013 [36], Hidost [38] and two variants of PDFRate [35] (termed PDFRate-R and PDFRate-B respectively), displayed in Table 1. SL2013 and its revised version, Hidost, are *structure-based* PDF classifiers, which use the logical structure of a PDF document to construct and extract features used in detecting malicious PDFs. PDFRate, on the other hand, is a *content-based* classifier, which constructs features based on *medadata* and *content* information in the PDF file to distinguish benign and malicious instances. Evasion attacks on both SL2013 and PDFRate classifiers, particularly of the realizable kind, have been developed in recent literature [36–38, 44], providing a natural evaluation framework for our purposes.

Classifier	Feature type	Number of features
SL2013	Binary	6,087
Hidost	Binary	961
PDFRate-R	Real-valued	135
PDFRate-B	Binary	135

Table 1: Target classifiers.

4.2.1 Structure-Based Classifiers

SL2013: SL2013 is a well-documented and open-source machine learning system using Support Vector Machines (SVM) with a radial basis function (RBF) kernel, and was shown to have state-of-the-art performance [36]. It employs structural properties of PDF files to discriminate between malicious and benign PDFs. Specifically, SL2013 uses the presence of particular *structural paths* as binary features to present PDF files in feature space. A structural path of an object is a sequence of edges in the reduced (tree) *logical structure*, starting from the catalog dictionary and ending at this object. Therefore, the structural path reveals the shortest reference path to an object. SL2013 uses 6,087 most common structural paths among 658,763 PDF files as a uniform set for classification.
Hidost: Hidost is an updated version of SL2013. It inherits all the characteristics of SL2013 and employs *structual path consolidation* (SPC), a technique to consolidate features which have the same or similar semantic meaning in a PDF. As the semantically equivalent structural paths are merged, Hidost reduces polymorphic paths and still preserves the semantics of logical structure, so as to improve evasion-robustness of SL2013 [38].

In our work, we employ the 961 features identified in the latest version of Hidost.

4.2.2 PDFRate: A Content-Based Classifier

The original PDFRate classifier uses a random forest algorithm, and employs PDF *metadata* and *content* features. The metadata features include the size of a file, author name, and creation date, while content-based features include position and counts of specific keywords. All features were manually defined by Smutz and Stavrou [35].

PDFRate uses a total of 202 features, but only 135 of these are publicly documented [34]. Consequently, in our work we employ the Mimicus implementation of PDFRate which was shown to be a close approximation [37]. Mimicus trained a surrogate SVM classifier with the documented 135 features

and the same dataset as PDFRate, using both the SVM and random forest classifiers, both performing comparably. We use the SVM implementation in our experiments to enable more direct comparisons with the structure-based classifiers that also use SVM. An important aspect of Mimicus is *feature standardization* on extracted data points performed by subtracting the mean of the feature value and dividing by standard deviation, transforming all features to be real-valued and zero-mean (henceforth, PDFRate-R). This surrogate was shown to have $\sim 99\%$ accuracy on the test data [35]. In addition, we construct a *binarized* variant of PDFRate (henceforth, PDFRate-B), where each feature is transformed into a binary feature by assigning 0 whenever the feature value is 0, and assigning 1 whenever the feature value is non-zero.

4.3 Realizable Evasion Attacks

4.3.1 EvadeML

The primary realizable attack in our study is EvadeML [44], which allows insertion, deletion, and swapping of objects, and is consequently a stronger attack than most other realizable attacks in the literature, which typically only allow insertion to ensure that malicious functionality is preserved. EvadeML assumes that the adversary has black-box access to the classifier and can only get classification scores of PDF files, and was shown to effectively evade both SL2013 and PDFRate [44]. It employs genetic programming (GP) to search the space of possible PDF instances to find ones that evade the classifier while maintaining malicious features. First, an initial population is produced by randomly manipulating a malicious PDF repeatedly. The manipulation is either a deletion, an insertion, or a swap operation on PDF objects. After the population is initialized, each variant is assessed by the Cuckoo sandbox [17] and the target classifier to evaluate its fitness. The sandbox is used to determine if a variant preserves malicious behavior, such as API or network anomalies. The target classifier provides a classification score for each variant. If a variant is classified as benign but displays malicious behavior, or if GP reaches the maximum number of generations, then GP terminates with the variant achieving the best fitness score and the corresponding mutation trace is stored in a pool for future population initialization. Otherwise, a subset of the population is selected for the next generation based on their fitness evaluation. Afterward, the variants selected are randomly manipulated to generate the next generation of the population.

We use EvadeML as the primary realizable evasion model for the first part of the paper. We set the GP parameters in EvadeML as the same as in the experiments by Xu et al. [44]. The population size in each generation is 48. The maximum number of generations is 20. The mutation rate for each PDF object is 0.1. The mutation traces that lead to successful evasion and promising variants are stored and applied in our experiments. The fitness threshold of a classifier is 0. We use the same external benign PDF files as Xu et al. [44] to provide ingredients for insertion and swap operations.

4.3.2 The Mimicry Attack

Mimicry assumes that an attacker has full knowledge of the features employed by a target classifier. The mimicry attack then manipulates a malicious PDF file so that it mimics a particular selected benign PDF as much as possible. The implementation of Mimicry is simple and independent of any particular classification model.

Our mimicry attack uses the Mimicus [37] implementation, which was shown to successfully evade the PDFRate classifier. To improve its evasion effectiveness, Mimicus chooses 30 different target benign PDF files for each attack file. It then produces one instance in feature space for each target-attack pair by merging the malicious features with the benign ones. The feature space instance is then transformed into a PDF file using a *content injection approach*. The resulting 30 files are evaluated by the target classifier, and only the PDF with the best evasion result is selected, which was submitted to WEPAWET [8] to verify malicious functionality. To make Mimicry consistent with our framework, we employ the Cuckoo sandbox [17] in place of WEPAWET (which was in any case discontinued) to validate maliciousness of the resulting PDF file.

In addition to the original version of Mimicry, we implement an enhanced variation, *Mimicry+*, with two modifications. First, Mimicry+ chooses the 30 most benign PDF files predicted by the target classifier as target files (instead of randomly selecting those, as in Mimicry). Second, for each attack file, all the resulting 30 files are evaluated by the sandbox and only those verified to have malicious functionality are selected to evade the target classifier.

4.3.3 MalGAN

MalGAN [19] is a Generative Adversarial Network [14] framework to generate malware examples which can evade a black-box malware detector with binary features. It assumes that an attacker knows the features, but has only black-box access to the detector decisions. MalGAN comprises three main components: a generator which transforms malware to its adversarial version, a black-box detector which returns detection results, and a substitute detector which is used to fit the black-box detector and train the generator. The generator and substitute detector are feed-forward neural networks which work together to evade the black-box detector. The results of [19] show that MalGAN is able to decrease the *True Positive Rate* on the generated examples from $> 90\%$ to 0%. We note that strictly speaking, MalGAN variants are not implemented as actual PDF files; however, we still treat it as a realizable attack since it only adds features to a malicious

Entry	Hexadecimal Representation
/Action	/#41#63#74#69#6f#6e
/Filter	/#46#69#6c#74#65#72
/Length	/#4c#65#6e#67#74#68
/JavaScript	/#4a#61#76#61#53#63#72#69#70#74
/JS	/#4a#53
/S	/#53
/Type	/#54#79#70#65

Table 2: Transformation of entry names in the custom attack.

file, which can be implemented (at least in structure-based detection) by adding the associated objects into the PDF file.

4.3.4 Reverse Mimicry

The *Reverse Mimicry* attack assumes that an attacker has zero knowledge of the target classifier. The basic idea is to inject malicious payloads into target benign files to minimize the structural difference between the resulting examples and targets. Our Reverse Mimicry attack employs the adversarial examples provided by Maiorca et al. [26] which was shown to successfully evade PDF classifiers based on structural analysis. Specifically, we use the 500 PDF files produced by injecting a malicious JavaScript code that does not contain references to other objects to target benign PDF files. We selected the 376 files out of 500 that display malicious behaviors detected by the Cuckoo sandbox.

4.3.5 The Custom Attack

We implemented a custom attack which exploits a feature extraction vulnerability in the Mimicus implementation of PDFRate. Normally, the characters used in the Name objects of a PDF file are limited to a specific set. Since PDF specification version 1.2, a lexical convention has been added to represent a character with its hexadecimal ANSI-code, e.g., #xx. Such a modification enables us to create an arbitrary string in the form of #xx#xx#xx. In our implementation, we replaced a set of entries in the attack PDF files with their hexadecimal representations (see Table 2). These features were selected with the goal to obfuscate tags crucial to the code execution in PDF, which are frequently used for feature extraction. With this technique, the scanner would not be able to detect malicious code without dynamically reconstructing the PDF structure. While it is theoretically possible to replace all the ASCII text inside the document, we chose not to do that due to the concern on the expansion of file size.

4.4 Feature-Space Evasion Model

In typical realizable attacks, including EvadeML, a consideration is not merely to move to the benign side of the classifier decision boundary, but to appear as benign as possible. This naturally translates into the following multi-objective optimization in feature space:

$$\underset{x}{\text{minimize}} \quad Q(x) = f(x) + \lambda c(x_M, x), \tag{2}$$

where $f(x)$ is the score of a feature vector x, with the actual classifier (such as SVM) $g(x) = \text{sgn}(f(x))$, x_M the malicious seed, x an evasion instance, $c(x_M, x)$ the cost of transforming x_M into x, and λ a parameter which determines the feature transformation cost. We use l_2 norm distance between x_M and x as the cost function: $c(x_M, x) = \sum_i |x_i - x_{M,i}|^2$. Since in most of our experiments features are binary, the choice of l_2 norm (as opposed to another l_p norm) is not critical.

As the optimization problem in Equation (2) is non-convex and variables are binary in three of the four cases we consider, we use a stochastic local search method designed for combinatorial search domains, *Coordinate Greedy* (alternatively known as iterative improvement), to compute a local optimum (the binary nature of the features is why we eschew gradient-based approaches) [18, 23]. In this method, we optimize one randomly chosen coordinate of the feature vector at a time, until a local optimum is reached. To improve the quality of the resulting solution, we repeat this process from several random starting points. This approach has been shown to be extremely effective for computing evasion instances in binary domains [23].

4.5 Datasets

The dataset we use is from the *Contagio Archive*.[2] We use 5,586 malicious and 4,476 benign PDF files for training, and another 5,276 malicious and 4,459 benign files as the non-adversarial test dataset. The training and test datasets also contain 500 seeds selected by Xu et al. [44], with 400 in the training data and 100 in the test dataset. These seeds are filtered from 10,980 PDF malware samples and are suitable for evaluation since they are detected with reliable malware signatures by the Cuckoo sandbox [17]. We randomly select 40 seeds from the training data as the retraining seeds and use the 100 seeds in the test data as the test seeds.

4.6 Implementation of Iterative Adversarial Retraining

We made a small modification to the general iterative retraining approach described in Section 3 when it uses EvadeML as the realizable attack $O(h; \mathsf{D})$. Specifically, we used only 40 malicious seeds to EvadeML to generate evasions, to reduce running time and make the experiment more consistent with realistic settings where a large proportion of malicious data is not adapting to the classifier. As shown below, this set of 40 instances was sufficient to generate a model robust to evasions from held out 100 malicious seed PDFs.

[2] Available at the following URL: http://contagiodump.blogspot.com/2013/03/16800-clean-and-11960-malicious-files.html.

We distribute both retraining and adversarial test tasks on two servers (Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz, 18 cores and 64 GB memory, running Ubuntu 16.04). For retraining using EvadeML as the attack, we assign each server 20 seeds; each seed is processed by EvadeML to produce the adversarial evasion instances. We then add the 40 examples obtained to the training data, retrain the classifier, and then split the seeds between the two servers in the next iteration. In the evaluation phase, we assign each server 50 seeds from the 100 test instances, and each seed is further used to evade the classifier by using EvadeML.

4.7 Evaluation Metrics

We evaluate performance in two ways: 1) evaluation of evasion robustness (which is central to our specific inquiry), and 2) traditional evaluation. To evaluate robustness, we compute the proportion of 100 malicious test seed PDFs for which EvadeML successfully evades the classifier; this is our metric of *evasion robustness*, evaluated with respect to EvadeML. Thus, evasion robustness of 0% means that the classifier is successfully evaded in every instance, while evasion robustness of 100% means that evasion fails every time. Our traditional evaluation metric uses test data of malicious and benign PDFs, where no evasions are attempted. On this data, we compute the ROC (receiver operating characteristic) curve and the corresponding AUC (area under the curve).

5 Efficacy of Feature-Space Attack Models

We now undertake our first task: evaluation of the effectiveness of robust ML obtained by using the abstract feature-space models of attack. We compare to a baseline classifier obtained by retraining with the most potent attack on our menu, EvadeML (which, in addition to inserting content, as done by other attacks [19, 26, 37], also allows the attacker to delete and swap PDF objects). We can think of our baseline as assuming that the defender knows that EvadeML is employed by the attacker, along with its hyperparameters. Throught this and next section, we also use EvadeML to evaluate the effectiveness of classifiers hardened using a feature-space model, in comparison with the above baseline.

5.1 Structure-Based PDF Malware Classification

Our first case study uses a state-of-the-art PDF malware classifier which engineers features based on PDF *structure*. Indeed, we evaluate two versions of this classifier: an earlier version, which we call *SL2013*, and a more recent version, which we call *Hidost*. The experiments by Xu et al. [44] demonstrate that SL2013 can be successfully evaded. Since Hidost was a recent redesign attempting in part to address its vulnerability to mimicry attacks by significantly reducing the feature space,

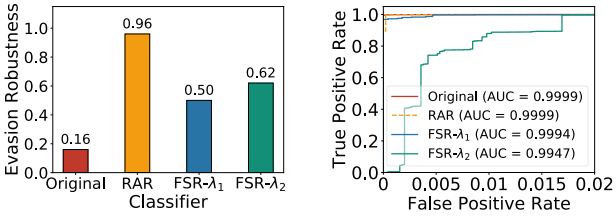


Figure 1: Evasion robustness under EvadeML test (left) and performance on non-adversarial data (right) of different classifiers for SL2013.

no data exists on its vulnerability to evasion attacks. Below we demonstrate that Hidost is also vulnerable to evasion attacks (indeed, more so than SL2013).

From the perspective of defense, we show that it is possible to harden both SL2013 and Hidost against a powerful realizable EvadeML attack by simply retraining with this attack (*RAR*, for *realizable-attack retraining*, henceforth refers to a model hardened using EvadeML). This serves as a baseline we use to evaluate the efficacy of a retraining defense with a feature-space attack model (henceforth, FSR for *feature-space retraining*). We then show that for both SL2013 and Hidost, FSR significantly underperforms RAR.

In our experiments, we empirically set the *RBF* parameters for training both SL2013 and Hidost to $C = 12$ and $\gamma = 0.0025$.

5.1.1 SL2013

Retraining with a Powerful Realizable Attack First, we replicated the EvadeML attack on the original SL2013; the classifier achieves only a 16% evasion robustness.[3] Next, to create a baseline, we conduct experiments in which EvadeML is employed to retrain SL2013. The process terminated after 10 iterations at which point no evasive variants of the 40 retraining seeds could be generated. We observe (Figure 1 (left)) that the retrained classifier (RAR) obtained by this approach achieves a 96% evasion robustness. Moreover, RAR is essentially as accurate as the baseline SL2013 on non-adversarial data (Figure 1 (right)). Thus, it is clearly possible to be highly robust to this evasion attack without significantly compromising effectiveness on data not featuring explicit evasion attacks.

Figure 2 (left) shows the gradual improvement of evasion robustness over the 10 retraining iterations. This plot demonstrates non-trivial effectiveness of EvadeML: the first few iterations are clearly insufficient, as re-running EvadeML creates many new evasions that cannot be correctly detected by

[3] This result differs from the experiments in [44] which show a 0% evasion robustness. We found a flaw in the implementation of feature extraction in EvadeML which causes evaluation to be performed using the wrong feature vectors. This bug has been fixed in the GitHub version of EvadeML.

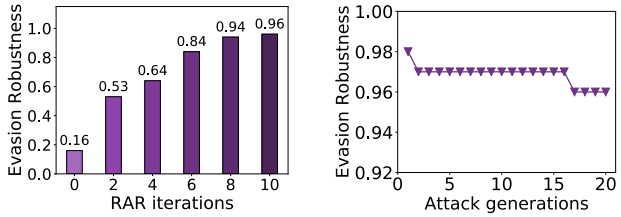


Figure 2: Evasion robustness with retraining iterations (left) and generations of the EvadeML attack test (right).

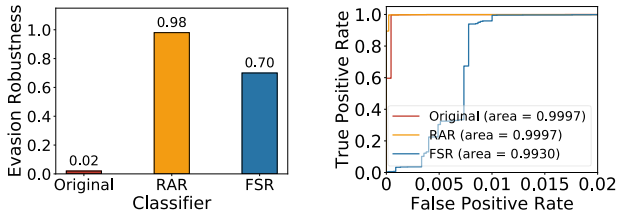


Figure 3: Evasion robustness under EvadeML test (left) and performance on non-adversarial data (right) of different classifiers for Hidost.

the classifier. Only after 6 iterations does EvadeML optimization loop begin to show significant signs of failing. Figure 2 (right) shows how increasing the number of generations in EvadeML attacks affects robustness of the RAR classifier. At this point, we can see that increasing the capability of the attack has minimal impact.

Feature-Space Retraining Next, we experimentally evaluate the effectiveness of retraining with a feature-space model of evasion attacks in obtaining robust ML in the face of the EvadeML realizable attack. We consider the setting with $\lambda = 0.05$ and $\lambda = 0.005$ in Equation 2 (henceforth, FSR-λ_1 and FSR-λ_2).

The robustness results are shown in Figure 1 (left). Compared to the SL2013 baseline, feature-space retraining (FSR) boosts evasion robustness from 16% to 62%. Crucially, *the robustness of the resulting classifier is far below the classifier achieved by RAR.* This illustrates that defense that relies on feature-space models of adversarial examples may not in fact lead to robustness when it is faced with a real attack.

We again consider performance of FSR classifier on non-adversarial test data (Figure 1 (right)). We can see that robustness boosting again does not much degrade performance, with AUC remaining above 99%. However, we do see a substantial degradation as we move from $\lambda = 0.05$ to 0.005; thus, as we increase adversarial power in the feature-space model, while we do obtain a slightly more robust model, we incur a nontrivial hit in performance on non-adversarial data.

5.1.2 Hidost

We now repeat our experiments above with another structure-based classifier, Hidost. We set the retraining parameter $\lambda = 0.005$, which appears to strike a reasonable balance between robustness and accuracy on non-adversarial data. As before, we first evaluated the robustness of the original Hidost [38] by EvadeML. The result shows a 2% robustness—remarkably, significantly worse than SL2013.

Evasion robustness of Hidost, as well as improvements achieved by RAR and FSR, are shown in Figure 3 (left), and the results are consistent with our observations for SL2013. First, by retraining with the realizable attack, evasion robustness is boosted to 98%, a rather dramatic improvement, and clear demonstration that successful defense is possible. In contrast, FSR achieves a 70% evasion robustness, a significant boost over the original Hidost, to be sure, but far below the evasion robustness of RAR.

Evaluating these classifiers on non-adversarial test data in terms of ROC curves (Figure 3 (right)), we can observe that RAR achieves comparable accuracy ($> 99.9\%$ AUC) with the original Hidost classifier on non-adversarial data, and provides even better *True Positive Rate (TPR)* when *False Positive Rate (FPR)* is close to zero. On the other hand, FSR achieves $> 99\%$ AUC, but yields a significant degradation of TPR when FPR< 0.01.

5.2 Content-Based PDF Malware Classification

Our next case study concerns another two PDF malware classifiers which use features based on PDF file content, rather than logical structure. We trained both real-valued and binarized PDFRate (henceforth, PDFRate-R and PDFRate-B) on the same dataset as SL2013 and Hidost, and achieved $> 99.9\%$ AUC for both classifiers on test data. In our experiments, we empirically set the SVM *RBF* parameters for training to $C = 10$ and $\gamma = 0.01$. In our evaluation of ML robustness, we again set the feature-space model parameter λ to be 0.005.

5.2.1 PDFRate with Real-Valued Features

We begin with the variant of PDFRate—PDFRate-R—which has been constructed in previous evaluations and shown comparable in performance to the original implementation [37]. We again begin by replicating the EvadeML evasion robustness evaluation of the baseline classifier. As expected, we find the classifier quite vulnerable, with only 2% evasion robustness.

Next, we retrain PDFRate-R with EvadeML for 10 iterations (RAR baseline), and perform feature-space retraining using the conventional feature space model above. Our results are shown in Figure 4 (left). Observe that while RAR

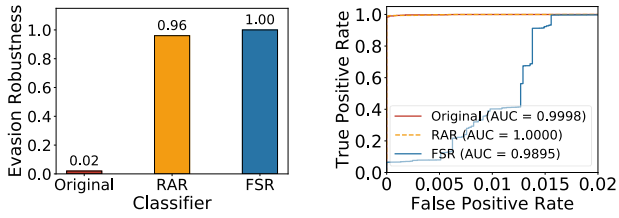


Figure 4: Evasion robustness under EvadeML test (left) and performance on non-adversarial data (right) of different classifiers for PDFRate-R.

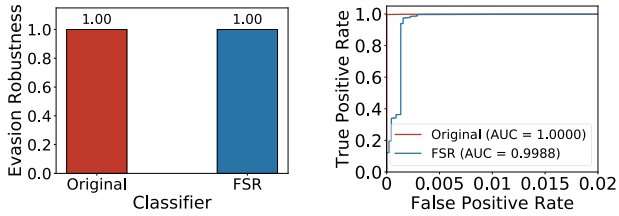


Figure 5: Evasion robustness under EvadeML test (left) and performance on non-adversarial data (right) of different classifiers for PDFRate-B.

indeed achieves a highly robust classifier (96% robustness), FSR actually performs *even better*, with 100% robustness.

Comparing RAR and FSR performance on non-adversarial data (Figure 4 (right)), we observe that the high robustness of FSR does incur a cost: while RAR remains exceptionally effective (>99.99% AUC), FSR achieves AUC slightly lower than 99%, although most significantly, the degradation is rather pronounced for low FPR regions (below 0.015).

5.2.2 PDFRate with Binarized Features

One of our great surprises is the robustness of the binarized PDFRate: despite the fact that the real-valued PDFRate is quite vulnerable, *the same classifier using binary features was 100% robust to EvadeML* (Figure 5 (left)). Consequently, this will serve as our robust baseline (equivalently, RAR would terminate with no iterations). Feature-space retrained PDFRate-B also exhibits 100% evasion robustness, although it does require a number of iterations to converge.

Considering now the performance of PDFRate-B and FSR on non-adversarial test data (Figure 5 (right)), we can make two interesting observations. First, the baseline PDFRate-B is remarkably good even on this data; in a sense, it appears to hit the sweet spot of adversarial robustness and non-adversarial performance. Second, FSR retrained classifier is competitive in terms of AUC ($\sim$ 99.9%), but is observably worse than the baseline classifier for very low false positive rates.

6 Evasion-Robust Classification with Conserved Features

Thus far, we had observed that ML hardened with the standard mathematically convenient feature-space evasion attack model may in some cases not yield satisfactory robustness against real attacks. The key issue is that feature-space models are entirely disembodied from the domain. This is crucial to enable us to have mathematical formulations of attacks, but clearly has limitations. The key question is whether we can devise a simple way of anchoring feature-space attacks in the application domain to allow us to meaningfully and minimally constrain abstract attacks to reflect some of the constraints that real attacks face. Next, we propose a refinement of the feature-space model that aims to do just that.

Specifically, we introduce the idea of *conserved features*, which we define to be *features, the unilateral modification of which compromises malicious functionality.* We develop this idea specifically for *binary features*, as this notion is particularly crisp in such a case (e.g., such features tend to correspond to the existence of particular objects in PDF).

Next, we present three major findings. First, conserved features do exist in all three of our classifiers over the binary feature space, and can be effectively identified (see our algorithm for identifying conserved features in Appendix A). Second, conserved features cannot be recovered using statistical feature reduction (in our case, sparse regularization), and feature reduction methods do not lead to robust classifiers. The reason is that conservation is connected to the relationship between features and malicious functionality, rather than statistical properties of non-evasion data; for example, features which are strongly correlated with malicious behavior are often a consequence of attacker "laziness" (such as whether a PDF file has an author), and are easy for attackers to change. Third, we demonstrate that the limitations of feature-space robust ML can be substantially alleviated by incorporating conserved features as attack invariants in the feature-space evasion model.

To develop intuition about the nature of conserved features, consider SL2013, which employs structural paths as features to discriminate between malicious and benign PDFs. On the one hand, the structural paths like `/Type` are unessential to preserve malicious behaviors, and we do not expect them to be conserved. On the other hand, as the shellcode which triggers malicious functionality is embedded in certain PDF objects, those corresponding structural paths are likely to be conserved in each variant crafted from the same malicious seed (e.g., `/OpenAction/JS`). In addition, structural paths that facilitate embedded script in PDF files also can be conserved features as removing them can break the script (e.g., `/Names` and `/Pages`). This further illustrates that conserved features are not necessarily optimal for statistically distinguishing benign and malicious instances (indeed, these may be common to both); rather, they serve to anchor the feature-

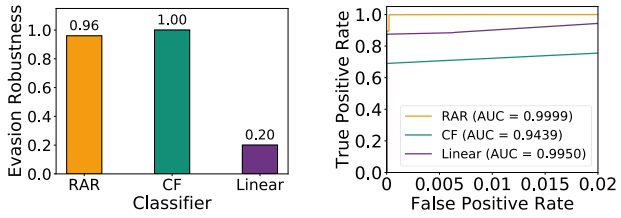


Figure 6: Classifying with conserved features: comparing evasion robustness (left) and ROC curves (right).

space attack model in the domain by connecting features to malicious functionality.

6.1 Classifying Using Only Conserved Features

We begin by exploring the effectiveness of using *only* conserved features for classification. We identified 8 conserved features for SL2013 (out of ∼6000), 7 for Hidost (out of ∼1000), and 4 for PDFRate-B (out of 135); these are detailed in Table 3 of the appendix, while our algorithm for identifying conserved features is presented in Appendix A.

We start by considering four natural questions pertaining to conserved features: 1) are they sufficient to make a classifier robust to evasions, 2) do they effectively discriminate between benign and malicious instances, 3) can they be identified using standard statistical methods (such as sparse regularization), and 4) are they just detecting the presence of JavaScript in PDF?

We explore these for SL2013. Specifically, we trained a classifier using *only* the 8 conserved features (*CF* henceforth). As we can see in Figure 6 (left), this classifier is 100% robust to EvadeML attacks, appearing to resolve the first question. However, we emphasize that conserved features alone need not capture the full spectrum of adversarial behavior and constraints. Indeed, in Section 7 we show that classifiers based solely on conserved features can also be evaded, particularly if attacks are *specifically designed to evade them*. Rather, as we show presently, they provide a *sufficient anchoring* in the problem domain for feature-space attack models to succeed.

To address question (2), consider Figure 6 (right): clearly, if we desire a low false positive rate, using only conserved features for classification yields subpar performance on non-adversarial data. To address the third question, we learn a linear SVM classifier for SL2013 with l_1 regularization (henceforth, Linear) where we empirically adjust the SVM parameter C to perform feature reduction until the number of the features is also 8; we find that only 3 of these are conserved features (see Appendix A.6 for a more detailed analysis of the relationship between statistically useful and conserved features). As we can see in Figure 6 (left), this classifier exhibits poor robustness; thus, statistical methods are insufficient to identify good conserved features.

To address the fourth question, we create a classifier using only one boolean feature which identifies the presence of JavaScript in a PDF file (henceforth, we refer to this feature as *JS*). We find that this classifier is also robust to EvadeML. On non-adversarial data, JS achieves FPR of 0.04 and FNR of 0.14 (in other words, 4% of the benign files in the non-adversarial dataset use JavaScript, while 14% of malicious instances use alternative attacks to Javascript).[4] To create an apples-to-apples comparison with the CF classifier, we empirically adjust the classification threshold of CF until we get the same FPR with JS. The resulting CF classifier exhibits FNR of 0.11, considerably better than JS. Nevertheless, it is clear that using either CF (only conserved features), or only JS, is impractical, since both FNR and FPR of these are quite high. Moreover, as we show in Section 7, classifiers based only on conserved features can be defeated by other realizable attacks. Next, we show that identification of conserved features is nevertheless crucial in creating highly effective feature-space attack models.

6.2 Feature-Space Model with Conserved Features

As discussed above, the feature-space evasion model in Equation (2) may not sufficiently boost ML robustness. Since conserved features allow us to minimally tie the abstract feature-space representation to malicious functionality, we offer a natural modification of the model in Equation (2), imposing the constraint that conserved features cannot be modified by the attacker. We formally capture this in the new optimization problem in Equation (3), where S is the set of conserved features:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & Q(x) = f(x) + \lambda c(x_M, x), \\ \text{subject to} \quad & x_i = x_{M,i},\ \forall i \in \mathsf{S}. \end{aligned} \tag{3}$$

Other than this modification, we use the same *Coordinate Greedy* algorithm with random restarts as before to compute adversarial examples. We adopt the evasion model in Equation (3) to retrain the target classifier using the retraining procedure from Section 4. We denote the classifier obtained by the retraining procedure using a feature-space model grounded by conserved features by *CFR*. We also study the effectiveness of our automated procedure for identifying conserved features as compared to using a subset that only considers Javascript features (we can think of these as expert-identified conserved features, as this is what an expert would naturally consider). To this end, we repeat the procedure above by replacing the conserved feature set S in Eq. 3 with a subset that involves Javascript. The classifier resulting from such restricted adver-

[4] We observe similar results for 5,000 benign PDFs obtained by using Google web searches [37], where 3% of benign files use Javacript.

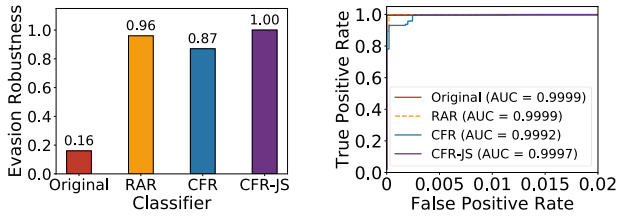


Figure 7: Evasion robustness (left) and performance on non-adversarial data (right) of different variants of SL2013.

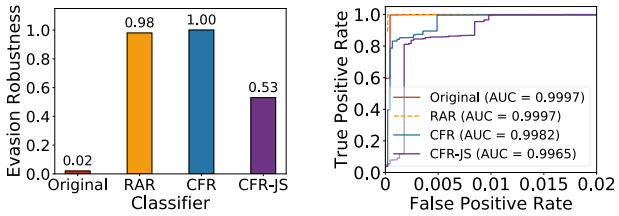


Figure 8: Evasion robustness (left) and performance on non-adversarial data (right) of different variants of Hidost.

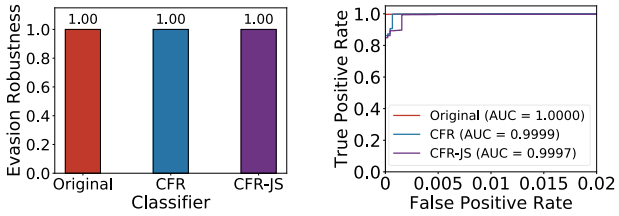


Figure 9: Evasion robustness (left) and performance on non-adversarial data (right) of different variants of PDFRate-B.

sarial retraining with "expert"-identified conserved features is termed *CFR-JS*.

6.2.1 SL2013

We now evaluate the robustness and effectiveness of the feature space retraining approach, which uses conserved features. We set the parameter $\lambda = 0.005$ as before. The robustness results are presented in Figure 7 (left). Observe that *CFR* now significantly improves robustness of the original classifier, with evasion robustness rising from 16% to 87%. Moreover, *CFR-JS* achieves a 100% evasion robustness against EvadeML. These results demonstrate that by leveraging the conserved features, the feature-space evasion models are now quite effective as a means to boost evasion robustness of SL2013.

In Figure 7 (right) we evaluate the quality of these classifiers on non-adversarial test data in terms of ROC curves. In all cases, be it original, RAR, CFR, and CFR-JS, AUC is $> 99.9\%$, although we can see a slight degradation of CFR for extremely low false positive rates compared to the others. It is noteworthy that CFR performs much better than FSR (robust ML using a standard feature-space approach, recall Figure 1 (right)).

6.2.2 Hidost

Next, we evaluate the effectiveness of CFR for Hidost. The results are shown in Figure 8 (left) and are largely consistent with SL2013. In particular, CFR boosts evasion robustness from 2% to 100% (slightly better than RAR), well above conventional FSR (recall Figure 3 (left)). In contrast, CFR-JS only boosts robustness to 53%, showing that our algorithmic approach can in some cases offer a considerable advantage to expert-chosen conserved features.

Evaluating the performance of CFR and CFR-JS on non-adversarial test data in terms of ROC curves in Figure 8 (right), we find that the CFR classifier can achieve $\sim 99.8\%$ AUC. This is somewhat worse than RAR, particularly for very low false positive rates, but better than CFR-JS—again, in this case using the full batch of conserved features exhibits a significant advantage over solely looking for Javascript.

6.2.3 Binarized PDFRate

Finally, we evaluate the effectiveness of the CFR variants of PDFRate-B. We observe that both the *CFR* and *CFR-JS* classifiers in the PDFRate-B family achieve 100% evasion robustness against EvadeML (Figure 9 (left)), just as the RAR and FSR counterparts had.

However, a close look at Figure 9 (right) demonstrates that CFR and CFR-JS achieve far better performance on non-adversarial data, with $>99.9\%$ AUC, where improvements are particularly significant for small false positive rates compared to FSR (recall Figure 5 (right)). Moreover, in this experiment, CFR achieves slightly higher TPR than CFR-JS for low FPR regions (below 0.003). The main takeaway here is that although the feature-space approach already yields high robustness in this setting, introducing conserved features significantly mitigates its degradation in performance on non-adversarial data.

7 Additional Realizable Evasion Attacks

So far we used EvadeML as the primary realizable attack in our experiments. This choice is defensible, as EvadeML explores a significantly larger attack space than many other evasion methods (e.g., Mimicry [37]), allowing deletions and swaps, in addition to insertions. Nevertheless, it is natural to wonder whether classifiers robust to EvadeML remain robust to other classes of evasion attacks. A particularly intriguing question is how the classifiers hardened against EvadeML fare in comparison with classifiers hardened against feature-space models, when faced with different realizable attacks.

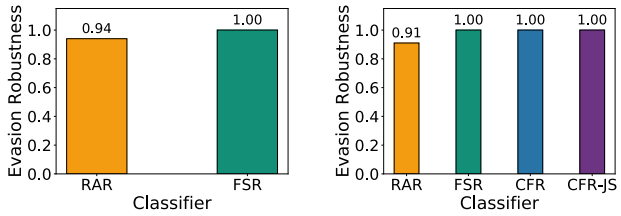


Figure 10: Robustness to Mimicry attack. Left: PDFRate-R (note that our notion of CFR is not applicable here). Right: PDFRate-B.

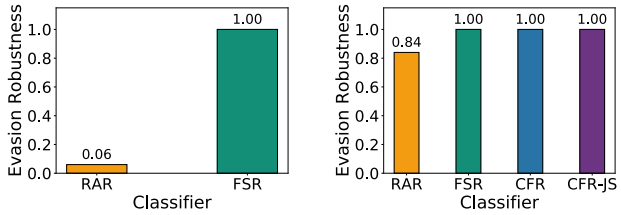


Figure 11: Robustness to Mimicry+ attack. Left: PDFRate-R (note that our notion of CFR is not applicable here). Right: PDFRate-B.

To answer these questions, we consider *five* additional realizable attacks: *Mimicry* [37], which was one of the first realizable attacks on PDF malware detectors, *Mimicry+*, an enhanced variant of Mimicry, *MalGAN* [19], which uses Generative Adversarial Networks (GANs) to create evasion attacks (but only targets binary classifiers), *Reverse Mimicry* [26], which inserts malicious payloads into target benign files, and a new custom attack aimed at defeating PDFRate-B conserved features. The Mimicry/Mimicy+ attacks are designed specifically for PDFRate, and cannot be usefully applied to SL2013 or Hidost, whereas the Reverse Mimicry attack and our custom attack require *zero knowledge* of target classifiers.

7.1 Mimicry and Mimicry+ Attacks

We start by considering the Mimicry and Mimicry+ attacks for both real-valued and binarized variants of PDFRate, with the same 100 malicious seeds employed in Section 5 and 6 as attack files.

The results are shown in Figures 10 and 11, and offer two noteworthy findings. First, as can be seen in Figure 11, RAR classifiers (hardened specifically against EvadeML, recall that the original PDFRate-B classifier is equivalent to RAR) can be quite vulnerable to the Mimicry+ attack, whereas both FSR and CFR classifiers remain robust. Second, Mimicry+ is indeed a much stronger attack than Mimicry: the original Mimicry fails to significantly degrade RAR performance, whereas Mimicry+ largely evades the RAR variant of PDFRate-R, and is somewhat more potent against PDFRate-B

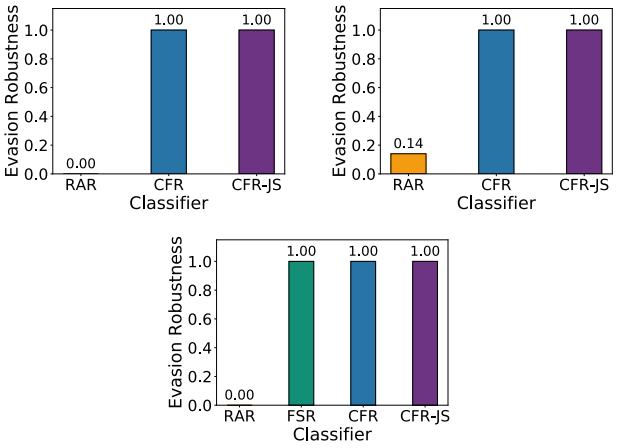


Figure 12: Robustness to MalGAN attack. SL2013 (top left), Hidost (top right), PDFRate-B (bottom).

than Mimicry. This demonstrates that besides its mathematical elegance, the abstract feature-space evasion models, once appropriately anchored to the domain, are rather generally robust to evasion attacks.

7.2 MalGAN Attack

Next, we consider the MalGAN attack on the three classifiers over binary feature space we have previously studied: SL2013, Hidost, and PDFRate-B, with RAR and FSR/CFR versions that have been shown robust to EvadeML.

The results, shown in Figure 12, demonstrate that despite EvadeML being a powerful attack, the RAR approaches which use it for hardening (with resulting classifiers no longer very vulnerable to EvadeML) are *highly* vulnerable to MalGAN, with evasion robustness of 0% in most cases. In contrast, CFR models which use conserved features remain highly robust (100% in all cases), just as we had observed earlier.

7.3 Reverse Mimicry Attack

Next, we employ the *Reverse Mimicry* attack on the EvadeML-robust variants of all the classifier types (SL2013, Hidost, PDFRate-R, and PDFRate-B).

Figure 13 presents the results, which are revealing in several ways. First, we again observe that RAR (hardened specifically against EvadeML) is roundly defeated in most cases. Second, consider the robustness results for the classifier using only the conserved features (CF), we can see that reverse mimicry succeeds in defeating conserved features for a non-trivial proportion of instances. It does so by including Javascript tags in structural paths that are not used as features by SL2013/Hidost (since these classifiers only consider commonly occurring sets of structural paths). Thus, this attack reveals an important vulnerability in the feature extraction

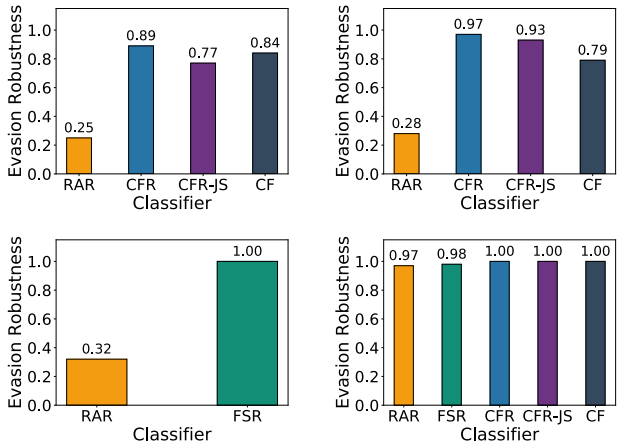


Figure 13: Robustness to Reverse Mimicry attack. SL2013 (top left), Hidost (top right), PDFRate-R (bottom left), PDFRate-B (bottom right). Note that our notions of CFR and CF for PDFRate-R is not applicable here.

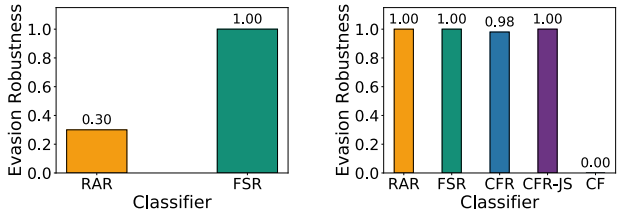


Figure 14: Robustness to the custom attack. Left: PDFRate-R (note that our notions of CFR and CF are not applicable here). Right: PDFRate-B.

approach employed by these classifiers; indeed, it suggests that structure-based classifiers may be *inherently* difficult to harden. Remarkably, CFR remains more robust than CF despite these vulnerabilities. The case of Hidost is particularly stark: CFR is nearly 20% more robust than CF!

7.4 The Custom Attack

Our final attack specifically targets a feature extraction bug in the Mimicus implementation of PDFRate in order to defeat the corresponding CF classifier.

The results are shown in Figure 14. We find that after this attack, CF robustness is 0. We also observe that the robustness of RAR classifier for PDFRate-R also drops, although to 0.3 rather than 0. Significantly, the FSR classifiers for both PDFRate-R and PDFRate-B remain 100% robust, and the CFR variant of PDFRate-B has nearly perfect robustness (0.98) against this attack. Our latter observation is particularly remarkable: although the conserved features are roundly defeated by this attack, the use of these as a part of a holistic retraining approach yields a classifier that remains robust. Thus, not only is it possible to construct a robust malware classifier without unduly relying on conserved features, but we can accomplish this through iterative retraining in feature space.

8 Related Work

Below we briefly describe some of the related literature on adversarial evasion or adversarial example attacks and defenses; we refer readers to Vorobeychik and Kantarcioglu [40] for a broader and more in-depth treatment of the subject of ML attacks and defenses.

Evasion and Adversarial Example Attacks: An early realizable evasion attack on machine learning was devised by Fogla et al. [11, 12], who developed an attack on anomaly-based intrusion detection systems. Šrndic and Laskov [37] present a case study of an evasion attack on a state-of-the-art PDF malware classifier, PDFRate. Xu et al. [44] propose EvadeML, a fully realizable attack on PDF malware classifiers which generates evasion instances by using genetic programming to modify PDF source directly, using a sandbox to ensure that malicious functionality is preserved. Grosse et al. [16] develop a method for generating evasion attacks against a deep learning-based Android malware classifier, using a gradient-based approach which is also a form of iterative improvement heuristics, but chooses the best coordinate to improve in each iteration as evaluated by the gradient, rather than random coordinate as in our case. Their approach likely requires fewer steps than coordinate greedy, but since we run coordinate greedy until convergence, this difference isn't important in our study. Moreover, we also optimize among several local optima through random restarts, which is likely to obtain better evasion solutions (Grosse et al. [16] stop as soon as an evasion is found, rather than trying to identify the most benign looking malware). This particular attack can be viewed as realizable, even though it wasn't implemented and evaluated in actual malware, since the attack space is significantly restricted to only add features that do not interfere with others already present. Similarly, MalGAN, an evasion attack based on generative adversarial networks developed by Hu and Tan [19], only adds features from benign to malicious malware, and we treat it as a realizable attack (since it's not difficult to implement).

In addition to classifier evasion methods which change the actual malicious instances (or are relatively direct to implement as such), a number of techniques have sprouted for modeling adversarial examples in feature space [1–4, 7, 9, 15, 21, 21–24, 28, 41, 45]. Moreover, a series of efforts explore evasion in the context of image classification by deep neural networks [15, 20, 31, 33], although Gilmer et al. [13] question the common threat models used in these works. Several recent approaches attempt to generate adversarial examples against computer vision systems in physical space, such as

adding stickers to a stop sign to cause misclassification, or wearing printed glass frames to fool face recognition, and are therefore somewhat analogous to our notion of realizable attacks [10, 33].

Evasion-Robust Classification: Dalvi et al. [9] presented the first approach for evasion-robust classification. A series of approaches formulate robust classification as minimizing maximum loss (i.e., following a robust optimization paradigm), where maximization is attributed to the evading attacker aiming to maximize the learner's loss through small feature-space transformations [25, 32, 39, 42, 46]. A number of alternative methods for designing classifiers consider the interaction as a non-zero-sum game [5, 6, 21–23]. Finally, a series of iterative retraining procedures have been proposed, both for general adversarial evasion [21, 23], and specifically for deep learning methods for vision [15, 20, 25] (note that Madry et al. [25] fall into both robust optimization and retraining buckets, since their approach is equivalent to retraining if stochastic gradient descent simply continues by processing adversarial examples as they are added). These diverse efforts share one common property: attack models that they leverage use feature-space manipulations, which are only a proxy for realizable attacks on ML.

9 Discussion and Conclusion

We undertook an extensive exploration of the extent to which robust ML that uses the conventional feature-space models of evasion attacks remains robust to "real" attacks that can be implemented in actual malware and preserve malicious functionality (what we call realizable attacks). Our first intriguing observation is that defense based on feature-space models can fail to achieve satisfactory robustness. This in itself raises some doubts about the nearly universal focus on such models as a means for ML defense, and suggests that practical usefulness of such approaches cannot be taken for granted. However, we also show that changing the nature of the feature space can make a difference: robust ML with feature-space models is quite robust in content-based detection (which uses content, rather than structural paths, as features). Additionally, we presented a refined version of the feature-space model that makes use of conserved features (which we can identify automatically, as shown in the Appendix), and showed that where feature-space defense previously failed, it now succeeds. Our final finding may well be the most intriguing: feature-space approaches exhibit generalized robustness, in that the resulting robust ML (after appropriate refinement using conserved features) exhibits robustness to multiple realizable attacks. This contrasts with defense that is hardened using a *specific* realizable attack—even one quite powerful on the surface (EvadeML)—which can fail dramatically when faced with a different attack. These findings demonstrate the power of effective mathematical abstractions in security.

It is natural to wonder how our approach and results are applied to other domains. In computer vision, the analog of realizable malware attacks are physical attacks, whereby the physical environment is modified, rather than the digital object, such as an image. Here, the corresponding foundational question is whether common robust ML methods based on small-l_p attacks successfully protect against physical attacks. The notion of conserved features can also be seen as more generally applicable. For example, in a bag-of-words representation for spam filtering, these could correspond to the existence of URL or file attachments, and in SQL injection attacks, these may refer to the existence of specific SQL commands, such as `Select`.

The main limitation of our study is in the specific choices we had to make to ensure that it is tractable. We chose a particular defensive paradigm—iterative retraining. As we have argued, it is the only paradigm that can fit every case that we investigate; for example, there is no other general approach for learning a robust SVM with non-linear kernels. However, it is possible that approaches based on robust optimization, if they were developed, can improve performance by taking advantage of the special structure of this problem. We implemented a particular class of feature-space attacks, using l_2 norm to measure the attacker's cost of feature manipulations, and stochastic local search to compute evasions. It is possible that better attack algorithms for generating attacks over binary domains will be developed, and, indeed, some alternatives exist. However, prior work suggests that this approach yields attacks that are close to optimal [23], with the use of random restarts playing a crucial role. Finally, our study was specific to PDF malware detection. However, our framework is quite general, and could be used in the future to consider other similar questions, such as the effectiveness of robust deep learning against physical attacks. Several additional limitations offer further opportunities for future work. One example is the fact that we only define conserved features when these are binary; it may be that finding meaningful conserved features in continuous feature spaces is inherently more difficult. Another issue is the surprising finding that sufficient anchoring of feature-space defense in the domain using conserved features allows us to achieve robustness, *even when conserved features can be circumvented*. It may be that conserved features are ultimately only a part of the solution, and only help if they adequately capture the attack surface in the abstract feature space. The extent to which small variations in the set of identified conserved features matters is also an open question: our evidence is mixed, with "expert"-defined features usually, but not always, sufficient for robustness.

Acknowledgments

This work was partially supported by the Army Research Office (W911NF1610069) and NSF CAREER award (IIS-1649972).

Appendix

A Identifying Conserved Features

We now describe a systematic automated procedure for identifying these. We first introduce how to identify conserved features of SL2013, and then describe how to generalize the approach to extract conserved features of Hidost.

The key to identifying the conserved features of a malicious PDF is to discriminate them from non-conserved ones. Since merely applying statistical approaches on training data is insufficient to discriminate between these two classes of features, as demonstrated above, we need a qualitatively different approach which relies on the nature of evasions (as implemented in EvadeML) and the sandbox (which determines whether malicious functionality is preserved) to identify features that are conserved.

We use a modified version of pdfrw [27][5] to parse the objects of PDF file and repack them to produce a new PDF file. We use Cuckoo [17] as the sandbox to evaluate malicious functionality. In the discussion below, we define x_i to be the malicious file, S_i the conserved feature set of x_i, and O_i the set of its non-conserved features. Initially, $\mathsf{S}_i = \mathsf{O}_i = \emptyset$.

At the high level, our first step is to sequentially delete each object of a malicious file and eliminate non-conserved features by evaluating the existence of a malware signature in a sandbox for each resulting PDF, which provides a preliminary set of conserved features. Then, we replace the object of each corresponding structural path in the resulting preliminary set with an external benign object and assess the corresponding functionality, which allows us to further prune non-conserved features. Next, we describe these procedures in detail.

A.1 Structural Path Deletion

In the first step, we filter out non-conserved features by deleting each object and its corresponding structural path, and then checking whether this eliminates malicious functionality (and should therefore be conserved). First, we obtain all the structural paths (objects) by parsing a PDF file. These objects are organized as a tree-topology and are sequentially deleted. Each time an object is removed, we produce a resulting PDF file by repacking the remaining objects. Then, we employ the sandbox to detect malicious functionality of the PDF after the object deletion. If any malware signature is captured, the corresponding structural path of the object is deleted as a non-conserved feature, and added to O_i. On the other hand, if no malware signature is detected, the corresponding feature is added in S_i as a *possibly* conserved feature.

One important challenge in this process is that features are not necessarily independent. Thus, in addition to identifying S_i and O_i, we explore *interdependence* between features by deleting objects. As the logic structure of a PDF file is with a tree-topology, the presence of some structural path depends on the presence of other structural paths whose object refers to the object of the prior one. We define that a structural path is a dependent of another if unilateral deleting the object associated with the latter causes a flip from 1 to 0 on the feature value of the former. For any feature j of x_i, we denote the set of features that depend on j by D_i^j. Note that for a given structural path (feature), there could be multiple corresponding PDF objects. In such a case, these objects are deleted simultaneously, so as the corresponding feature value is shifted from 1 to 0.

A.2 Structural Path Replacement

In the second step, we subtract the remaining non-conserved features in the preliminary S_i and move them to O_i. Similar to the prior step, we first obtain all the structural paths and objects of the malicious PDF file. Then for each object of the PDF that is in S_i, we replace it with an external object from a benign PDF file and produce the resulting PDF, which is further evaluated in the sandbox. If the sandbox detects any malware signature, then the corresponding structural path of the object replaced is moved from S_i to O_i. Otherwise, the structural path is a conserved feature since both deletion and replacement of the corresponding object removes the malicious functionality of the PDF file. Note that in the case of multiple corresponding and identical objects of a structural path, all of these objects are replaced simultaneously.

After structural path deletion and replacement, for each malicious PDF file x_i, we can get its conserved feature set S_i, non-conserved feature set O_i, and dependent feature set D_j for any feature $j \in \mathsf{S}_i \cup \mathsf{O}_i$, which could be further leveraged to design evasion-robust classifiers.

A.3 Obtaining a Uniform Conserved Feature Set

The systematic approach discussed above provides a conserved feature set for each malicious seed to retrain a classifier. Our goal, however, is to identify a single set of conserved features which is *independent* of the specific malicious PDF seed file. We now develop an approach for transforming a collection of S_i, O_i, and D_i^j for a set of malicious seeds i into a *uniform* set of conserved features.

Obtaining a uniform set of conserved features faces two challenges: 1) minimizing conflicts among different conserved features, as a conserved feature for one malicious instance could be a non-conserved feature for another, and 2) abiding by feature interdependence if a conserved feature should be further eliminated.

To address these challenges, we propose a *Forward Elimination* algorithm to compute the uniform conserved feature

[5] The modified version is available at https://github.com/mzweilin/pdfrw.

Algorithm 1 Forward Elimination for uniform conserved feature set.

Input:
The set of conserved features for $x_i (i \in [1,n])$, S_i;
The set of non-conserved features for $x_i (i \in [1,n])$, O_i;
The set of dependent features for $j \in \mathsf{S}_i \cup \mathsf{O}_i$, D_i^j;
Output:
The uniform conserved feature set for $\{x_1, x_2, ..., x_n\}$, S;

$\mathsf{S} \leftarrow \bigcup_{i=1}^{n} \mathsf{S}_i$;
$\mathsf{S}' \leftarrow \mathsf{S}$;
$\mathsf{Q} \leftarrow \emptyset$;
$\mathsf{D}^j = \bigcup_{i=1}^{n} \mathsf{D}_i^j$;
for each $j \in \mathsf{S}'$ **do**
  if $j \notin \mathsf{Q}$ **then**
    if $\sum_{i=1}^{n} \mathbb{1}_{j \in \mathsf{O}_i} \geq \beta \cdot \sum_{i=1}^{n} \mathbb{1}_{j \in \mathsf{S}_i}$ **then**
      $\mathsf{S} \leftarrow \mathsf{S} \setminus (\{j\} \cup \mathsf{D}^j)$;
      $\mathsf{Q} \leftarrow \mathsf{Q} \cup (\{j\} \cup \mathsf{D}^j)$;
    end if
  end if
end for
return S;

set for a set of malicious seeds $\{x_1, x_2, ..., x_n\}$, given the conserved feature sets, non-conserved feature sets and dependent sets for each seed. As Algorithm 1 shows, we first obtain a union of the conserved feature sets. Then, we explore the contradiction of each feature in the union with the others, by comparing the total number of the feature being selected as a non-conserved feature and conserved feature. If the former one is greater than β times the latter one, then this feature, together with its dependents, are eliminated from the union. Otherwise, the feature is added to the uniform feature set. We use β as a parameter to adjust the balance between conserved and non-conserved features. Typically, $\beta > 1$ as we are inclined to preserve malicious functionality associated with a conserved feature, even it could be a non-conserved feature of another PDF file. We set $\beta = 3$ in our experiments.

A.4 Identifying Conserved Features for Other Classifiers

Once we obtain conserved features of SL2013 for each malicious seeds, we can employ these features to identify conserved features for other classifiers using binary features. As our approach relies on the existence of malicious functionality and corresponding features, such a relation is not obvious for real-valued features; we therefore leave the question of how to define and identify conserved features in real space for future work.

Hidost Hidost and SL2013 are similar in nature in such a way that they employ structural paths as features. The only difference is that Hidost consolidates features of SL2013 as described in Section 4. Therefore, once the conserved features of SL2013 are identified, we can simply apply the *PDF structural path consolidation rules* described in Srndic and Laskov [38] to transform these features to the corresponding conserved features for Hidost.

Binarized PDFRate We identify the conserved features for PDFRate-B by using the conserved feature set S_i of each seed x_i. For each x_i, we generate $|\mathsf{S}_i|$ PDF files, each of which corresponds to the PDF file when an element (structural path) in S_i is deleted. We then compare PDFRate-B features of these PDFs to the original x_i. If any feature value of x_i is flipped from 1 to 0, then this feature will be added in the conserved feature set of x_i for PDFRate-B. Afterward, we use Algorithm 1 to obtain the uniform conserved feature set. This approach can in fact be used for arbitrary PDF malware detectors over binary features (leveraging conserved structural paths identified using SL2013).

Classifier	Conserved features	Involve JS?
SL2013	/Names	No
	/Names/JavaScript	Yes
	/Names/JavaScript/Names	Yes
	/Names/JavaScript/Names/JS	Yes
	/OpenAction	No
	/OpenAction/JS	Yes
	/OpenAction/S	No
	/Pages	No
Hidost	/Names	No
	/Names/JavaScript	Yes
	/Names/JavaScript/Names	Yes
	/Names/JavaScript/Names/JS	Yes
	/OpenAction	No
	/OpenAction/JS	Yes
	/Pages	No
PDFRate-B	count_box_other	No
	count_javascript	Yes
	count_js	Yes
	count_page	No

Table 3: Conserved features and their relevance to JavaScript.

A.5 Conserved Features

Table 3 presents the full list of conserved features we identified for each classifier.

A.6 Conserved vs. Regularized Features

In our experiments, we empirically adjust the SVM parameter C to study the overlap between *conserved features* and those selected by l_1 regularization. We first adjust C to perform feature reduction until the number of features is identical to the number of conserved features. In this case, sparse versions of both SL2013 and Hidost include only 3 of the conserved features, while sparse PDFRate-B includes only 1. In another experiment, we adjusted C until all conserved features were selected. In this case, SL2013 requires 510 features, Hidost needs 154, and PDFRate-B needs 83.

References

[1] ATHALYE, A., CARLINI, N., AND WAGNER, D. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *International Conference on Machine Learning* (2018), pp. 274–283.

[2] BARRENO, M., NELSON, B., SEARS, R., JOSEPH, A. D., AND TYGAR, J. D. Can machine learning be secure? In *ACM Asia Conference Computer and Communications Security* (2006), pp. 16–25.

[3] BIGGIO, B., CORONA, I., MAIORCA, D., NELSON, B., SRNDIC, N., LASKOV, P., GIACINTO, G., AND ROLI, F. Evasion attacks against machine learning at test time. In *European Conference on Machine Learning and Knowledge Discovery in Databases* (2013), pp. 387–402.

[4] BIGGIO, B., FUMERA, G., AND ROLI, F. Security evaluation of pattern classifiers under attack. *IEEE Transactions on Knowledge and Data Engineering 26*, 4 (2014), 984–996.

[5] BRÜCKNER, M., AND SCHEFFER, T. Stackelberg games for adversarial prediction problems. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2011), pp. 547–555.

[6] BRÜCKNER, M., AND SCHEFFER, T. Static prediction games for adversarial learning problems. *Journal of Machine Learning Research*, 13 (2012), 2617–2654.

[7] CARLINI, N., AND WAGNER, D. Towards evaluating the robustness of neural networks. In *IEEE Symposium on Security and Privacy* (2017), pp. 39–57.

[8] COVA, M., KRUEGEL, C., AND VIGNA, G. Detection and analysis of drive- by-download attacks and malicious javascript code. In *International Conference on World Wide Web* (2010), pp. 281–290.

[9] DALVI, N., DOMINGOS, P., MAUSAM, SANGHAI, S., AND VERMA, D. Adversarial classification. In *SIGKDD International Conference on Knowledge Discovery and Data Mining* (2004), pp. 99–108.

[10] EYKHOLT, K., EVTIMOV, I., FERNANDES, E., LI, B., RAHMATI, A., XIAO, C., PRAKASH, A., KOHNO, T., AND SONG, D. Robust physical-world attacks on deep learning visual classification. In *Computer Vision and Pattern Recognition* (2018).

[11] FOGLA, P., AND LEE, W. Evading network anomaly detection systems: Formal reasoning and practical techniques. In *ACM Conference on Computer and Communications Security* (2006), pp. 59–68.

[12] FOGLA, P., SHARIF, M., PERDISCI, R., KOLESNIKOV, O., AND LEE, W. Polymorphic blending attacks. In *USENIX Security Symposium* (2006).

[13] GILMER, J., ADAMS, R. P., GOODFELLOW, I. J., ANDERSEN, D., AND DAHL, G. E. Motivating the rules of the game for adversarial example research. arXiv preprint.

[14] GOODFELLOW, I., POUGET, J., MIRZA, M., XU, B., WARDE, D., OZAIR, S., COURVILLE, A., AND BENGIO, Y. Generative adversarial nets. In *Neural Information Processing Systems* (2014), pp. 2672–2680.

[15] GOODFELLOW, I. J., SHLENS, J., AND SZEGEDY, C. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations* (2015).

[16] GROSSE, K., PAPERNOT, N., MANOHARAN, P., BACKES, M., AND MCDANIEL, P. Adversarial perturbations against deep neural networks for malware classification. In *European Symposium on Research in Computer Security* (2017).

[17] GUARNIERI, C., TANASI, A., BREMER, J., AND SCHLOESSER, M. Cuckoo sandbox: A malware analysis system, 2012. http://www.cuckoosandbox.org/.

[18] HOOS, H. H., AND STUTZLE, T. *Stochastic Local Search : Foundations & Applications*. Morgan Kaufmann, 2004.

[19] HU, W., AND TAN, Y. Generating adversarial malware examples for black-box attacks based on GAN. arXiv preprint.

[20] HUANG, R., XU, B., SCHUURMANS, D., AND SZEPESVÁRI, C. Learning with a strong adversary. In *International Conference on Learning Representations* (2016).

[21] KANTCHELIAN, A., TYGAR, J. D., AND JOSEPH, A. D. Evasion and hardening of tree ensemble classifiers. In *International Conference on Machine Learning* (2016), pp. 2387–2396.

[22] LI, B., AND VOROBEYCHIK, Y. Feature cross-substitution in adversarial classification. In *Neural Information Processing Systems* (2014), pp. 2087–2095.

[23] LI, B., AND VOROBEYCHIK, Y. Evasion-robust classification on binary domains. *ACM Transactions on Knowledge Discovery from Data* (2018).

[24] LOWD, D., AND MEEK, C. Adversarial learning. In *ACM SIGKDD International Conference on Knowledge Discovery in Data Mining* (2005), pp. 641–647.

[25] MADRY, A., MAKELOV, A., SCHMIDT, L., TSIPRAS, D., AND VLADU, A. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations* (2018).

[26] MAIORCA, D., CORONA, I., AND GIACINTO, G. Looking at the bag is not enough to find the bomb: an evasion of structural methods for malicious PDF files detection. In *ACM Asia Conference on Computer and Communications Security* (2013), pp. 119–130.

[27] MAUPIN, P. Pdfrw: A pure python library that reads and writes pdfs. https://github.com/pmaupin/pdfrw, 2017. Accessed: 2017-05-18.

[28] NELSON, B., RUBINSTEIN, B. I., HUANG, L., JOSEPH, A. D., LEE, S. J., RAO, S., AND TYGAR, J. Query strategies for evading convex-inducing classifiers. *Journal of Machine Learning Research* (2012), 1293–1332.

[29] PAPERNOT, N., MCDANIEL, P., SINHA, A., AND WELLMAN, M. Towards the science of security and privacy in machine learning. In *IEEE European Symposium on Security and Privacy* (2018).

[30] PAPERNOT, N., MCDANIEL, P., WU, X., AND JHA, S. Distillation as a defense to adversarial perturbations against deep neural networks. In *IEEE Symposium on Security and Privacy,* (2016).

[31] PAPERNOT, N., MCDANIEL, P. D., AND GOODFELLOW, I. J. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples, 2016. arxiv preprint.

[32] RAGHUNATHAN, A., STEINHARDT, J., AND LIANG, P. Certified defenses against adversarial examples. In *International Conference on Learning Representations* (2018).

[33] SHARIF, M., BHAGAVATULA, S., BAUER, L., AND REITER, M. K. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. In *ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 1528–1540.

[34] SMUTZ, C., AND STAVROU, A. Malicious pdf detection using matadata and structural features. Tech. rep., 2012.

[35] SMUTZ, C., AND STAVROU, A. Malicious pdf detection using matadata structural features. In *Annual Computer Security Applications Conference* (2012), pp. 239–248.

[36] ŠRNDIC, N., AND LASKOV, P. Detection of malicious PDF files based on hierarchical document structure. In *Network and Distributed System Security Symposium* (2013).

[37] ŠRNDIC, N., AND LASKOV, P. Practical evasion of a learning-based classifier: A case study. In *IEEE Symposium on Security and Privacy* (2014), pp. 197–211.

[38] ŠRNDIĆ, N., AND LASKOV, P. Hidost: a static machine-learning-based detector of malicious files. *EURASIP Journal on Information Security 2016*, 1 (2016), 22.

[39] TEO, C. H., GLOBERSON, A., ROWEIS, S., AND SMOLA, A. J. Convex learning with invariances. In *Neural Information Processing Systems* (2007).

[40] VOROBEYCHIK, Y., AND KANTARCIOGLU, M. *Adversarial Machine Learning*. Morgan and Claypool, 2018.

[41] VOROBEYCHIK, Y., AND LI, B. Optimal randomized classification in adversarial settings. In *International Conference on Autonomous Agents and Multiagent Systems* (2014), pp. 485–492.

[42] WONG, E., AND KOLTER, J. Z. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *International Conference on Machine Learning* (2018).

[43] XU, H., CARAMANIS, C., AND MANNOR, S. Robustness and regularization of support vector machines. *Journal of Machine Learning Research 10* (2009), 1485–1510.

[44] XU, W., QI, Y., AND EVANS, D. Automatically evading classifiers: A case study on PDF malware classifiers. In *Network and Distributed System Security Symposium* (2016).

[45] ZHANG, F., CHAN, P., BIGGIO, B., YEUNG, D., AND ROLI, F. Adversarial feature selection against evasion attacks. *IEEE Transactions on Cybernetics* (2015).

[46] ZHOU, Y., KANTARCIOGLU, M., THURAISINGHAM, B. M., AND XI, B. Adversarial support vector machine learning. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2012), pp. 1059–1067.

ALOHA: Auxiliary Loss Optimization for Hypothesis Augmentation

Ethan M. Rudd*, Felipe N. Ducau*, Cody Wild, Konstantin Berlin, and Richard Harang*
Sophos PLC

Abstract

Malware detection is a popular application of Machine Learning for Information Security (ML-Sec), in which an ML classifier is trained to predict whether a given file is malware or benignware. Parameters of this classifier are typically optimized such that outputs from the model over a set of input samples most closely match the samples true malicious/benign (1/0) target labels. However, there are often a number of other sources of contextual metadata for each malware sample, beyond an aggregate malicious/benign label, including multiple labeling sources and malware type information (e.g. *ransomware*, *trojan*, etc.), which we can feed to the classifier as auxiliary prediction targets. In this work, we fit deep neural networks to multiple additional targets derived from metadata in a threat intelligence feed for Portable Executable (PE) malware and benignware, including a multi-source malicious/benign loss, a count loss on multi-source detections, and a semantic malware attribute tag loss. We find that incorporating multiple auxiliary loss terms yields a marked improvement in performance on the main detection task. We also demonstrate that these gains likely stem from a more informed neural network representation and are not due to a regularization artifact of multi-target learning. Our auxiliary loss architecture yields a significant reduction in detection error rate (false negatives) of 42.6% at a false positive rate (FPR) of 10^{-3} when compared to a similar model with only one target, and a decrease of 53.8% at 10^{-5} FPR.

1 Introduction

Machine learning (ML) for computer security (ML-Sec) has proven to be a powerful tool for malware detection. ML models are now integral parts of commercial anti-malware engines and multiple vendors in the industry have dedicated ML-Sec teams. For the malware detection problem, these models are typically tuned to predict a binary label (malicious or benign) using features extracted from sample files. Unlike signature engines, where the aim is to reactively blacklist/whitelist malicious/benign samples that hard-match manually-defined patterns (signatures), ML engines employ numerical optimization on parameters of highly parameterized models that aim to learn more general concepts of *malware* and *benignware*. This allows some degree of proactive detection of previously unseen malware samples that is not typically provided by signature-only engines.

Frequently, malware classification is framed as a binary classification task using a simple binary cross-entropy or two-class softmax loss function. However, there often exist substantial metadata available at training time that contain more information about each input sample than just an aggregate label of whether it is malicious or benign. Such metadata might include malicious/benign labels from multiple sources (e.g., from various security vendors), malware family information, file attributes, temporal information, geographical location information, counts of affected endpoints, and associated tags. In many cases this metadata will not be available when the model is deployed, and so in general it is difficult to include this data as features in the model (although see Vapnik et al. [28, 29] for one approach to doing so with Support Vector Machines).

It is a popular practice in the domain of malware analysis to derive binary malicious/benign labels based on a heuristic combination of multiple detection sources for a given file, and then use these noisy labels for training ML models [9]. However, there is nothing that precludes training a classifier to predict each of these source labels simultaneously optimizing classifier parameters over these predictions + labels. In fact, one might argue intuitively that guiding a model to develop representations capable of predicting multiple targets simultaneously may have a smoothing or regularizing effect conducive to generalization, particularly if the auxiliary targets are related to the main target of interest. These auxiliary targets can be ignored during test time if they are ancillary to the task at hand (and in many cases the extra

* Equal contribution.
Contact: <firstname>.<lastname>@sophos.com

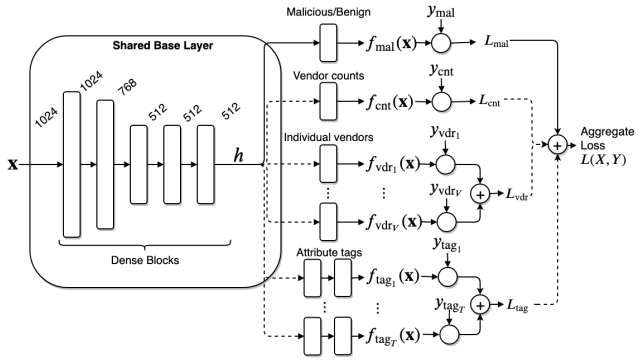
weights required to produce them can be removed from the model prior to deployment), but nevertheless, there is much reason to believe that forcing the model to fit multiple targets simultaneously can improve performance on the key output of interest. In this work, we take advantage of multi-target learning [2] by exploring the use of metadata from threat intelligence feeds as auxiliary targets for model training.

Research in other domains of applied machine learning supports this intuition [14, 19, 12, 31, 1, 22], however outside of the work of Huang et al. [11], multi-target learning has not been widely applied in anti-malware literature. In this paper, we present a wide-ranging study applying auxiliary loss functions to anti-malware classifiers. In contrast to [11], which studies the addition of a single auxiliary label for a fundamentally different task, i.e., malware family classification – we study both the addition of multiple label sources for the same task and multiple label sources for multiple separate tasks. Also, in contrast to [11], we do not presume the presence of all labels from all sources, and introduce a per-sample weighting scheme on each loss term to accommodate missing labels in the metadata. We further explore the use of multi-objective training as a way to expand the number of trainable samples in cases where the aggregate malicious/benign label is unclear, and where those samples would otherwise be excluded from purely binary training.

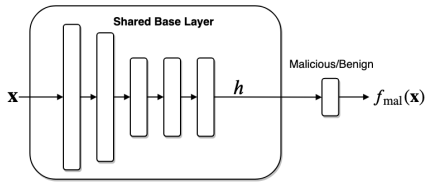
Having established for which loss types and in which contexts auxiliary loss optimization works well, we then explore *why it works well*, via experiments designed to test whether performance gains are a result of a regularization effect from multi-objective training or information from the content of the target labels that the network has learned to correlate.

In summary, this paper makes the following contributions:

- A demonstration that including auxiliary losses yields improved performance on the main task. When all of our auxiliary loss terms are included, we see a reduction of 53.8% in detection error (false negative) rate at 10^{-5} false positive rate (FPR) and a 42.6% reduction in detection error rate at 10^{-3} FPR compared to our baseline model. We also see a consistently better and lower-variance ROC curve across all false positive rates.
- A breakdown of performance improvements from different auxiliary loss types. We find that an auxiliary Poisson loss on detection counts tends to yield improved detection rates at higher FPR areas ($\geq 10^{-3}$) of the ROC curve, while multiple binary auxiliary losses tend to yield improved detection performance in lower FPR areas of the ROC curve ($< 10^{-3}$). When combined we see a net improvement across the entire ROC curve over using any single auxiliary loss type.
- An investigation into the mechanism by which multi-objective optimization yields enhanced performance, including experiments to assess possible regularization effects.



(a) Training



(b) Deployment

Figure 1: Schematic overview of our neural network architecture. (a) During training multiple output layers with corresponding loss functions are optionally connected to a common base topology consisting of five dense blocks (see Section 3) of sizes 1024, 768, 512, 512, and 512. This base, connected to our main malicious/benign output (solid line in the figure) with a loss on the aggregate label, constitutes our baseline architecture. Auxiliary outputs and their respective losses are represented as dashed lines. The auxiliary losses fall into three types: count loss, multi-label vendor loss, and multi-label attribute tag loss. The formulation of each of these auxiliary loss types is explained in Section 3. (b) At deployment time, these auxiliary outputs are removed and we predict only the main label.

We see our auxiliary loss approach as specifically useful for both endpoint and cloud deployed models in cases when the auxiliary information cannot be directly used as input into the model at prediction time, but can be collected for a training dataset. This could be due to high cost, performance issues, latency concerns, or a multitude of other constraints during prediction time. For example, it is not feasible to scan every new file executed on an endpoint via a threat intelligence feed, because of prohibitive licensing fees, endpoint latency and bandwidth limitations, as well as customer privacy concerns. However, procuring reports from such a feed for large training sets might be feasible offline.

The remainder of this paper is laid out as follows: First, in Section 2 we discuss some of the metadata available for use as auxiliary targets, and feature extraction methods for portable executable (PE) files. We then provide details on how we converted the available metadata into auxiliary tar-

gets and losses, as well as how the individual losses are weighted and combined on a per-sample basis in Section 3. We finish that Section with a description of how our dataset was collected and provide summary statistics. In Section 4 we describe our experimental evaluations across a number of combinations of auxiliary targets, and demonstrate the impact of fitting these targets on the performance of the main malware detection task. Section 5 presents discussion of our results, as well as a set of experiments on synthetic targets to explore potential explanations for the observed improvements. Section 6 presents related work and Section 7 concludes.

2 ML-Sec Detection Pipelines: From Single Objective to Multi-Objective

In the following, we describe a simplified ML-Sec pipeline for training a malicious file classifier, and propose a simple extension that allows the use of metadata available during training (but not at test time) and improves performance on the classification task.

ML-Sec detection pipelines use powerful machine learning models that require many labeled malicious and benign samples to train. Data is typically gathered from several sources, including deployed anti-malware products and vendor aggregation services, which run uploaded samples through vendor products and provide reports containing per-vendor detections and metadata. The exact nature of the metadata varies, but typically, malicious and benign scores are provided for each of M individual samples on a per-vendor basis, such that, given V vendors, between 0 and V of them will designate a sample malicious. For a given sample, some vendors may choose not to answer, resulting in a missing label for that vendor. Furthermore, many vendors also provide a detection name (or malware family name) when they issue a detection for a given file. Additional information may also be available, but crucially, the following metadata are presumed present for the models presented in this paper: i) per-vendor labels for each sample, either malicious/benign (mapped to binary $1/0$, respectively) or NULL; ii) textual per-vendor labels on the sample describing the family and variant of the malware (an empty string if benign or NULL); and iii) time at which the sample was first seen.

Using the individual vendor detections, an aggregate label can be derived either by a voting mechanism or by thresholding the net number of vendors that identify a given sample as malicious. The use of aggregated anti-malware vendor detections as a noisy labeling source presumes that the vendor diagnoses are generally accurate. While this is not necessarily a valid assumption, e.g., for novel malware and benignware, this is typically accounted for by using samples and metadata that are slightly dated so that vendors can correct their respective mistakes (e.g., by blacklisting samples in their signature databases).

Each malware/benignware sample must also be converted to a numerical vector to be able to train a classifier, a process called *feature extraction*. In this work we focus on static malware detection, meaning that we assume only access to the binary file, as opposed to dynamic detection, in which the features used predominantly come from the execution of the file. The feature extraction mechanism varies depending on the format type at hand, but consists of some numerical transformation that preserves aggregate and fine-grained information throughout each sample, for example, the feature extraction proposed by Saxe et al. [25] – which we use in this work – uses windowed byte statistics, 2D histograms of delimited string hash vs. length, and histograms of hashes of PE-format specific metadata – e.g., imports from the import address table (IAT).

Given extracted features and derived labels, a classifier is then trained, tuning parameters to minimize misclassification as measured by some loss criterion, which under the constraints of a statistical noise model measures the deviation of predictions from their ground truth. For both neural networks and ensemble methods a logistic sigmoid is commonly used to constrain predictions to a [0,1] range, and a cross-entropy loss between predictions and labels is used as the minimization criterion under a Bernoulli noise model per-label.

While the prior description roughly characterizes ML-Sec pipelines discussed in literature to date, note that much information in the metadata, which is often not used to determine the sample label but *is* correlated to the aggregate classification, is not used in training, e.g., the individual vendor classifications, the combined number of detections across all vendors, and information related to malware type that could be derived from the detection names. In this work, we augment a deep neural network malicious/benign classifier with additional output predictions of vendor counts, individual vendor labels, and/or attribute tags. These separate prediction arms were given their own loss functions which we jointly minimized through backpropagation. The difference between a conventional malware detection pipeline and our model can be visualized by considering Figure 1 in the absence and presence of auxiliary outputs (and their associated losses) connected by the dashed lines. In the next section, we shall explore the precise formulation and implementation of these different loss functions.

3 Implementation Details

In this section we describe our implementation of the experiments sketched above. We first introduce our model immediately below, followed by the various loss functions – denoted by $L_{\text{loss type}}(X,Y)$ for some input features X and targets Y – associated with the various outputs of the model, as well as how the labels Y representing the targets of these outputs are

constructed. Finally we discuss how our data set of M samples associated with V vendor targets is collected. We use the same feature representation as well as the same general model class and topology for all experiments. Each portable executable file is converted into a feature vector as described in [25].

The base for our model (see Figure 1) is a feedforward neural network incorporating multiple blocks composed of Dropout [27], a dense layer, batch normalization [13], and an exponential linear unit (ELU) activation [7]. The core of the model contains five such blocks with 1024, 768, 512, 512, and 512 hidden units, respectively. This base topology applies the function $f(\cdot)$ to the input vector to produce an intermediate 512 dimensional representation of the input file $\mathbf{h} = f(\mathbf{x})$. We then append to this model an additional block, consisting of one or more dense layers and activation functions, for each output of the model. We denote the composition of our base topology and our target-specific final dense layers and activations applied to features $\mathbf{x}$ by $f_{\text{target}}(\mathbf{x})$. The output for the main malware/benign prediction task – $f_{\text{mal}}(\mathbf{x})$ – is always present and consists of a single dense layer followed by a sigmoid activation on top of the base shared network that aims to estimate the probability of the input sample being malicious. A network architecture with only this malware/benign output serves as the baseline for our evaluations. To this baseline model we add auxiliary outputs with similar structure as described above: one fully connected layer (two for the *tag* prediction task in Section 3.4) which produces some task-specific number of outputs (a single output, with the exception of the restricted generalized Poisson distribution output, which uses two) and some task-specific activation described in the associated sections below.

Except where noted otherwise, all multi-task losses were produced by computing the sum, across all tasks, of the per-task loss multiplied by a task-specific weight (1.0 for the malware/benign task and 0.1 for all other tasks; see Section 4). Training was standardized at 10 epochs; for all experiments we used a training set of 9 million samples and a test set of approximately 7 million samples. Additional details about the training and test data are reported in Section 3.6. Additionally, we used a validation set of 100,000 samples to ensure that each network had converged to an approximate minimum on validation loss after 10 epochs. All of our models were implemented in Keras [6] and optimized using the Adam optimizer [15] with Keras's default parameters.

3.1 Malware Loss

As explained in Section 2, for the task of predicting if a given binary file, represented by its features $\mathbf{x}^{(i)}$, is malicious or benign we used a binary cross-entropy loss between the malware/benign output of the network $\hat{y}^{(i)} = f_{\text{mal}}(\mathbf{x}^{(i)})$ and the malicious label $y^{(i)}$. This results in the following loss for a dataset with M samples:

$$\begin{aligned} L_{\text{mal}}(X,Y) &= \frac{1}{M}\sum_{i=1}^{M} \ell_{\text{mal}}(f_{\text{mal}}(\mathbf{x}^{(i)}), y^{(i)}) \\ &= -\frac{1}{M}\sum_{i=1}^{M} y^{(i)} \log(\hat{y}^{(i)}) + (1-y^{(i)}) \log(1-\hat{y}^{(i)}). \end{aligned} \tag{1}$$

In this paper, we use a "1-/5+" criterion for labeling a given file as malicious or benign: if a file has one or fewer vendors reporting it as malicious, we label the file as 'benign' and use a weight of 1.0 for the malware loss for that sample. Similarly, if a sample has five or more vendors reporting it as malicious, we label the file as 'malicious' and use a weight of 1.0 for the malware loss for that sample.

3.2 Vendor Count Loss

To more finely distinguish between positive results, we investigate the use of the total number of 'malicious' reports for a given sample from the vendor aggregation service as an additional target; the rationale being that a sample with a higher number of malicious vendor reports should, all things being equal, be more likely to be malicious. In order to properly model this target, we require a suitable noise model for count data. A popular candidate is a Poisson noise model, parameterized by a single parameter μ, which assumes that counts follow a Poisson process, where μ is the mean and variance of the Poisson distribution. The probability of an observation of y counts conditional on μ is

$$P(y|\mu) = \mu^y e^{-\mu}/y!. \tag{2}$$

In our problem, as we expect the mean number of positive results for a given sample to be related to the file itself, we attempt to learn to *estimate* μ conditional on each sample $\mathbf{x}^{(i)}$ in such a way that the likelihood of $y^{(i)}|\mu^{(i)}$ is maximized (or, equivalently, the negative log-likelihood is minimized). Denote the output of the neural network with which we are attempting to estimate the mean count of vendor positives for sample i as $f_{\text{cnt}}(\mathbf{x}^{(i)})$. Note that under a non-distributional loss, this would be denoted by $\hat{y}^{(i)}$, however since we are fitting a parameter of a distribution, and not the sample label y directly, we use different notation in this section. By taking some appropriate activation function $a(\cdot)$ that maps $f_{\text{cnt}}(\mathbf{x}^{(i)})$ to the non-negative real numbers, we can write $\mu^{(i)} = a\left(f_{\text{cnt}}(\mathbf{x}^{(i)})\right)$. Consistent with generalized linear model (GLM) literature [18], we use an exponential activation for a, though one could equally well employ some other transformation with the correct output range, for instance the ReLU function.

Letting $y^{(i)}$ here denote the actual number of vendors that recognized sample $\mathbf{x}^{(i)}$ as malicious, the corresponding negative log-likelihood loss over the dataset is

$$L_{\mathrm{p}}(X,Y) = \frac{1}{M}\sum_{i=1}^{M} \ell_{\mathrm{p}}\left(a\left(f_{\mathrm{cnt}}(\mathbf{x}^{(i)})\right), y^{(i)}\right)$$
$$= \frac{1}{M}\sum_{i=1}^{M} \mu^{(i)} - y^{(i)}\log(\mu^{(i)}) + \log(y^{(i)}!), \quad (3)$$

which we will refer to as the *Poisson* or *vendor count* loss. In practice, we ignore the $\log(y^{(i)}!)$ term when minimizing this loss since it does not depend on the parameters of the network.

A Poisson loss is more intuitive for dealing with count data than other common loss functions, even for count data not generated by a Poisson process. This is partly due to the discrete nature of the distribution and partly because the assumption of increased variance with predicted mean is more accurate than a homoscedastic – i.e., constant variance – noise model.

While the assumption of increasing variance with predicted count value seems reasonable, it is very unlikely that vendor counts perfectly follow a Poisson process – where the mean *is* the variance – due to correlations between vendors, which might occur from modeling choice and licensing/OEM between vendor products. The variance might increase at a higher or lower rate than the count and might not even be directly proportional to or increase monotonically with the count. Therefore, we also implemented a Restricted Generalized Poisson distribution [10] – a slightly more intricate noise model that accommodates dispersion in the variance of vendor counts. Given dispersion parameter α, the Restricted Generalized Poisson distribution has a probability mass function (pmf):

$$P(y|\alpha,\mu) = \left(\frac{\mu}{1+\alpha\mu}\right)^{y}(1+\alpha y)^{y-1}\exp\left(\frac{-\mu(1+\alpha y)}{1+\alpha\mu}\right)/y!. \quad (4)$$

When $\alpha = 0$, this reduces to Eq. 2. $\alpha > 0$ accounts for over-dispersion, while $\alpha < 0$ accounts for under-dispersion. Note that in our use case α, like μ, is estimated by the neural network and conditioned on the feature vector, allowing varying dispersion per-sample. Given the density function in Eq. 4, the resultant log-likelihood loss for a dataset with M samples is defined as:

$$L_{\mathrm{gp}}(X,Y) = -\frac{1}{M}\sum_{i=1}^{M}\Big[y^{(i)}\left(\log\mu^{(i)} - \log(1+\alpha^{(i)}\mu^{(i)})\right) + (y^{(i)}-1)\log(1+\alpha^{(i)}y^{(i)}) - \frac{\mu^{(i)}(1+\alpha^{(i)}y^{(i)})}{1+\alpha^{(i)}\mu^{(i)}} + \log(y^{(i)}!)\Big], \quad (5)$$

where $\alpha^{(i)}$ and $\mu^{(i)}$ are obtained as transformed outputs of the neural network in a similar fashion as we obtain $\mu^{(i)}$ for the Poisson loss. In practice, as for the Poisson loss, we dropped the term related to $y!$ since it does not affect the optimization of the network parameters.

Note also that restrictions must be placed on the negative value of the $\alpha^{(i)}$ term to keep the arguments of the logarithm positive. For numerical convenience, we used an exponential activation over the dense layer for our $\alpha^{(i)}$ estimator, which accommodates over-dispersion but not under-dispersion. Results from experiments comparing the use of Poisson and Generalized Poisson auxiliary losses are presented in Section 4.1.

While the Poisson distribution is a widely used model for count data, other discrete probability distributions could also be used to model the count of vendor positive results. During early experimentation we also examined the binomial, geometric, and negative binomial distributions as models for vendor counts, but found that they produced unsatisfactory results and so do not discuss them further.

3.3 Per-Vendor Malware Loss

The aggregation service from which we collected our data sets contains a breakout of individual vendor results per sample. We identified a subset $\mathcal{V} = \{v_1,\ldots,v_V\}$ of 9 vendors that each produced a result for (nearly) every sample in our data. Each vendor result was added as a target in addition to the malware target by adding an extra fully connected layer per vendor followed by a sigmoid activation function to the end of the shared architecture. We employed a binary cross-entropy loss per vendor during training. Note that this differs from the vendor count loss presented above in that each high-coverage vendor is used as an individual binary target, rather than being aggregated into a count. The aggregate *vendors* loss L_{vdr} for the $V = 9$ selected vendors is simply the sum of the individual vendor losses:

$$L_{\mathrm{vdr}}(X,Y) = \frac{1}{M}\sum_{i=1}^{M}\sum_{j=1}^{V}\ell_{\mathrm{vdr}}\left(f_{\mathrm{vdr}_j}\left(\mathbf{x}^{(i)}\right), y_{v_j}^{(i)}\right)$$
$$= -\frac{1}{M}\sum_{i=1}^{M}\sum_{j=1}^{V} y_{v_j}^{(i)}\log(\hat{y}_{v_j}^{(i)}) + (1-y_{v_j}^{(i)})\log(1-\hat{y}_{v_j}^{(i)}), \quad (6)$$

where ℓ_{vdr} is the per-sample binary cross-entropy function and $f_{\mathrm{vdr}_j}\left(\mathbf{x}^{(i)}\right) = \hat{y}_{v_j}^{(i)}$ is the output of the network that is trained to predict the label $y_{v_j}^{(i)}$ assigned by vendor j to input sample $\mathbf{x}^{(i)}$.

Results from experiments exploring the use of individual vendor targets in addition to malware label targets are presented in Section 4.2.

3.4 Malicious Tags Loss

In this experiment we attempt exploit information contained in family detection names provided by different vendors in the form of malicious tags. We define each tag as a high level description of the purpose of a given malicious sample. The tags used as auxiliary targets in our experiments are: *flooder*, *downloader*, *dropper*, *ransomware*, *crypto-miner*, *worm*, *adware*, *spyware*, *packed*, *file-infector*, and *installer*.

We create these tags from a parse of individual vendor detection names, using a set of 10 vendors which from our experience provide high quality detection names. Once we have extracted the most common tokens, we filter them to keep only tokens related to well-known malware family names or tokens that could easily be associated with one or more of our tags, for example, the token *xmrig* – even though it is not a malware family – can be recognized as referring to a crypto-currency mining software and therefore can be associated with the *crytpo-miner* tag. We then create a mapping from tokens to tags based on prior knowledge. We label a sample as associated with tag t_i if any of the tokens associated with t_i are present in any of the detection names assigned to the sample by the set of trusted vendors.

Annotating our dataset with these tags, allows us to define the tag prediction task as multi-label binary classification, since zero or more tags from the set of possible tags $\mathscr{T} = \{t_1, \ldots, t_T\}$ can be present at the same time for a given sample. We introduce this prediction task in order to have targets in our loss function that are not not directly related to the number of vendors that recognize the sample as malicious. The vendor counts and the individual vendor labels are closely related with the definition of our main target, i.e. the malicious label, which classifies a sample as malicious if 5 or more vendors identify the sample as malware (see Section 3.1). In the case of the tag targets, this information is not present. For instance, if all the vendors recognize a given sample as coming from the *WannaCry* family in their detection names, the sample will be associated only once with the *ransomware* tag. On the converse, because of our tagging mechanism, if only one vendor considers that a given sample is malicious and classifies it as coming from the *WannaCry* family, the *ransomware* tag will be present (although our malicious label will be 0).

In order to predict these tags, we use a *multi-headed* architecture in which we add two additional layers per tag to the end of the shared base architecture, a fully connected layer of size 512-to-256, followed by a fully connected layer of size 256-to-1, followed by a sigmoid activation function, as shown in Figure 1. Each tag t_j out of the possible $T = 11$ tags has its own loss term computed with binary cross-entropy. Like the per-vendor malware loss, the aggregate tag loss is the sum of the individual tag losses. For the dataset with M samples it becomes:

$$\begin{aligned} L_{\text{tag}}(X,Y) &= \frac{1}{M}\sum_{i=1}^{M}\sum_{j=1}^{T} \ell_{\text{tag}}\left(f_{\text{tag}_j}\left(\mathbf{x}^{(i)}\right), y_{t_j}^{(i)}\right) \\ &= -\frac{1}{M}\sum_{i=1}^{M}\sum_{j=1}^{T} y_{t_j}^{(i)} \log(\hat{y}_{t_j}^{(i)}) + (1-y_{t_j}^{(i)}) \log(1-\hat{y}_{t_j}^{(i)}), \end{aligned} \tag{7}$$

where $y_{t_j}^{(i)}$ indicates if sample i is annotated with tag j, and $\hat{y}_{t_j}^{(i)} = f_{\text{tag}_j}\left(\mathbf{x}^{(i)}\right)$ is the prediction issued by the network for that value.

3.5 Sample Weights

While our multi-objective network has the advantage that multiple labels and loss functions serve as additional sources of information, this introduces an additional complexity: given many (potentially missing) labels for each sample, we cannot rely on having all labels for a large quantity of the samples. Moreover, this problem gets worse as more labels are added. To address this, we incorporated per-sample weights, depending on the presence and absence of each label. For labels that are missing, we assign them to a default value and then set the associated weights to zero in the loss computation so a sample with a missing target label will not add to the loss computation for that target. Though this introduces slight implementation overhead, it allows us to train our network, even in the presence of partially labeled samples (e.g., when a vendor decides not to answer).

3.6 Dataset

We collected two datasets of PE files and associated metadata from a threat intelligence feed: a set for training/validation and a test set. For the training/validation set, we pulled 20M PE files and associated metadata, randomly sub-selecting over a year – from September 6, 2017 to September 6, 2018. For the test set, we pulled files from October 6, 2018 to December 6, 2018. Note also that we indexed files based on unique SHA for first seen time, so every PE in the test set comes temporally after the ones in the training set. We do not use a randomized cross-validation training/test split as is common in other fields, because that would allow the set on which the classifier was trained to contain files "from the future", leading to spuriously optimistic results. The reason for the one month gap between the end of the training/validation set and the start of the test set is to simulate a realistic worst-case deployment scenario where the detection model of interest is updated on a monthly basis. All files used in the following experiments – both malicious and benign – were obtained from the threat intelligence feed.

We then extracted 1024-element feature vectors for all those files using feature type described in [25] and derived

an aggregate malicious/benign label using a 1-/5+ criterion as described above. Invalid PE files were discarded.

Of the valid PE files from which we were able to extract features we further subsampled our training dataset to 9,000,000 training samples, with 7,188,150 (79.87%) malicious and 1,587,035 (17.63%) benign. The remaining 224,815 (2.5%) are *gray* samples, without a benign or malicious label, i.e., samples where the total number of vendor detections is between 2 and 4 and thus do not meet our 1-/5+ labeling criterion. Our validation set was also randomly subsampled from the same period as the training data and used to monitor convergence during training. It consisted of 100,000 samples; of these, 17,620 were benign (17.62%), 79,819 were malicious (79.82%) , and 2,561 were gray (2.56%). Our test set exhibited similar statistics, with 7,656,573 total samples, 1,606,787 benign (21.8%), 5,762,064 malicious (78.2%), and 287,722 gray (3.76%). Further statistics for the distribution of vendor counts and tags in our datasets are presented in Appendix A.1.

The ratios of malicious to benign data in our training, test, and validation sets are comparable, with malicious samples more prevalent than benign samples. Note that this class balance differs substantially from a real-world deployment scenario, where malware is rarely seen. Increasing the prevalence of this low-occurrence class when training on unbalanced data sets is commonly done to avoid overfitting [3] (we have also observed this in practice), and using a data set with a higher proportion of malicious samples assuming a sufficient number of benign samples – may lead to a more precise decision boundary, and better overall performance as measured by the full ROC curve. Further, when using our malicious tags loss, a greater diversity in malware can yield a more diverse tag set to learn from during training.

Note that ROC curves, which we use as performance measures in Sections 4 and 5, are independent of class ratio in the test set (unlike accuracy), since false positive rate (FPR) values depend only on the benign data, and true positive rate (TPR) values depend only on malware. We also focus on improvements in detection at the very low FPR of 0.1% or below, where we see the most dramatic improvements, since several publications by anti-virus vendors [25, 30] and our experience suggest that 0.1% or lower is indeed a practical FPR target for most deployment scenarios. Our model outputs can also be easily (without retraining) rescaled to the desired deployment class ratio, based on the provided ROC curve and/or standard calibration methods, e.g., fitting a weighted isotonic regressor on scores from the validation set with each score contribution weighted according to its ground truth label to correct the class balance discrepancy between the validation set and the expected deployment setting, then using that regressor to calibrate scores during test/deployment.

4 Experimental Evaluation

In this section, we apply the auxiliary losses presented in in Section 3, first individually, each loss in conjunction with a main malicious/benign loss, and then simultaneously in one combined model. We then compare to a baseline model, finding that each loss term yields an improvement, either in Receiver Operating Characteristic (ROC) net area under the curve (AUC) or in terms of detection performance at low false positive rates (FPR). We note that none of the auxiliary losses we present below harmed classification relative to the baseline model; at worst, our loss-augmented models had equivalent performance to the baseline model with respect to AUC and low-FPR ROC performance on the aggregate malicious/benign label. Each model used a loss weight of 1.0 on the aggregate malicious/benign loss and 0.1 on each auxiliary loss, i.e. when we add K targets to the main loss, the final loss that gets backpropagated through the model becomes

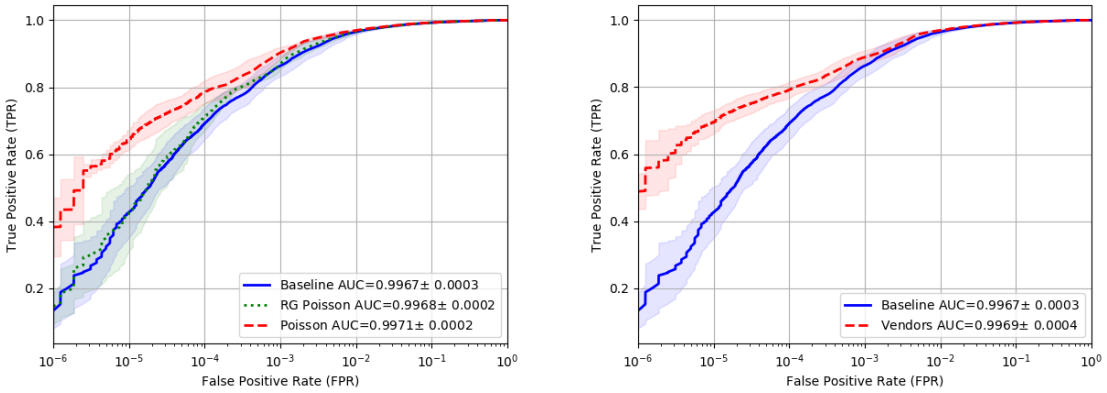
$$L(X,Y) = L_{\text{mal}}(X,Y) + 0.1 \sum_{k=1}^{K} L_k(X,Y). \tag{8}$$

Results are depicted in graphical form in Figure 2 and in tabular form in Table 1.

As the training process for deep neural networks has some degree of intrinsic randomness, which can result in variations in their performance, we report our results in terms of both the mean and standard deviation for the test statistics of interest across five runs. Each model was trained five times, each time with a different random initialization and different randomization over minibatches, and all other parameters (optimizer and learning rate, training data, model structure, number of epochs, etc.) held identical. We compute the test statistic of interest (e.g. the detection rate at a false positive rate of 10^{-3}) for each model, and then compute the average and standard deviation of those results. Notice that the ROC curves in Figure 2 are plotted on a logarithmic scale for visibility, since the baseline performance is already quite high and significant marginal improvements are difficult to discern. For this reason, we also include relative reductions in mean true positive detection error (the rate at which the model fails to detect malware samples – or false negative rate – averaged over the five model results) and in standard deviation from the baseline for our best model in Table 1, and for all models in Table C.1 in the appendix.

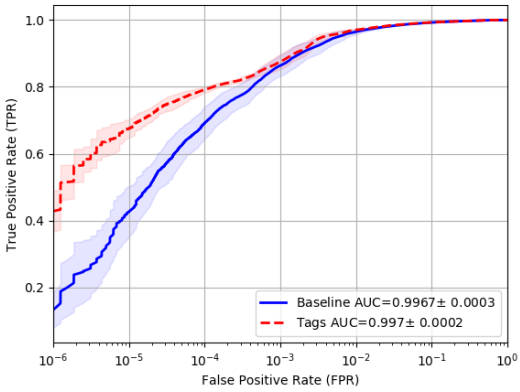
4.1 Vendor Count Loss

We employed the same base PE model topology as for our other experiments, with a primary malicious/benign binary cross-entropy loss, and an auxiliary count loss. We experimented with two different loss functions for the count loss – a Poisson loss and a Restricted Generalized Poisson loss

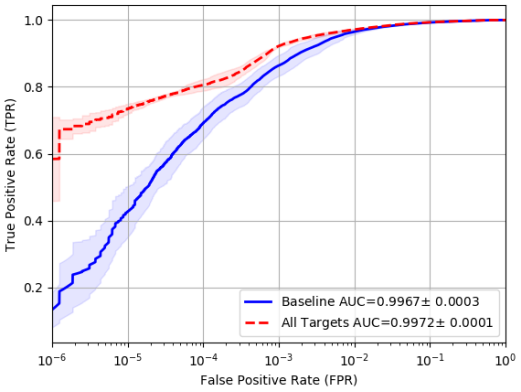


(a) Count Loss

(b) Vendor Loss



(c) Tag Loss



(d) Combined Loss

Figure 2: ROC curves and AUC statistics for count, vendor, and tag experiments compared to our baseline. Lines represent the mean TPR at a given FPR, while shaded regions represent ± 1 standard deviation. Statistics were computed over 5 training runs, each with random parameter initialization. (a) *Count loss*. Our baseline model (blue solid line) is shown compared to a model employing a Poisson auxiliary loss (red dashed line), and a dispersed Poisson auxiliary loss (green dotted line). (b) Auxiliary loss on multiple vendors malicious/benign labels (red dashed line) and baseline (blue solid line). (c) Auxiliary loss on semantic attribute tags (red dashed line) and baseline (blue solid line). (d) Our combined model (red dashed line) and baseline (blue solid line). The combined model utilizes an aggregate malicious/benign loss with an auxiliary Poisson count loss, a multi-vendor malicious/benign loss, and a malware attribute tag loss.

(equations 3 and 5 respectively). For the Poisson loss, we used an exponential activation over a dense layer atop the base to estimate $\mu^{(i)}$. For the Restricted Generalized Poisson (RG-Poisson) loss, we followed a similar pattern using two separate dense layers with exponential activations on top; one for the $\mu^{(i)}$ parameter and another for the $\alpha^{(i)}$ parameter. The choice of an exponential activation is consistent with statistics literature on Generalized Linear Models (GLMs) [18].

Results on the malware detection task using Poisson and RG-Poisson losses as an auxiliary loss function are shown in Figure 2a. When compared to a baseline using no auxiliary loss, we see a statistically significant improvement with the Poisson loss function in both AUC and ROC curve, particularly in low false positive rate (FPR) regions. The RG-Poisson loss, by contrast, yields no statistically significant gains over the baseline in terms of AUC, nor does it appear to yield statistically significant gains at any point along the ROC curve.

This suggests that the RG-Poisson loss model is ill-fit, which could stem from a variety of issues. First, if counts are under-dispersed, an over-dispersed Poisson loss could be an inappropriate model. Under-dispersion could occur if certain vendors disproportionately trigger simultaneously or be-

	FPR				
	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}
TPR Baseline	0.427 ± 0.076	0.692 ± 0.049	0.864 ± 0.031	0.965 ± 0.007	0.9928 ± 0.0007
TPR Poisson	0.645 ± 0.029	0.785 ± 0.034	0.903 ± 0.016	$0.970 \pm \mathbf{0.001}$	$0.9932 \pm \mathbf{0.0002}$
TPR RG Poisson	0.427 ± 0.116	0.711 ± 0.041	0.870 ± 0.016	0.966 ± 0.003	0.9930 ± 0.0003
TPR Vendors	0.697 ± 0.034	0.792 ± 0.024	0.889 ± 0.020	0.970 ± 0.004	0.9928 ± 0.0014
TPR Tags	0.677 ± 0.027	$0.792 \pm \mathbf{0.009}$	0.875 ± 0.022	0.971 ± 0.004	0.9932 ± 0.0008
TPR All Targets	$\mathbf{0.735} \pm \mathbf{0.014}$	$\mathbf{0.806} \pm 0.017$	$\mathbf{0.922} \pm \mathbf{0.004}$	$\mathbf{0.972} \pm 0.003$	$\mathbf{0.9934} \pm 0.0004$
% Error Reduction (All Targets)	**53.8%**	**37.0%**	**42.7%**	**20.0%**	**8.3%**
% Variance Reduction (All Targets)	**81.6%**	65.3%	**87.1%**	57.1%	94.3%

Table 1: Top: Mean and standard deviation true positive rates (TPRs) for the different experiments in Section 4 at false positive rates (FPRs) of interest. Results were aggregated over five training runs with different weight initializations and minibatch orderings. Best results consistently occurred when using all auxiliary losses and are shown in bold. Bottom: Percentage reduction in missed true positive detections and percentage reductions in ROC curve standard deviation resulting from the best model (All Targets) compared to the baseline across various FPRs. State-of-the-art results are shown in **bold**.

cause counts are inherently bounded by the net number of vendors. Second, a Poisson model, even with added dispersion parameters, is an ill-posed model of count data, but removing the dispersion parameter removes a dimension in the parameter space to over-fit on. Inspecting the dispersion parameters predicted by the RG-Poisson model, we noted that they were relatively large, which supports the latter hypothesis. We also noticed that the RG-Poisson model converged significantly faster than the Poisson model in terms of malware detection loss.

4.2 Modeling Individual Vendor Responses

Incorporating an auxiliary multi-label binary cross-entropy loss across vendors (cf. Section 3.3) in conjunction with the main malicious/benign loss yields a similar increase in the TPR at low FPR regions of the ROC curve (see Figure 2b) to the Poisson experiment. Though we do not see a significant increase in AUC, since the improvement is integrated across an extremely narrow range of FPRs, this improvement in TPR at lower FPRs may still be operationally significant, and does indicate an improvement in the model.

4.3 Incorporating Tags as Targets

In this experiment we extend the architecture of our base network to predict, not only the malware/benign label, but also the set of 11 tags defined in Section 3.4. For this, we add two fully connected layers per tag to the end of the base architecture (see Section 3.4) which serve to identify each tag from the shared representation. Each of these tag-specialized layers predicts a binary output corresponding to presence/absence of the tag and has an associated binary cross-entropy loss that gets added to the other tag losses and the main malicious/benign loss. The overall loss for this experiment is a sum containing one term per tag, weighted by a loss weight of 0.1 (as mentioned at the beginning of this section), and one term for the loss incurred on the main task.

The result of this experiment is represented via the ROC curves of Figure 2c. Similar to section 4.2 we see no statistical difference in the AUC values with respect to the baseline, but we do observe substantial statistical improvement in the predictions of the model in low FPR regions, particularly for FPR values lower than 10^{-3}. Furthermore, we also witness a substantial decrease in the variance of the ROC curve.

4.4 Combined Model

Finally, we extend our model to predict all auxiliary targets in conjunction with the aggregate label, with a net loss term containing a combination of all auxiliary losses used in previous experiments. The final loss function for the experiment is the sum of all the individual losses where the malware/benign loss has a weight of 1.0 while the rest of the losses have a weight of 0.1.

The resulting ROC curve and AUC are shown in Figure 2d. The AUC of 0.9972 ± 0.0001 is the highest obtained across all the experiments conducted in this study. Moreover, in contrast to utilizing any single auxiliary loss, we see a noticeable improvement in the ROC curve not only in very low FPR regions, but also at 10^{-3} FPR. Additionally, variance is consistently lower across a range of low-FPR values for this combined model than for our baseline or any previous models. An exception is near 10^{-6} FPR where measuring variance is an ill-posed problem because even with a test dataset of over 7M samples, detecting or misdetecting even one or two of them can significantly affect detection rate.

In order to account for the effect of gray samples in the evaluation of our detection model, we re-scanned a subset of those at a later point in time, giving the AV community time to update their detection rules, and evaluated the prediction issued by the model. Even though it is naturally harder to

determine maliciousness of these samples (otherwise they would not initially have been categorized as gray), we find that our model predicts the correct labels for more than 77% of them. A more in depth analysis of grey samples is deferred to Appendix B.

5 Discussion

In this section, we examine the effects of different types of auxiliary loss functions on main task ROC curve. We then perform a sanity check to establish whether our performance increases result from additional information added to our neural networks by the auxiliary loss functions or are the artifact of some regularization effect.

5.1 Modes of Improvement

Examining the plots in Figure 2, we see three different types of improvement that result from our auxiliary losses:

1. A bump in TPR at low FPR ($< 10^{-3}$).
2. A net increase in AUC and a small bump in performance at higher FPRs ($\geq 10^{-3}$).
3. A reduction in variance.

Improvement 1 is particularly pronounced in the plots due to the logarithmic scale, but it does not substantially contribute to net AUC due to the narrow FPR range. However, this low-FPR part of the ROC is important from an operational perspective when deploying a malware detection model in practice. Substantially higher TPRs at low FPR could allow for novel use cases in an operational scenario. Notice that this effect is more pronounced for auxiliary losses containing multi-objective binary labels (Figs. 2b, 2c, and 2d) than for the Poisson loss, suggesting that it occurs most prominently when employing our multi-objective binary label losses. Let us consider why a multi-objective binary loss might cause such an effect to occur: At low FPRs, we see high thresholds on the detection score from the main output of the network. To surpass this threshold and register as a detection, the main sigmoid output must be very close to 1.0, i.e., very high response. Under a latent correlation with the main output, a high-response hit for an auxiliary target label could also boost the response for the main detector output, while a baseline model without this information might wrongly classify the sample as benign. We hypothesize that improvement 1 occurs from having many objectives simultaneously and thereby increasing chances for a high-response target hit. The loss type may or may not be incidental, which is consistent with its noticeable but less pronounced presence under a single-objective Poisson auxiliary loss (Figure 2a).

Improvement 2 likely stems from improvements in detection rate that we see around 10^{-3} FPR and higher. Notice that these effects are more pronounced in Figs. 2a and 2d, are somewhat noticeable in Figure 2b, and are not noticeable in Figure 2c, consistent with the resultant AUCs. This suggests that the effect occurs most prominently in the presence of an auxiliary count loss. We postulate that this occurs because our aggregate detection label is derived by *thresholding* the net number of vendor detections for each sample but doing so removes a notional view of confidence that a sample is malicious. Alternatively stated, thresholding removes information on the difficulty of classifying a malicious sample or the extent of "maliciousness" that the number of detection counts provides. Bear in mind that some detectors are better at detecting different types of malware than others, so more detections suggest a *more malicious* file, e.g., with more malware components, or a more widely blacklisted file (higher confidence). Providing information on the number of counts in an auxiliary loss function may therefore provide the classifier more principled information on how to order detection scores, thus allowing for more effective thresholding and a better ROC curve.

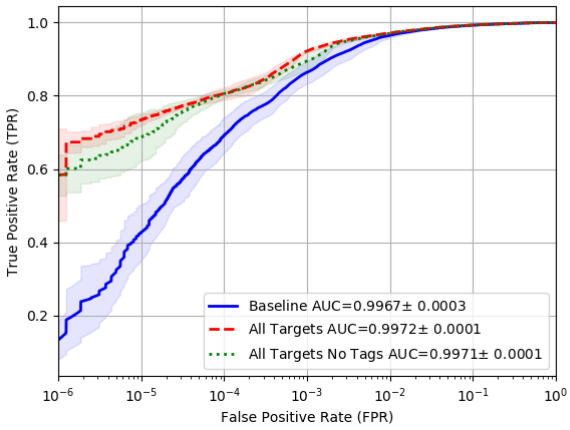


Figure 3: When we remove the attribute tags loss (green dotted line) we get a similar shaped ROC curve with similar ROC compared to using all losses (red dashed line), but with slightly higher variance in the ROC. This supports our hypotheses about effects of different loss functions on the shape of the ROC curve. The baseline is shown as a blue solid line for comparison.

Improvement 3 occurs across all loss types, particularly in low FPR ranges, with the exception of *very low* FPRs (e.g., 10^{-6}), where accurately measuring mean and variance is an ill-posed problem due to the size of the dataset (cf. section 4.4). Comparing the ROC plots in Figure 2, the reduction in variance appears more pronounced as the number of losses increases. Intuitively, this is not a surprising result since adding objectives/tasks imposes constrains the allowable weight space – while many choices of weights might allow a network to perform a single task well only a subset of these choices will work well for all tasks simultaneously.

Thus, assuming equivalent base topology, we expect a network that is able to perform at least as well on multiple tasks as many single-task networks to exhibit lower variance.

Combining all losses seems to accentuate all improvements (1-3) with predictable modes which we attribute to our various loss types (Figure 2d) – higher detection rate at low FPR brought about primarily by multi-objective binary losses, a net AUC increase and a detection bump at 10^{-3} FPR brought about by the count loss, and a reduced variance brought about by many loss functions. To convince ourselves that this is not a coincidence, we also trained a network using only Poisson and vendor auxiliary losses but no attribute tags (cf. Figure 3). As expected, we see that this curve exhibits similar general shape and AUC characteristics that occur when training with all loss terms, but the variance appears slightly increased.

In the variance reduction sense, we can view our auxiliary losses as regularizers. This raises a question: are improvements 1 and 2 actually occurring for the reasons that we hypothesize or are they merely naive result of regularization?

5.2 Representation or Regularization?

While the introduction of some kinds of auxiliary targets appears to improve the model's performance on the main task, it is less clear why this is the case. The reduction in variance produced by the addition of extra targets suggests one potential alternative explanation for the observed improvement: rather than inducing a more discriminative representation in the hidden layers of the network, the additional targets may be acting as constraints on the network, reducing the space of viable weights for the final trained network, and thus acting as a form of additional regularization. Alternatively, the addition of extra targets may simply be accelerating training by amplifying the gradient; while this seems unlikely given our use of a validation set to monitor approximate convergence, we nevertheless also investigate this possibility.

To evaluate these hypotheses, we constructed three additional targets (and associated loss functions) that provided uninformative targets to the model: i) a pseudo-random target that is approximately independent of either the input features or the malware/benign label; ii) an additional copy of the main malware target transformed to act as a regression target; and iii) an extra copy of the main malware target.

The random target approach attempts to directly evaluate whether or not an additional pseudo-random target might improve network performance by 'using up' excess capacity that might otherwise lead to overfitting. We generate pseudo-random labels for each sample based off of the parity of a hash of the file contents. While this value is effectively random and independent of the actual malware/benignware label of the file, the use of a hash value ensures that a given sample will always produce the same pseudo-random target. This target is fit via standard binary cross-entropy loss against a sigmoid output,

$$L_{\text{rnd}}(X,Y) = -\frac{1}{M}\sum_{i=1}^{M} y^{(i)} \log f_{\text{rnd}}(\mathbf{x}^{(i)}) + (1-y^{(i)}) \log\left(1 - f_{\text{rnd}}(\mathbf{x}^{(i)})\right), \quad (9)$$

where $f_{\text{rnd}}\left(\mathbf{x}^{(i)}\right)$ is the output of the network which is being fit to the random target $y^{(i)}$.

In contrast, the duplicated regression target evaluates whether further constraining the weights *without* requiring excess capacity to model additional independent targets has an effect on the performance on the main task. The model is forced to adopt an internal representation that can satisfy two different loss functions for perfectly correlated targets, thus inducing a constraint that does not add additional information. To do this, we convert our binary labels (taking on values of 0 and 1 for benign and malware, respectively) to -10 and 10, and add them as additional *regression* targets fit via mean squared error (MSE). Taking $y^{(i)}$ as the ith binary target and $f_{\text{MSE}}(\mathbf{x}^{(i)})$ as the regression output of the network, we can express the MSE loss as:

$$L_{\text{mse}}(X,Y) = \sum_{i=1}^{M} \left(f_{\text{mse}}\left(\mathbf{x}^{(i)}\right) - 20\left(y^{(i)} - 0.5\right)\right)^2. \quad (10)$$

Finally, in the case of the duplicated target, the model effectively receives a larger gradient due to a duplication of the loss. The loss for this label uses the same cross-entropy loss as for the main target, obtained by substituting $f_{\text{dup}}(\mathbf{x}^{(i)})$ for $f_{\text{mal}}(\mathbf{x}^{(i)})$ in equation 1 as the additional model output that is fit to the duplicated target. Note that we performed two variants of the duplicated target experiment: one in which both the dense layer prior to the main malware target and the dense layer prior to the duplicated target were trainable, and one in which the dense layer for the duplicate target was frozen at its initialization values to avoid the trivial solution where the pre-activation layer for both the main and duplicate target were identical. In both cases, the results were equivalent; only results for the trainable case are shown.

Both f_{rnd} and f_{dup} are obtained by applying a dense layer followed by a sigmoid activation function to the intermediate output of the input sample from the shared base layer (**h** in Figure 1), while $f_{\text{mse}}(\mathbf{x}^{(i)})$ is obtained by passing the intermediate representation of the input sample **h** through a fully connected layer with no output non-linearity.

Results of all three experiments are shown in Figure 4. In no case did the performance of the model on the main task improve statistically significantly over the baseline. This suggests that auxiliary tasks must encode relevant information to improve the model's performance on the main task. For each of the three auxiliary loss types in Figure 4, there is

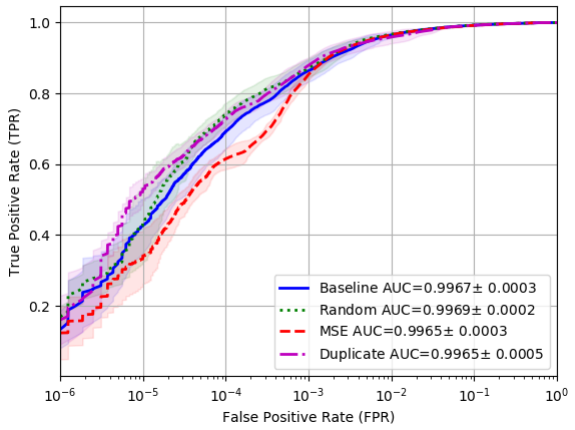


Figure 4: ROC curves comparing classification capabilities of models on the malware target when either random (green dotted line), regression (red dashed line), or duplicated targets (magenta dashed and dotted line) are added as auxiliary losses. Note that with the exception of the regression loss – which appears to harm performance – there is little discernible difference between the remaining ROC curves. The baseline is shown as a blue solid line for comparison.

no additional information provided by the auxiliary targets: the random target is completely uncorrelated from any information in the file (and indeed the final layer is ultimately dominated by the bias weights and produces a constant output of 0.5 regardless of the inputs to the layer), while the duplicated and MSE layers are perfectly correlated with the final target. In either case, there is no incentive for the network to develop a richer representation in layers closer to the input; the final layer alone is sufficient given an adequate representation in the core of the model.

6 Related Work

Applications of ML to computer security date back to the 1990's [21], but large-scale commercial deployments of deep neural networks (DNNs) that have led to transformative performance gains are a more recent phenomenon. Several works from the ML-Sec community have leveraged DNNs for statically detecting malicious content across a variety of different formats and file types [25, 26, 23]. However, these works predominantly focus on applying regularized cross-entropy loss functions between single network outputs and malicious/benign labels per-sample, leaving the potential of multiple-objective optimization largely untapped.

A notable exception, which we build upon in this work, is [11], in which Huang et al. add a multiclass label for Microsoft's malware families to their classification model using a categorical cross-entropy loss function atop a softmax output as an auxiliary objective. They observed that adding targets in this fashion increased performance both on the detection task and on the malware family classification task. Our work builds upon theirs in several respects. First, while their work used 4000-dimensional dynamic features derived from Windows API calls, we extend multi-target approaches to lower-dimensional static features on a larger data set. In this respect, our work pioneers a more scalable approach, but lacks the advantages of dynamic features that their approach provides. Second, we demonstrate that improvements from multi-target learning also occur using far more targets, and we introduce heterogeneous loss functions, i.e., binary cross-entropy and Poisson, whereas their work employs only two categorical cross-entropy losses. Finally, our work introduces loss-weighting to account for potentially missing labels, which may not be problematic for only two targets but become more prevalent with additional targets.

Despite the lack of attention from the ML-Sec community, multi-target learning has been applied to other areas of ML for a long time. The work of Abu-Mostafa [2] predates most explicit references to multi-task learning by introducing the concept of *hints*, in which known invariances of a solution (e.g., translation invariance, invariance under negation) can be incorporated into network structure and used to generate additional training samples by applying the invariant operation to the existing samples, or – most relevant to our work – used as an additional target by enforcing that samples modified by an invariant function should be *both* correctly classified *and* explicitly classified identically. Caruna [5] first introduced multi-task learning in neural networks as a "source of inductive bias" (also reframed as inductive transfer in [4]), in which more difficult tasks could be combined in order to exploit similarities between tasks that could serve as complementary signals during training. While his work predates the general availability of modern GPUs, and thus the models and tasks he examines are fairly simple, Caruna nevertheless demonstrates that jointly learning related tasks produces better generalization on a task-by-task basis than learning them individually. It is interesting to note that in [5] he also demonstrated that learning multiple copies of the same task can also lead to a modest improvement in performance (which we did not observe in this work, possibly due to the larger scale and complexity of our task).

Kumar and Duame [17] consider a refinement on the basic multi-task learning approach that leads to clustering related tasks, in an effort to mitigate the potential of *negative transfer* in which unrelated tasks degrade performance on the target task. Similarly, the work of Rudd et al. [22] explores the use of domain-adaptive weighting of tasks during the training process.

Multi-target learning has been applied to extremely complex image classification tasks, including predicting characters and ngrams within unconstrained images of text [14], joint facial landmark localization and detection [19], image

tagging and retrieval [12, 31], and attribute prediction [1, 22] where a common auxiliary task is to challenge the network to classify additional attributes of the image, such as manner of dress for full-body images of people or facial attributes (e.g., smiling, narrow eyes, race, gender). While a range of neural network structures are possible, common exemplars include largely independent networks with a limited number of shared weights (e.g., [1]), a single network with minimal separation between tasks (e.g., [22]), or a number of parallel single-task classifiers in which the weights are constrained to be similar to each other. A more complex approach may be found in [24], in which the sharing between tasks is learned automatically in an online fashion.

Other, more distantly connected domains of ML research reinforce the intuition that learning on disparate tasks can improve model performance. Work in semi-supervised learning, such as [16] and [20], has shown the value of additional reconstruction and denoising tasks to learn representations valuable for a core classification model, both through regularization and through access to a larger dataset than is available with labels. The widespread success of transfer learning is also a testament to the value of training a single model on nominally distinct tasks. BERT [8], a recent example from the Natural Language Processing literature, shows strong performance gains from pre-training a model on masked-word prediction and predictions of whether two sentences appear in sequence, even when the true task of interest is quite distinct (e.g. question answering, translation).

Multi-view learning (see [32] for a survey) is a related approach in which multiple inputs are trained to a single target. This approach also arguably leads to the same general mechanism for improvement: the model is forced to learn relationships between sets of features that improve the performance using any particular set. While this approach often requires all sets of features to be available at test time, there are other approaches, such as [28], that relax this constraint.

7 Conclusion

In this paper, we have demonstrated the effectiveness of auxiliary losses for malware classification. We have also provided experimental evidence which suggests that performance gains result from an improved and more informative representation, not merely a regularization artifact. This is consistent with our observation that improvements occur as additional auxiliary losses and different loss types are added. We also note that different loss types have different effects on the ROC; multi-label vendor and semantic attribute tag losses have greatest effect at low false positive rates ($\leq 10^{-3}$), while Poisson counts have a substantial net impact on AUC, the bulk of which stems from detection boosts at higher FPR.

While we experimented on PE malware in this paper, our auxiliary loss technique could be applied to many other problems in the ML-Sec community, including utilizing a label on format/file type for format-agnostic features (e.g., office document type in [23]) or file type under a given format, for example APKs and JARs both share an underlying ZIP format; a zip-archive malware detector could use tags on the file type for auxiliary targets. Additionally, tags on topics and classifications of embedded URLs could serve as auxiliary targets when classifying emails or websites.

One open question is whether or not multiple auxiliary losses improve each others' performances as well as the main task's. If the multiple outputs of operational interest (such as the tagging output) can be trained simultaneously while also increasing (or at least not decreasing) their joint accuracy, this could lead to models that are both more compact and more accurate than individually deployed ones. In addition to potential accuracy gains, this has significant potential operational benefits, particularly when it comes to model deployment and updates. We defer a more complete evaluation of this question to future work.

While this work has focused on applying auxiliary losses in the context of deep neural networks, there is nothing mathematically that precludes using them in conjunction with a number of other classifier types. Notably, gradient boosted classifier ensembles, which are also popular in the ML-Sec community could take very similar auxiliary loss functions even though the structure of these classifiers is much different. We encourage the ML-Sec research community to implement multi-objective ensemble classifiers and compare with our results. Our choice of deep neural networks for this paper is infrastructural more than anything else; while several deep learning platforms, including PyTorch, Keras, and Tensorflow among others easily support multiple objectives and custom loss functions, popular boosting frameworks such as lightGBM and XGBoost have yet to implement this functionality.

The analyses conducted herein used metadata that can naturally be transformed into a label source and impart additional information to the classifier with no extra data collection burden on behalf of the threat intelligence feed. Moreover, our auxiliary loss technique does not change the underlying feature space representation. Other types of metadata, e.g., the file path of the malicious binary or URLs extracted from within the binary might be more useful in a multi-view context, serving as input to the classifier, but this approach raises challenges associated with missing data that our loss weighting scheme trivially addresses. Perhaps our weighting scheme could even be extended, e.g., by weighting each sample's loss contribution according to certainty/uncertainty in that sample's label, or re-balancing the per-task loss according to the expected frequency of the label in the target distribution. This could open up novel applications, e.g., detectors customized to a particular user endpoints and remove sampling biases inherent to multi-task data.

8 Acknowledgments

This research was sponsored by Sophos PLC. We would additionally like to thank Adarsh Kyadige, Andrew Davis, Hillary Sanders, and Joshua Saxe for their suggestions that greatly improved this manuscript.

References

[1] ABDULNABI, A. H., WANG, G., LU, J., AND JIA, K. Multi-task cnn model for attribute prediction. *IEEE Transactions on Multimedia 17*, 11 (2015), 1949–1959.

[2] ABU-MOSTAFA, Y. S. Learning from hints in neural networks. *J. Complexity 6*, 2 (1990), 192–198.

[3] ANDERSON, H. S., AND ROTH, P. Ember: an open dataset for training static pe malware machine learning models. *arXiv preprint arXiv:1804.04637* (2018).

[4] CARUANA, R. A dozen tricks with multitask learning. In *Neural networks: tricks of the trade*. Springer, 1998, pp. 165–191.

[5] CARUNA, R. Multitask learning: A knowledge-based source of inductive bias. In *Machine Learning: Proceedings of the Tenth International Conference* (1993), pp. 41–48.

[6] CHOLLET, F., ET AL. Keras, 2015.

[7] CLEVERT, D.-A., UNTERTHINER, T., AND HOCHREITER, S. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289* (2015).

[8] DEVLIN, J., CHANG, M.-W., LEE, K., AND TOUTANOVA, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[9] DU, P., SUN, Z., CHEN, H., CHO, J.-H., AND XU, S. Statistical Estimation of Malware Detection Metrics in the Absence of Ground Truth. *arXiv e-prints* (Sept. 2018), arXiv:1810.07260.

[10] FAMOYE, F. Restricted generalized poisson regression model. *Communications in Statistics-Theory and Methods 22*, 5 (1993), 1335–1354.

[11] HUANG, W., AND STOKES, J. W. Mtnet: a multi-task neural network for dynamic malware classification. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2016), Springer, pp. 399–418.

[12] HUANG, Y., WANG, W., AND WANG, L. Unconstrained multimodal multi-label learning. *IEEE Transactions on Multimedia 17*, 11 (2015), 1923–1935.

[13] IOFFE, S., AND SZEGEDY, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).

[14] JADERBERG, M., SIMONYAN, K., VEDALDI, A., AND ZISSERMAN, A. Deep structured output learning for unconstrained text recognition. *arXiv preprint arXiv:1412.5903* (2014).

[15] KINGMA, D. P., AND BA, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[16] KINGMA, D. P., MOHAMED, S., REZENDE, D. J., AND WELLING, M. Semi-supervised learning with deep generative models. In *Advances in neural information processing systems* (2014), pp. 3581–3589.

[17] KUMAR, A., AND DAUME III, H. Learning task grouping and overlap in multi-task learning. *arXiv preprint arXiv:1206.6417* (2012).

[18] MCCULLAGH, P. *Generalized linear models*. Routledge, 2018.

[19] RANJAN, R., PATEL, V. M., AND CHELLAPPA, R. Hyperface: A deep multi-task learning framework for face detection, landmark localization, pose estimation, and gender recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2017).

[20] RASMUS, A., BERGLUND, M., HONKALA, M., VALPOLA, H., AND RAIKO, T. Semi-supervised learning with ladder networks. In *Advances in Neural Information Processing Systems* (2015), pp. 3546–3554.

[21] RUDD, E., ROZSA, A., GUNTHER, M., AND BOULT, T. A survey of stealth malware: Attacks, mitigation measures, and steps toward autonomous open world solutions. *IEEE Communications Surveys & Tutorials 19*, 2 (2017), 1145–1172.

[22] RUDD, E. M., GÜNTHER, M., AND BOULT, T. E. Moon: A mixed objective optimization network for the recognition of facial attributes. In *European Conference on Computer Vision* (2016), Springer, pp. 19–35.

[23] RUDD, E. M., HARANG, R., AND SAXE, J. Meade: Towards a malicious email attachment detection engine. *arXiv preprint arXiv:1804.08162* (2018).

[24] RUDER12, S., BINGEL, J., AUGENSTEIN, I., AND SØGAARD, A. Sluice networks: Learning what to share between loosely related tasks. *stat 1050* (2017), 23.

[25] SAXE, J., AND BERLIN, K. Deep neural network based malware detection using two dimensional binary program features. In *Malicious and Unwanted Software (MALWARE), 2015 10th International Conference on* (2015), IEEE, pp. 11–20.

[26] SAXE, J., HARANG, R., WILD, C., AND SANDERS, H. A deep learning approach to fast, format-agnostic detection of malicious web content. *arXiv preprint arXiv:1804.05020* (2018).

[27] SRIVASTAVA, N., HINTON, G., KRIZHEVSKY, A., SUTSKEVER, I., AND SALAKHUTDINOV, R. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research 15*, 1 (2014), 1929–1958.

[28] VAPNIK, V., AND IZMAILOV, R. Learning using privileged information: similarity control and knowledge transfer. *Journal of machine learning research 16*, 2023-2049 (2015), 2.

[29] VAPNIK, V., AND VASHIST, A. A new learning paradigm: Learning using privileged information. *Neural networks 22*, 5-6 (2009), 544–557.

[30] WEISS, G. M. Mining with rarity: a unifying framework. *ACM Sigkdd Explorations Newsletter 6*, 1 (2004), 7–19.

[31] WU, F., WANG, Z., ZHANG, Z., YANG, Y., LUO, J., ZHU, W., AND ZHUANG, Y. Weakly semi-supervised deep learning for multi-label image annotation. *IEEE Trans. Big Data 1*, 3 (2015), 109–122.

[32] XU, C., TAO, D., AND XU, C. A survey on multi-view learning. *arXiv preprint arXiv:1304.5634* (2013).

A Dataset Statistics

A.1 Vendor Counts Distribution

To better characterize the distribution of the Vendor Counts auxiliary target for our Poisson loss experiment, we plot a histogram representing the distribution of the number of vendor convictions in Figure A.1. The x-axis depicts the number of vendors that identify a given sample as malicious, while the y-axis represents the number of samples for which that number of detections was observed in our training dataset (note the logarithmic scale). The statistics of the test and validation datasets are similar and not shown here due to space considerations.

We note that there is a peak at zero detections, accounting for the majority of the benign files. Most of the samples considered malicious by our labeling scheme have more than 20 individual detections out of 67 total vendors considered, with a peak around 57 detections.

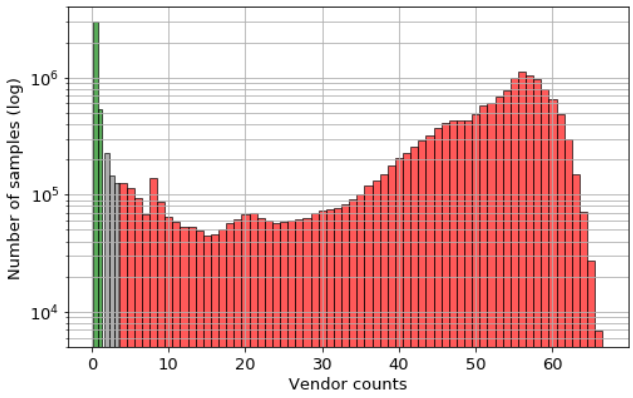


Figure A.1: Histogram of vendor detections per file. Files with zero or one detections (green bars) are considered benign under our labeling scheme, samples with two, three or four vendor detections (gray bars) are considered gray files, and files with more than four detections (red bars) are considered malicious.

A.2 Individual Vendor Responses

Table A.1 summarizes the number of samples identified as malware, benign and number of missing samples per vendor for the nine vendors used to compute the auxiliary per-vendor malware loss. In Figure A.2 we plot the pairwise similarity of the predictions between vendors. The value in the j, k position of the matrix is the fraction of samples for which the predictions of vendor j are equal to the predictions for vendor k. Even though the predictions by each vendor are created in a quasi independent manner, they tend to agree for most of the samples. The diagonal elements of the matrix

indicate the fraction of samples for which we have a classification by the vendor (fraction of non-missing values).

	Malware	Benign	None
v1	13,752,004 (69%)	6,110,180 (31%)	20,979 (<1%)
v2	14,751,413 (74%)	5,122,728 (26%)	9,022 (<1%)
v3	14,084,689 (71%)	5,713,116 (29%)	85,358 (<1%)
v4	14,438,043 (73%)	5,239,896 (26%)	205,224 (1 %)
v5	13,778,367 (69%)	5,922,859 (30%)	181,937 (1 %)
v6	15,065,196 (76%)	4,704,695 (24%)	113,272 (1 %)
v7	14,935,624 (75%)	4,927,436 (25%)	20,103 (<1%)
v8	12,704,512 (64%)	7,009,855 (35%)	168,796 (1 %)
v9	14,234,545 (72%)	5,613,604 (28%)	35,014 (<1%)

Table A.1: Individual vendor counts for files in the training set identified as malicious, benign, or missing value for the set of nine vendors used in the per-vendor malware loss 3.3.

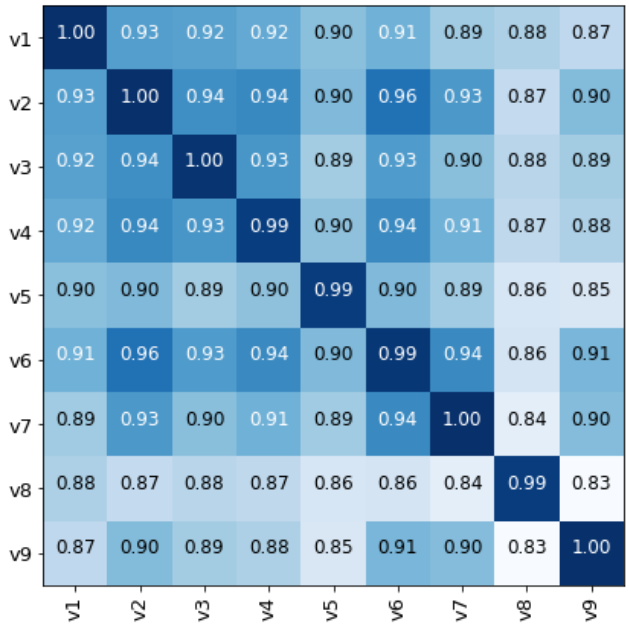


Figure A.2: Vendor predictions similarity matrix. Each entry in the matrix represents the percentage of samples that are the same for any two vendors. The elements in the diagonal of the matrix represent the percentage of the samples for predictions from the vendor are present (i.e., not missing). Note that diagonal values of less than 1.0 are due to missing labels (compare to the final column of table A.1), which we treat as disagreeing with *any* label.

A.3 Semantic Tags Distribution

In this section we analyze the distribution of the semantic tags over three sets of samples: i) samples in the training set; ii) samples in the test set; and iii) those which the baseline model classifies incorrectly but our model trained with all targets classifies correctly either as malicious or benign samples. The percentages in Table A.2 represent the number of samples in each set labeled with a given tag. The total number of samples in the test set for which the improved model makes correct conviction classification but the baseline model fails is 665,944. The binarization of the predictions for the baseline and the final model was done such that each would have a FPR of 10^{-3} in the test set. As shown below, those samples with the *adware* tag are the ones that most benefit from the addition of auxiliary losses during training, however we also see notable improvements on *packed* samples, *spyware*, and *droppers*.

	Train Set	Test Set	Improvement over baseline
adware	21 %	18%	41 %
crypto-miner	7 %	2%	1 %
downloader	25 %	18%	11 %
dropper	29 %	22%	17 %
file-infector	19 %	12%	9 %
flooder	1 %	1%	<1%
installer	7 %	1 %	5 %
packed	34 %	25%	19 %
ransomware	5 %	6%	1 %
spyware	40 %	25%	18 %

Table A.2: Tag statistics for three sets of interest: train set; test set; and the set of samples for which the full model classifies correctly but the baseline model fails.

B Gray Samples Evaluation

In Section 3.6 we observed that 2.5% of the samples in our training set, and 3.7% of the samples in our test set are considered gray samples by our labeling function. While training this is not necessarily an issue since we can assign a weight of zero for those samples in their malware/benign label as noted in Section 3.5. For the evaluation of the detection algorithms though, the performance on those becomes more relevant. To evaluate how our proposed detection model performs on those samples we re-scanned a random selection of 10,000 gray samples in the test set 5 months later than the original collection. From these, 5,000 were predicted by the model as benign and 5,000 as malicious. We expect, after this time-lag, that, with updated detection rules from the AV community, that samples originally labeled as "gray" that are effectively malicious will accrue additional detections and samples originally labeled as "gray" that are effectively benign will accrue fewer detections as vendors have re-written their rules to suppress false positives

and recognize false negatives. Thus, the gray sample labels will tend to converge to either malicious or benign under our 1-/5+ criterion. Out of these 10,000 rescans, we were able to label 5,653 gray samples: 3,877 (68.6%) as malicious and 1,776 (31.4%) as benign.

In Figure B.1 we plot the ROC curve for the predictions on the re-scanned samples for our final model trained with all targets, which achieves an AUC of 0.84. Even though the AUC is much lower than the one obtained on our test set, our model trained with knowledge from 5 months earlier (after the first collection) still correctly predicts more than 77% of the samples correctly. We measure this by binarizing the predictions using a threshold that would achieve an FPR of 10^{-3} on the original test set. Furthermore we note that the samples we are evaluating on in this case are more difficult or even ambiguous in nature, to the point that at the original collection time there was not consensus across the AV community.

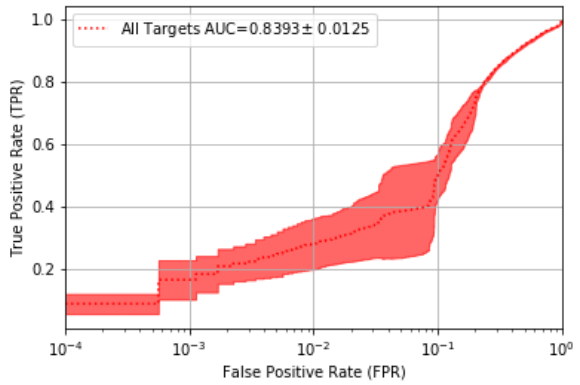


Figure B.1: Mean and standard deviation ROC curve over rescanned samples.

C Relative Improvements

In Table C.1 we present the relative percentage reduction both in true positive detection error and standard deviation with respect to the baseline model trained only using the malware/benign target for various values of false positive rates.

	FPR				
	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}
Poisson	38.05, 61.84	30.19, 30.61	28.68, 48.39	14.29, **85.71**	5.56, **97.14**
RG Poisson	0.00, -52.63	6.17, 16.33	4.41, 48.39	2.86, 57.14	2.78, 95.71
Vendors	47.12, 55.26	32.47, 51.02	18.38, 35.48	14.29, 42.86	0.00, 80.00
Tags	43.63, 64.47	32.47, **81.63**	8.09, 29.03	17.14, 42.86	5.56, 88.57
All Targets	**53.75**, **81.58**	**37.01**, 65.31	**42.65**, **87.10**	**20.00**, 57.14	**8.33**, 94.29

Table C.1: Relative percentage reductions in true positive detection error and standard deviation compared to the baseline model (displayed as *detection error reduction*, *standard deviation reduction*) at different false positive rates (FPRs) for the different experiments in Section 4. Results were evaluated over five different weight initializations and minibatch orderings. Best detection error reduction consistently occurred when using all auxiliary losses. Best results are shown in **bold**.

Why Do Adversarial Attacks Transfer? Explaining Transferability of Evasion and Poisoning Attacks

Ambra Demontis[†], Marco Melis[†], Maura Pintor[†], Matthew Jagielski[*], Battista Biggio[†,‡], Alina Oprea[*], Cristina Nita-Rotaru[*], and Fabio Roli[†,‡]

[†]Department of Electrical and Electronic Engineering, University of Cagliari, Italy
[‡]Pluribus One, Italy
[*]Northeastern University, Boston, MA, USA

Abstract

Transferability captures the ability of an attack against a machine-learning model to be effective against a different, potentially unknown, model. Empirical evidence for transferability has been shown in previous work, but the underlying reasons why an attack transfers or not are not yet well understood. In this paper, we present a comprehensive analysis aimed to investigate the transferability of both test-time evasion and training-time poisoning attacks. We provide a unifying optimization framework for evasion and poisoning attacks, and a formal definition of transferability of such attacks. We highlight two main factors contributing to attack transferability: the intrinsic *adversarial vulnerability* of the target model, and the *complexity* of the surrogate model used to optimize the attack. Based on these insights, we define three metrics that impact an attack's transferability. Interestingly, our results derived from theoretical analysis hold for both evasion and poisoning attacks, and are confirmed experimentally using a wide range of linear and non-linear classifiers and datasets.

1 Introduction

The wide adoption of machine learning (ML) and deep learning algorithms in many critical applications introduces strong incentives for motivated adversaries to manipulate the results and models generated by these algorithms. Attacks against machine learning systems can happen during multiple stages in the learning pipeline. For instance, in many settings training data is collected online and thus can not be fully trusted. In *poisoning availability attacks*, the attacker controls a certain amount of training data, thus influencing the trained model and ultimately the predictions at testing time on most points in testing set [4, 18, 20, 28–30, 34, 36, 41, 48]. *Poisoning integrity attacks* have the goal of modifying predictions on a few targeted points by manipulating the training process [20, 41]. On the other hand, *evasion attacks* involve small manipulations of testing data points that results in misprediction at testing time on those points [3, 8, 10, 14, 32, 38, 42, 45, 49].

Creating poisoning and evasion attack points is not a trivial task, particularly when many online services avoid disclosing information about their machine learning algorithms. As a result, attackers are forced to craft their attacks in *black-box* settings, against a surrogate model instead of the real model used by the service, hoping that the attack will be effective on the real model. The *transferability* property of an attack is satisfied when an attack developed for a particular machine learning model (i.e., a surrogate model) is also effective against the target model. Attack transferability was observed in early studies on adversarial examples [14, 42] and has gained a lot more interest in recent years with the advancement of machine learning cloud services. Previous work has reported empirical findings about the transferability of evasion attacks [3, 13, 14, 21, 26, 32, 33, 42, 43, 47] and, only recently, also on the transferability of poisoning integrity attacks [41]. In spite of these efforts, the question of *when and why do adversarial points transfer* remains largely unanswered.

In this paper we present the first comprehensive evaluation of transferability of evasion and poisoning availability attacks, understanding the factors contributing to transferability of both attacks. In particular, we consider attacks crafted with gradient-based optimization techniques (e.g., [4, 8, 23]), a popular and successful mechanism used to create attack data points. We unify for the first time evasion and poisoning attacks into an optimization framework that can be instantiated for a range of threat models and adversarial constraints. We provide a formal definition of transferability and show that, under linearization of the loss function computed under attack, several main factors impact transferability: the intrinsic *adversarial vulnerability* of the target model, the *complexity* of the surrogate model used to optimize the attacks, and its alignment with the target model. Furthermore, we derive a new poisoning attack for logistic regression, and perform a comprehensive evaluation of both evasion and poisoning attacks on multiple datasets, confirming our theoretical analysis.

In more detail, the contributions of our work are:

Optimization framework for evasion and poisoning attacks. We introduce a unifying framework based on gradient-

descent optimization that encompasses both evasion and poisoning attacks. Our framework supports threat models with different adversarial goals (integrity and availability), amount of knowledge available to the adversary (white-box and black-box), as well as different adversarial capabilities (causative or exploratory). Our framework generalizes existing attacks proposed by previous work for evasion [3, 8, 14, 23, 42] and poisoning [4, 18, 20, 24, 27, 48]. Under our framework, we derive a novel gradient-based poisoning availability attack against logistic regression. We remark here that poisoning attacks are more difficult to derive than evasion ones, as they require computing hypergradients from a bilevel optimization problem, to capture the dependency on how the machine-learning model changes while the training poisoning points are modified [4, 18, 20, 24, 27, 48].

Transferability definition and theoretical bound. We give a formal definition of transferability of evasion and poisoning attacks, and an upper bound on a transfer attack's success. This allows us to derive three metrics connected to *model complexity*. Our formal definition unveils that transferability depends on: (1) the size of input gradients of the target classifier; (2) how well the gradients of the surrogate and target models align; and (3) the variance of the loss landscape optimized to generate the attack points.

Comprehensive experimental evaluation of transferability. We consider a wide range of classifiers, including logistic regression, SVMs with both linear and RBF kernels, ridge regression, random forests, and deep neural networks (both feed-forward and convolutional neural networks), all with different hyperparameter settings to reflect different model complexities. We evaluate the transferability of our attacks on three datasets related to different applications: handwritten digit recognition (MNIST), Android malware detection (DREBIN), and face recognition (LFW). We confirm our theoretical analysis for both evasion and poisoning attacks.

Insights into transferability. We demonstrate that attack transferability depends strongly on the *complexity* of the target model, i.e., on its inherent vulnerability. This confirms that reducing the size of input gradients, e.g., via regularization, may allow us to learn more robust classifiers not only against evasion [22, 35, 39, 44] but also against poisoning availability attacks. Second, transferability is also impacted by the surrogate model's alignment with the target model. Surrogates with better alignments to their targets (in terms of the angle between their gradients) are more successful at transferring the attack points. Third, surrogate loss functions that are stabler and have lower variance tend to facilitate gradient-based optimization attacks to find better local optima (see Figure 1). As less complex models exhibit a lower variance of their loss function, they typically result in better surrogates.

Organization. We discuss background on threat modeling against machine learning in Section 2. We introduce our unifying optimization framework for evasion and poisoning attacks, as well as the poisoning attack for logistic regression in Section 3. We then formally define transferability for both evasion and poisoning attacks, and show its approximate connection with the input gradients used to craft the corresponding attack samples (Section 4). Experiments are reported in Section 5, highlighting connections among regularization hyperparameters, the size of input gradients, and transferability of attacks, on different case studies involving handwritten digit recognition, Android malware detection, and face recognition. We discuss related work in Section 6 and conclude in Section 7.

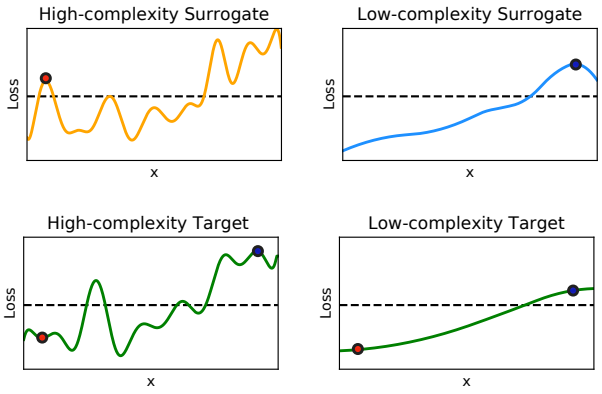


Figure 1: Conceptual representation of transferability. We show the loss function of the attack objective as a function of a single feature x. The top row includes 2 surrogate models (*high* and *low* complexity), while the bottom row includes both models as targets. The adversarial samples are represented as red dots for the high-complexity surrogate and as blue dots for the low-complexity surrogate. If the adversarial sample loss is below a certain threshold (i.e., the black horizontal line), the point is correctly classified, otherwise it is misclassified. The adversarial point computed against the high-complexity model (top left) lays in a local optimum due to the irregularity of the objective. This point is not effective even against the same classifier trained on a different dataset (bottom left) due to the variance of the high-complexity classifier. The adversarial point computed against the low complexity model (top right), instead, succeeds against both low and high-complexity targets (left and right bottom, respectively).

2 Background and Threat Model

Supervised learning includes: (1) a training phase in which training data is given as input to a learning algorithm, resulting in a trained ML model; (2) a testing phase in which the model is applied to new data and a prediction is generated. In this paper, we consider a range of adversarial models against machine learning classifiers at both training and testing time. Attackers are defined by: (*i*) their goal or objective in attacking the system; (*ii*) their knowledge of the system; (*iii*) their capabilities in influencing the system through manipulation

of the input data. Before we detail each of these, we introduce our notation, and point out that the threat model and attacks considered in this work are suited to binary classification, but can be extended to multi-class settings.

Notation. We denote the sample and label spaces with $\mathcal{X}$ and $\mathcal{Y} \in \{-1,+1\}$, respectively, and the training data with $\mathcal{D} = (\mathbf{x}_i, y_i)_{i=1}^n$, where n is the training set size. We use $L(\mathcal{D}, \mathbf{w})$ to denote the *loss* incurred by classifier $f : \mathcal{X} \mapsto \mathcal{Y}$ (parameterized by $\mathbf{w}$) on $\mathcal{D}$. Typically, this is computed by averaging a loss function $\ell(y, \mathbf{x}, \mathbf{w})$ computed on each data point, i.e., $L(\mathcal{D}, \mathbf{w}) = \frac{1}{n} \sum_{i=1}^n \ell(y_i, \mathbf{x}_i, \mathbf{w})$. We assume that the classifier f is learned by minimizing an objective function $\mathcal{L}(\mathcal{D}, \mathbf{w})$ on the training data. Typically, this is an estimate of the generalization error, obtained by the sum of the empirical loss L on training data $\mathcal{D}$ and a regularization term.

2.1 Threat Model: Attacker's Goal

We define the attacker's goal based on the desired security violation. In particular, the attacker may aim to cause either an *integrity* violation, to evade detection without compromising normal system operation; or an *availability* violation, to compromise the normal system functionalities available to legitimate users.

2.2 Threat Model: Attacker's Knowledge

We characterize the attacker's knowledge κ as a tuple in an abstract knowledge space $\mathcal{K}$ consisting of four main dimensions, respectively representing knowledge of: (*k.i*) the training data $\mathcal{D}$; (*k.ii*) the feature set $\mathcal{X}$; (*k.iii*) the learning algorithm f, along with the objective function $\mathcal{L}$ minimized during training; and (*k.iv*) the parameters $\mathbf{w}$ learned after training the model. This categorization enables the definition of many different kinds of attacks, ranging from *white-box* attacks with full knowledge of the target classifier to *black-box* attacks in which the attacker has limited information about the target system.

White-Box Attacks. We assume here that the attacker has full knowledge of the target classifier, i.e., $\kappa = (\mathcal{D}, \mathcal{X}, f, \mathbf{w})$. This setting allows one to perform a worst-case evaluation of the security of machine-learning algorithms, providing empirical upper bounds on the performance degradation that may be incurred by the system under attack.

Black-Box Attacks. We assume here that the input feature representation $\mathcal{X}$ is known. For images, this means that we do consider pixels as the input features, consistently with other recent work on black-box attacks against machine learning [32, 33]. At the same time, the training data $\mathcal{D}$ and the type of classifier f are not known to the attacker. We consider the most realistic attack model in which the attacker does not have querying access to the classifier.

The attacker can collect a surrogate dataset $\hat{\mathcal{D}}$, ideally sampled from the same underlying data distribution as $\mathcal{D}$, and train a *surrogate model* $\hat{f}$ on such data to approximate the target function f. Then, the attacker can craft the attacks against $\hat{f}$, and then check whether they successfully *transfer* to the target classifier f. By denoting limited knowledge of a given component with the *hat* symbol, such black-box attacks can be denoted with $\hat{\kappa} = (\hat{\mathcal{D}}, \mathcal{X}, \hat{f}, \hat{\mathbf{w}})$.

2.3 Threat Model: Attacker's Capability

This attack characteristic defines how the attacker can influence the system, and how data can be manipulated based on application-specific constraints. If the attacker can manipulate both training and test data, the attack is said to be *causative*. It is instead referred to as *exploratory*, if the attacker can only manipulate test data. These scenarios are more commonly known as *poisoning* [4, 18, 24, 27, 48] and *evasion* [3, 8, 14, 42].

Another aspect related to the attacker's capability depends on the presence of application-specific constraints on data manipulation; e.g., to evade malware detection, malicious code has to be modified without compromising its intrusive functionality. This may be done against systems leveraging static code analysis, by injecting instructions that will never be executed [11, 15, 45]. These constraints can be generally accounted for in the definition of the optimal attack strategy by assuming that the initial attack sample $\mathbf{x}$ can only be modified according to a space of possible modifications $\Phi(\mathbf{x})$.

3 Optimization Framework for Gradient-based Attacks

We introduce here a general optimization framework that encompasses both evasion and poisoning attacks. Gradient-based attacks have been considered for evasion (e.g., [3, 8, 14, 23, 42]) and poisoning (e.g., [4, 18, 24, 27]). Our optimization framework not only unifies existing evasion and poisoning attacks, but it also enables the design of new attacks. After defining our general formulation, we instantiate it for evasion and poisoning attacks, and use it to derive a new poisoning availability attack for logistic regression.

3.1 Gradient-based Optimization Algorithm

Given the attacker's knowledge $\kappa \in \mathcal{K}$ and an attack sample $\mathbf{x}' \in \Phi(\mathbf{x})$ along with its label y, the attacker's goal can be defined in terms of an objective function $\mathcal{A}(\mathbf{x}', y, \kappa) \in \mathbb{R}$ (e.g., a loss function) which measures how effective the attack sample $\mathbf{x}'$ is. The optimal attack strategy can be thus given as:

$$\mathbf{x}^\star \in \arg\max_{\mathbf{x}' \in \Phi(\mathbf{x})} \mathcal{A}(\mathbf{x}', y, \kappa)\,. \tag{1}$$

Note that, for the sake of clarity, we consider here the optimization of a single attack sample, but this formulation can be easily extended to account for multiple attack points. In

Algorithm 1 Gradient-based Evasion and Poisoning Attacks

Input: $\mathbf{x}, y$: the input sample and its label; $\mathcal{A}(\mathbf{x}, y, \kappa)$: the attacker's objective; $\kappa = (\mathcal{D}, \mathcal{X}, f, \mathbf{w})$: the attacker's knowledge parameter vector; $\Phi(\mathbf{x})$: the feasible set of manipulations that can be made on $\mathbf{x}$; $t > 0$: a small number.

Output: $\mathbf{x}'$: the adversarial example.

1: Initialize the attack sample: $\mathbf{x}' \leftarrow \mathbf{x}$
2: **repeat**
3: Store attack from previous iteration: $\mathbf{x} \leftarrow \mathbf{x}'$
4: Update step: $\mathbf{x}' \leftarrow \Pi_\Phi\left(\mathbf{x} + \eta \nabla_{\mathbf{x}} \mathcal{A}(\mathbf{x}, y, \kappa)\right)$, where the step size η is chosen with line search (bisection method), and Π_Φ ensures projection on the feasible domain Φ.
5: **until** $|\mathcal{A}(\mathbf{x}', y, \kappa) - \mathcal{A}(\mathbf{x}, y, \kappa)| \leq t$
6: **return** $\mathbf{x}'$

particular, as in the case of poisoning attacks, the attacker can maximize the objective by iteratively optimizing one attack point at a time [5, 48].

Attack Algorithm. Algorithm 1 provides a general projected gradient-ascent algorithm that can be used to solve the aforementioned problem for both evasion and poisoning attacks. It iteratively updates the attack sample along the gradient of the objective function, ensuring the resulting point to be within the feasible domain through a projection operator Π_Φ. The gradient step size η is determined in each update step using a line-search algorithm based on the bisection method, which solves $\max_\eta \mathcal{A}(\mathbf{x}'(\eta), y, \kappa)$, with $\mathbf{x}'(\eta) = \Pi_\Phi\left(\mathbf{x} + \eta \nabla_{\mathbf{x}} \mathcal{A}(\mathbf{x}, y, \kappa)\right)$. For the line search, in our experiments we consider a maximum of 20 iterations. This allows us to reduce the overall number of iterations required by Algorithm 1 to reach a local or global optimum. We also set the maximum number of iterations for Algorithm 1 to 1,000, but convergence (Algorithm 1, line 5) is typically reached only after a hundred iterations.

We finally remark that non-differentiable learning algorithms, like decision trees and random forests, can be attacked with more complex strategies [17, 19] or using gradient-based optimization against a differentiable surrogate learner [31, 37].

3.2 Evasion Attacks

In evasion attacks, the attacker manipulates test samples to have them misclassified, i.e., to evade detection by a learning algorithm. For white-box evasion, the optimization problem given in Eq. (1) can be rewritten as:

$$\max_{\mathbf{x}'} \quad \ell(y, \mathbf{x}', \mathbf{w}), \tag{2}$$

$$\text{s.t.} \quad \|\mathbf{x}' - \mathbf{x}\|_p \leq \varepsilon, \tag{3}$$

$$\mathbf{x}_{\text{lb}} \preceq \mathbf{x}' \preceq \mathbf{x}_{\text{ub}}, \tag{4}$$

where $\|\mathbf{v}\|_p$ is the ℓ_p norm of $\mathbf{v}$, and we assume that the classifier parameters $\mathbf{w}$ are known. For the black-box case, it suffices to use the parameters $\hat{\mathbf{w}}$ of the surrogate classifier $\hat{f}$. In this work we consider $\ell(y, \mathbf{x}', \mathbf{w}) = -y f(\mathbf{x}')$, as in [3].

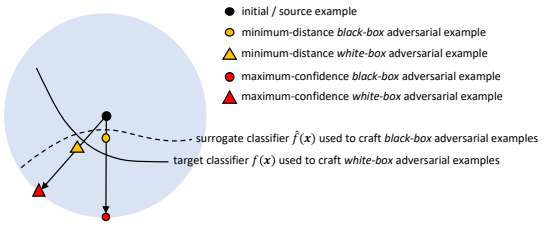


Figure 2: Conceptual representation of maximum-confidence evasion attacks (within an ℓ_2 ball of radius ε) vs. minimum-distance adversarial examples. Maximum-confidence attacks tend to transfer better as they are misclassified with higher confidence (though requiring more modifications).

The intuition here is that the attacker maximizes the loss on the adversarial sample with the original class, to cause misclassification to the opposite class. The manipulation constraints $\Phi(\mathbf{x})$ are given in terms of: (*i*) a distance constraint $\|\mathbf{x}' - \mathbf{x}\|_p \leq \varepsilon$, which sets a bound on the maximum input perturbation between $\mathbf{x}$ (i.e., the input sample) and the corresponding modified adversarial example $\mathbf{x}'$; and (*ii*) a box constraint $\mathbf{x}_{\text{lb}} \preceq \mathbf{x}' \preceq \mathbf{x}_{\text{ub}}$ (where $\mathbf{u} \preceq \mathbf{v}$ means that each element of $\mathbf{u}$ has to be not greater than the corresponding element in $\mathbf{v}$), which bounds the values of the attack sample $\mathbf{x}'$.

For images, the former constraint is used to implement either *dense* or *sparse* evasion attacks [12, 25, 37]. Normally, the ℓ_2 and the ℓ_∞ distances between pixel values are used to cause an indistinguishable image blurring effect (by slightly manipulating all pixels). Conversely, the ℓ_1 distance corresponds to a sparse attack in which only few pixels are significantly manipulated, yielding a salt-and-pepper noise effect on the image [12, 37]. The box constraint can be used to bound each pixel value between 0 and 255, or to ensure manipulation of only a specific region of the image. For example, if some pixels should not be manipulated, one can set the corresponding values of $\mathbf{x}_{\text{lb}}$ and $\mathbf{x}_{\text{ub}}$ equal to those of $\mathbf{x}$.

Maximum-confidence vs. minimum-distance evasion. Our formulation of evasion attacks aims to produce adversarial examples that are misclassified with *maximum confidence* by the classifier, within the given space of feasible modifications. This is substantially different from crafting minimum-distance adversarial examples, as formulated in [42] and in follow-up work (e.g., [33]). This difference is conceptually depicted in Fig. 2. In particular, in terms of transferability, it is now widely acknowledged that higher-confidence attacks have better chances of successfully transfering to the target classifier (and even of bypassing countermeasures based on gradient masking) [2, 8, 13]. For this reason, in this work we consider evasion attacks that aim to craft adversarial examples misclassified with *maximum* confidence.

Initialization. There is another factor known to improve trans-

ferability of evasion attacks, as well as their effectiveness in the white-box setting. It consists of running the attack starting from different initialization points to mitigate the problem of getting stuck in poor local optima [3, 13, 50]. In addition to starting the gradient ascent from the initial point $\mathbf{x}$, for non-linear classifiers we also consider starting the gradient ascent from the projection of a randomly-chosen point of the opposite class onto the feasible domain. This double-initialization strategy helps finding better local optima, through the identification of more promising paths towards evasion [13, 47, 50].

3.3 Poisoning Availability Attacks

Poisoning attacks consist of manipulating training data (mainly by injecting adversarial points into the training set) to either favor intrusions without affecting normal system operation, or to purposely compromise normal system operation to cause a denial of service. The former are referred to as poisoning integrity attacks, while the latter are known as poisoning availability attacks [5, 48]. Recent work has mostly addressed transferability of poisoning integrity attacks [41], including backdoor attacks [9, 16]. In this work we focus on poisoning availability attacks, as their transferability properties have not yet been widely investigated. Crafting transferable poisoning availability attacks is much more challenging than crafting transferable poisoning integrity attacks, as the latter have a much more modest goal (modifying prediction on a small set of targeted points).

As for the evasion case, we formulate poisoning in a white-box setting, given that the extension to black-box attacks is immediate through the use of surrogate learners. Poisoning is formulated as a bilevel optimization problem in which the outer optimization maximizes the attacker's objective $\mathcal{A}$ (typically, a loss function L computed on untainted data), while the inner optimization amounts to learning the classifier on the poisoned training data [4, 24, 48]. This can be made explicit by rewriting Eq. (1) as:

$$\max_{\mathbf{x}'} \quad L(\mathcal{D}_{\text{val}}, \mathbf{w}^\star) = \sum_{j=1}^{m} \ell(y_j, \mathbf{x}_j, \mathbf{w}^\star) \tag{5}$$

$$\text{s.t.} \quad \mathbf{w}^\star \in \arg\min_{\mathbf{w}} \mathcal{L}(\mathcal{D}_{\text{tr}} \cup (\mathbf{x}', y), \mathbf{w}) \tag{6}$$

where $\mathcal{D}_{\text{tr}}$ and $\mathcal{D}_{\text{val}}$ are the training and validation datasets available to the attacker. The former, along with the poisoning point $\mathbf{x}'$, is used to train the learner on poisoned data, while the latter is used to evaluate its performance on untainted data, through the loss function $L(\mathcal{D}_{\text{val}}, \mathbf{w}^\star)$. Notably, the objective function implicitly depends on $\mathbf{x}'$ through the parameters $\mathbf{w}^\star$ of the poisoned classifier.

The attacker's capability is limited by assuming that the attacker can inject only a small fraction α of poisoning points into the training set. Thus, the attacker solves an optimization problem involving a set of poisoned data points (αn) added to the training data.

Poisoning points can be optimized via gradient-ascent procedures, as shown in Algorithm 1. The main challenge is to compute the gradient of the attacker's objective (i.e., the validation loss) with respect to each poisoning point. In fact, this gradient has to capture the implicit dependency of the optimal parameter vector $\mathbf{w}^\star$ (learned after training) on the poisoning point being optimized, as the classification function changes while this point is updated. Provided that the attacker function is differentiable w.r.t. $\mathbf{w}$ and $\mathbf{x}$, the required gradient can be computed using the chain rule [4, 5, 24, 27, 48]:

$$\nabla_{\mathbf{x}} \mathcal{A} = \nabla_{\mathbf{x}} L + \frac{\partial \mathbf{w}}{\partial \mathbf{x}}^\top \nabla_{\mathbf{w}} L, \tag{7}$$

where the term $\frac{\partial \mathbf{w}}{\partial \mathbf{x}}$ captures the implicit dependency of the parameters $\mathbf{w}$ on the poisoning point $\mathbf{x}$. Under some regularity conditions, this derivative can be computed by replacing the inner optimization problem with its stationarity (Karush-Kuhn-Tucker, KKT) conditions, i.e., with its implicit equation $\nabla_{\mathbf{w}} \mathcal{L}(\mathcal{D}_{\text{tr}} \cup (\mathbf{x}', y), \mathbf{w}) = \mathbf{0}$ [24, 27].[1] By differentiating this expression w.r.t. the poisoning point $\mathbf{x}$, one yields:

$$\nabla_{\mathbf{x}} \nabla_{\mathbf{w}} \mathcal{L} + \frac{\partial \mathbf{w}}{\partial \mathbf{x}}^\top \nabla^2_{\mathbf{w}} \mathcal{L} = \mathbf{0}. \tag{8}$$

Solving for $\frac{\partial \mathbf{w}}{\partial \mathbf{x}}$, we obtain $\frac{\partial \mathbf{w}}{\partial \mathbf{x}}^\top = -(\nabla_{\mathbf{x}} \nabla_{\mathbf{w}} \mathcal{L})(\nabla^2_{\mathbf{w}} \mathcal{L})^{-1}$, which can be substituted in Eq. (7) to obtain the required gradient:

$$\nabla_{\mathbf{x}} \mathcal{A} = \nabla_{\mathbf{x}} L - (\nabla_{\mathbf{x}_c} \nabla_{\mathbf{w}} \mathcal{L})(\nabla^2_{\mathbf{w}} \mathcal{L})^{-1} \nabla_{\mathbf{w}} L. \tag{9}$$

Gradients for SVM. Poisoning attacks against SVMs were first proposed in [4]. Here, we report a simplified expression for SVM poisoning, with $\mathcal{L}$ corresponding to the dual SVM learning problem, and L to the hinge loss (in the outer optimization):

$$\nabla_{\mathbf{x}_c} \mathcal{A} = -\alpha_c \frac{\partial \mathbf{k}_{kc}}{\partial \mathbf{x}_c} \mathbf{y}_k + \alpha_c \begin{bmatrix} \frac{\partial \mathbf{k}_{sc}}{\partial \mathbf{x}_c} & 0 \end{bmatrix} \begin{bmatrix} \mathbf{K}_{ss} & \mathbf{1} \\ \mathbf{1}^\top & 0 \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{K}_{sk} \\ \mathbf{1}^\top \end{bmatrix} \mathbf{y}_k \,. \tag{10}$$

We use c, s and k here to respectively index the attack point, the support vectors, and the validation points for which $\ell(y, \mathbf{x}, \mathbf{w}) > 0$ (corresponding to a non-null derivative of the hinge loss). The coefficient α_c is the dual variable assigned to the poisoning point by the learning algorithm, and $\mathbf{k}$ and $\mathbf{K}$ contain kernel values between the corresponding indexed sets of points.

Gradients for Logistic Regression. Logistic regression is a linear classifier that estimates the probability of the positive class using the sigmoid function. A poisoning attack against logistic regression has been derived in [24], but maximizing a different outer objective and not directly the validation loss.

[1] More rigorously, we should write the KKT conditions in this case as $\nabla_{\mathbf{w}} \mathcal{L}(\mathcal{D}_{\text{tr}} \cup (\mathbf{x}', y), \mathbf{w}) \in \mathbf{0}$, as the solution may not be unique.

One of our contributions is to compute gradients for logistic regression under our optimization framework. Using logistic loss as the attacker's loss, the poisoning gradient for logistic regression can be computed as:

$$\nabla_{\mathbf{x}_c}\mathcal{A} = -\begin{bmatrix}\nabla_{\mathbf{x}_c}\nabla_\theta\mathcal{L}\\ C\,z_c\,\theta\end{bmatrix}^\top \begin{bmatrix}\nabla^2_\theta\mathcal{L} & \mathbf{X}\,\mathbf{z}\,C\\ C\,\mathbf{z}\,\mathbf{X} & C\sum_i^n z_i\end{bmatrix}^{-1}\begin{bmatrix}\mathbf{X}(\mathbf{y}\circ\sigma-\mathbf{y})\\ \mathbf{y}^\top(\sigma-1)\end{bmatrix}C,$$

where θ are the classifier weights (bias excluded), $\circ$ is the element-wise product, $\mathbf{z}$ is equal to $\sigma(1-\sigma)$, σ is the sigmoid of the signed discriminant function (each element of that vector is therefore: $\sigma_i = \frac{1}{1+\exp(-y_i f_i)}$ with $f_i = \mathbf{x}_i\theta + b$), and:

$$\nabla^2_\theta\mathcal{L} = C\sum_i^n \mathbf{x_i} z_i \mathbf{x}_i^\top + \mathbb{I}, \tag{11}$$

$$\nabla_{\mathbf{x}_c}\nabla_\theta\mathcal{L} = C(\mathbb{I}\circ(y_c\sigma_c - y_c) + z_c\theta\mathbf{x}_c^\top) \tag{12}$$

In the above equations, $\mathbb{I}$ is the identity matrix.

4 Transferability Definition and Metrics

We discuss here an intriguing connection among transferability of both evasion and poisoning attacks, input gradients and model complexity, and highlight the factors impacting transferability between a surrogate and a target model. Model complexity is a measure of the capacity of a learning algorithm to fit the training data. It is typically penalized to avoid overfitting by reducing either the number of classifier parameters to be learnt or their size (e.g., via regularization) [6]. Given that complexity is essentially controlled by the hyperparameters of a given learning algorithm (e.g., the number of neurons in the hidden layers of a neural network, or the regularization hyperparameter C of an SVM), *only models that are trained using the same learning algorithm should be compared in terms of complexity*. As we will see, this is an important point to correctly interpret the results of our analysis. For notational convenience, we denote in the following the attack points as $\mathbf{x}^\star = \mathbf{x} + \hat{\delta}$, where $\mathbf{x}$ is the initial point and $\hat{\delta}$ the adversarial perturbation optimized by the attack algorithm against the *surrogate* classifier, for both evasion and poisoning attacks. We start by formally defining transferability for evasion attacks, and then discuss how this definition and the corresponding metrics can be generalized to poisoning.

Transferability of Evasion Attacks. Given an evasion attack point $\mathbf{x}^\star$, crafted against a surrogate learner (parameterized by $\hat{\mathbf{w}}$), we define its *transferability* as the loss attained by the target classifier f (parameterized by $\mathbf{w}$) on that point, i.e., $T = \ell(y, \mathbf{x}+\hat{\delta}, \mathbf{w})$. This can be simplified through a linear approximation of the loss function as:

$$T = \ell(y,\mathbf{x}+\hat{\delta},\mathbf{w}) \approxeq \ell(y,\mathbf{x},\mathbf{w}) + \hat{\delta}^\top\nabla_{\mathbf{x}}\ell(y,\mathbf{x},\mathbf{w}). \tag{13}$$

This approximation may not only hold for sufficiently-small input perturbations. It may also hold for larger perturbations if the classification function is linear or has a small curvature (e.g., if it is strongly regularized). It is not difficult to see that, for any given point $\mathbf{x}, y$, the evasion problem in Eqs. (2)-(3) (without considering the feature bounds in Eq. 4) can be rewritten as:

$$\hat{\delta} \in \arg\max_{\|\delta\|_p\le\varepsilon} \ell(y,\mathbf{x}+\delta,\hat{\mathbf{w}}). \tag{14}$$

Under the same linear approximation, this corresponds to the maximization of an inner product over an ε-sized ball:

$$\max_{\|\delta\|_p\le\varepsilon} \delta^\top\nabla_{\mathbf{x}}\ell(y,\mathbf{x},\hat{\mathbf{w}}) = \varepsilon\|\nabla_{\mathbf{x}}\ell(y,\mathbf{x},\hat{\mathbf{w}})\|_q, \tag{15}$$

where ℓ_q is the dual norm of ℓ_p.

The above problem is maximized as follows:

1. For $p=2$, the maximum is $\hat{\delta} = \varepsilon\frac{\nabla_{\mathbf{x}}\ell(y,\mathbf{x},\hat{\mathbf{w}})}{\|\nabla_{\mathbf{x}}\ell(y,\mathbf{x},\hat{\mathbf{w}})\|_2}$;
2. For $p=\infty$, the maximum is $\hat{\delta} \in \varepsilon\cdot\text{sign}\{\nabla_{\mathbf{x}}\ell(y,\mathbf{x},\hat{\mathbf{w}})\}$;
3. For $p=1$, the maximum is achieved by setting the values of $\hat{\delta}$ that correspond to the maximum absolute values of $\nabla_{\mathbf{x}}\ell(y,\mathbf{x},\hat{\mathbf{w}})$ to their sign, i.e., ±1, and 0 otherwise.

Substituting the optimal value of $\hat{\delta}$ into Eq. (13), we can compute the loss increment $\Delta\ell = \hat{\delta}^\top\nabla_{\mathbf{x}}\ell(y,\mathbf{x},\mathbf{w})$ under a transfer attack in closed form; e.g., for $p=2$, it is given as:

$$\Delta\ell = \varepsilon\frac{\nabla_{\mathbf{x}}\hat{\ell}^\top}{\|\nabla_{\mathbf{x}}\hat{\ell}\|_2}\nabla_{\mathbf{x}}\ell \le \varepsilon\|\nabla_{\mathbf{x}}\ell\|_2, \tag{16}$$

where, for compactness, we use $\hat{\ell} = \ell(y,\mathbf{x},\hat{\mathbf{w}})$ and $\ell = \ell(y,\mathbf{x},\mathbf{w})$. In this equation, the left-hand side is the increase in the loss function in the black-box case, while the right-hand side corresponds to the white-box case. The upper bound is obtained when the surrogate classifier $\hat{\mathbf{w}}$ is equal to the target $\mathbf{w}$ (white-box attacks). Similar results hold for $p=1$ and $p=\infty$ (using the dual norm in the right-hand side).

Intriguing Connections and Transferability Metrics. The above findings reveal some interesting connections among transferability of attacks, model complexity (controlled by the classifier hyperparameters) and input gradients, as detailed below, and allow us to define simple and computationally-efficient transferability metrics.

(1) Size of Input Gradients. The first interesting observation is that transferability depends on the size of the gradient of the loss ℓ computed using the *target* classifier, regardless of the surrogate: the larger this gradient is, the larger the attack impact may be. This is inferred from the upper bound in Eq. (16). We define the corresponding metric $S(\mathbf{x},y)$ as:

$$S(\mathbf{x},y) = \|\nabla_{\mathbf{x}}\ell(y,\mathbf{x},\mathbf{w})\|_q, \tag{17}$$

where q is the dual of the perturbation norm.

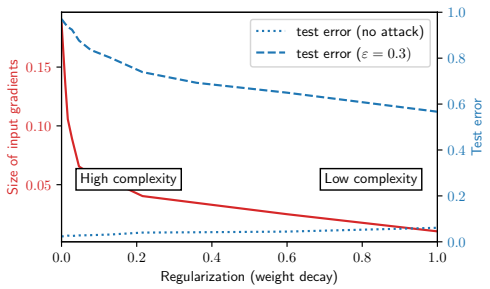


Figure 3: Size of input gradients (averaged on the test set) and test error (in the absence and presence of evasion attacks) against regularization (controlled via weight decay) for a neural network trained on MNIST89 (see Sect. 5.1.1). Note how the size of input gradients and the test error under attack decrease as regularization (complexity) increases (decreases).

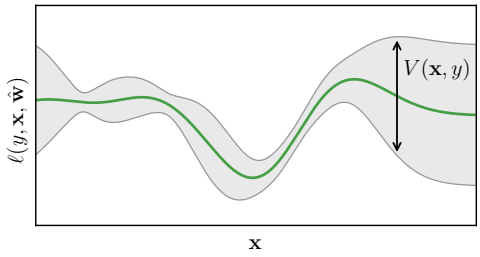


Figure 4: Conceptual representation of the variability of the loss landscape. The green line represents the expected loss with respect to different training sets used to learn the surrogate model, while the gray area represents the variance of the loss landscape. If the variance is too large, local optima may change, and the attack may not successfully transfer.

The size of the input gradient also depends on the complexity of the given model, controlled, e.g., by its regularization hyperparameter. Less complex, strongly-regularized classifiers tend to have smaller input gradients, i.e., they learn smoother functions that are more robust to attacks, and vice-versa. Notably, this holds for both evasion and poisoning attacks (e.g., the poisoning gradient in Eq. 10 is proportional to α_c, which is larger when the model is weakly regularized). In Fig. 3 we report an example showing how increasing regularization (i.e., decreasing complexity) for a neural network trained on MNIST89 (see Sect. 5.1.1), by controlling its *weight decay*, reduces the average size of its input gradients, improving adversarial robustness to evasion. It is however worth remarking that, since complexity is a model-dependent characteristic, the size of input gradients cannot be directly compared across different learning algorithms; e.g., if a linear SVM exhibits larger input gradients than a neural network, we cannot conclude that the former will be more vulnerable.

Another interesting observation is that, if a classifier has large input gradients (e.g., due to high-dimensionality of the input space and low level of regularization), for an attack to succeed it may suffice to apply only tiny, *imperceptible* perturbations. As we will see in the experimental section, this explains why adversarial examples against deep neural networks can often only be slightly perturbed to mislead detection, while when attacking less complex classifiers in low dimensions, modifications become more evident.

(2) Gradient Alignment. The second relevant impact factor on transferability is based on the alignment of the input gradients of the loss function computed using the target and the surrogate learners. If we compare the increase in the loss function in the black-box case (the left-hand side of Eq. 16) against that corresponding to white-box attacks (the right-hand side), we find that the relative increase in loss, at least for ℓ_2 perturbations, is given by the following value:

$$R(\mathbf{x}, y) = \frac{\nabla_{\mathbf{x}} \hat{\ell}^{\top} \nabla_{\mathbf{x}} \ell}{\|\nabla_{\mathbf{x}} \hat{\ell}\|_2 \|\nabla_{\mathbf{x}} \ell\|_2} . \tag{18}$$

Interestingly, this is exactly the cosine of the angle between the gradient of the loss of the surrogate and that of the target classifier. This is a novel finding which explains why the cosine angle metric between the target and surrogate gradients can well characterize the transferability of attacks, confirming empirical results from previous work [21]. For other kinds of perturbation, this definition slightly changes, but gradient alignment can be similarly evaluated. Differently from the gradient size S, gradient alignment is a pairwise metric, allowing comparisons across different surrogate models; e.g., if a surrogate SVM is better aligned with the target model than another surrogate, we can expect that attacks targeting the surrogate SVM will transfer better.

(3) Variability of the Loss Landscape. We define here another useful metric to characterize attack transferability. The idea is to measure the variability of the loss function $\hat{\ell}$ when the training set used to learn the surrogate model changes, even though it is sampled from the same underlying distribution. The reason is that the loss $\hat{\ell}$ is exactly the objective function $\mathcal{A}$ optimized by the attacker to craft evasion attacks (Eq. 1). Accordingly, if this loss landscape changes dramatically even when simply resampling the surrogate training set (which may happen, e.g., for surrogate models exhibiting a large error variance, like neural networks and decision trees), it is very likely that the local optima of the corresponding optimization problem will change, and this may in turn imply that the attacks will not transfer correctly to the target learner.

We define the variability of the loss landscape simply as the *variance* of the loss, estimated at a given attack point $\mathbf{x}, y$:

$$V(\mathbf{x}, y) = \mathbb{E}_{\mathcal{D}}\{\ell(y, \mathbf{x}, \hat{\mathbf{w}})^2\} - \mathbb{E}_{\mathcal{D}}\{\ell(y, \mathbf{x}, \hat{\mathbf{w}})\}^2 , \tag{19}$$

where $\mathbb{E}_{\mathcal{D}}$ is the expectation taken with respect to different (surrogate) training sets. This is very similar to what is typically done to estimate the variance of classifiers' predictions. This notion is clarified also in Fig. 4. As for the size of input gradients S, also the loss variance V should only be compared across models trained with the same learning algorithm.

The transferability metrics S, R and V defined so far depend on the initial attack point $\mathbf{x}$ and its label y. In our experiments, we will compute their mean values by averaging on different initial attack points.

Transferability of Poisoning Attacks. For poisoning attacks, we can essentially follow the same derivation discussed before. Instead of defining transferability in terms of the loss attained on the modified test point, we define it in terms of the validation loss attained by the target classifier under the influence of the poisoning points. This loss function can be linearized as done in the previous case, yielding: $T \approxeq L(\mathcal{D},\mathbf{w}) + \hat{\delta}^\top \nabla_{\mathbf{x}} L(\mathcal{D},\mathbf{w})$, where $\mathcal{D}$ are the untainted validation points, and $\hat{\delta}$ is the perturbation applied to the initial poisoning point $\mathbf{x}$ against the surrogate classifier. Recall that L depends on the poisoning point through the classifier parameters $\mathbf{w}$, and that the gradient $\nabla_{\mathbf{x}} L(\mathcal{D},\mathbf{w})$ here is equivalent to the generic one reported in Eq. (9). It is then clear that the perturbation $\hat{\delta}$ maximizes the (linearized) loss when it is best aligned with its derivative $\nabla_{\mathbf{x}} L(\mathcal{D},\mathbf{w})$, according to the constraint used, as in the previous case. The three transferability metrics defined before can also be used for poisoning attacks provided that we simply replace the evasion loss $\ell(y,\mathbf{x},\mathbf{w})$ with the validation loss $L(\mathcal{D},\mathbf{w})$.

5 Experimental Analysis

In this section, we evaluate the transferability of both evasion and poisoning attacks across a range of ML models. We highlight some interesting findings about transferability, based on the three metrics developed in Sect. 4. In particular, we analyze attack transferability in terms of its connection to the size of the input gradients of the loss function, the gradient alignment between surrogate and target classifiers, and the variability of the loss function optimized to craft the attack points. We provide recommendations on how to choose the most effective surrogate models to craft transferable attacks in the black-box setting.

5.1 Transferability of Evasion Attacks

We start by reporting our experiments on evasion attacks. We consider here two distinct case studies, involving handwritten digit recognition and Android malware detection.

5.1.1 Handwritten Digit Recognition

The MNIST89 data includes the MNIST handwritten digits from classes 8 and 9. Each digit image consists of 784 pixels ranging from 0 to 255, normalized in $[0,1]$ by dividing such values by 255. We run 10 independent repetitions to average the results on different training-test splits. In each repetition, we run white-box and black-box attacks, using $5,900$ samples to train the target classifier, $5,900$ distinct samples to train the surrogate classifier (without even relabeling the surrogate data with labels predicted by the target classifier; i.e., we do not perform any query on the target), and $1,000$ test samples. We modified test digits in both classes using Algorithm 1 under the ℓ_2 distance constraint $\|\mathbf{x}-\mathbf{x}'\|_2 \leq \varepsilon$, with $\varepsilon \in [0,5]$.

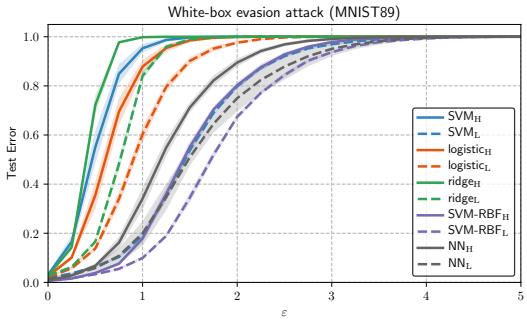


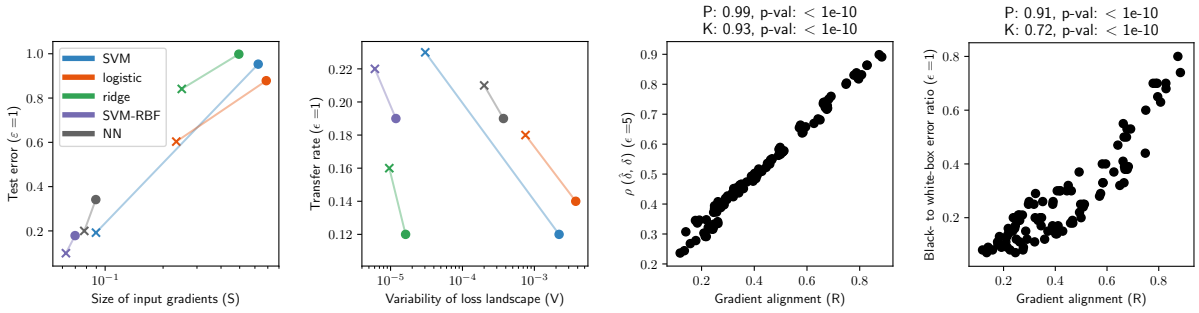
Figure 5: White-box evasion attacks on MNIST89. Test error against increasing maximum perturbation ε.

For each of the following learning algorithms, we train a high-complexity (H) and a low-complexity (L) model, by changing its hyperparameters: (*i*) SVMs with linear kernel (SVM_H with $C = 100$ and SVM_L with $C = 0.01$); (*ii*) SVMs with RBF kernel (SVM-RBF_H with $C = 100$ and SVM-RBF_L with $C = 1$, both with $\gamma = 0.01$); (*iii*) logistic classifiers (logistic_H with $C = 10$ and logistic_L with $C = 1$); (*iv*) ridge classifiers (ridge_H with $\alpha = 1$ and ridge_L with $\alpha = 10$);[2] (*v*) fully-connected neural networks with two hidden layers including 50 neurons each, and ReLU activations (NN_H with no regularization, i.e., weight decay set to 0, and NN_L with weight decay set to 0.01), trained via cross-entropy loss minimization; and (*vi*) random forests consisting of 30 trees (RF_H with no limit on the depth of the trees and RF_L with a maximum depth of 8). These configurations are chosen to evaluate the robustness of classifiers that exhibit similar test accuracies but different levels of complexity.

How does model complexity impact evasion attack success in the white-box setting? The results for white-box evasion attacks are reported for all classifiers that fall under our framework and can be tested for evasion with gradient-based attacks (SVM, Logistic, Ridge, and NN). This excludes random forests, as they are not differentiable. We report the complete *security evaluation curves* [5] in Fig. 5, showing the mean test error (over 10 runs) against an increasing maximum admissible distortion ε. In Fig. 6a we report the mean test error at $\varepsilon = 1$ for each target model against the size of its input gradients (S, averaged on the test samples and on the 10 runs).

The results show that, for each learning algorithm, the low-complexity model has smaller input gradients, and it is less vulnerable to evasion than its high-complexity counterpart, confirming our theoretical analysis. This is also confirmed by the p-values reported in Table 1 (first column), obtained by

[2] Recall that the level of regularization increases as α increases, and as C decreases.



(a) (b) (c) (d)

Figure 6: Evaluation of our metrics for evasion attacks on MNIST89. (a) Test error under attack vs average size of input gradients (S) for low- (denoted with '$\times$') and high-complexity (denoted with '$\circ$') classifiers. (b) Average transfer rate vs variability of loss landscape (V). (c) Pearson correlation coefficient $\rho(\hat{\delta}, \delta)$ between black-box ($\hat{\delta}$) and white-box (δ) perturbations (values in Fig. 8, right) vs gradient alignment (R, values in Fig. 8, left) for each target-surrogate pair. Pearson (P) and Kendall (K) correlations between ρ and R are also reported along with the p-values obtained from a permutation test to assess statistical significance.

	Evasion				**Poisoning**			
	MNIST89		**DREBIN**		**MNIST89**		**LFW**	
	$\varepsilon = 1$	$\varepsilon = 1$	$\varepsilon = 5$	$\varepsilon = 30$	5%	20%	5%	20%
SVM	<1e-2	<1e-2	<1e-2	<1e-2	<1e-2	<1e-2	<1e-2	0.75
logistic	<1e-2	<1e-2	<1e-2	0.02	<1e-2	<1e-2	0.10	0.21
ridge	<1e-2	<1e-2	<1e-2	<1e-2	0.02	<1e-2	0.02	0.75
SVM-RBF	<1e-2	<1e-2	<1e-2	<1e-2	<1e-2	<1e-2	<1e-2	0.11
NN	<1e-2	<1e-2	<1e-2	0.02				

Table 1: Statistical significance of our results. For each attack, dataset and learning algorithm, we report the p-values of two two-sided binomial tests, to respectively reject the null hypothesis that: (*i*) for white-box attacks, the test errors of the high- and low-complexity target follow the same distribution; and (*ii*) for black-box attacks, the transfer rates of the high- and low-complexity surrogate follow the same distribution. Each test is based on 10 samples, obtained by comparing the error of the high- and low-complexity models for each learning algorithm in each repetition. In the first (second) case, success corresponds to a larger test (transfer) error for the high-complexity target (low-complexity surrogate).

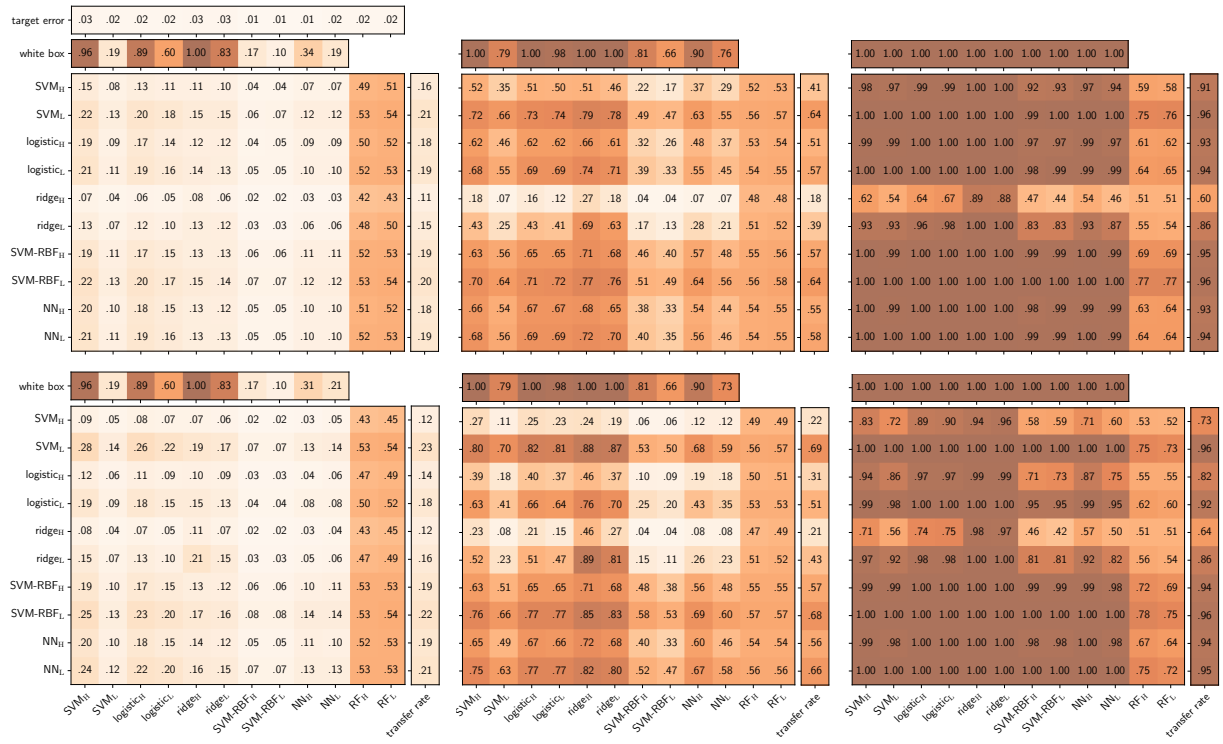
running a binomial test for each learning algorithm to compare the white-box test error of the corresponding high- and low-complexity models. All the p-values are smaller than 0.05, which confirms 95% statistical significance. Recall that these results hold only when comparing models trained using the same learning algorithm. This means that we can compare, e.g., the S value of SVM_H against SVM_L, but not that of SVM_H against logistic_H. In fact, even though logistic_H exhibits the largest S value, it is not the most vulnerable classifier. Another interesting finding is that nonlinear classifiers tend to be less vulnerable than linear ones.

How do evasion attacks transfer between models in black-box settings? In Fig. 7 we report the results for black-box evasion attacks, in which the attacks against surrogate models (in rows) are transferred to the target models (in columns). The top row shows results for surrogates trained using only 20% of the surrogate training data, while in the bottom row surrogates are trained using all surrogate data, i.e., a training set of the same size as that of the target. The three columns report results for $\varepsilon \in \{1, 2, 5\}$.

It can be noted that lower-complexity models (with stronger regularization) provide better surrogate models, on average. In particular, this can be seen best in the middle column for medium level of perturbation, in which the lower-complexity models (SVM_L, logistic_L, ridge_L, and SVM-RBF_L) provide on average higher error when transferred to other models. The reason is that they learn smoother and stabler functions, that are capable of better approximating the target function. Surprisingly, this holds also when using only 20% of training data, as the black-box attacks relying on such low-complexity models still transfer with similar test errors. This means that most classifiers can be attacked in this black-box setting with almost no knowledge of the model, no query access, but provided that one can get a small amount of data similar to that used to train the target model.

These findings are also confirmed by looking at the variability of the loss landscape, computed as discussed in Sect. 4 (by considering 10 different training sets), and reported against the average transfer rate of each surrogate model in Fig. 6b. It is clear from that plot that higher-variance classifiers are less effective as surrogates than their less-complex counterparts, as the former tend to provide worse, unstable approximations of the target classifier. To confirm the statistical significance of this result, for each learning algorithm we also compare the mean transfer errors of high- and low-complexity surrogates with a binomial test whose p-values (always lower than 0.05) are reported in Table 1 (second column).

Another interesting, related observation is that the adversarial examples computed against lower-complexity surrogates have to be perturbed more to evade (see Fig. 9), whereas the perturbation of the ones computed against complex models



(a) $\varepsilon = 1$ (b) $\varepsilon = 2$ (c) $\varepsilon = 5$

Figure 7: Black-box (transfer) evasion attacks on MNIST89. Each cell contains the test error of the target classifier (in columns) computed on the attack samples crafted against the surrogate (in rows). Matrices in the top (bottom) row correspond to attacks crafted against surrogate models trained with 20% (100%) of the surrogate training data, for $\varepsilon \in \{1, 2, 5\}$. The test error of each target classifier in the absence of attack (target error) and under (white-box) attack are also reported for comparison, along with the mean transfer rate of each surrogate across targets. Darker colors mean higher test error, i.e., better transferability.

can be smaller. This is again due to the instability induced by high-complexity models into the loss function optimized to craft evasion attacks, whose sudden changes cause the presence of closer local optima to the initial attack point.

On the vulnerability of random forests. A noteworthy finding is that random forests can be effectively attacked at small perturbation levels using most other models (see last two columns in Fig. 7). We looked at the learned trees and discovered that trees often are susceptible to small changes. In one example, a node of the tree checked if a particular feature value was above 0.002, and classified samples as digit 8 if that condition holds (and digit 9 otherwise). The attack modified that feature from 0 to 0.028, causing it to be immediately misclassified. This vulnerability is intrinsic in the selection process of the threshold values used by these decision trees to split each node. The threshold values are selected among the existing values in the dataset (to correctly handle categorical attributes). Therefore, for pixels which are highly discriminant (e.g., mostly black for one class and white for the other), the threshold will be either very close to one extreme or the other, making it easy to subvert the prediction by a small change. Since ℓ_2-norm attacks change almost all feature values, with high probability the attack modifies at least one feature on every path of the tree, causing misclassification.

Is gradient alignment an effective transferability metric? In Fig. 8, we report on the left the gradient alignment computed between surrogate and target models, and on the right the Pearson correlation coefficient $\rho(\hat{\delta}, \delta)$ between the perturbation optimized against the surrogate (i.e., the black-box perturbation $\hat{\delta}$) and that optimized against the target (i.e., the white-box perturbation δ). We observe immediately that gradient alignment provides an accurate measure of transferability: the higher the cosine similarity, the higher the correlation (meaning that the adversarial examples crafted against the two models are similar). We correlate these two measures in Fig. 6c, and show that such correlation is statistically significant for both Pearson and Kendall coefficients. In Fig. 6d we also correlate gradient alignment with the ratio between the test error of the target model in the black- and white-box setting (extrapolated from the matrix corresponding to $\varepsilon = 1$ in the bottom row of Fig. 7), as suggested by our theoretical derivation. The corresponding permutation tests confirm statistical significance. We finally remark that gradient alignment is extremely fast to evaluate, as it does not require simulating any attack, but it is only a relative measure of the attack trans-

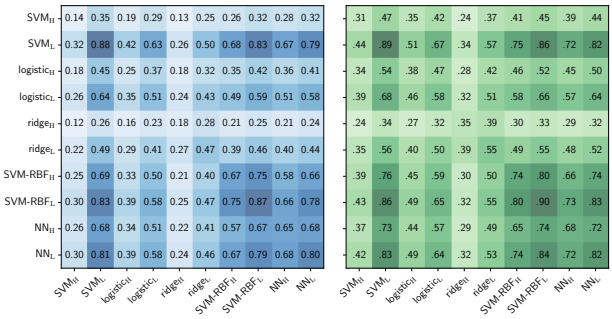


Figure 8: Gradient alignment and perturbation correlation for evasion attacks on MNIST89. *Left:* Gradient alignment R (Eq. 18) between surrogate (rows) and target (columns) classifiers, averaged on the unmodified test samples. *Right:* Pearson correlation coefficient $\rho(\delta, \hat{\delta})$ between white-box and black-box perturbations for $\varepsilon = 5$.

ferability, as the latter also depends on the complexity of the target model; i.e., on the size of its input gradients.

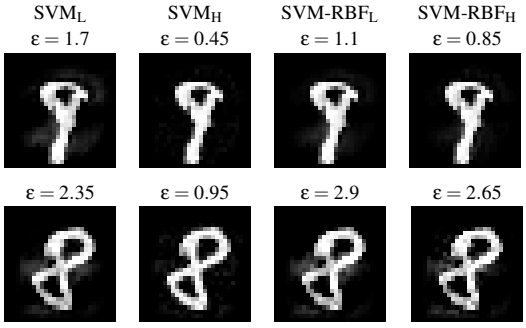


Figure 9: Digit images crafted to evade linear and RBF SVMs. The values of ε reported here correspond to the minimum perturbation required to evade detection. Larger perturbations are required to mislead low-complexity classifiers (L), while smaller ones suffice to evade high-complexity classifiers (H).

5.1.2 Android Malware Detection

The Drebin data [1] consists of around 120,000 legitimate and around 5000 malicious Android applications, labeled using the VirusTotal service. A sample is labeled as malicious (or positive, $y = +1$) if it is classified as such from at least five out of ten anti-virus scanners, while it is flagged as legitimate (or negative, $y = -1$) otherwise. The structure and the source code of each application is encoded as a *sparse* feature vector consisting of around a million binary features denoting the presence or absence of permissions, suspicious URLs and other relevant information that can be extracted by statically analyzing Android applications. Since we are working with sparse binary features, we use the ℓ_1 norm for the attack.

We use $30,000$ samples to learn surrogate and target classifiers, and the remaining $66,944$ samples for testing. The

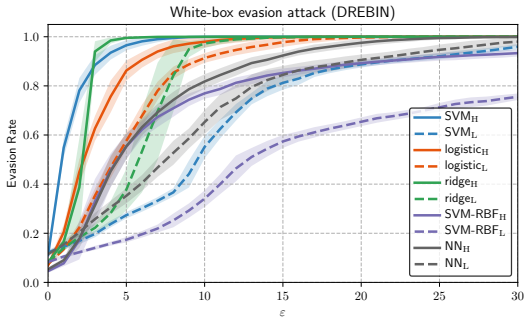


Figure 10: White-box evasion attacks on DREBIN. Evasion rate against increasing maximum perturbation ε.

classifiers and their hyperparameters are the same used for MNIST89, apart from (*i*) the number of hidden neurons for NN_H and NN_L, set to 200, (*ii*) the weight decay of NN_L, set to 0.005; and (*iii*) the maximum depth of RF_L, set to 59.

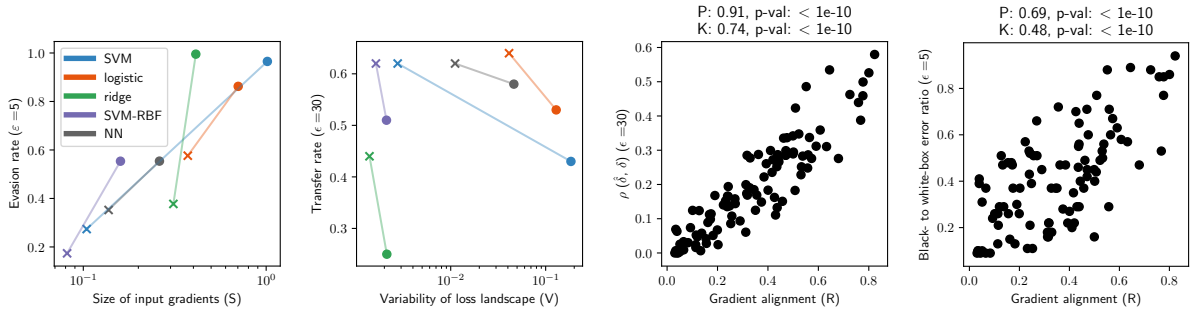
We perform feature selection to retain those $5,000$ features which maximize information gain, i.e., $|p(x_k = 1|y = +1) - p(x_k = 1|y = -1)|$, where x_k is the kth feature. While this feature selection process does not significantly affect the detection rate (which is only reduced by 2%, on average, at 0.5% false alarm rate), it drastically reduces the computational complexity of classification.

In each experiment, we run white-box and black-box evasion attacks on $1,000$ distinct malware samples (randomly selected from the test data) against an increasing number of modified features in each malware $\varepsilon \in \{0, 1, 2, \ldots, 30\}$. This is achieved by imposing the ℓ_1 constraint $\|\mathbf{x}' - \mathbf{x}\|_1 \leq \varepsilon$. As in previous work, we further restrict the attacker to only *inject* features into each malware sample, to avoid compromising its intrusive functionality [3, 11].

To evaluate the impact of the aforementioned evasion attack, we measure the evasion rate (i.e., the fraction of malware samples misclassified as legitimate) at 0.5% false alarm rate (i.e., when only 0.5% of the legitimate samples are misclassified as malware). As in the previous experiment, we report the complete *security evaluation curve* for the white-box attack case, whereas we report only the value of test error for the black-box case. The results, reported in Figs. 10, 11, 12, and 13, along with the statistical tests in Table 1 (third and fourth columns) confirm the main findings of the previous experiments. One significant difference is that random forests are much more robust in this case. The reason is that the ℓ_1-norm attack (differently from the ℓ_2) only changes a small number of features, and thus the probability that it will change features in all the ensemble trees is very low.

5.2 Transferability of Poisoning Attacks

For poisoning attacks, we report experiments on handwritten digits and face recognition.



(a) (b) (c) (d)

Figure 11: Evaluation of our metrics for evasion attacks on DREBIN. See the caption of Fig. 6 for further details.

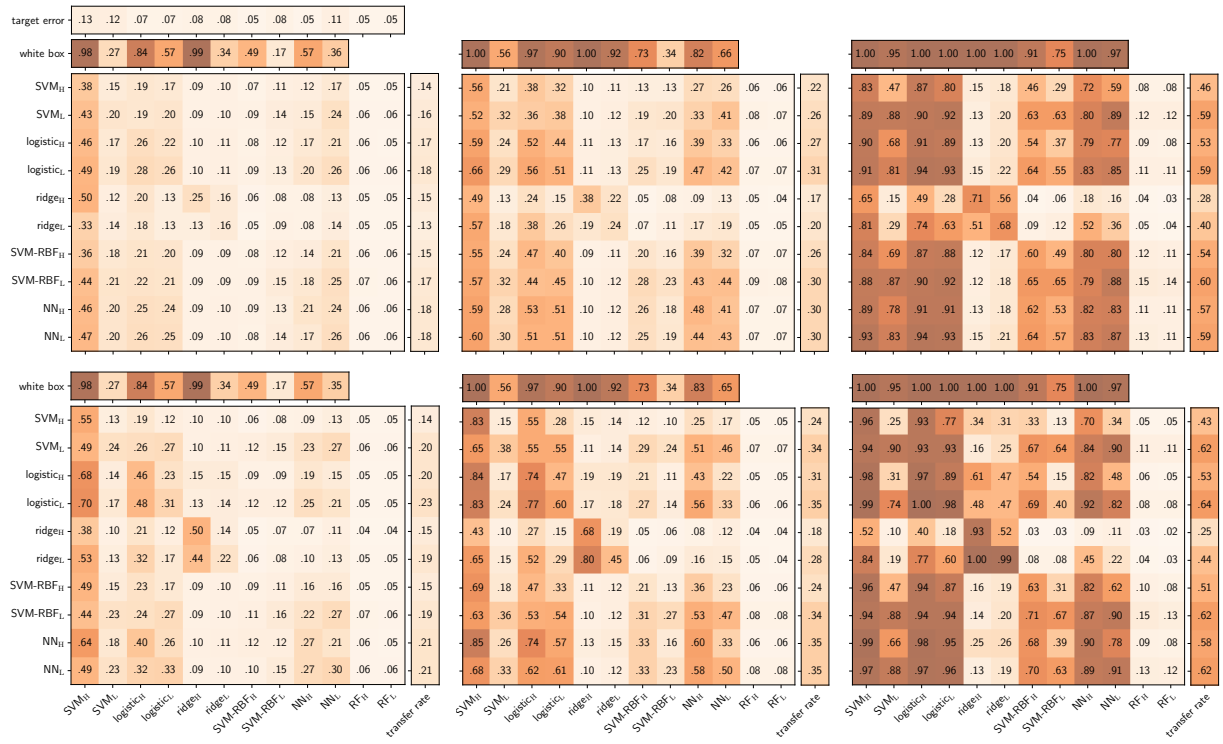
5.2.1 Handwritten Digit Recognition

We apply our optimization framework to poison SVM, logistic, and ridge classifiers in the white-box setting. Designing efficient poisoning availability attacks against neural networks is still an open problem due to the complexity of the bilevel optimization and the non-convexity of the inner learning problem. Previous work has mainly considered integrity poisoning attacks against neural networks [5, 20, 41], and it is believed that neural networks are much more resilient to poisoning availability attacks due to their memorization capability. Poisoning random forests is not feasible with gradient-based attacks, and we are not aware of any existing attacks for this ensemble method. We thus consider as surrogate learners: (*i*) linear SVMs with $C = 0.01$ (SVM_L) and $C = 100$ (SVM_H); (*ii*) logistic classifiers with $C = 0.01$ (logistic_L) and $C = 10$ (logistic_H); (*iii*) ridge classifiers with $\alpha = 100$ (ridge_L) and $\alpha = 10$ (ridge_H); and (*iv*) SVMs with RBF kernel with $\gamma = 0.01$ and $C = 1$ (SVM-RBF_L) and $C = 100$ (SVM-RBF_H). We additionally consider as target classifiers: (*i*) random forests with 100 base trees, each with a maximum depth of 6 for RF_L, and with no limit on the maximum depth for RF_H; (*ii*) feed-forward neural networks with two hidden layers of 200 neurons each and ReLU activations, trained via cross-entropy loss minimization with different regularization (NN_L with weight decay 0.01 and NN_H with no decay); and (*iii*) the Convolutional Neural Network (CNN) used in [7].

We consider 500 training samples, 1,000 validation samples to compute the attack, and a separate set of 1,000 test samples to evaluate the error. The test error is computed against an increasing number of poisoning points into the training set, from 0% to 20% (corresponding to 125 poisoning points). The reported results are averaged on 10 independent, randomly-drawn data splits.

How does model complexity impact poisoning attack success in the white-box setting? The results for white-box poisoning are reported in Fig. 14. Similarly to the evasion case, high-complexity models (with larger input gradients, as shown in Fig. 15a) are more vulnerable to poisoning attacks than their low-complexity counterparts (i.e., given that the same learning algorithm is used). This is also confirmed by the statistical tests in the fifth column of Table 1. Therefore, model complexity plays a large role in a model's robustness also against poisoning attacks, confirming our analysis.

How do poisoning attacks transfer between models in black-box settings? The results for black-box poisoning are reported in Fig. 16. For poisoning attacks, the best surrogates are those matching the complexity of the target, as they tend to be better aligned and to share similar local optima, except for low-complexity logistic and ridge surrogates, which seem to transfer better to linear classifiers. This is also witnessed by gradient alignment in Fig. 17, which is again not only correlated to the similarity between black- and white-box perturbations (Fig. 15c), but also to the ratio between the black- and white-box test errors (Fig. 15d). Interestingly, these error ratios are larger than one in some cases, meaning that attacking a surrogate model can be more effective than running a white-box attack against the target. A similar phenomenon has been observed for evasion attacks [33], and it is due to the fact that optimizing attacks against a *smoother* surrogate may find better local optima of the target function (e.g., by overcoming gradient obfuscation [2]). According to our findings, for poisoning attacks, reducing the variability of the loss landscape (V) of the surrogate model is less important than finding a good alignment between the surrogate and the target. In fact, from Fig. 15b it is evident that increasing V is even beneficial for SVM-based surrogates (and all these results are statistically significant according to the *p*-values in the sixth column of Table 1). A visual inspection of the poisoning digits in Fig. 18 reveals that the poisoning points crafted against high-complexity classifiers are only minimally perturbed, while the ones computed against low-complexity classifiers exhibit larger, visible perturbations. This is again due to the presence of closer local optima in the former case. Finally, a surprising result is that RFs are quite robust to poisoning, as well as NNs when attacked with low-complexity linear surrogates. The reason may be that these target classifiers have a large capacity, and can thus fit *outlying* samples (like the digits crafted against low-complexity classifiers in Fig. 18) without affecting the classification of the other training samples.



(a) $\varepsilon = 5$ (b) $\varepsilon = 10$ (c) $\varepsilon = 30$

Figure 12: Black-box (transfer) evasion attacks on DREBIN. See the caption of Fig. 7 for further details.

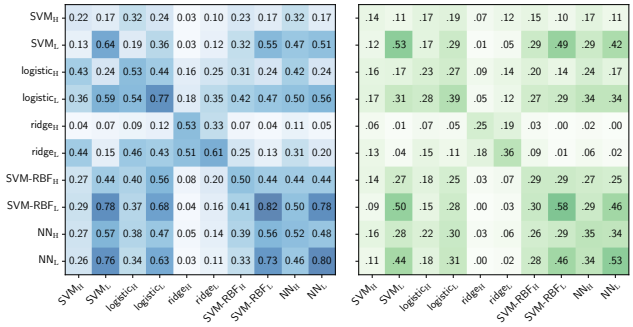


Figure 13: Gradient alignment and perturbation correlation (at $\varepsilon = 30$) for evasion attacks on DREBIN. See the caption of Fig. 8 for further details.

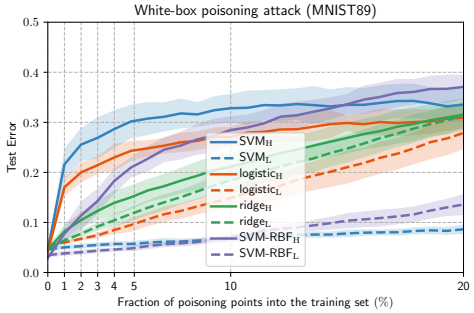
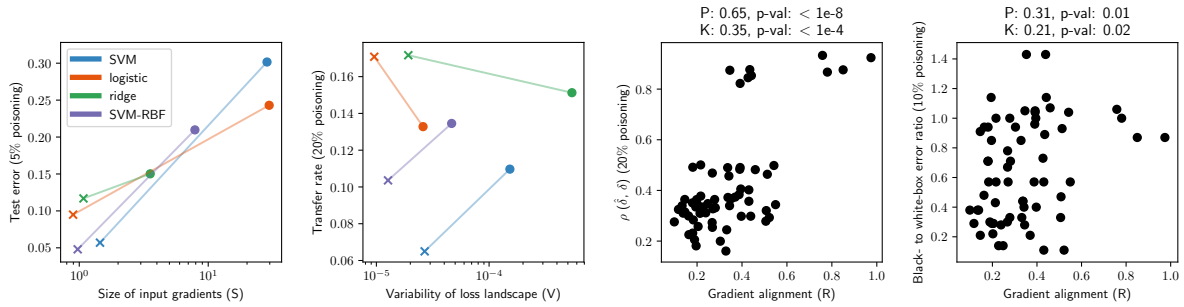


Figure 14: White-box poisoning attacks on MNIST89. Test error against an increasing fraction of poisoning points.

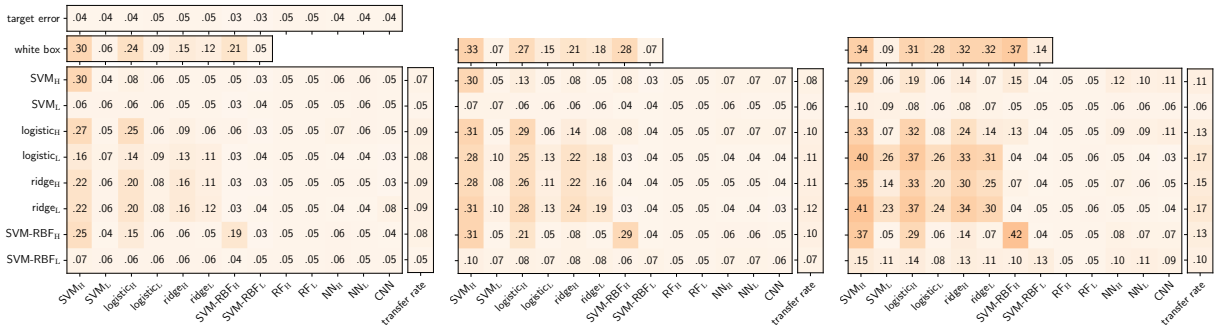
5.2.2 Face Recognition

The Labeled Face on the Wild (LFW) dataset consists of faces of famous peoples collected on Internet. We considered the six identities with the largest number of images in the dataset. We considered the person with most images as positive class, and all the others as negative. Our dataset consists of 530 positive and 758 negative images. The classifiers and their hyperparameters are the same used for MNIST89, except that we set: (*i*) $C = 0.1$ for logistic_L, (*ii*) $\alpha = 1$ for ridge_H, (*iii*) $\gamma = 0.001, C = 10$ for SVM-RBF_L, (*iv*) $\gamma = 0.001, C = 1000$ for SVM-RBF_H, and (*v*) weight decay to 0.001 for NN_L. We run 10 repetitions with 300 samples in each training, validation and test set. The results are shown in Figs 19, 20, 21 and 22, confirming the main findings discussed for poisoning attacks on MNIST89. Statistical tests for significance are reported in Table 1 (seventh and eighth columns). In this case, there is not a significant distinction between the mean transfer rates of high- and low-complexity surrogates, probably due to the reduced size of the training sets used. Finally, in Fig. 23 we report examples of perturbed faces against surrogates with different complexities, confirming again that larger perturbations are required to attack lower-complexity models.



(a) (b) (c) (d)

Figure 15: Evaluation of our metrics for poisoning attacks on MNIST89. See the caption of Fig. 6 for further details.



(a) 5% poisoning (b) 10% poisoning (c) 20% poisoning

Figure 16: Black-box (transfer) poisoning attacks on MNIST89. See the caption of Fig. 7 for further details.

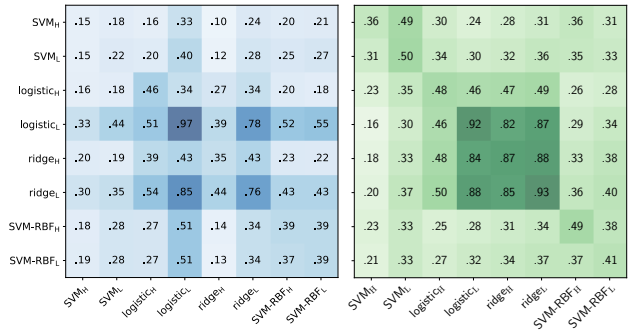


Figure 17: Gradient alignment and perturbation correlation (at 20% poisoning) for poisoning attacks on MNIST89. See the caption of Fig. 8 for further details.

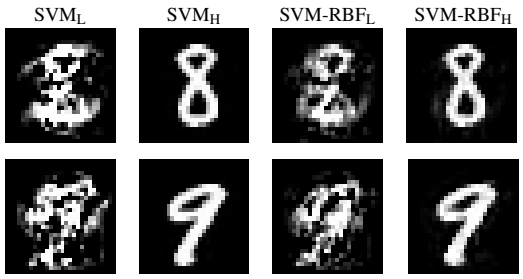


Figure 18: Poisoning digits crafted against linear and RBF SVMs. Larger perturbations are required to have significant impact on low-complexity classifiers (L), while minimal changes are very effective on high-complexity SVMs (H).

5.3 Summary of Transferability Evaluation

We summarize the results of transferability for evasion and poisoning attacks below.

(1) Size of input gradients. Low-complexity target classifiers are less vulnerable to evasion and poisoning attacks than high-complexity target classifiers trained with the same learning algorithm, due to the reduced size of their input gradients. In general, nonlinear models are more robust than linear models to both types of attacks.

(2) Gradient alignment. Gradient alignment is correlated with transferability. Even though it cannot be directly measured in black-box scenarios, some useful guidelines can be derived from our analysis. For evasion attacks, low-complexity surrogate classifiers provide stabler gradients which are better aligned, on average, with those of the target models; thus, it is generally preferable to use strongly-regularized surrogates. For poisoning attacks, instead, gradient alignment tends to improve when the surrogate matches the complexity (regularization) of the target (which may be estimated using techniques from [46]).

(3) Variability of the loss landscape. Low-complexity surro-

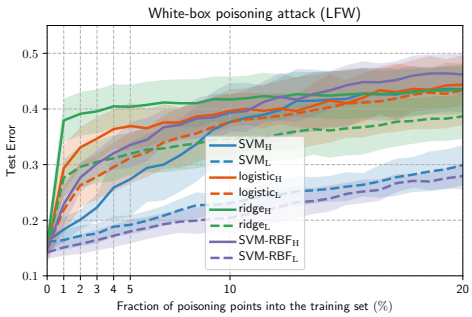


Figure 19: White-box poisoning attacks on LFW. Test error against an increasing fraction of poisoning points.

gate classifiers provide loss landscapes with lower variability than high-complexity surrogate classifiers trained with the same learning algorithm, especially for evasion attacks. This results in better transferability.

To summarize, for evasion attacks, decreasing complexity of the surrogate model by properly adjusting the hyperparameters of its learning algorithm provides adversarial examples that transfer better to a range of models. For poisoning attacks, the best surrogates are generally models with similar levels of regularization as the target model. The reason is that the poisoning objective function is relatively stable (i.e., it has low variance) for most classifiers, and gradient alignment between surrogate and target becomes a more important factor.

Understanding attack transferability has two main implications. First, even when attackers do not know the target classifier, our findings suggest that low-complexity surrogates have a better chance of transferring to other models. Our recommendation to performing black-box evasion attacks is to choose surrogates with low complexity (e.g., by using strong regularization and reducing model variance). To perform poisoning attacks, our recommendation is to acquire additional information about the level of regularization of the target and train a surrogate model with a similar level of regularization. Second, our analysis also provides recommendations to defenders on how to design more robust models against evasion and poisoning attacks. In particular, lower-complexity models tend to have more resilience compared to more complex models. Of course, we need to take into account the bias-variance trade-off and choose models that still perform relatively well on the original prediction tasks.

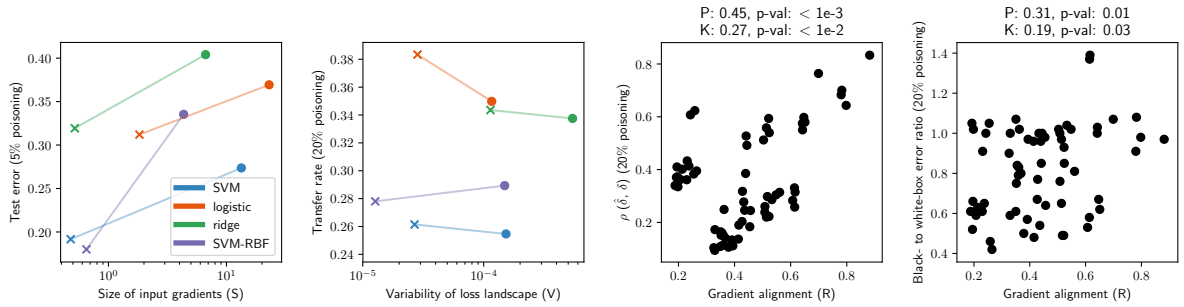
6 Related Work

Transferability for evasion attacks. Transferability of evasion attacks has been studied in previous work, e.g., [3, 13, 14, 21, 26, 32, 33, 42, 43, 47]. Biggio et al. [3] have been the first to consider evasion attacks against surrogate models in a limited-knowledge scenario. Goodfellow et al. [14], Tramer et al. [43], and Moosavi et al. [26] have made the observation that different models might learn intersecting decision boundaries in both benign and adversarial dimensions and in that case adversarial examples transfer better. Tramer et al. have also performed a detailed study of transferability of model-agnostic perturbations that depend only on the training data, noting that adversarial examples crafted against linear models can transfer to higher-order models. We answer some of the open questions they posed about factors contributing to attack transferability. Liu et al. [21] have empirically observed the gradient alignment between models with transferable adversarial examples. Papernot et al. [32, 33] have observed that adversarial examples transfer across a range of models, including logistic regression, SVMs and neural networks, without providing a clear explanation of the phenomenon. Prior work has also investigated the role of input gradients and Jacobians. Some authors have proposed to decrease the magnitude of input gradients during training to defend against evasion attacks [22, 35] or improve classification accuracy [40, 44]. In [35, 39], the magnitude of input gradients has been identified as a cause for vulnerability to evasion attacks. A number of papers have shown that transferability of adversarial examples is increased by averaging the gradients computed for ensembles of models [13, 21, 43, 47]. We highlight that we obtain similar effect by attacking a strongly-regularized surrogate model with a smoother and stabler decision boundary (resulting in a lower-variance model). The advantage of our approach is to reduce the computational complexity compared to attacking classifier ensembles. Through our formalization, we shed light on the most important factors for transferability. In particular, we identify a set of conditions that explain transferability including the gradient alignment between the surrogate and targeted models, and the size of the input gradients of the target model, connected to model complexity. We demonstrate that adversarial examples crafted against lower-variance models (e.g., those that are strongly regularized) tend to transfer better across a range of models.

Transferability for poisoning attacks. There is very little work on the transferability of poisoning availability attacks, except for a preliminary investigation in [27]. That work indicates that poisoning examples are transferable among very simple network architectures (logistic regression, MLP, and Adaline). Transferability of targeted poisoning attacks has been addressed recently in [41]. We are the first to study in depth transferability of poisoning availability attacks.

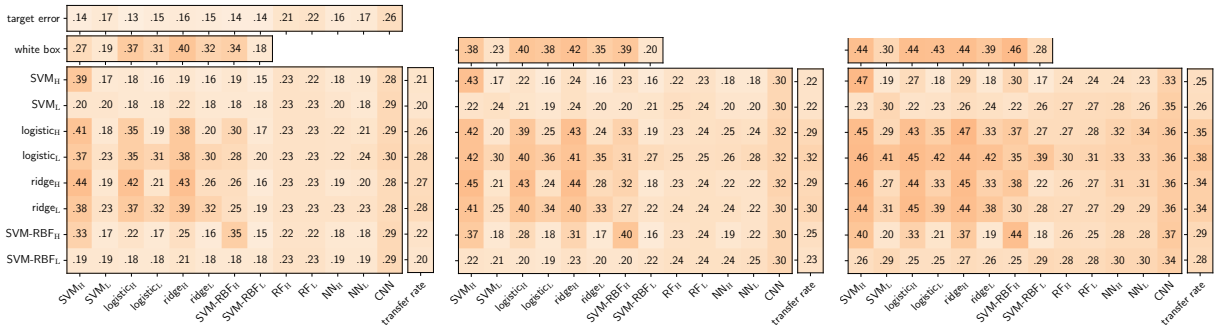
7 Conclusions

We have conducted an analysis of the transferability of evasion and poisoning attacks under a unified optimization framework. Our theoretical transferability formalization sheds light on various factors impacting the transfer success rates. In particular, we have defined three metrics that impact the transferability of an attack, including the complexity of the target model, the gradient alignment between the surrogate and



(a) (b) (c) (d)

Figure 20: Evaluation of our metrics for poisoning attacks on LFW. See the caption of Fig. 6 for further details.



(a) 5% poisoning (b) 10% poisoning (c) 20% poisoning

Figure 21: Black-box (transfer) poisoning attacks on LFW. See the caption of Fig. 7 for further details.

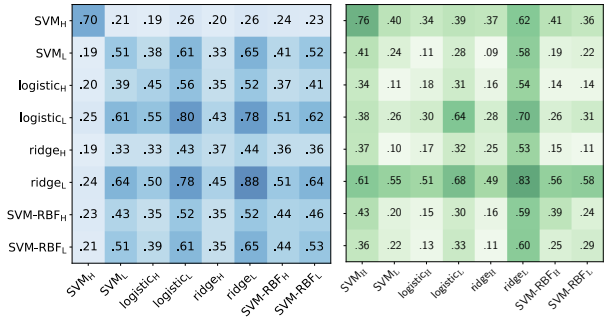


Figure 22: Gradient alignment and perturbation correlation (at 20% poisoning) for poisoning attacks on LFW. See the caption of Fig. 8 for further details.

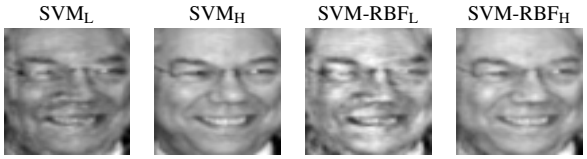


Figure 23: Adversarial examples crafted against linear and RBF SVMs. Larger perturbations are required to have significant impact on low-complexity classifiers (L), while minimal changes are very effective on high-complexity SVMs (H).

target models, and the variance of the attacker optimization objective. The lesson to system designers is to evaluate their classifiers against these criteria and select lower-complexity, stronger regularized models that tend to provide higher robustness to both evasion and poisoning. Interesting avenues for future work include extending our analysis to multi-class classification settings, and considering a range of gray-box models in which attackers might have additional knowledge of the machine learning system (as in [41]). Application-dependent scenarios such as cyber security might provide additional constraints on threat models and attack scenarios and could impact transferability in interesting ways.

Acknowledgements

The authors would like to thank Neil Gong for shepherding our paper and the anonymous reviewers for their constructive feedback. This work was partly supported by the EU H2020 project ALOHA, under the European Union's Horizon 2020 research and innovation programme (grant no.780788). This research was also sponsored by the Combat Capabilities Development Command Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Combat Capabilities Development Command Army Research Laboratory

or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes not withstanding any copyright notation here on. We would also like to thank Toyota ITC for funding this research.

References

[1] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck. Drebin: Efficient and explainable detection of android malware in your pocket. In *21st NDSS*. The Internet Society, 2014.

[2] A. Athalye, N. Carlini, and D. A. Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *ICML*, vol. 80 of *JMLR W&CP*, pp. 274–283. JMLR.org, 2018.

[3] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli. Evasion attacks against machine learning at test time. In H. Blockeel et al., editors, ECML PKDD, Part III, vol. 8190 of *LNCS*, pp. 387–402. Springer Berlin Heidelberg, 2013.

[4] B. Biggio, B. Nelson, and P. Laskov. Poisoning attacks against support vector machines. In J. Langford and J. Pineau, editors, *29th Int'l Conf. on Machine Learning*, pp. 1807–1814. Omnipress, 2012.

[5] B. Biggio and F. Roli. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition*, 84:317–331, 2018.

[6] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Stats)*. Springer, 2007.

[7] N. Carlini and D. A. Wagner. Adversarial examples are not easily detected: Bypassing ten detection methods. In B. M. Thuraisingham et al., editors, *10th ACM Workshop on Artificial Intelligence and Security*, AISec '17, pp. 3–14, New York, NY, USA, 2017. ACM.

[8] N. Carlini and D. A. Wagner. Towards evaluating the robustness of neural networks. In *IEEE Symp. on Sec. and Privacy*, pp. 39–57. IEEE Computer Society, 2017.

[9] X. Chen, C. Liu, B. Li, K. Lu, and D. Song. Targeted backdoor attacks on deep learning systems using data poisoning. *ArXiv e-prints*, abs/1712.05526, 2017.

[10] H. Dang, Y. Huang, and E.-C. Chang. Evading classifiers by morphing in the dark. In *24th ACM SIGSAC Conf. on Computer and Comm. Sec.*, CCS, 2017.

[11] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, and F. Roli. Yes, machine learning can be more secure! a case study on android malware detection. *IEEE Trans. Dependable and Secure Computing*, In press.

[12] A. Demontis, P. Russu, B. Biggio, G. Fumera, and F. Roli. On security and sparsity of linear classifiers for adversarial settings. In A. Robles-Kelly et al., editors, *Joint IAPR Int'l Workshop on Structural, Syntactic, and Statistical Patt. Rec.*, vol. 10029 of *LNCS*, pp. 322–332, Cham, 2016. Springer International Publishing.

[13] Y. Dong, F. Liao, T. Pang, X. Hu, and J. Zhu. Boosting adversarial examples with momentum. In *CVPR*, 2018.

[14] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. In *ICLR*, 2015.

[15] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. D. McDaniel. Adversarial examples for malware detection. In *ESORICS (2)*, vol. 10493 of *LNCS*, pp. 62–79. Springer, 2017.

[16] T. Gu, B. Dolan-Gavitt, and S. Garg. Badnets: Identifying vulnerabilities in the machine learning model supply chain. In *NIPS Workshop on Machine Learning and Computer Security*, vol. abs/1708.06733, 2017.

[17] A. Ilyas, L. Engstrom, A. Athalye, and J. Lin. Black-box adversarial attacks with limited queries and information. In J. Dy and A. Krause, editors, *35th ICML*, vol. 80, pp. 2137–2146. PMLR, 2018.

[18] M. Jagielski, A. Oprea, B. Biggio, C. Liu, C. Nita-Rotaru, and B. Li. Manipulating machine learning: Poisoning attacks and countermeasures for regression learning. In *IEEE Symp. S&P*, pp. 931–947. IEEE CS, 2018.

[19] A. Kantchelian, J. D. Tygar, and A. D. Joseph. Evasion and hardening of tree ensemble classifiers. In *33rd ICML*, vol. 48 of *JMLR W&CP*, pp. 2387–2396. JMLR.org, 2016.

[20] P. W. Koh and P. Liang. Understanding black-box predictions via influence functions. In *Proc. of the 34th Int'l Conf. on Machine Learning*, ICML, 2017.

[21] Y. Liu, X. Chen, C. Liu, and D. Song. Delving into transferable adversarial examples and black-box attacks. In *ICLR*, 2017.

[22] C. Lyu, K. Huang, and H.-N. Liang. A unified gradient regularization family for adversarial examples. In *IEEE Int'l Conf. on Data Mining (ICDM)*, vol. 00, pp. 301–309, Los Alamitos, CA, USA, 2015. IEEE CS.

[23] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks. In *ICLR*, 2018.

[24] S. Mei and X. Zhu. Using machine teaching to identify optimal training-set attacks on machine learners. In *29th AAAI Conf. Artificial Intelligence (AAAI '15)*, 2015.

[25] M. Melis, A. Demontis, B. Biggio, G. Brown, G. Fumera, and F. Roli. Is deep learning safe for robot vision? Adversarial examples against the iCub humanoid. In *ICCVW ViPAR*, pp. 751–759. IEEE, 2017.

[26] S.-M. Moosavi-Dezfooli, A. Fawzi, O. Fawzi, and P. Frossard. Universal adversarial perturbations. In *CVPR*, 2017.

[27] L. Muñoz-González, B. Biggio, A. Demontis, A. Paudice, V. Wongrassamee, E. C. Lupu, and F. Roli. Towards poisoning of deep learning algorithms with back-gradient optimization. In B. M. Thuraisingham et al., editors, *10th ACM Workshop on AI and Sec.*, AISec '17, pp. 27–38, New York, NY, USA, 2017. ACM.

[28] B. Nelson, M. Barreno, F. J. Chi, A. D. Joseph, B. I. P. Rubinstein, U. Saini, C. Sutton, J. D. Tygar, and K. Xia. Exploiting machine learning to subvert your spam filter. In *LEET '08*, pp. 1–9, Berkeley, CA, USA, 2008. USENIX Association.

[29] A. Newell, R. Potharaju, L. Xiang, and C. Nita-Rotaru. On the practicality of integrity attacks on document-level sentiment analysis. In *AISec*, 2014.

[30] J. Newsome, B. Karp, and D. Song. Paragraph: Thwarting signature learning by training maliciously. In *RAID*, pp. 81–105. Springer, 2006.

[31] N. Papernot, P. McDaniel, and I. Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. arXiv:1605.07277, 2016.

[32] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami. Practical black-box attacks against machine learning. In *ASIA CCS '17*, pp. 506–519, New York, NY, USA, 2017. ACM.

[33] N. Papernot, P. D. McDaniel, and I. J. Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *ArXiv e-prints*, abs/1605.07277, 2016.

[34] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. Sharif. Misleading worm signature generators using deliberate noise injection. In *IEEE Symp. Sec. & Privacy*, 2006.

[35] A. S. Ross and F. Doshi-Velez. Improving the adversarial robustness and interpretability of deep neural networks by regularizing their input gradients. In *AAAI*. AAAI Press, 2018.

[36] B. I. Rubinstein, B. Nelson, L. Huang, A. D. Joseph, S.-h. Lau, S. Rao, N. Taft, and J. D. Tygar. Antidote: understanding and defending against poisoning of anomaly detectors. In *9th ACM SIGCOMM Internet Measurement Conf.*, IMC '09, pp. 1–14, NY, USA, 2009. ACM.

[37] P. Russu, A. Demontis, B. Biggio, G. Fumera, and F. Roli. Secure kernel machines against evasion attacks. In *9th ACM Workshop on AI and Sec.*, AISec '16, pp. 59–69, New York, NY, USA, 2016. ACM.

[38] M. Sharif, S. Bhagavatula, L. Bauer, and M. K. Reiter. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. In *ACM SIGSAC Conf. on Comp. and Comm. Sec.*, pp. 1528–1540. ACM, 2016.

[39] C. J. Simon-Gabriel, Y. Ollivier, B. Schölkopf, L. Bottou, and D. Lopez-Paz. Adversarial vulnerability of neural networks increases with input dimension. *ArXiv*, 2018.

[40] J. Sokolić, R. Giryes, G. Sapiro, and M. R. D. Rodrigues. Robust large margin deep neural networks. *IEEE Trans. on Signal Proc.*, 65(16):4265–4280, 2017.

[41] O. Suciu, R. Marginean, Y. Kaya, H. D. III, and T. Dumitras. When does machine learning FAIL? Generalized transferability for evasion and poisoning attacks. In *27th USENIX Sec.*, pp. 1299–1316, 2018. USENIX Assoc.

[42] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. In *ICLR*, 2014.

[43] F. Tramèr, N. Papernot, I. Goodfellow, D. Boneh, and P. McDaniel. The space of transferable adversarial examples. *ArXiv e-prints*, 2017.

[44] D. Varga, A. Csiszárik, and Z. Zombori. Gradient Regularization Improves Accuracy of Discriminative Models. *ArXiv e-prints ArXiv:1712.09936*, 2017.

[45] N. Šrndic and P. Laskov. Practical evasion of a learning-based classifier: A case study. In *IEEE Symp. Sec. and Privacy*, SP '14, pp. 197–211, 2014. IEEE CS.

[46] B. Wang and N. Z. Gong. Stealing hyperparameters in machine learning. In *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 36–52. IEEE, 2018.

[47] L. Wu, Z. Zhu, C. Tai, and W. E. Enhancing the transferability of adversarial examples with noise reduced gradient. *ArXiv e-prints*, 2018.

[48] H. Xiao, B. Biggio, G. Brown, G. Fumera, C. Eckert, and F. Roli. Is feature selection secure against training data poisoning? In F. Bach and D. Blei, editors, *JMLR W&CP - 32nd ICML*, vol. 37, pp. 1689–1698, 2015.

[49] W. Xu, Y. Qi, and D. Evans. Automatically evading classifiers a case study on PDF malware classifiers. In *NDSS*. Internet Society, 2016.

[50] F. Zhang, P. Chan, B. Biggio, D. Yeung, and F. Roli. Adversarial feature selection against evasion attacks. *IEEE Trans. on Cybernetics*, 46(3):766–777, 2016.

Stack Overflow Considered Helpful! Deep Learning Security Nudges Towards Stronger Cryptography

Felix Fischer, Huang Xiao[†], Ching-Yu Kao[*], Yannick Stachelscheid, Benjamin Johnson, Danial Razar
Paul Fawkesley[◇], Nat Buckley[◇], Konstantin Böttinger[*], Paul Muntean, Jens Grossklags
Technical University of Munich, [†]Bosch Center for Artificial Intelligence
[*]Fraunhofer AISEC, [◇]Projects by IF
{*flx.fischer, yannick.stachelscheid, benjamin.johnson, danial.razar, paul.muntean, jens.grossklags*}*@tum.de*
{*huang.xiao*}*@de.bosch.com,* {*nat, paul*}*@projectsbyif.com,* {*ching-yu.kao, konstantin.boettinger*}*@aisec.fraunhofer.de*

Abstract

Stack Overflow is the most popular discussion platform for software developers. However, recent research identified a large amount of insecure encryption code in production systems that has been inspired by examples given on Stack Overflow. By copying and pasting functional code, developers introduced exploitable software vulnerabilities into security-sensitive high-profile applications installed by millions of users every day.

Proposed mitigations of this problem suffer from usability flaws and push developers to continue shopping for code examples on Stack Overflow once again. This motivates us to fight the proliferation of insecure code directly at the root before it even reaches the clipboard. By viewing Stack Overflow as a market, implementation of cryptography becomes a decision-making problem. In this context, our goal is to simplify the selection of helpful and secure examples. More specifically, we focus on supporting software developers in making better decisions on Stack Overflow by applying nudges, a concept borrowed from behavioral economics and psychology. This approach is motivated by one of our key findings: For 99.37% of insecure code examples on Stack Overflow, similar alternatives are available that serve the same use case and provide strong cryptography.

Our system design that modifies Stack Overflow is based on several nudges that are controlled by a deep neural network. It learns a representation for cryptographic API usage patterns and classification of their security, achieving average AUC-ROC of 0.992. With a user study, we demonstrate that nudge-based security advice significantly helps tackling the most popular and error-prone cryptographic use cases in Android.

1 Introduction

Informal documentation such as Stack Overflow outperforms formal documentation in effectiveness and efficiency when helping software developers implementing functional code. The fact that 78% of software developers primarily seek help on Stack Overflow on a daily basis[1] underlines the usability and perceived value of community and example-driven documentation [2].

Reuse of code examples is the most frequently observed user pattern on Stack Overflow [17]. It reduces the effort for implementing a functional solution to its minimum and the functionality of the solution can immediately be tested and verified. However, when implementing encryption, its security, being a non-functional property, is difficult to verify as it necessitates profound knowledge of the underlying cryptographic concepts. Moreover, most developers are unaware of pitfalls when applying cryptography and that misuse can actually harm application security. Instead, it is often assumed that mere application of any encryption is already enough to protect private data [13, 14]. Stack Overflow users also cannot rely on the community to correctly verify the security of available code examples [9]. Security advice given by community members and moderators is mostly missing and oftentimes overlooked. This is due to only a few security experts being available as community moderators and a feedback system which is not sufficient to communicate security advice effectively. Consequently, highly insecure code examples are frequently reused in production code [17]. Exploiting these insecure samples, high-profile applications were successfully attacked, leading to theft of user credentials, credit card numbers and other private data [13].

While mainly focused on the negative impact of Stack Overflow on code security, recent research has also reported that there is a full range of code snippets providing strong security for symmetric, asymmetric and password-based encryption, as well as TLS, message digests, random number generation, and authentication [17]. However, it was previously unknown whether useful alternatives can be found for most use cases. In our work, we show that for 99.37% of insecure encryption code examples on Stack Overflow a similar secure alternative is available that serves the same use

[1] `https://insights.stackoverflow.com/survey/2016#community`

case. So, why are they not used in a consistent fashion?

We take a new perspective and see implementation of cryptography as a decision-making problem between available secure and insecure examples on Stack Overflow. In order to assist developers in making better security decisions, we apply nudges, a concept borrowed from experimental economics and psychology to attempt altering individuals' behaviors in a predictable way without forbidding any options or significantly changing their economic incentives. Nudging interventions typically address problems associated with cognitive and behavioral biases, such as anchoring, loss aversion, framing, optimism, overconfidence, post-completion errors, and status-quo bias [5, 31]. They have been applied in the security and privacy domain in a successful fashion [4, 5, 7, 19, 22, 32]. In contrast to these approaches, which focused on systems for end-users, we translate the concept of nudges to the software developer domain by modifying the choice architecture of Stack Overflow. It nudges developers towards reusing secure code examples without interfering with their primary goals.

Our designed security nudges are controlled by a code analysis system based on deep learning. It learns general features that allow the separation of secure and insecure cryptographic usage patterns, as well as their similarity-based clustering and use-case classification. Applying this system, we can directly derive a choice architecture that is based on providing similar, secure, and use-case preserving code examples for insecure encryption code on Stack Overflow.

In summary, we make the following contributions:

- We present a deep learning-based representation learning approach of cryptographic API usage patterns that encodes their similarity, use case and security.
- Our trained security classification model which uses the learned representations achieves average AUC-ROC of 0.992 for predicting insecure usage patterns.
- We design and implement several security nudges on Stack Overflow that apply our similarity, use case and security models to help developers make better decisions when reusing encryption code examples.
- We demonstrate the effectiveness of nudge-based security advice within a user study where participants had to implement the two most popular and error-prone cryptographic use cases in Android [13, 17]: nudged participants provided significantly more secure solutions, while achieving the same level of functionality as the control group.

We proceed as follows. After reviewing related work (Section 2), we present our system design that combines deep learning-based representation learning with nudge-based security advice (Sections 3 – 6). Then, we present our model evaluation and user study (Sections 7 & 8), as well as limitations, future work, and conclusions (Sections 9 – 11).

2 Related Work

2.1 Getting Cryptography Right

Acar et al. [2] have investigated the impact of formal and informal information sources on Android application security. With a lab study, they found that developers prefer informal documentation such as Stack Overflow over official Android documentation and textbooks when implementing encryption code. Solutions based on advice from Stack Overflow provided significantly more functional – but less secure – solutions than those based on formal documentation. Work by Fischer et al. [17] showed that 30% of cryptographic code examples on Stack Overflow were insecure. Many severely vulnerable samples were reused in over 190,000 Android applications from Google Play including high-profile applications from security-sensitive categories. Moreover, they have shown that the community feedback given on Stack Overflow was not helpful in preventing reuse of insecure code.

Chen et al. studied the impact of these community dynamics on Stack Overflow in more detail [9]. Based on manual inspection of a subset of posts, they found that (on average) posts with insecure snippets garnered higher view counts and higher scores, and had more duplicates compared to posts with secure snippets. Further, they demonstrated that a sizable subset of posts from trusted users were insecure. Taken together, these works show that developers (by copying and pasting insecure code) are imposing negative externalities on millions of users who eventually bear the cost of apps harboring vulnerabilities [8].

Oliveira et al. focus on developers' misunderstandings of ambiguities in APIs (including cryptography), which may contribute to vulnerabilities in the developed code [26]. They studied the impact of personality characteristics and contextual factors (such as problem complexity), which impact developers' ability to identify such ambiguities. Likewise, Acar et al. [1] investigated whether current cryptographic API design had an impact on cryptographic misuse. They selected different cryptographic APIs; including some particularly simplified APIs in order to prevent misuse. However, while indeed improving security, these APIs produced significantly less functional solutions and oftentimes were not applicable to specific use cases at all. As a consequence, developers searched for code examples on Stack Overflow again.

Nguyen et al. [25] developed FixDroid, a static code analysis tool integrated in Android Studio which checks cryptographic code flaws and suggests quick fixes.

2.2 Security Nudges

Wang et al. [32] implemented privacy nudges on Facebook in order to make users consider the content and audience of their online publications more carefully, as research has

shown that users eventually regret some of their disclosure decisions. They found that a reminder nudge about the audience effectively and non-intrusively prevents unintended disclosure. Almuhimedi et al. [6] implemented an app permissions manager that sends out nudges to the user in order to raise awareness of data collected by installed apps. With the help of a user study they were able to show that 95% of the participants reassessed their permissions, while 58% of them further restricted them. Liu et al. [23] created a personalized privacy assistant that predicts personalized privacy settings based on a questionnaire. In a field study, 78.7% of the recommendations made by the assistant were adopted by users, who perceived these recommendations as usable and useful. They were further motivated to review and modify the proposed settings with daily privacy nudges.

2.3 Deep Learning Code

Fischer et al. [17] proposed an approach based on machine learning to predict the security score of encryption code snippets from Stack Overflow. They used *tf-idf* to generate features from source code and trained a support vector machine (SVM) using an annotated dataset of code snippets. The resulting model was able to predict the security score of code snippets with an accuracy of 0.86, with precision and recall of 0.85 and 0.75, respectively. However, security predictions were only available for the complete code snippet. It did not allow indicating and marking specific code parts within the snippet to be insecure. This lack of explainability is detrimental for security advice.

Xiaojun et al. [33] introduced neural network-based representation learning of control flow graphs (CFGs) generated from binary code. Using a Siamese network architecture they learned similar graph embeddings using *Structure2vec* [11] from similar binary functions over different platforms. These embeddings were used to detect vulnerabilities in binary blobs by applying code-similarity search. Their approach significantly outperformed the state-of-the-art [16] in both, efficiency and effectiveness, by providing shorter training times and higher area under the curve (AUC) on detecting vulnerabilities. The approach does not allow identification and description of code parts within binary functions that cause the vulnerabilities. To allow better explainability, we depend on our new approach to provide statement-level granularity. It enables identifying and classifying multiple code patterns within a single function.

Li et al. [21] developed VulDeePecker, a long short-term memory (LSTM) neural network that predicts buffer overflows and resource management error vulnerabilities of source code gadgets. Code gadgets are backward and forward slices, considering data and control flow, that are generated from arguments used in library function calls. Further, they use *word2Vec* to create embeddings for the symbolic representation of code gadgets. These embeddings are then used together with their security label to train a bi-directional LSTM. VulDeePecker outperforms several pattern-based and code similarity-based vulnerability detection systems with respect to false positive and false negative rates. However, their LSTM model has a very long training time. Our convolutional approach leverages transfer learning to achieve much faster training.

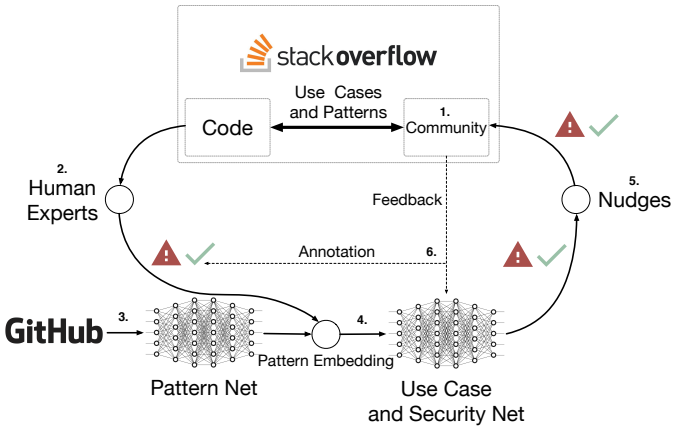


Figure 1: Learn-to-Nudge Loop Overview

3 Overview

We present an overview of our system design for nudge- and deep learning-based security advice on Stack Overflow in Figure 1. It depicts a learn-to-nudge loop that represents the interaction and interference of the community behavior, classification models and proposed security nudges on Stack Overflow. The community behavior on Stack Overflow (1) triggers the loop by continuously providing and reusing code examples that introduce new use cases and patterns of cryptographic application programming interfaces (APIs). In the initial step (2), a representative subset of these code examples is extracted and annotated by human experts. The annotations provide ground truth about the use cases and security of cryptographic patterns in the given code. Then (3), a representation for these patterns is learned by an unsupervised neural network based on open source projects provided by GitHub. In combination with the given annotations, the pattern embeddings are used to train an additional model to predict their use cases and security (4). Based on these predictions, we can apply security nudges on Stack Overflow by providing security warnings, reminders, recommendations and defaults for encryption code examples (5). Further, we allow assigned security moderators[2] within the community to annotate unknown patterns and provide feedback to predictions of our models (6). Therefore, our system creates a

[2] https://stackexchange.com/about/moderators

learn-to-nudge loop that is supposed to iteratively improve the classification models, which in turn help improving the security decisions made by the community and the security of code provided on Stack Overflow.

4 Nudge-Based System Design

We apply five popular nudges [30, 31] and describe their translation to security advice in this section.

Simplification A simplification nudge promotes building upon existing and established infrastructures and programs. We apply this nudge by integrating our system in Stack Overflow, a platform that is already used by the majority of software developers worldwide. By integrating developer tools on a platform that is already used by almost everyone, we unburden developers from installing additional tools. Moreover, it allows us to create awareness of the problem of cryptographic misuse in general.

Warnings A warning nudge aims at raising the user's attention in order to counteract the natural human tendency towards unrealistic optimism [5]. We apply this nudge by integrating security warnings on Stack Overflow. Whenever an insecure code example has been detected, a warning is displayed to the developer to inform about the security problem and potential risks in reusing the code sample.

Increases in Ease and Convenience (IEC) Research has also shown that users oftentimes discount security warnings. However, if they additionally describe available alternative options to make a less risky decision, warnings tend to be much more effective [5]. Therefore, our design combines security warnings with recommendations for similar code examples with strong cryptography. With this nudge, we make code examples with better security visible to the user. To provide an easy choice, we present the recommended code examples by displaying a list of the related posts. This aims at encouraging the user to consider the recommendations as it only demands clicking on a link.

Reminders Users might not engage in the expected conduct of paying attention to the warning and following the recommendations. This might be due to inertia, procrastination, competing priorities, and simple forgetfulness [5]. Oftentimes seeking functional solutions is considered as a competing priority to secure solutions [2]. Therefore, we apply a reminder nudge, which is triggered whenever the user copies an insecure code example.

Defaults The default nudge is the most popular and effective nudge in improving decision-making. Popular examples are automated enrollment in healthcare plans or corporate wellness programs, or double-sided printing which can promote environmental protection [5]. We apply this nudge by up-ranking posts that only contain secure code examples in Stack Overflow search results by default.

The goal of our approach is to thoughtfully develop a new user interface (UI) design that implements the proposed nudges (see Section 6) and to test whether it improves developer behavior on Stack Overflow. Please note that we do not intend to comparatively evaluate multiple UI candidates for our design patterns to identify the most effective one. We consider this out of scope for this paper and leave this task for future work.

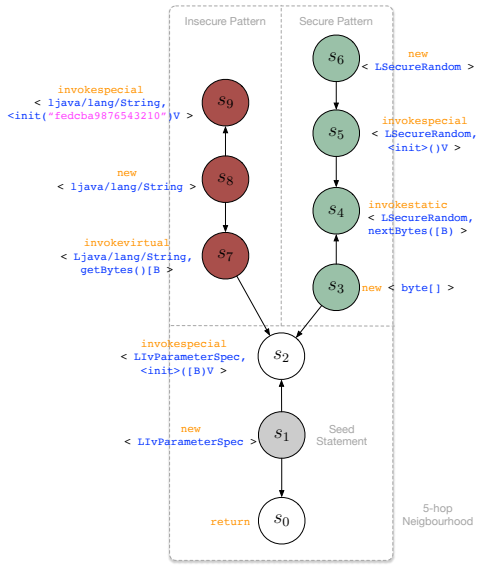
5 Neural Network-Based Learning of Cryptographic Use Cases and Security

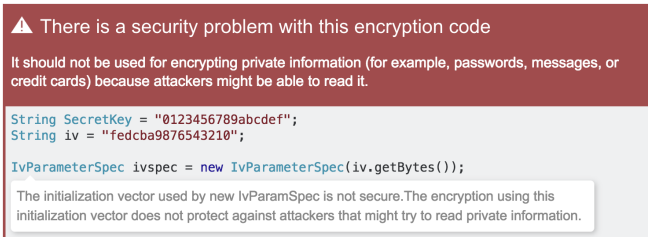
The nudge-based system design requires algorithmic decisions about the security and similarity of code examples. In order to display security warnings, code examples have to be scanned for encryption flaws. To further recommend helpful alternatives without common encryption problems, Stack Overflow posts have to be scanned for similar examples with strong cryptography.

Due to *Simplification* (see Section 4), we already chose a platform that provides us with a large amount of secure and insecure samples that contain cryptographic API usage patterns to learn from in order to design the code analysis approach [17]. Instead of defining rule-based algorithms [12, 13, 20] that would have to be updated whenever samples with unknown patterns are added to Stack Overflow, we simplify and increase the flexibility of our system by applying deep learning to automatically learn the similarity, use-case and security features from the ever-increasing dataset of available code on Stack Overflow. Based on the learned features, our models are able to predict insecure code examples and similar but secure alternatives that serve the same use case. However, newly added code examples that provide unknown use cases and security flaws might be underrepresented in the data and therefore difficult to learn. Therefore, we apply transfer learning where we reuse already obtained knowledge that facilitates learning from a small sample set of a similar domain.

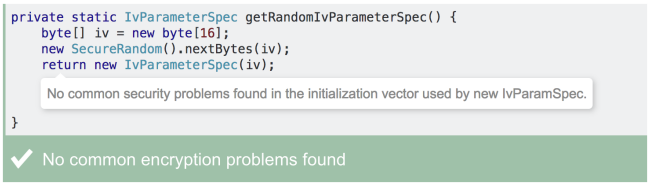
5.1 Cryptographic Use Cases

Stack Overflow offers a valuable source for common use cases of cryptographic APIs in Android. As developers post questions whenever they have a particular problem with an API, a collection of error-prone or difficult cryptographic problems is aggregating over time. Moreover, frequencies of similar posted questions, view counts, and scores of questions posted on Stack Overflow indicate very common and important problems developers encounter when writing





(a) Example for an insecure pattern: The initialization vector (IV) is created from a static string value stored in the code.



(b) Example for a secure pattern: A secure random source is used to generate the IV.

Figure 2: Example for a secure and insecure usage pattern of *new IvParameterSpec*. It shows the program dependency graph (PDG) of the 5-hop neighborhood of the seed statement s_1 for the secure and insecure code example displayed in (a) and (b). Next to each node in the graph we provide the shortened signature of the related statement, highlighting a subset of its attributes we store in the feature vector. Bytecode instruction types are highlighted yellow, Java types blue and constants magenta.

security-related code. Therefore, Stack Overflow can be seen as a dataset of different cryptographic use cases that are frequently required in production code. Previous work identified the most popular and error-prone use cases of cryptography in Android apps [17]. The authors scanned Stack Overflow for insecure code examples that use popular cryptographic APIs, e g. Oracle's Java Cryptography Architecture (JCA), and detected their reuse in Android applications. We summarize the identified use cases in Table 1.

Use Case Identifier	Usage Pattern Description	API Seed Statement
Cipher	Initialization of cipher, mode and padding	*Cipher.getInstance*
Key	Generation of symmetric key	*new SecretKeySpec*
IV	Generation of initialization vector	*new IvParameterSpec*
Hash	Initialization of cryptographic hash function	*MessageDigest. getInstance*
TLS	Initialization of TLS protocol	*SSLContext.getInstance*
HNV	Setting the hostname verifier	*setHostnameVerifier*
HNVOR	Overriding the hostname verification	*verify*
TM	Overriding server certificate verification	*checkServerTrusted*

Table 1: Common cryptographic use cases in Android

5.2 Learning API Usage Patterns

In order to predict similarity, use case and security of encryption code, we need to learn a numerical representation of the related patterns that can be understood by a neural network. Therefore, our first step is learning an embedding of cryptographic API usage patterns.

Usage Pattern As shown in Table 1, a cryptographic API element, e. g., *javax.crypto.Cipher.getInstance*, can have different usage patterns that belong to the same use case. A usage pattern consists of a particular API element, all statements it depends on, and all its dependent statements within the given code. In other words, a pattern can be seen as a subgraph of the PDG, which represents the control and data dependencies of statements. The subgraph is created by pruning the graph from anything but the forward and backward slices of the API element, as shown in Figure 2. We call this element the *seed statement*. This pruned graph can become very large and therefore might contain noise with respect to the identification of patterns. Our goal is to learn an optimal representation of usage patterns that allows accurate classification of their use cases and security. Ideally, the related subgraph is minimized to a *neighborhood* of the seed statement in the pruned PDG such that it provides enough information to solve the classification tasks.

Neighborhood Aggregation Our approach learns *pattern embeddings* for the K-hop neighborhood of cryptographic API elements within the PDG, as shown in Figure 2. To generate these embeddings we use the *neighborhood-aggregation* algorithm provided by *Structure2vec* [11]. This method leverages node features (e. g., instruction types of a statement node) and graph statistics (e. g., node degrees) to

inform their embeddings. It provides a convolutional method that represents a node as a function of its surrounding neighborhood. The parameter K allows us to search for a neighborhood that optimally represents usage patterns to solve given classification tasks. In other words, we learn the code representation in a way such that its features improve use case and security prediction of the code. As we will show throughout this work, this representation is very helpful for classifying cryptographic API usage patterns. We further argue that the learned pattern representation is not restricted to cryptographic APIs, as the used features are general code graph properties.

Neighborhood Similarity We learn pattern embeddings such that similar patterns have similar embeddings by minimizing their distance in the embedding space. Therefore, next to the neighborhood information, pattern embeddings additionally encode their similarity information. On the one hand, this allows us to apply efficient and accurate search for similar usage patterns on Stack Overflow [33]. On the other hand, we can transfer knowledge from the similarity domain to the use case and security domain. This knowledge transfer is leveraged by our use case and security classification models.

Code similarity is very helpful to predict code security. Therefore, we expect that the similarity feature of our pattern embeddings will improve the accuracy and efficiency of the security classification model. However, code similarity is oftentimes not enough for predicting security. Therefore, the main effort of our classification models lies in learning the additional unknown conditions where code similarity becomes insufficient.

To learn our embeddings, we apply a modified architecture of the graph embedding network *Gemini* [33].

5.3 Feature Engineering

The embedding network should learn a pattern embedding that is general enough to allow several classification tasks. This means that the embedding has to be learned from general code features or attributes, e. g., statistical and structural features [33] from each statement within the PDG representation of the code. Further, pattern embeddings should represent very small neighborhoods. As we want to minimize the neighborhood size K, patterns might consist only of a few lines of code. Therefore, considering only graph statistics as features might not be sufficient and may result in similar features for dissimilar patterns. In order to overcome these insufficiencies, we additionally combine structural and statistical with lexical and numerical features for each statement in a neighborhood.

Structure and Statistics We first create the PDG of the given input program using WALA[3], a static analysis framework for Java. Note that WALA creates a PDG for each Java method. Then, we extract the resulting statistical and structural features for each statement. We store the bytecode instruction type of a statement using a one-hot indicator vector. Additionally, we store the count of string and numerical constants that are used by the statement. We further add structural features by storing the offspring count and node degree of a statement in the PDG [33]. Finally, we store the indexes of the statement's direct neighbors in the graph.

Element Names and String Constants Method and field names of APIs are strings and have to be transformed into a numerical representation first. We learn feature vectors for these tokens by training a simple unsupervised neural network to predict the Java type that defines the given method or field name. Thereby, each name is represented in a one-hot encoding vector with dimension 23,545, corresponding to the number of unique element names provided by the cryptographic APIs [17]. To learn features, we use a network architecture with one hidden layer and apply categorical cross-entropy as a loss function during training. Finally, we apply the trained model on all names and extract the neurons of the hidden layer as they can be seen as learned features necessary to solve the classification task. This way, each name obtains a unique feature vector which preserves its type information. We use the same approach for learning feature vectors for the 763 unique string constants given by the APIs.

5.4 Pattern Embedding Network

Many code examples on Stack Overflow typically do not provide sound programs as they mostly consist of loose code parts [17]. In contrast to complete programs, compiling these partial programs might introduce multiple types of ambiguities in the resulting PDG such that the extracted statement features x_u are not sound [10]. Whenever we generate sound and unsound features x_s, x_u from a complete and a partial program, respectively, that provide the same usage pattern for a given seed statement, both sets of feature vectors extracted from the patterns might be different. Therefore, we need to learn a representation for patterns that preserves their similarity properties independently from the shape of the containing program. With a Siamese network architecture [33], we can learn similar pattern embeddings independently from the completeness of the code example. It learns embeddings from similar and dissimilar input pairs. We create similar input pairs by extracting sound and unsound features for the same pattern and dissimilar pairs by extracting sound and unsound features from different patterns. The

[3] `https://github.com/wala/WALA`

Algorithm 1 Neighborhood-aggregation algorithm

Input: PDG $G(V,E)$ input features $\{x_v, \forall v \in V\}$;
Output: Pattern embedding $p_v, \forall v \in V$

$\phi_v^0 \leftarrow 0, \forall v \in V$,
for $k = 1...K$ **do**
  for $v \in V$ **do**
    $\phi_{N(v)}^k \leftarrow AGGREGATE(\phi_n^{k-1}, \forall n \in N(v))$
    $\phi_v^k \leftarrow \tanh(W_1 x_v \cdot \sigma(\phi_{N(v)}^k)$
return $\{p_v = W_2 \phi_v^K, \forall v \in V\}$

trained model will then generate similar embeddings independently from the completeness of the program.

The pattern embeddings are generated with *Structure2vec* as depicted in Algorithm 1. We provide the abstract description of the algorithm and refer to *Gemini's* neural network architecture that gives information about its implementation, which we use as the basis for our approach. The update function calculates a pattern embedding p_v for each feature vector x_v of statements (i. e., nodes) $v \in V$ in the PDG $G(V,E)$. An embedding p_v is generated by recursively aggregating previously generated embeddings $\{\phi_n^{k-1}, \forall n \in N(v)\}$ of direct neighbors $N(v)$ in the graph, combining it with the weighted feature vector x_v. Unlike *Gemini*, which outputs an aggregation of p_v to return an embedding for the complete graph, our network returns the set of pattern embeddings $P = \{p_v, \forall v \in V\}$.

We give an overview of the pattern embedding network in Figure 3. Here, the insecure pattern $G(x_3, x_4, x_5, E)$ informs the embedding of its direct neighbors in each iteration step, finally informing the seed statement in iteration $k = 2$. After this step, the seed statement knows that it is part of an insecure pattern and its embedding preserves this information accordingly. We extract the pattern embedding ϕ_1^K of the seed statement and apply weights W_2. Note, we train W_2 based on classification loss of aggregated pattern similarity p_a as explained in Section 5.5. However, we use the trained model to generate and output embeddings for each individual pattern p_v in the graph (see Figure 3).

5.5 Training

For unsupervised training of pattern embeddings, we need to generate similar and dissimilar input pairs from data that provides ground truth. We use two different sets of PDGs that are compiled from the same source code. One set S contains the sound graph representations of the code, the other one the unsound graphs U. A sound graph $G_s(V_s, E_s)$ in the first set is compiled from a complete program using a standard Java compiler. An unsound graph $G_u(V_u, E_u)$ in the second set is generated by a partial compiler [10] that compiles each Java class of a program individually. Then we construct the feature vectors $X_s = \{x_s\}_{v_s \in V_s}$ and $X_u = \{x_u\}_{v_u \in V_u}$ for all

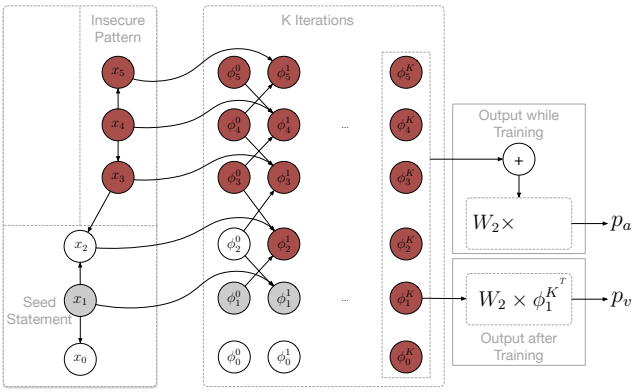


Figure 3: Pattern embedding network overview

statements v in the respective graphs.

To obtain ground truth for similar and dissimilar usage patterns we need to create similar pairs $\left(\langle v_s, v_u \rangle, 1\right)$ and dissimilar pairs $\left(\langle v_s, v_u \rangle, -1\right)$. However, we do not have information about the relationship of statements in $G_u(V_u, E_u)$ with statements $G_s(V_s, E_s)$, which is necessary to create these pairs. Note that source code statements do not correspond one-to-one with statements in the PDG. The compiler may divide a source code statement into multiple instructions, which may have different associated statements in the PDG. Since we use different compilers, the resulting PDG statements in V_s and V_u may look different even though they represent the same source code. Therefore, we aggregate all pattern embeddings from a method into p_a and use the resulting embedding for training. The network calculates the loss based on the cosine similarity of the aggregated pattern embedding pairs $\{\langle p_a^s, p_a^u \rangle\}_{x_s, x_u \in X_s, X_u}$ and their given similarity label $y \in \{-1, 1\}$

We downloaded 824 open source Android apps from GitHub and compiled the complete and sound graph $G_s(V_s, E_s)$ for each method. Further, we used the partial compiler to obtain the unsound graph for each method $G_u(V_u, E_u)$. After creating the feature vectors X_s and X_u from the graphs, similar method pairs $\left(\langle X_s, X_u \rangle, 1\right)$ were created by extracting X_s and X_u from the same source code, and dissimilar $\left(\langle X_s, X_u \rangle, -1\right)$ by extracting X_s and X_u from different source code. From the 824 downloaded apps, we extracted 91,075 methods to create 157,162 input pairs in total. These pairs have been split up into the training and validation set, where 80% have been randomly allocated for training and 20% for validation. Note that the intersection of both sets is empty.

5.6 Learning Use Cases and Security

From a given source code example, we want to be able to predict the cryptographic use case and security of patterns within the code. We apply transfer learning by reusing the

previously learned pattern embedding that already encodes their similarity information.

Pattern embeddings are learned unsupervised and we can obtain almost arbitrarily large training datasets from open source projects. However, code examples on Stack Overflow provide a very different distribution of data [17]. Many use case and security classes are under- or overrepresented and availability of encryption code examples is limited in general. We transfer knowledge from the similarity domain to the use case and security domain in order to tackle these problems. We argue that the similarity information preserved in our pattern embeddings will be helpful for classifying their use cases and security.

5.7 Labeling

We extracted 10,558 code examples from Stack Overflow[4] by searching for code that contained at least one of the seed statements. Each code sample has been manually reviewed in order to label use case and security of the contained usage patterns. Labeling was done by two security experts individually applying the labeling rules given by [12, 13, 17]. This leads to conservative binary security labeling, which might at times be too strict. For instance, depending on the context, MD5 can be the better trade-off and secure enough. However, our approach aims at developers that are layman in cryptography and we consider binary classification preferable to encourage safe defaults.

Initially, 100 samples for each of the different seed statements have been selected randomly to apply dual control labeling. After clearing up disagreements, the remaining samples have been annotated individually to speed up the labeling. The whole process took approximately 10 man days to complete. To evaluate individual annotation accuracy, we randomly selected 200 samples from both experts and report agreement of 98.32% on given labels. We further publish the annotated dataset in order to allow verification of annotation accuracy and reproduction of our results. Please refer to Appendix C for further details on the annotation process.

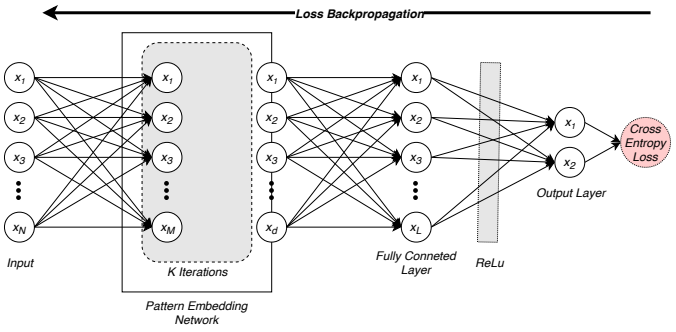


Figure 4: End-to-End Network Architecture

[4] I.e., from the Official Stack Overflow Data Dump.

5.8 End-to-end Architecture

We introduce an architecture that allows classification of different uses cases and security, while improving the pattern-embedding model in order to forward optimized code representations to the classification layer (see Figure 4).

To achieve this, we add a fully connected layer with dimension 1,024 using Rectified Linear Unit (ReLu) as the activation function on top of the pattern-embedding network. We use a softmax layer for binary and multi-class classification and trained our network to optimize cross-entropy loss. Applying transfer learning, we initialize the pattern-embedding network using the previously learned weights for pattern similarity (see Section 5.4). The end-to-end network now connects the pattern-embedding network with the classification layers. Within training, the latter backpropagates cross-entropy loss from the classification task all the way to the input of the pattern-embedding network. This allows the similarity network to adjust the pattern representation in order to better perform on the classification. Therefore, both coupled networks now generate a new pattern representation for the given classification problem in such a way that it is optimally solved.

For instance, security classification of *Cipher, Key* and *TM* rely on very different features. Only using the pre-trained "static" pattern embeddings might therefore be disadvantageous for some use cases. However, by dynamically customizing the pattern embedding with respect to classification loss minimization, the network learns a code representation that preserves the necessary code features to improve classification.

5.9 Training

The Stack Overflow dataset provides 16,539 pattern embeddings extracted from 10,558 code examples. Note that a single code example might contain several patterns, e. g. *IV, Key* is used to initialize *Cipher*. We test pattern embeddings generated from several models that were trained on different neighborhood sizes K and different output dimensions d for the embeddings. Thereby, we search for the optimal hyperparameter K and d to achieve the best performance on both classification tasks. We first train the network to learn the use case identifier of pattern embeddings using the complete dataset. Then, we train a different model to learn the security labels. Here, patterns with the same security label belong to the same class independently of their use case. Finally, we divide the dataset into combinations of several use case classes, testing the effect on performance of security prediction.

6 Security Nudges

The neural network architecture described in the previous section provides everything needed to apply the security nudges on Stack Overflow. In this section, we explain the design of each nudge including its implementation on Stack Overflow and how it applies the predictions from the similarity and classification models[5].

6.1 Security Warnings

Whenever an insecure code example is detected, a security warning, as shown in Figure 8, which surrounds the code, is displayed to the user. The warning is triggered by the prediction result of the security model that classifies each pattern in the snippet.

The difficulties in designing effective security warnings are widely known and have been extensively investigated. We base our approach on the design patterns of Google Chrome's security warning for insecure server communication, whose effectiveness has been comprehensively field-tested [3]. The header of the warning informs the user that a security problem has been detected in the encryption code of the sample. Note that we assume users with a very diverse background, knowledge and expertise in cryptography. Users and even experienced developers might not be aware of flawed or out-dated encryption that does not provide sufficient security. Therefore, we inform the user about the consequences that might occur when reusing insecure code examples in production code, e. g., private information might be at risk in an attack scenario.

We further provide code annotations for each seed statement in the code whose usage pattern has been classified as insecure (see Figure 2). The annotation is attached below and points at the statement. It gives further information about the statement, while additionally highlighting the consequences of reusing it. In order to select the correct annotation for the insecure statement, we apply use case prediction of the related pattern. Each use case identifier has an assigned security annotation to be displayed in the code snippet.

6.2 Security Recommendations

Security warnings should always offer a way out of a situation where the user seems to be unable to continue with her current action due to the warning. Whenever the user decides to follow the advice given by the warning, she would refuse to reuse the code example that was originally considered a candidate for solving her problem. In this situation, she has been thrown out of her usual user pattern as she has to restart searching for another example. Therefore, for each insecure code example, we recommend a list of similar examples, as shown in Figure 8, that serve the same use case and provide stronger encryption. Ideally, the user would only have to click on a single link to the recommended alternative. Our nudge design pattern does not claim that the recommended code is generally secure, as it still might contain insecure patterns that are unknown to the model. However, for simplicity, we refer to code examples, which do not contain any detected insecure patterns and do contain detected secure patterns, as *secure* code examples throughout the paper.

We create this list of recommendations by applying similarity search, use case and security prediction of usage patterns in code examples. We start with predicting the use case of each insecure usage pattern. I_q contains all insecure use cases of a method $q \in M_q$, where M_q is the set of query methods in the snippet. We create the set $\{I_q\}_{q \in M_q}$, which consists of the sets of insecure use cases over all methods in the snippet. Then, we generate the set of aggregated pattern embeddings $\{e_q\}_{q \in M_q}$, as described in Section 5.4. Afterwards, we analogously create the set of secure use cases for all target methods $\{S_t\}_{t \in M_t}$ where M_t is the set of methods available on Stack Overflow that only contain usage patterns our model has classified as secure. Likewise, we create the aggregated pattern embeddings $\{e_t\}_{t \in M_t}$. We rank M_t for given I_q, S_t and e_q, e_t based on ascending Jaccard distance $d_J(I_q, S_t)$, ranking pairs with the same distance using cosine similarity $cos(e_q, e_t)$. We create the ranked list of recommended posts R by adding the related Stack Overflow post for each t of the top-fifty results in the ranking. Beneath the security warning, we display a scrollable list of R, as displayed in Figure 8. Each post is displayed by showing the title of the related question. When the user clicks on the title, a new browser tab opens and the web page automatically scrolls down to the recommended code example, highlighting it with a short flash animation.

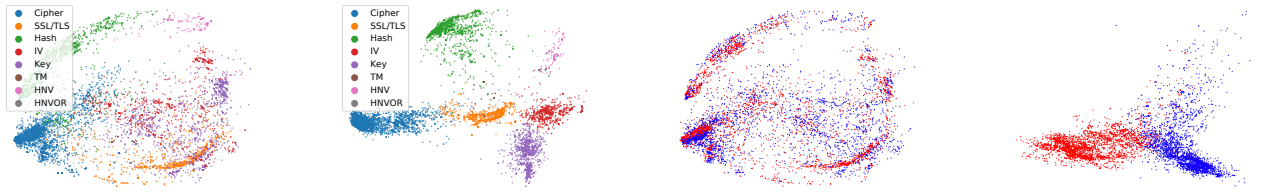
Recommended examples are displayed inside a green box, annotated with a check mark and message informing the user that no common encryption problems have been found within the code. This way, we avoid declaring the code example to be secure, which would be a too strong claim. However, the statement intends to be strong enough to reach the users and make them follow the advice. Similar to warnings, we provide code annotations for each statement in the code whose usage pattern has been classified as secure.

6.3 Security Reminders and Defaults

We further caution the user – in addition to prompting the security warning and recommendations – by blurring out the remainder of the web page, whenever a copy event of an insecure code example is triggered.

We additionally apply a search filter which up-ranks posts that only contain secure code examples. Posts with insecure code examples are appended to the list of secure posts. The original ranking of posts within its security class is maintained. This approach lowers the risk of reusing code exam-

[5] We provide further example figures of our nudges in the Appendix.



(a) Use cases before training (b) Use cases after training (c) Security before training. (d) Security after training.

Figure 5: Visualizing the pattern embeddings of different use cases and security using PCA. Each color indicates one use case in (a) and (b), and security in (c) and (d). The legend provides the use case identifier.

ples that have been predicted to be insecure. This also means that whenever a post consists of secure and insecure samples it is ranked lower than posts with only secure samples.

7 Model Evaluation

7.1 Pattern Similarity

We evaluate the learned pattern embeddings by measuring cosine similarity for all pairs in the validation set and calculate the receiver operating characteristic curve (ROC) given the similarity label $y \in \{-1, 1\}$ for each pair. Our approach reaches an optimal AUC of 0.978, which slightly outperforms *Gemini* with AUC of 0.971. *Gemini* was originally applied to similarity prediction of binary functions by learning embeddings for CFGs and may not be a suitable benchmark.

We observe that the model converges already after five epochs. For the remaining epochs, AUC stays around 0.978 and does not improve significantly. This allows for a very short training time, as five epochs only need 27 minutes on average on our system[6]. However, we choose the model with the best AUC for generating the pattern embeddings.

7.2 Use Case Classification

For training the use case and security models, we apply the dataset consisting of 16,539 pattern embeddings extracted from Stack Overflow split up into subsets for training (80%) and validation (20%). Note that the validation set is constructed such that none of its samples appear in the training set. Therefore, we evaluate the performance of use case prediction on unseen pattern embeddings.

Visualization To illustrate the transfer learning process, we plot the pattern embeddings in $2D$ using principal component analysis (PCA) before and after the training of the classification model. Figure 5(a) shows the complete set of pattern embeddings before training, displayed in the color of their use case. We observe that some use cases already build clusters in the plot, while others appear overlapping and intermixed. Therefore, we apply an additional neural network on top that leverages supervision on use cases in addition to the similarity knowledge preserved in the input embeddings. Figure 5(b) plots the pattern embeddings again after supervised training of the model. Here, we input the initial pattern embeddings into the trained model and extract the last hidden layer of the network to obtain new embeddings that preserve information about their use case. We observe that the new embeddings now create dense and separable clusters for each use case in the plot. The network has moved pattern embeddings that belong to the same use case closer together, and the resulting clusters further away from each other in the embedding space.

Accuracy The promising observations from the visualization of pattern embeddings are confirmed by the accuracy results of the classification model. We performed a grid search that revealed the optimal neighborhood size of $K = 5$. The average AUC for predicting the different use cases already achieves its optimum of 0.999 after 20 epochs. As already indicated by the PCA plots, pattern embeddings provide a very good representation of use cases as the average AUC for all classes before training (epoch zero) is already above 0.998. However, precision and recall of *IV, HNVOR* and *TM* start below 0.878 and have been improved up to above 0.986 within 30 epochs of training.

7.3 Security Classification

Visualization We start again with illustrating the transfer-learning process for security classification by plotting pattern embeddings before and after training. Figure 5(c) displays pattern embeddings before training with their respective security score, Figure 5(d) plots the new embeddings after training. Samples that were labeled as secure are depicted in blue, insecure samples in red. When comparing Figure 5(a) and Figure 5(c), we can already observe several secure and insecure clusters within the use case clus-

[6] Intel Xeon E5-2660 v2 ("Sandy Bridge"), 20 CPU cores, 240GB memory

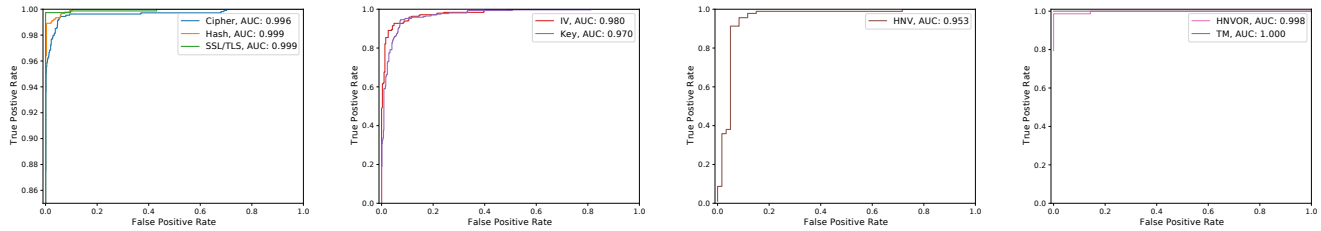


Figure 6: ROC for security classification of different use cases. The legend provides use case identifier and respective AUC.

ters, e. g., *Hash*, *Cipher* and *SSL/TLS*. However, again many secure and insecure samples appear to have a wide distribution because PCA does not plot them in dense clusters. After training the security classification model, we input the complete set of pattern embeddings and plot the last layer of the neural network for each sample in Figure 5(d) again. Now, we observe dense and separated clusters for secure and insecure samples. The network has adjusted the pattern embeddings such that samples within both security classes have been moved closer together in the embedding space. Samples with different security have been moved further away from each other, finally dividing samples into two security clusters.

Accuracy We trained a single model using the labeled dataset of 16,539 pattern embeddings. Thereby, a single model learns security classification for all use cases. Our grid search revealed $K = 5$ as the optimal neighbourhood size. The model provides a good fit because training and validation loss already converge after 50 epochs. A single epoch takes 0.58 seconds on average on our system, resulting in roughly five minutes for complete training time. We plot ROC curves for security prediction for each use case class in Figure 6. We observe that the three use cases *Hash*, *Cipher* and *SSL/TLS* that provide the largest percentage of samples in the dataset achieve the best results. The model achieves very good classification accuracy with AUC values of 0.999, 0.996 and 0.999, respectively, similar to *HNVOR* and *TM*. However, performance drops marginally for *IV*, *Key* and *HNV* to 0.980, 0.970 and 0.953, respectively.

Comparison In Table 2, we compare our approach on security prediction on Stack Overflow with [17], where the authors use *tf-idf* to create a feature vector as a representation for the complete input snippets and to train a SVM predicting its binary security score. Our deep learning approach (marked as CNN in the table) significantly outperforms their classifier in all use cases; especially *IV, Key* and *HNVOR*, where security evaluation heavily relies on data and control flow. In contrast to our approach, the work by Fischer et al. [17] does not inform the learning model about these properties, but solely relies on lexical features.

Moreover, our deep learning approach allows a higher level of explainability to the user. While [17] can only report security warnings for the complete snippet, our more fine-grained approach is able to directly highlight statements in the code and provides annotations that explain the security issue. Since we learn a representation of code patterns that allows prediction of different code properties beyond security, we can provide this additional explanation, which is crucial for developer advice.

	CNN		tfidf+SVM	
	AUC-ROC	Explanation	AUC-ROC	Explanation
Cipher	0.996	SW, CA	0.960	SW
Hash	0.999	SW, CA	0.956	SW
TLS	0.999	SW, CA	0.902	SW
IV	0.980	SW, CA	0.881	SW
Key	0.970	SW, CA	0.886	SW
HNV	0.953	SW, CA	0.922	SW
HNVOR	0.998	SW, CA	0.850	SW
TM	1.000	SW, CA	0.982	SW

Table 2: Performance and explainability comparison of security prediction on Stack Overflow. SW: Provides security warnings for the complete snippet. CA: Additionally provides code annotation that explains the issue in detail.

7.4 Recommendations

We applied our trained models in order to evaluate whether Stack Overflow provides secure alternative code snippets, which preserve the use case and are similar to detected insecure code examples. Thereby, we extracted all methods from the complete set of 10,558 snippets, generated their aggregated embeddings and separated them into two sets. The first set contains all 6,442 distinct insecure *query* embeddings and the second one all 3,579 distinct secure *target* embeddings. We created these two sets by applying the security model and predicted the security of each pattern within a given method. Finally, we ranked the embeddings based on their Jaccard distance, applying the use case model, and cosine similarity, as described in Section 6.2. We found 6,402 (99.37%) query methods that have Jaccard distance of 0.0 to at least one target method. This means that for almost every insecure method, a secure one exist on Stack Overflow

that serves the same use case. When additionally demanding code similarity, we found 6,047 (93.86%) query methods with a cosine similarity above 0.81 and 4,805 (75.58%) query methods with a similarity above 0.9 with at least one target method.

8 Evaluation of Security Nudges

To evaluate the impact of our system including the security nudges on the security of programming results, we perform a laboratory user study. Thereby, participants had to solve programming tasks with the help from Stack Overflow.

8.1 User Study Setup

Participants were randomly assigned to one of two treatment conditions. For the nudge treatment, we provided security warnings (Figure 2a) for insecure code examples, recommendations for secure snippets (Figure 2b) and recommendations lists attached to each warning (Figure 8). Further, security reminders were enabled. In the control treatment, all security nudges on Stack Overflow were disabled.

Participants were advised to use Stack Overflow to solve the tasks. In all treatments, we restricted Stack Overflow search results to posts that contain a code example from the set of 10,558 code examples we extracted from Stack Overflow[7]. Further, we applied a whitelist filter to restrict access to Stack Overflow in the Chrome browser. Any requests to different domains were redirected to the Stack Overflow search page. Participants were provided with the Google Chrome browser and Eclipse pre-loaded with two Java class templates. Both class templates provided code skeletons that were intended to reduce the participants' workload and simplify the programming tasks. Using additional applications was prohibited. All tasks had to be solved within one hour. We avoided security or privacy priming during the introduction and throughout the study. Moreover, we did not name or explain any of the security nudges on Stack Overflow.

8.2 Tasks

All participants had to solve five programming tasks related to symmetric encryption and certificate pinning. We chose these two use cases as they provide the most error-prone cryptographic problems in Android [17].

Symmetric Encryption The first three tasks dealt with initializing a symmetric cipher in order to encrypt and decrypt a message. Task *Cipher*: a symmetric cipher had to be initialized by setting the algorithm, block mode and padding. The main security pitfalls in this task are choosing a weak cipher and block mode. Task *Key*: a symmetric cryptographic key had to be generated. Participants had to create a key having the correct and secure key length necessary for the previously defined cipher. It had to be generated from a secure random source and should not have been stored in plaintext. Task *IV*: an initialization vector had to be instantiated. Like key generation, this task is particularly error-prone as choosing the correct length, secure random source and storage can be challenging.

Certificate Pinning Within these two tasks, a SSL/TLS context had to be created to securely communicate with a specific server via HTTPS. In the end, the program should have been able to perform a successful GET request on the server, while denying connection attempts to domains that provide a different server certificate. A solution for Task *TLS* would have been to select a secure TLS version to initialize the context. Task *TM*: the server's certificate had to be added to an empty custom trust manager replacing the default manager. This way, the program would pin the server's certificate and create a secure communication channel, while rejecting attempts to any other server with a different certificate.

8.3 Preliminaries and Participants

We advertised the study in lectures and across various university communication channels. 30 subjects participated in the study, however, three subjects dropped out, because they misunderstood a basic participation requirement (i.e., having at least basic Java programming knowledge). Of the remaining 27 subjects, 16 were assigned to the nudge treatment, and 11 to the control treatment. While being students, our sample varied across demographics and programming skill, but none of the self-reported characteristics systematically differed across the two treatments (see Appendix A for details).

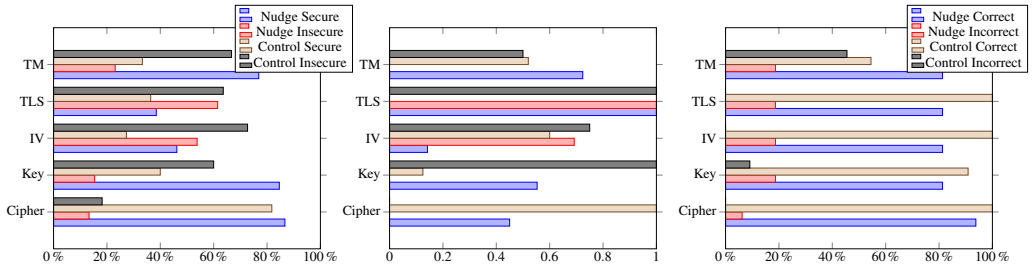
We followed well-established community principles for conducting security and privacy studies [29]. Participants were presented with a comprehensive consent form and separate study instructions on paper. Participants were compensated with 20 Euros.

After submission of the solutions, participants were asked to complete a short exit survey. We asked specific questions addressing the effectiveness of the security nudges and whether they were noticed by the participants. Also, we only asked demographic questions at this point to avoid any bias during the study. See Table 3 in the Appendix for details.

8.4 User Study Results

Functional Correctness Our system is not designed to address difficulties of programmers to deliver functionally correct code. However, it is important that using the system does not create obstacles to programmers. Participants predominantly submitted functionally correct code in both treatments

[7] This aims at simplifying search for participants. All Stack Overflow posts that contain a seed statement are available during the study.



(a) Security results (b) Copy-paste rate (average) (c) Correctness results

Figure 7: User study results for security, copy-paste rate and correctness of the submitted solutions across both treatments.

with some differences across tasks (cf. Figure 7c). Applying ordinal logistic regression (cf. Table 4 in the Appendix) indicates that the nudge treatment has – as anticipated – no effect on functional correctness of submitted tasks. However, non-professionals submitted significantly less functional code ($p < 0.05$). *Cipher* submissions are more often functional, irrespective of the treatment ($p < 0.05$).

Security Figure 7a shows the security results per task for both treatments. Performing ordinal logistic regression (see Table 5 in the Appendix), we show that the nudge treatment is significantly outperforming the control group in producing secure solutions (with an estimate of 1.303 and $p < 0.01$; Model 4). While the main effect of the nudge treatment dominates the regression models, we can observe from Figure 7a that comparatively more secure submissions are made for *TM* and *Key*. Indeed, pairwise testing using Chi-Square tests reveals $p < 0.001$ for both tasks. Participants from the nudge group provided 84.6% secure solutions for *Key* and 76.9% for *TM*, while 60.0% and 66.7% of the respective solutions submitted by the control group were insecure. These observations for *TM* are somewhat encouraging given previous findings: [17] have shown that reused insecure *TM* code snippets from Stack Overflow were responsible for 91% (183,268) of tested apps from Google Play being vulnerable. Only 0.002% (441) of apps contained secure *TM* code from Stack Overflow. Based on these insecure *TM* snippets, [13] were able to attack several high-profile apps extracting private data. Moreover, [27] found that only 45 out of 639,283 Android apps applied certificate pinning, while 25% of developers find certificate pinning too complex to use. [2] reported that tasks very similar to *TM* could not be solved with the help of simplified cryptographic APIs within a user study.

For *Cipher*, the nudge treatment performs very well, but only slightly better than the control treatment, as both achieved 86.7% and 81.8% secure solutions, respectively.

For *IV* and *TLS*, the nudge results are less desirable with 46.2% and 38.5% secure solutions, while performing better but not significantly ($p < 0.077$ and $p < 0.53$; Chi-Square) than the control treatment. To better understand these observations, we analyzed visited posts, copy-and-paste history and the submitted code of participants that provided insecure solutions for these two tasks. In the case of *IV*, we found that four insecure solutions reused insecure patterns from code snippets that were falsely marked as recommended code. To encounter false predictions of the security model was *a priori* extremely unlikely. Interestingly, the remaining insecure solutions were created by users combining secure code from different correctly marked recommendations (true negatives) into insecure code. Thereby, users reused the seed statement for *IV* from one snippet and initialized it with an empty array obtained from another code snippet that did not make use of *IV* at all. In the case of *TLS*, all insecure solutions were copied from code snippets that were clearly marked as insecure.

Copy-and-Paste Behavior We calculated the average copy-paste rate per task for both treatments, which reports the relative frequency copied code has also been reused in a submitted solution (see Figure 7b). Importantly, in the nudge treatment, *not a single* insecure copy-and-paste event was observed for *Cipher, Key* and *TM*, while secure code that was copied into the clipboard was reused at a rate of 0.45, 0.55, and 0.72 on average, respectively. This goes in line with observed security outcomes depicted in Figure 7a, where more secure than insecure solutions were provided for these tasks. However, insecure copy-and-paste events were observed for *IV* and *TLS*, partly explaining the comparatively higher number of insecure solutions. In the control treatment, the copy-paste rate for insecure snippets closely follows the observed frequencies of insecure results for all tasks except *Cipher*.

Warnings/Recommendations/Reminder Even though all users within the nudge group saw security warnings during their journey, we observed an insecure-to-secure copy event ratio of 0.27 for both treatments indicating that warnings alone are not sufficient for preventing users from placing insecure code on the clipboard. However, the copy-paste rate measuring the relative frequency of copy to paste events (see Figure 7b) offers more nuanced results. It shows that the nudge group tends to discard insecure copies, while pasting more secure copies into their solutions. This is most likely

the result of the reminder nudge, which was triggered by insecure copy events. As a result, users dropped copied insecure samples and started looking for a secure alternative. In contrast, the copy-paste rate for control shows that copied insecure snippets were not dropped, but rather pasted into the solution. Therefore, the interaction of several nudges was responsible for improving the security decisions of the participants. In the exit survey, users also marked the relevant nudges with high average Likert score values of above 4 (on 5-point scales).

We only observed 22 events where users clicked on a proposed recommendation link as shown in the bottom part of Figure 8a. Therefore, only 10.1% of secure posts (of 219 in total) were visited following such a proposed recommendation. However, 86.6% remembered the feature during the exit survey. With 4 paste and 8 copy events (i.e., a copy-paste rate of 0.5) only a very small amount of reused secure code in the submitted solutions was directly related to this nudge. Though contributing to the improvement of code security, we can state that this nudge was surprisingly the least effective one. The average Likert score for the list of recommendations was also comparatively low with 3.2.

9 Limitations

The response rate during recruitment for our developer study was quite low. However, we achieved a participation count per treatment which was very similar to comparable peer-reviewed studies (e.g., [2]). However, participation may introduce self-selection bias. Therefore, we avoided any security framing during recruitment and have no reason to believe that the final group of participants was systematically different in terms of security knowledge. The study was performed within a laboratory under strict time constraints. By enforcing a time limit, we intended to create a more realistic scenario and to obtain a comparable outcome for both treatments. Participants had to solve their programming tasks using a given code editor, browser, as well as operating system which they might not have been familiar with. Most of our participants were students, while only a minority had a professional background, which may limit the generalizability of our results. Professionals performed slightly better in achieving functional solutions, but not in security across both treatments. Therefore, comparisons among both treatments remain valid.

For implementing custom trust managers in Android (see Section 8.2), current best practices suggest a declarative solution which uses a static configuration file instead of Java code.[8] Being able to include other formats, such as formal documentation in our recommendations would additionally allow suggesting this solution. One possible way to achieve that is to create a link between code examples from Stack Overflow and natural language text in official documentation. I.e., we would have to extend our framework such that it embeds code examples and natural language text into the same vector space. This can be done with sequence-to-sequence models, which are usually applied for natural language translation. GitHub is currently testing a similar approach for their semantic code search engine.[9]

10 Future Work

Our recommendation approach may be subject to attacks. More specifically, in an adversarial setting, machine learning algorithms are often not robust against manipulated input data. Similar to efforts in malware obfuscation and spam filter bypassing, an attacker might be able to craft malicious code that gets mistakenly classified as secure. This way, the attacker could spread malicious code into the ecosystem on a large scale. However, a number of novel techniques have been proposed to counter the adversarial effect [18, 24, 28].

Stack Overflow provides code examples for almost each and every programming language. Since our framework learns the optimal code representation for a given classification task based on general code features, we do not see major issues in applying it to different programming languages. A language-specific compiler or a universal parser can be used to generate the PDG, which is then fed to our pattern embedding network (see Section 5.4). The representation learning of API-specific lexical features (see Section 5.3) is completely independent from the programming language and therefore straightforward.

We suggest to conduct additional UI testing as we might not have identified the optimal design, yet. Following Felt et al. [15], different security indicators such as alternative candidate icons and text have to be tested, for instance within user surveys or by repeating our developer study. Stack Overflow recently proposed a partnership program with academia that would allow to extend their developer survey and to test design tweaks on their website.[10]

11 Conclusion

In this paper, we propose an approach for deep learning security nudges that help software developers write strong encryption code. We propose a system design integrated in Stack Overflow whose components consist of several security nudges, namely *warnings*, *recommendations*, *reminders*, and *defaults* and a neural network architecture that controls these nudges by learning and predicting secure and insecure cryptographic usage patterns from community-provided

[8] https://developer.android.com/training/articles/security-config

[9] https://githubengineering.com/towards-natural-language-semantic-code-search/

[10] https://meta.stackoverflow.com/questions/377152/stack-overflow-academic-research-partnership-program

code examples. We propose a novel approach on deep learning optimized code representations for given code classification tasks and train a classification model that is able to predict use cases and security scores of encryption code examples with an AUC-ROC of 0.999 and 0.992, respectively. Applying this model within our nudge-based system design on Stack Overflow, we performed a user study where participants had to solve the most error-prone cryptographic programming tasks reported in recent research. Our results demonstrate the effectiveness of nudges in helping software developers to make better security decisions on Stack Overflow.

Acknowledgements

The authors would like to thank Fraunhofer AISEC for technical support, DIVSI for support of our research efforts, and the anonymous reviewers for their helpful comments.

References

[1] ACAR, Y., BACKES, M., FAHL, S., GARFINKEL, S., KIM, D., MAZUREK, M. L., AND STRANSKY, C. Comparing the usability of cryptographic APIs. In *IEEE Symposium on Security and Privacy* (2017), pp. 154–171.

[2] ACAR, Y., BACKES, M., FAHL, S., KIM, D., MAZUREK, M. L., AND STRANSKY, C. You Get Where You're Looking For: The Impact of Information Sources on Code Security. In *IEEE Symposium on Security and Privacy* (2016), pp. 289–305.

[3] ACER, M., STARK, E., FELT, A. P., FAHL, S., BHARGAVA, R., DEV, B., BRAITHWAITE, M., SLEEVI, R., AND TABRIZ, P. Where the wild warnings are: Root causes of Chrome HTTPS certificate errors. In *ACM Conference on Computer & Communications Security* (2017), pp. 1407–1420.

[4] ACQUISTI, A. Nudging privacy: The behavioral economics of personal information. *IEEE Security & Privacy 7*, 6 (2009), 82–85.

[5] ACQUISTI, A., ADJERID, I., BALEBAKO, R., BRANDIMARTE, L., CRANOR, L. F., KOMANDURI, S., LEON, P. G., SADEH, N., SCHAUB, F., SLEEPER, M., ET AL. Nudges for privacy and security: Understanding and assisting users' choices online. *ACM Computing Surveys 50*, 3 (2017), Article No. 44.

[6] ALMUHIMEDI, H., SCHAUB, F., SADEH, N., ADJERID, I., ACQUISTI, A., GLUCK, J., CRANOR, L. F., AND AGARWAL, Y. Your location has been shared 5,398 times! A field study on mobile app privacy nudging. In *ACM Conference on Human Factors in Computing Systems* (2015), pp. 787–796.

[7] BALEBAKO, R., LEON, P. G., ALMUHIMEDI, H., KELLEY, P. G., MUGAN, J., ACQUISTI, A., CRANOR, L. F., AND SADEH, N. Nudging users towards privacy on mobile devices. In *CHI Workshop on Persuasion, Nudge, Influence and Coercion* (2011), pp. 193–201.

[8] BÖHME, R., AND GROSSKLAGS, J. The security cost of cheap user interaction. In *ACM New Security Paradigms Workshop* (2011), pp. 67–82.

[9] CHEN, M., FISCHER, F., MENG, N., WANG, X., AND GROSSKLAGS, J. How reliable is the crowdsourced knowledge of security implementation? In *ACM/IEEE International Conference on Software Engineering* (2019).

[10] DAGENAIS, B., AND HENDREN, L. Enabling static analysis for partial Java programs. *ACM Sigplan Notices 43*, 10 (2008), 313–328.

[11] DAI, H., DAI, B., AND SONG, L. Discriminative embeddings of latent variable models for structured data. In *International Conference on Machine Learning* (2016), pp. 2702–2711.

[12] EGELE, M., BRUMLEY, D., FRATANTONIO, Y., AND KRUEGEL, C. An empirical study of cryptographic misuse in Android applications. In *ACM Conference on Computer & Communications Security* (2013), pp. 73–84.

[13] FAHL, S., HARBACH, M., MUDERS, T., SMITH, M., BAUMGÄRTNER, L., AND FREISLEBEN, B. Why Eve and Mallory love Android: An analysis of Android SSL (in)security. In *ACM Conference on Computer & Communications Security* (2012), pp. 50–61.

[14] FAHL, S., HARBACH, M., PERL, H., KOETTER, M., AND SMITH, M. Rethinking SSL development in an appified world. In *ACM Conference on Computer & Communications Security* (2013), pp. 49–60.

[15] FELT, A. P., REEDER, R., AINSLIE, A., HARRIS, H., WALKER, M., THOMPSON, C., ACER, M. E., MORANT, E., AND CONSOLVO, S. Rethinking connection security indicators. In *Symposium on Usable Privacy and Security* (2016), pp. 1–14.

[16] FENG, Q., ZHOU, R., XU, C., CHENG, Y., TESTA, B., AND YIN, H. Scalable graph-based bug search for firmware images. In *ACM Conference on Computer & Communications Security* (2016), pp. 480–491.

[17] FISCHER, F., BÖTTINGER, K., XIAO, H., STRANSKY, C., ACAR, Y., BACKES, M., AND FAHL, S. Stack overflow considered harmful? The impact of copy&paste on Android application security. In *IEEE Symposium on Security and Privacy* (2017).

[18] GANIN, Y., USTINOVA, E., AJAKAN, H., GERMAIN, P., LAROCHELLE, H., LAVIOLETTE, F., MARCHAND, M., AND LEMPITSKY, V. Domain-adversarial training of neural networks. *Journal of Machine Learning Research 17*, 59 (2016), 1–35.

[19] GROSSKLAGS, J., RADOSAVAC, S., CÁRDENAS, A., AND CHUANG, J. Nudge: Intermediaries role in interdependent network security. In *International Conference on Trust and Trustworthy Computing* (2010), pp. 323–336.

[20] KRÜGER, S., SPÄTH, J., ALI, K., BODDEN, E., AND MEZINI, M. CrySL: An extensible approach to validating the correct usage of cryptographic apis. In *European Conference on Object-Oriented Programming* (2018), pp. 10:1–10:27.

[21] LI, Z., ZOU, D., XU, S., OU, X., JIN, H., WANG, S., DENG, Z., AND ZHONG, Y. VulDeePecker: A deep learning-based system for vulnerability detection. In *Network and Distributed Systems Security Symposium* (2018).

[22] LIU, B., ANDERSEN, M. S., SCHAUB, F., ALMUHIMEDI, H., ZHANG, S. A., SADEH, N., AGARWAL, Y., AND ACQUISTI, A. Follow my recommendations: A personalized privacy assistant for mobile app permissions. In *Symposium on Usable Privacy and Security* (2016), pp. 27–41.

[23] LIU, B., LIN, J., AND SADEH, N. Reconciling mobile app privacy and usability on smartphones: Could user privacy profiles help? In *International Conference on World Wide Web* (2014), pp. 201–212.

[24] MIYATO, T., MAEDA, S., KOYAMA, M., NAKAE, K., AND ISHII, S. Distributional smoothing by virtual adversarial examples. *CoRR abs/1507.00677* (2015).

[25] NGUYEN, D. C., WERMKE, D., ACAR, Y., BACKES, M., WEIR, C., AND FAHL, S. A stitch in time: Supporting Android developers in writing secure code. In *ACM Conference on Computer and Communications Security* (2017), pp. 1065–1077.

[26] OLIVEIRA, D. S., LIN, T., RAHMAN, M. S., AKEFIRAD, R., ELLIS, D., PEREZ, E., BOBHATE, R., DELONG, L. A., CAPPOS, J., AND BRUN, Y. API Blindspots: Why experienced developers write vulnerable code. In *Symposium on Usable Privacy and Security* (2018), pp. 315–328.

[27] OLTROGGE, M., ACAR, Y., DECHAND, S., SMITH, M., AND FAHL, S. To pin or not to pin – Helping app developers bullet proof their TLS connections. In *USENIX Security Symposium* (2015), pp. 239–254.

[28] PAPERNOT, N., MCDANIEL, P., WU, X., JHA, S., AND SWAMI, A. Distillation as a defense to adversarial perturbations against deep neural networks. In *IEEE Symposium on Security and Privacy* (2016), pp. 582–597.

[29] SCHECHTER, S. Common pitfalls in writing about security and privacy human subjects experiments, and how to avoid them. Tech. rep., Microsoft Research, 2013.

[30] SUNSTEIN, C. Nudging: A very short guide. *Journal of Consumer Policy 37*, 4 (2014), 583–588.

[31] THALER, R., AND SUNSTEIN, C. *Nudge: Improving decisions about health, wealth, and happiness*. Penguin, 2008.

[32] WANG, Y., LEON, P. G., SCOTT, K., CHEN, X., ACQUISTI, A., AND CRANOR, L. F. Privacy nudges for social media: An exploratory Facebook study. In *International Conference on World Wide Web* (2013), pp. 763–770.

[33] XU, X., LIU, C., FENG, Q., YIN, H., SONG, L., AND SONG, D. Neural network-based graph embedding for cross-platform binary code similarity detection. In *ACM Conference on Computer & Communications Security* (2017), pp. 363–376.

Appendix A: Additional Participant Data

Table 3 includes additional data about the 27 participants, who completed the study.

We also conducted a series of statistical tests to verify that the self-reported characteristics of the recruited participants did not systematically vary across treatments. Indeed, using the Mann-Whitney U Test, we found that participants did not differ in their reported age across treatments ($p = 0.79$). Applying Fisher's Exact Test, we also observed the absence of a statistically significant difference for country of origin ($p = 0.809$), gender ($p = 0.551$), level of education ($p = 0.217$), security knowledge/background ($p = 0.124$), and professional programming experience ($p = 0.315$). Using the Mann-Whitney U Test, we did not find any statistically significant difference for years of experience with Java programming ($p = 0.422$). We also did not find any reportable differences regarding participants' awareness of encryption flaws ($p = 0.363$) using Fisher's Exact Test. The percentage of participants who had to program Java as primary activity for their work ($p = 1$) or for whom writing Java code was part of their primary job in the last 5 years ($p = 0.696$) also did not differ across treatments (using Fisher's Exact Test).

Appendix B: Detailed Regression Results

Based on the user study data and self-reported survey responses, we follow an ordinal (Logit link) regression ap-

proach, which is primarily focused on evaluating the effectiveness of the nudge treatment.

First, we report a series of four models (M1 - M4) to evaluate whether the nudge treatment significantly impacts the *functional correctness* of the submitted programs for the five different tasks (see Table 4). We iteratively add factors to the regression model to also test whether programming expertise or security expertise positively impact the outcome variable. Most importantly, as the nudge treatment is not designed to address this aspect of programming, we did *not* expect any significantly positive effect. Indeed, across all model specifications that we tested, we did not observe any significant (positive or negative) effect. Regarding the different programming tasks, we found that the Cipher task was associated with a significantly increased likelihood of being functionally correct (M2 - M4). Further, not being a security professional (as reported by the participants) significantly impacts the likelihood that functional programs were submitted in a negative fashion (M3 - M4). In contrast, a higher degree of security knowledge (as reported by the participants) did not significantly impact the results (M4).

Note that the regression statistics for tasks IV and TLS are identical as the aggregate results for functional correctness happen to be the same (see Figure 7c).

Age				
Mean = 22.93	Median = 22	Stddev = 3.9	Min = 19	Max = 38
Country of Origin				
Germany = 16				Other = 11
Gender				
Male = 9				Female = 18
Achieved Level of Education				
Highschool = 15	Bachelor = 8		Master = 3	Ph.D. = 0
Professional at Programming				
Yes = 12	No = 15			
Security Background				
Yes = 10	No = 17			
Java Years Experience				
Mean = 3.81	Median = 3	Stddev = 2.304	Min = 1	Max = 8
Encryption Flaw Awareness				
Yes = 17	No = 10			
Java primary focus of job				
Yes = 5	No = 21	No Data = 1		
Java part of any job				
Yes = 12	No = 15			

Table 3: Detailed data about demographics of participants (N = 27). One missing response for the question whether Java is primary focus of current job.

Second, we report a series of four models (M1 - M4) to evaluate whether the nudge treatment significantly impacts the *security* of the submitted programs for the five different tasks (see Table 5). For consistency, we iteratively add the same factors to the regression model to also test whether programming expertise or security expertise positively impact the outcome variable.

Most importantly, as the nudge treatment is designed to improve the security of cryptography-related programming, we did expect a significantly positive effect. Indeed, across all model specifications that we tested, we did observe a significant and positive effect. Regarding the different programming tasks, we did not find that they significantly differed from each other regarding the security property (M2 - M4). Being a security professional did not impact the security of the submitted programs in a significant way (M3 - M4). Perhaps surprisingly, a higher degree of security knowledge (as

FACTORS	**M1**	**M2**	**M3**	**M4**
Treatment: Nudge	-0.460	-0.489	-0.263	-0.226
	(0.523)	(0.544)	(0.568)	(0.605)
Task: Cipher	-	**2.407***	**2.539***	**2.539***
		(1.105)	(1.125)	(1.125)
Task: IV	-	1.224	1.324	1.324
		(0.746)	(0.775)	(0.775)
Task: Key	-	0.892	0.974	0.974
		(0.690)	(0.72)	(0.721)
Task: TLS	-	1.224	1.324	1.324
		(0.746)	(0.775)	(0.775)
Not Professional	-	-	**-1.701***	**-1.698***
			(0.679)	(0.680)
Sec. Knowledge	-	-	-	-0.106
				(0.605)

Table 4: Results for Ordinal Regression of Functional Correctness. Series of non-interaction models (M1 – M4) with factors iteratively added. Significant values are highlighted in bold, and marked with: * $p < 0.05$. Standard errors are included in parentheses. The baseline for Treatment is Control (i.e., the unmodified Stack Overflow), and the baseline for Task is TM.

FACTORS	**M1**	**M2**	**M3**	**M4**
Treatment: Nudge	**0.920***	**1.018***	**1.113***	**1.303****
	(0.388)	(0.426)	(0.438)	(0.480)
Task: Cipher	-	1.388	1.377	1.405
		(0.745)	(0.754)	(0.758)
Task: IV	-	-0.963	-1.001	-0.990
		(0.654)	(0.665)	(0.668)
Task: Key	-	0.224	0.200	2.13
		(0.668)	(0.677)	(0.679)
Task: TLS	-	-0.963	-1.001	-0.990
		(0.654)	(0.665)	(0.668)
Not Professional	-	-	-0.702	-0.686
			(0.432)	(0.434)
Sec. Knowledge	-	-	-	-0.517
				(0.481)

Table 5: Results for Ordinal Regression of Security. Series of non-interaction models (M1 – M4) with factors iteratively added. Significant values are highlighted in bold, and marked with: * $p < 0.05$ and ** $p < 0.01$. Standard errors are included in parentheses. The baseline for Treatment is Control (i.e., the unmodified Stack Overflow), and the baseline for Task is TM.



(a) Security warning provided by the security and use case model



(b) Recommendation provided by the similarity and use case model.

Figure 8: Security warning and recommendations provided by the similarity, use case and security model. The security model predicted the usage pattern of *setHostnameVerifier* as insecure. Further, it predicted its use case *HNV*, being able to select and display the related security annotation under the insecure statement. Below the security warning the similarity, use case and security model provide the ranked list of recommendations, that contains code examples with similar and secure patterns of *HNV*. We display the recommended code example that appears when clicking on the first link in (b).

reported by the participants) did not significantly impact the results either (M4).

We created regression models including further demographic and explanatory variables. However, none of them had a significant effect on the security of submitted solutions.

Appendix C: Pattern Annotation Tool

Our security annotations generally comply with rules and annotation heuristics given by [12, 13, 17]. However, manual analysis of patterns was not restricted to simple application of these heuristics, but was based on detecting insecure patterns in general. Whenever an unknown pattern has been detected, both annotators discussed them until agreement on a label. For example, [13] only reports empty trust manager implementations, while many insecure *TM* patterns on Stack Overflow are not empty, but provide insufficient certificate verification (e. g., only validating that the certificate is not expired).

To further speed up the labeling process and manage the large amount of samples, we created a code annotation tool. It automatically iterates through code snippets and displays them to the user, using a source code editor. Seed statements were already highlighted in order to allow the annotator to detect relevant patterns quickly. The annotator was able to assign labels (e. g., secure/insecure) to different keyboard buttons. While iterating through the seed statements, the annotator would investigate the related pattern and label it accordingly. Moreover, the annotator had the option to add seed statements, that she wanted to have highlighted and labeled. Whenever the annotator identified new patterns or wanted to share and discuss a pattern, the related code snippet was marked and other annotators were notified to comment on it. After agreement, the pattern was labeled by the initial annotator. Further, annotation heuristics obtained during the discussion were shared among all annotators.

Wireless Attacks on Aircraft Instrument Landing Systems

Harshad Sathaye, Domien Schepers, Aanjhan Ranganathan, and Guevara Noubir
Khoury College of Computer Sciences
Northeastern University, Boston, MA, USA

Abstract

Modern aircraft heavily rely on several wireless technologies for communications, control, and navigation. Researchers demonstrated vulnerabilities in many aviation systems. However, the resilience of the aircraft landing systems to adversarial wireless attacks have not yet been studied in the open literature, despite their criticality and the increasing availability of low-cost software-defined radio (SDR) platforms. In this paper, we investigate the vulnerability of aircraft instrument landing systems (ILS) to wireless attacks. We show the feasibility of spoofing ILS radio signals using commercially-available SDR, causing last-minute *go around* decisions, and even missing the landing zone in low-visibility scenarios. We demonstrate on aviation-grade ILS receivers that it is possible to fully and in fine-grain control the course deviation indicator as displayed by the ILS receiver, in *real-time*. We analyze the potential of both an overshadowing attack and a lower-power single-tone attack. In order to evaluate the complete attack, we develop a tightly-controlled closed-loop ILS spoofer that adjusts the adversary's transmitted signals as a function of the aircraft GPS location, maintaining power and deviation consistent with the adversary's target position, causing an undetected off-runway landing. We systematically evaluate the performance of the attack against an FAA certified flight-simulator (X-Plane)'s AI-based autoland feature and demonstrate systematic success rate with offset touchdowns of 18 meters to over 50 meters.

1 Introduction

Today, the aviation industry is experiencing significant growth in air traffic with more than 5000 flights [14] in the sky at any given time. It has become typical for air traffic control towers to handle more than a thousand takeoffs and landings every day. For example, Atlanta's Hartsfield-Jackson International airport handles around 2500 takeoffs and landings every day. Boston's Logan airport which is not one of the busiest airports in the world managed an average of 1100 flights every day in August 2018. The modern aviation ecosystem heavily relies on a plethora of wireless technologies for their safe and efficient operation. For instance, air traffic controllers verbally communicate with the pilots over the VHF (30 to 300 MHz) radio frequency channels. The airplanes continuously broadcast their position, velocity, callsigns, altitude, etc. using the automatic dependent surveillance-broadcast (ADS-B) wireless communication protocol. Primary and secondary surveillance radars enable aircraft localization and provide relevant target information to the air traffic controllers. Traffic Alert and Collision Avoidance System (TCAS), an airborne wireless system independent of the air traffic controller enables the aircraft to detect potential collisions and alert the pilots. Air traffic information, flight information and other operational control messages between the aircraft and ground stations are transferred using the Aircraft Communications Addressing and Reporting System (ACARS) which uses the VHF and HF radio frequency channels for communication. Similarly, many radio navigation aids such as GPS, VHF Omnidirectional Radio Range (VOR), Non-directional radio beacons (NDB), Distance Measuring Equipment (DME), and Instrument Landing System (ILS) play crucial roles during different phases of an airplane's flight.

Many studies have already demonstrated that a number of the above-mentioned aviation systems are vulnerable to attacks. For example, researchers [22] injected non-existing aircraft in the sky by merely spoofing ADS-B messages. Some other attacks [37] modified the route of an airplane by jamming and replacing the ADS-B signals of specific victim aircraft. ACARS, the data link communications system between aircraft and ground stations was found to leak a significant amount of private data [50], e.g., passenger information, medical data and sometimes even credit card details were transferred. GPS, one of the essential navigation aids is also vulnerable to signal spoofing attacks [32]. Furthermore, an attacker can spoof TCAS messages [42] creating false resolution advisories and forcing the pilot to initiate avoidance maneuvers. Given the dependence on wireless technologies, the threats described above are real and shows the importance of building secure aviation control, communication and navigation systems.

One of the most critical phases of an airplane's flight plan is the final approach or landing phase as the plane descends towards the ground actively maneuvered by the pilot. For example, 59% of the fatal accidents documented by Boeing [16] occurred during descent, approach and landing. Several technologies and systems such as GPS, VOR, DME assist the pilot in landing the aircraft safely. The Instrument Landing System (ILS) [17] is today the de-facto approach system used by planes at a majority of the airports as it is the most precise system capable of providing accurate horizontal and vertical guidance. At Boston's Logan International Airport, 405,822 [1] flight plans were filed in 2017. Out of these 405,822 flight plans, 95% were instrument flight rule (IFR) plans. Instrument flight rules are a set of instructions established by the FAA to govern flight under conditions in which flying by visual reference is either unsafe or just not allowed. Also, several European airports [9] prohibit aircraft from landing using visual flight rules during the night. ILS incorporates radio technology to provide all-weather guidance to pilots which ensures safe travel and any interference can be catastrophic.

As recently as September 2018, the pilots of Air India flight AI-101 reported an instrument landing system (ILS) malfunction and were forced to do an emergency landing. Even worse, TCAS, ACARS, and a majority of other systems that aid a smooth landing were unusable. Furthermore, NASA's Aviation Safety Reporting System [25] indicate over 300 ILS related incidents where pilots reported the erratic behavior of the localizer and glideslope–two critical components of ILS. ILS also plays a significant role in autoland systems that are capable of landing aircraft even in the most adverse conditions without manual interference. Autoland systems have significantly advanced over the years since its first deployment in De Havilland's DH121 Trident, the first airliner to be fitted with an autoland system [15]. However, several near-catastrophic events [4, 8, 12] have been reported due to the failure or erratic behavior of these autoland systems with ILS interference as one of the principal causes. With increasing reliance on auto-pilot systems and widespread availability of low-cost software-defined radio hardware platforms, adversarial wireless interference to critical infrastructure systems such as the ILS cannot be ruled out.

In this work, we investigate the security of aircraft instrument landing system against wireless attacks. To the best of our knowledge, there has been no prior study on the security guarantees of the instrument landing system. Specifically, our contributions are as follows.

- We analyze the ILS localizer and glideslope waveforms, the transmitters and receivers, and show that ILS is vulnerable to signal spoofing attacks. We devise two types of wireless attacks i) overshadow, and ii) single-tone attacks.

- For both the attacks, we generate specially crafted radio signals similar to the legitimate ILS signals using low-cost software-defined radio hardware platform and successfully induce aviation-grade ILS receivers, in *real-time*, to lock and display arbitrary alignment to both horizontal and vertical approach path. This demonstrates the potential for an adversary to the least be able to trigger multiple aborted landings causing air traffic disruption, and in the worst case, cause the aircraft to overshoot the landing zone or miss the runway entirely.

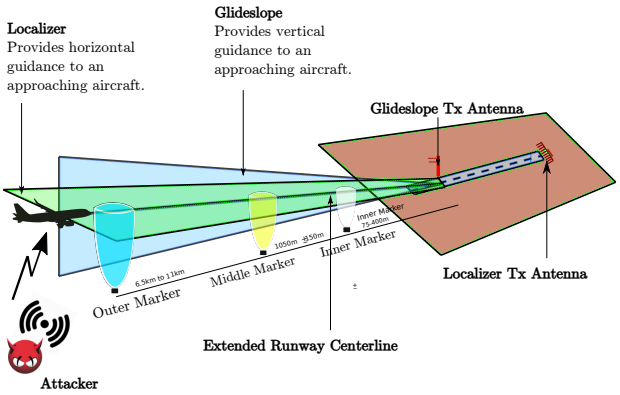


Figure 1: Overview of ILS sub-systems. The ILS consists of three subsystems: i) Localizer, ii) glideslope, and (iii) marker beacons.

- In order to evaluate the complete attack, we develop a tightly-controlled closed-loop ILS spoofer. It adjusts the the adversary's transmitted signals as a function of the aircraft GPS location, maintaining power and deviation consistent with the adversary's target position, causing an undetected off-runway landing. We demonstrate the integrated attack on an FAA certified flight-simulator (X-Plane), incorporating a spoofing region detection mechanism, that triggers the controlled spoofing on entering the landing zone to reduce detectability.

- We systematically evaluate the performance of the attack against X-Plane's AI-based autoland feature, and demonstrate the systematic success rate with offset touchdowns of 18 meters to over 50 meters.

- We discuss potential countermeasures including failsafe systems such as GPS and show that these systems also do not provide sufficient security guarantees. We highlight that implementing cryptographic authentication on ILS signals is not enough as the the system would still be vulnerable to record and replay attacks. Therefore, through this research, we highlight an open research challenge of building secure, scalable and efficient aircraft landing systems.

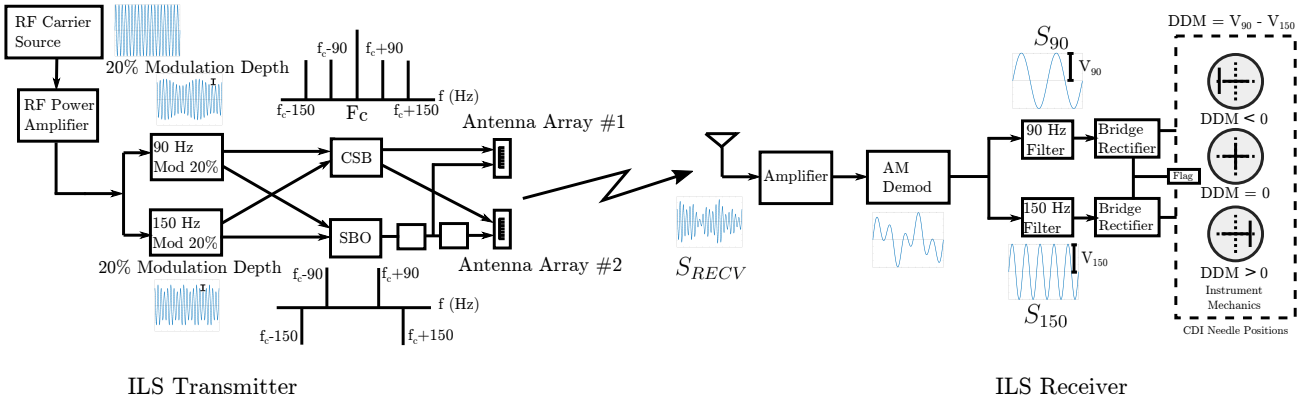


Figure 2: Block diagram of ILS transmitter and receiver describing the process of generation and reception of ILS signal along with waveforms at each stage.

2 Background

Approach systems enable pilots to land airplanes even in extreme weather conditions and are classified into non-precision and precision approach systems based on the accuracy and type of approach guidance provided to an aircraft. Non-precision approach systems provide only horizontal or lateral guidance (heading/bearing). Examples of non-precision approach systems are VHF Omnidirectional Range (VOR) [58], Non-Directional Beacon (NDB) [57], and satellite systems such as GPS. With the development of precision approach systems, the use of non-precision approach systems such as VOR and NDB has significantly decreased today. Precision approach systems provide both horizontal (heading/bearing) as well as vertical (glide path) guidance to an approaching aircraft. The Instrument Landing System (ILS) is the most commonly deployed precision approach system in use today. Other examples of precision approach systems include Microwave Landing System (MLS), Transponder Landing System (TLS), Ground Based Augmentation Landing System (GLS), and Joint Precision Approach and Landing System (JPALS). It is important to note that these alternate landing systems fundamentally still use existing ILS concepts and equipment mostly in scenarios where ILS is unavailable. For example, TLS enables precision landing guidance in places where the terrain is uneven, and the ILS signal reflections off the ground cause undesirable needle deflections by *emulating* the ILS signals using only one base tower (in contrast to two for ILS) whose placement allows more flexibility. However, TLS still leverages the same fundamental concepts of ILS. In short, ILS plays a key, de-facto role in providing precision landing guidance at the majority of airports today and it is, therefore, essential to evaluate its resilience to modern-day cyber-physical attacks.

2.1 Instrument Landing System (ILS)

The first fully operational ILS was deployed in 1932 at the Berlin Tempelhof Central Airport, Germany. ILS enables the pilot to align the aircraft with the centerline of the runway and maintain a safe descent rate. ITU defines ILS [28] as "a radio navigation system which provides aircraft with horizontal and vertical guidance just before and during landing and at certain fixed points, indicates the distance to the reference point of landing". Autopilot systems on some modern aircraft [49] use ILS signals to execute a fully autonomous approach and landing, especially in low visibility settings. ILS (Figure 1) comprises of three independent subsystems: i) localizer, (ii) glideslope and iii) marker beacons. The localizer and the glideslope guide the aircraft in the horizontal and vertical plane respectively. The marker beacons act as checkpoints that enable the pilot to determine the aircraft's distance to the runway. ILS has three operational categories: i) CAT I, ii) CAT II and, iii) CAT III. CAT III further has three sub-standards IIIa, IIIb and, IIIc. These operational categories are decided based on ILS installations at the airport [1] and is independent of the receiver on the aircraft. With the advent of GPS and other localization technologies, the marker beacons are less important today and increasingly obsolete. However, the localizer and the glideslope play a major role in an aircraft's safe landing today and is expected to remain so for many years.

2.1.1 ILS Signal Generation

ILS signals are generated and transmitted such that the waves form a specific radio frequency signal pattern in space to create guidance information related to the horizontal and vertical

[1] Procedures for the Evaluation and Approval of Facilities for Special Authorization Category I Operations and All Category II and III Operations http://fsims.faa.gov/wdocs/Orders/8400_13.htm

positioning. ILS signal generators leverage *space modulation* i.e., use multiple antennas to transmit an amplitude modulated radio frequency signals with various powers and phases. The transmitted signals combine in the airspace to form signals with different depths of modulation (DDM) at various points within the 3D airspace. Each DDM value directly indicates a specific deviation of the aircraft from the correct touchdown position. For example, the signals combine in space to produce a signal with zero difference in the depth of modulation (DDM) along the center-line of the runway. It is important to note that unlike traditional modulation techniques where the modulation occurs within the modulating hardware, in space modulation, the signals mix within the airspace.

The process of generating the localizer and glideslope signals (Figure 2) are similar with differences mainly in the carrier frequency used and how they are combined in space to provide the relevant guidance information. The carrier signal is amplitude modulated with 90 Hz and 150 Hz tones to a certain depth of modulation. The depth of modulation or modulation index is the measure of the extent of amplitude variation about an un-modulated carrier. The depth of modulation is set at 20% and 40% respectively for localizer and glideslope signals. The output of both the 90 Hz and 150 Hz modulator is then combined to yield two radio frequency signals: a carrier-plus-sidebands (CSB) and a sidebands-only (SBO) signal. The names of the signal directly reflect their spectral energy configuration with the CSB containing both the sideband energy and the assigned carrier frequency while in the SBO signal the carrier frequency component is suppressed. The CSB and SBO signals are subjected to specific phase shifts before being transmitted. The phase shifts are carefully chosen such that when the CSB and SBO signals combine in space, the resulting signal enables the aircraft to determine its horizontal and vertical alignment with the approach path.

Localizer. The localizer subsystem consists of an array of multiple antennas that emit the CSB and SBO signals such that the 150 Hz modulation predominates to the right of the runway centerline and the 90 Hz signal prevails to the left. In other words, if the flight is aligned to the right of the runway during the approach, the 150 Hz dominant signal will indicate the pilot to steer left and vice versa. The antenna array of the localizer is located at the opposite end (from the approach side) of the runway. Each runway operates its localizer at a specific carrier frequency (between 108.1 MHz to 111.95 MHz) and the ILS receiver automatically tunes to this frequency as soon as pilot inputs the runway identifier in the cockpit receiver module. Additionally, the runway identifier is transmitted using a 1020 Hz morse code signal over the localizer's carrier frequency.

Glideslope. The glideslope subsystem uses two antennas to create a signal pattern similar to that of the localizer except on a vertical plane. The two antennas are mounted on a tower at specific heights defined by the glide-path angle suitable for that particular airport's runway. In contrast to the localizer, the glideslope produces the signal pattern in the airspace based on the sum of the signals received from each antenna via the direct line-of-sight path and the reflected path. The mixing of the CSB and SBO signals results in a pattern in which the 90 Hz component of the signal predominates in the region above the glide-path while the 150 Hz prevails below the glide-path. The glideslope uses carrier frequencies between 329.15 MHz and 335.0 MHz, and the antenna tower is located near the touchdown zone of the runway. Typically, the center of the glide-slope defines a glide path angle of approximately 3°. For every localizer frequency, the corresponding glideslope frequency is hardcoded i.e., the localizer-glideslope frequencies occur in pairs and the instrument automatically tunes to the right glideslope frequency when the pilot tunes to a specific runway's localizer frequency.

2.1.2 ILS Receiver

The combined signals received at the aircraft are amplified, demodulated, and filtered to recover the 90 Hz and 150 Hz components. A bridge rectifier is used to convert the amplitude of the recovered tones to DC voltage levels. The DC voltage output is directly proportional to the depth of the modulation of the 90 Hz and 150 Hz tones–a direct measure of the dominating frequency signal. The DC voltage causes the course deviation indicator needle to deflect based on the difference in the depth of the modulation of the two tones thereby precisely indicating the aircraft's lateral and vertical deviation from approach path.

For example, an aircraft that is on-course will receive both 90 and 150 Hz signals with the same amplitude, i.e., equal depth of modulation and will result in zero *difference in the depth of modulation* and therefore cause no needle deflections. However, an aircraft that is off-course and not aligned with the approach path will receive signals with a non-zero difference in the depth of modulation resulting in a corresponding deflection of the needle. The instruments are calibrated to show full scale deflection if $DDM > 0.155$ or $DDM < -0.155$ for localizer and if $DDM > 0.175$ or $DDM < -0.175$ for glideslope [20]. These values correspond to 2.5° offset on the left side of the runway, 2.5° offset on the right side of the runway, 0.7° offset above the glide path angle and 0.7° below the glide path angle respectively.

2.2 Typical Approach Sequence

Pilots use aeronautical charts containing vital information about the terrain, available facilities and their usage guidelines throughout a flight. Approach plates are a type of navigation chart used for flying based on instrument readings. Every pilot is required to abide by the routes and rules defined in an approach plate unless ordered otherwise by the air traffic controller. The approach plate contains information like active localizer frequency of the runway, the runway identifier

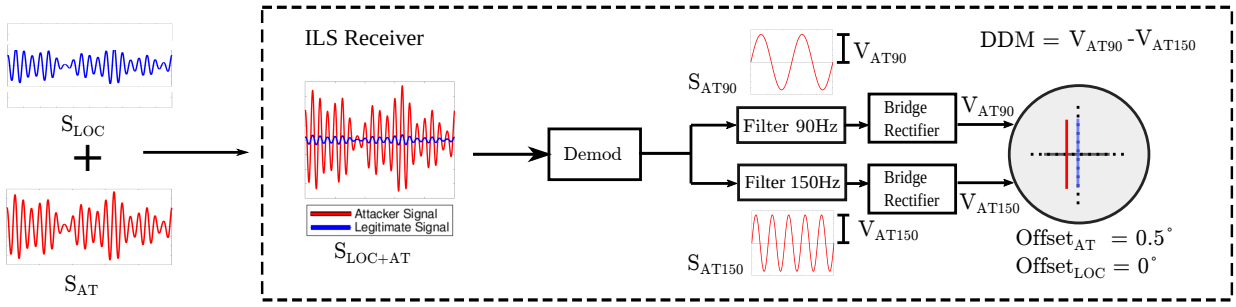


Figure 3: Schematic of the overshadow attack. The attacker's signal has a preset DDM corresponding to 0.5° to the right of the runway. Attacker's signal overshadows the legitimate signal. The blue line represents the needle position without attack.

in Morse code, glideslope interception altitude, ATC tower frequencies, and other information crucial for a safe landing.

Once the pilot receives the clearance to land at an assigned runway, the pilot enters the localizer frequency associated with the designated runway and enters the course of the runway into the auto-pilot. Note that the localizer and glideslope frequencies occur in pairs and therefore the pilot does not have to manually enter the corresponding glideslope frequency. When the pilot intercepts the localizer, the course deviation indicator needle is displayed on the cockpit. The pilot then verifies whether the receiver is tuned to the right localizer by confirming the runway identifier which is transmitted as morse code on the localizer frequency. For example, for landing on runway 4R (Runway Ident - IBOS) at Logan International Airport, Boston, the pilot will tune to 110.3 MHz and will verify this by confirming the Morse code: .. / --... / --- / ... Based on the deviation of the aircraft from the runway and the approach angle, the indicator will guide the pilot to appropriately maneuver the aircraft. Modern autopilot systems are capable of receiving inputs from ILS receivers and autonomously land the aircraft without human intervention.

In fact, pilots are trained and instructed to trust the instruments more than their intuition. If the instruments ask them to fly right, the pilots will fly right. This is true specifically when flying in weather conditions that force the pilots to follow the instruments. Detecting and recovering from any instrument failures during crucial landing procedures is one of the toughest challenges in modern aviation. Given the heavy reliance on ILS and instruments in general, malfunctions and adversarial interference can be catastrophic especially in autonomous approaches and flights. In this paper, we demonstrate vulnerabilities of ILS and further raise awareness towards the challenges of building secure aircraft landing systems.

3 Wireless Attacks on ILS

We demonstrate two types of wireless attacks: i) Overshadow attack and ii) Single-tone attack. In the overshadow attack, the attacker transmits pre-crafted ILS signals of higher signal strength; thus overpowering the legitimate ILS signals. The single-tone attack is a special attack where it is sufficient for the attacker to transmit a single frequency tone signal at a specific signal strength (lower than the legitimate ILS signal strength) to interfere and control the deflections of the course deviation indicator needle.

Attacker model. We make the following assumptions regarding the attacker. Given that the technical details of ILS are in the public domain, we assume that the attacker has complete knowledge of the physical characteristics of ILS signals e.g., frequencies, modulation index etc. We also assume that the attacker is capable of transmitting these radio frequency signals over the air. The widespread availability of low-cost (less than a few hundred dollars) software-defined radio platforms has put radio transmitters and receivers in the hands of the masses. Although not a necessary condition, in the case of single-tone, the knowledge of the flight's approach path, the airplane's manufacturer and model will allow the attacker to significantly optimize their attack signal. We do not restrict the location of the attacker and discuss pros and cons of both an on-board attacker as well as a attacker on the ground.

3.1 Overshadow attack

The overshadow attack is an attack where the attacker transmits specially crafted ILS signals at a power level such that the legitimate signals get overpowered by the attacker's signal at the receiver. The main reason why such an attack works is that the receivers "lock" and process only the strongest received signal. Figure 3 shows how the attacker's fake ILS signal completely overshadows the legitimate ILS signal resulting in the deflection of the CDI needle. We note that the attacker signal can be specially crafted to force the CDI needle to indicate a specific offset as demonstrated in Section 4.2.

Attack Signal Generation. Recall that the ILS receiver onboard receives a mix of the transmitted CSB and SBO signals that contain the 90 and 150 Hz tones (Figure 2). The amplitude of received 90 and 150 Hz tones depends on the position of the aircraft relative to the runway and its approach path

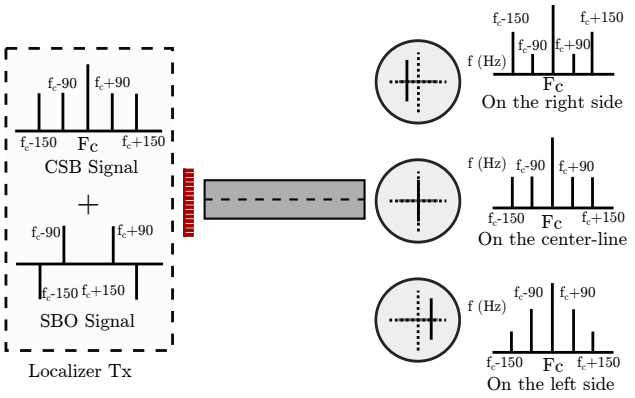


Figure 4: Frequency domain representation of the received signal showing the amplitudes of the sidebands as observed at various lateral offsets

angle. For example, as shown in Figure 4, the 90 Hz tone will dominate if the aircraft is offset to the left of the runway and the 150 Hz dominates to the right. Similarly, for glideslope, the 90 Hz tone dominates glide angles steeper than the recommended angle, and the 150 Hz tone dominates otherwise. Both 90 and 150 Hz will have equal amplitudes for a perfectly aligned approach. Therefore, to execute an overshadow attack, it is sufficient to generate signals similar to the received legitimate ILS signals and transmit at a much higher power as compared to legitimate ILS signals. In other words, the attacker need not generate CSB and SBO signals separately; instead can directly transmit the combined signal with appropriate amplitude differences between the 90 and 150 Hz tones. The amplitude differences are calculated based on the offset the attacker intends to introduce at the aircraft. The attacker's signal (Figure 5) is generated as follows. There are two tone generators for generating the 90 and the 150 Hz signals. It is important to enable configuration of each individual tone's amplitude to construct signals with a preset difference in the depth of modulation corresponding to the required deviation to spoof. The tones are then added and amplitude modulated using the runway's specific localizer or glideslope frequency. Recall that the amplitude differences i.e., difference in depth of modulation (DDM) between the two tones directly corresponds to the required offset to spoof. In the absence of the adversarial signals the estimated $DDM = V_{LOC90} - V_{LOC150}$. In the presence of the attacker's spoofing signals, the estimated $DDM = [V_{LOC90} + V_{AT90}] - [V_{LOC150} + V_{AT150}]$. Since $V_{AT90} >> V_{LOC90}$ and $V_{AT150} >> V_{LOC150}$, the resulting $DDM = V_{AT90} - V_{AT150}$. Thus by manipulating the amplitude *differences* between the transmitted 90 Hz and 150 Hz tones, the attacker can acquire precise control of the aircraft's course deviation indicator and the aircraft's approach path itself.

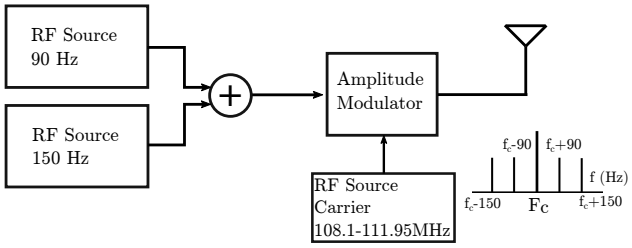


Figure 5: Signal generator used for generating the required attack signal with specific amplitudes of the 90 Hz and 150 Hz components

3.2 Single-tone attack

Single-tone attack is an attack where the attacker transmits only one of the sideband tones (either the 90 Hz or the 150 Hz) to cause deflections in the course deviation indicator needle. In contrast to the overshadow attack, single-tone attack does not require high powered spoofing signals. Recall that the aircraft's horizontal and vertical offset is estimated based on the difference in the depth of the modulation of the 90 Hz and the 150 Hz tones. As indicated in Figure 4, depending on the offset either of the frequency tones dominates. In the case of an overshadow attack, the spoofing signal was constructed with all the necessary frequency components. However, in the single-tone attack, the attacker aims to interfere with only one of the two sideband frequencies directly affecting the estimated offset.

Attack Signal Generation. The working of the single-tone attack is shown in Figure 6. The legitimate localizer signal's spectrum contains the carrier and both the sideband tones of 90 Hz and 150 Hz. As described previously, the amplitudes of the sideband tones depend on the true offset of the aircraft. In a single-tone attack, the attacker generates only one of the two sideband tones i.e., $f_c \pm 90$ or $f_c \pm 150$ with appropriate amplitude levels depending on the spoofing offset (e.g., left or right off the runway) introduced at the aircraft. For example, consider the scenario where the attacker intends to force the aircraft to land on the left of the runway with an offset of 0.5°. The legitimate difference in depth of modulation will be zero as the aircraft is centered over the runway. To cause the aircraft to go left, the attacker must transmit signals that will spoof the current offset to be at the right side of the runway. As shown in Figure 4, the 150 Hz component dominates in the right side of the runway approach and therefore the attacker needs to transmit the $f_c \pm 150$ signal with an appropriate amplitude to force the aircraft to turn left. For the specific example of 0.5° offset, the amplitude of the $f_c \pm 150$ component should be such that the difference in the depth of modulation equals 0.03 [20].

Notice that the single-tone attack signal is similar to a double-sideband suppressed-carrier signal which is well-

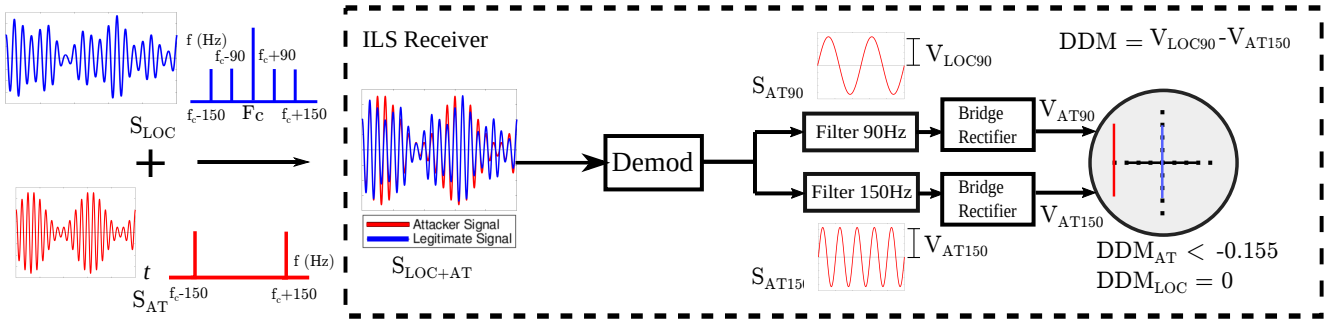


Figure 6: Schematic of the single-tone attack. Attacker constructs a DSB-SC signal without the 90 Hz component and the carrier. The blue line represents the needle position without the attack

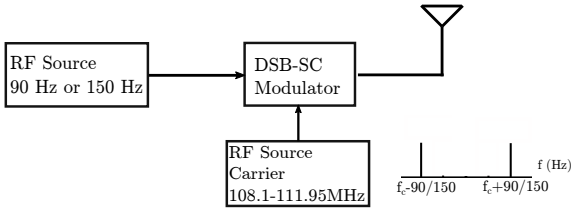


Figure 7: Single-tone attack signal generator with a DSB-SC modulator

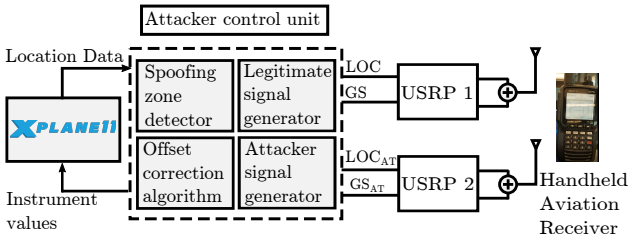


Figure 8: Schematic of the experiment setup used for evaluating the attacks on ILS. The attacker control unit interfaces with the simulator and USRP B210s. A flight yoke and throttle system is connected to the machine running X-Plane flight simulator software. Attacker control unit interfaces with the flight simulator over a UDP/IP network.

known to be spectrally efficient than the normal amplitude modulation signal. Specifically, it is possible for the attacker to reduce the required power to almost 50% of the overshadow attack as there is no need to transmit the carrier signal and one of the sideband signals. One of the important limitations of the single-tone is the effect of the attacker's synchronization with the legitimate signal. To precisely control the spoofing offset, the attacker needs to coarsely control the spoofing signal such that the phase difference between the attacker and the legitimate signals remain constant throughout the attack. We evaluate and show in Section 4.3.1 the effect of phase synchronization on this attack. Additionally, the spectral efficiency of the single-tone attack can be exploited to execute a low-power last-minute denial of service on the ILS system. This is specifically dangerous while an aircraft is executing an auto-pilot assisted approach. The block diagram of the single-tone attack signal generator is shown in Figure 7.

4 Implementation and Evaluation of Attacks

In this section, we demonstrate the feasibility and evaluate the effectiveness of the attack with the help of both simulations and actual experiments conducted using commercial aviation-grade receivers and an advanced flight simulator qualified for FAA certification.

4.1 Experimental Setup

Our experimental setup is shown in Figure 8 and Figure 9. The setup consists of four main components: i) X-Plane 11 flight simulator, ii) attacker control unit, iii) software-defined radio hardware platforms (USRP B210s) and iv) commercial aviation grade handheld navigation receiver. We use X-Plane 11 flight simulator to test the effects of spoofing attack on the ILS. X-Plane is a professional flight simulator capable of simulating several commercial, military, and other aircraft. X-Plane can also simulate various visibility conditions and implements advanced aerodynamic models to predict an aircraft's performance in abnormal conditions. It is important to note that X-Plane qualifies for FAA-certified flight training hours when used with computer systems that meet the FAA's minimum frame rate requirements. The certified versions of the software are used in numerous pilot training schools. X-Plane allows interaction with the simulator and instruments through a variety of mobile apps and UDP/IP networks. This feature allowed us to manipulate the instrument readings for evaluating our ILS attacks. Additionally, X-Plane has autopi-

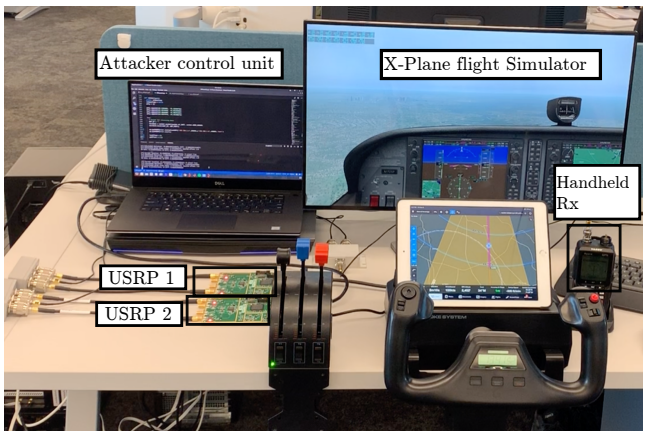


Figure 9: Photo of the experiment setup.

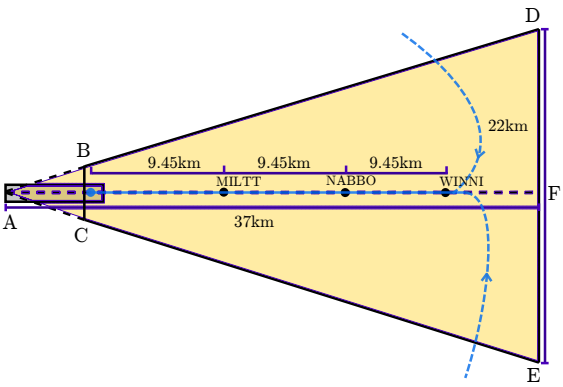


Figure 10: The spoofing zone is defined by points B, C, D, and E. WINNI, NABBO, and MILTT are the waypoints for the final approach as published for a mid-sized airport. The spoofing zone has a wide aperture as the air-traffic controller can vector in the aircraft onto the final approach in multiple ways.

lot and AI-based autoland features which we leverage in our experiments. In other words, X-Plane contains all the features and flexibility to evaluate our proposed attacks in a close to the real-world setting. The second component of our setup is the attacker control unit module which takes the location of the aircraft as input from X-Plane and generates signals for the attack. The module is also responsible for manipulating X-Plane's instrument panel based on the effect of the spoofing signal on the receiver. The attacker control unit module is a laptop running Ubuntu and contains four submodules: spoofing zone detector, offset correction algorithm, legitimate signal generator, and attacker signal generator. The spoofing zone detector identifies whether an aircraft is entering its first waypoint of the final approach and triggers the start of spoofing. The spoofing zone detector plays an important role in timely starting of the spoofing attack so as to prevent any abrupt changes in the instrument panel and therefore avoid suspicion. The offset correction algorithm uses the current location of the aircraft to continuously correct its spoofing signals taking into consideration aircraft's corrective actions. Note that the location data received from X-Plane can be analogous to receiving the location data through ADS-B signals [29] in the real world. The output of the offset correction algorithm is used to generate fake ILS signals. We also generate legitimate signals to evaluate the effect of overshadow and single-tone attacks. We use two USRP B210s [2], one each for transmitting legitimate ILS signals and attacker signals. We conducted the experiments in both wired and wireless settings. For the experiments conducted in wireless settings, the receiver was placed at a distance of 2 meters from the transmitter. Northeastern University has access to a Department of Homeland Security laboratory which provides RF shielding thus preventing signal leakage. This is necessary as it is illegal to transmit ILS signals over the air. We use two different ILS receivers, a Yaesu FTA-750L [10] and a Sporty's SP-400 Handheld NAV/COM Aviation [3] to evaluate the attacks.

4.1.1 Spoofing Zone Detection

The spoofing zone detection algorithm enables automated and timely triggering of the spoofing signal. One of the key requirements of the zone detector is to trigger the spoofing signals without causing any abrupt changes to the instrument readings; thereby avoiding detection by the pilots. The spoofing region is shaped like a triangle following the coverage of the localizer and glideslope signals. For example, the localizer covers 17.5° on either side of the extended runway centerline and extends for about 35 km beyond the touchdown zone. Figure 10 shows the zone measurements. The attacker signals are triggered when the aircraft approaches the shaded region. The shaded region is decided based on the final approach patterns for a specific runway. We used even-odd algorithm [27] for detecting the presence of the aircraft within this spoofing zone. Absolute locations cannot be used as aircraft enter the final approach path in many different ways based on their arrival direction and air traffic controller instructions. The even-odd algorithm is extensively used in graphics software for region detection and has low computational overhead. The attacker automatically starts transmitting the signals as soon as the aircraft enters the spoofing region from the sides and the needle is yet to be centered. This prevents any sudden noticeable jumps thus allowing a seamless takeover.

4.1.2 Offset correction algorithm

The attacker's signals are pre-crafted to cause the aircraft to land with a specific offset without being detected. The pilot or the autopilot system will perform course correction maneuvers to align with the runway centerline based on the instrument readings. At this point, the instruments will continuously indicate the spoofed offset irrespective of the aircraft's location and maneuvers raising suspicion of an instrument failure. To prevent this, we developed a real-time offset correc-

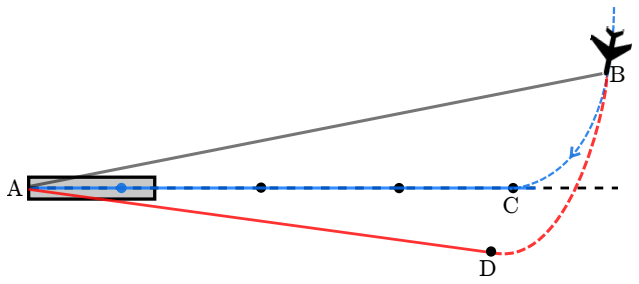


Figure 11: Offset correction algorithm takes into account aircraft's current position to calculate the difference in the spoofed offset and the current offset.

tion and signal generation algorithm that crafts the spoofing signals based on the aircraft's current location in real-time. The attacker can use the GPS coordinates if present inside the aircraft or leverage the ADS-B packets containing location information on the ground. We explain the offset correction algorithm using Figure 11. Consider an aircraft at point B, cleared to land and entering the spoofing zone. The air-traffic controller instructs the aircraft to intercept point C on the extended runway centerline. Assuming that the attacker's spoofing signal contains a pre-crafted offset to the left of the runway forcing the aircraft to follow path DA instead of CA. The offset correction module computes the current offset of the aircraft with respect to the centerline and subtracts the current offset from the spoofed offset to estimate the desired change in the course. Thus, the correction Δ required to be introduced is the difference between required offset angle $\angle DAC$ and the current offset angle $\angle BAC$. Note that offsets to the left of centerline are considered negative offsets and offsets to the right are considered positive offsets. The current offset θ can be estimated using $\theta = tan^{-}1[(m_{CA} - m_{BA})/(1 + m_{BA} * m_{CA})]$, where m is the slope. m_{CA} is typically hardcoded and is specific for each runway. m_{BA} can be estimated using the longitude and latitudes of the touchdown point and the current location of the aircraft. Now, the correction Δ is converted to the respective difference in depth of modulation value using the formula $DDM = (DDM_{fullscale} * \Delta)/2.5$, where 2.5 is the angle that results in full-scale deviation and $DDM_{fullscale}$ is the difference in depth of modulation that causes full-scale deviation. The amplitude of the individual 90 and 150 Hz components is estimated using the formula $0.2 + (DDM/2)$ and sent to the signal generator module which then transmits the required signal. Note that the value 0.2 comes from the legitimate signal's depth of modulation. The algorithm was implemented on a laptop running Ubuntu and took less than 5 ms on average to compute the offsets. The complete algorithm is shown in Algorithm 1.

Algorithm 1 Offset correction algorithm.

```
1:  procedure GETANGLEDIFFERENCE
2:      ∠DAC ← TargetedLocalizerOffset
3:      ∠BAC ← GetAngle(location)
4:      difference ← ∠DAC − ∠BAC
5:      return difference
6:  procedure CALCULATEDDM
7:      difference ← GetAngleDifference
8:      ddm ← (0.155 ∗ difference)/2.5
9:      AT90 ← 0.2 + (ddm)/2
10:     AT150 ← 0.2 − (ddm)/2
11:     ChangeAmplitude(AT90, AT150)
```

4.1.3 Setup Validation

We verified the working of our experimental setup as follows. First, we ensure consistency between the CDI needle displayed on the flight simulator and the handheld receiver. To this extent, we disabled the attacker signal and output only the legitimate signal to the handheld receiver based on the aircraft's location obtained from X-Plane. We manually validated that the alignment shown on the handheld receiver is the same as that of the flight simulator throughout the final approach. The uploaded attack demonstration video [2] also contains this validation for reference. We conducted the same experiment over the air in a controlled environment and verified consistency between the handheld receiver and the flight simulator cockpit. Second, we test our offset correction algorithm by maneuvering (swaying) the aircraft during its final approach. During this experiment, the offset correction algorithm should account for the maneuvers and generate corresponding ILS signals to the handheld receiver. We ensure the correctness of the algorithm by validating the consistency between the handheld receiver's CDI needle and the flight simulator cockpit. Note that we do not update the flight simulator's instrument readings for this experiment and the readings displayed in the simulator cockpit are only because of the simulator software engine. Finally, we validate the spoofing zone detector algorithm by entering the final approach from various directions and checking the trigger for beginning the spoofing attack. We are now ready to perform our attack evaluations.

4.2 Evaluation of Overshadow Attack

We evaluate the effectiveness of overshadow attack as follows. We leverage the autopilot and autoland feature of X-Plane to analyze the attack's effects avoiding any inconsistency that might arise due to human error. We configured X-Plane to land on the runway of a midsized airport in the US. This configuration is analogous to the pilot following approach instructions from the air-traffic controller. As soon as the aircraft entered the spoofing zone, the spoofing signals were transmitted along with the legitimate signals. The spoofing

[2] Video demonstration of the attack https://youtu.be/Wp4CpyxYJq4

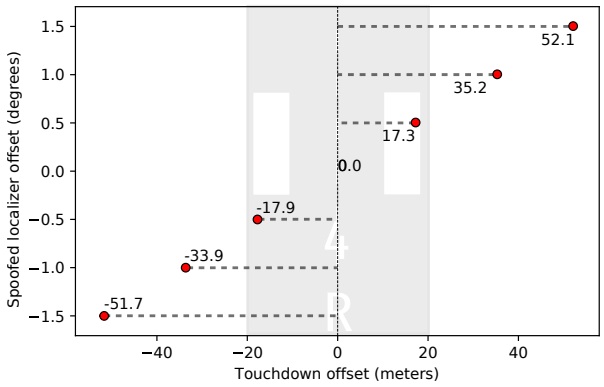


Figure 12: Results of localizer spoofing. 5 automated landings per spoofed localizer offset were executed and the touchdown offset in meters from the runway centerline was recorded.

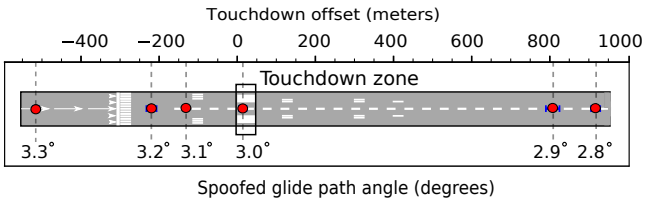


Figure 13: Results of glideslope spoofing. 5 automated landings per spoofed glideslope angle offset were executed and the touchdown offset in meters beyond the touchdown zone was recorded.

signals were generated to fake various vertical and horizontal offsets. Note that the spoofing signals were generated in real-time based on the current position of the aircraft. For the localizer (horizontal offset), spoofing signals corresponding to 0.5, 1.0, and 1.5° offset on both sides of the runway were generated. The spoofing glideslope angles were between 2.8° and 3.3°. For each spoofing angle and offset, we performed five automated landings and the results are shown in Figure 12 and Figure 13. Throughout the attack, we continuously monitored the path of the aircraft using Foreflight [3], a popular app used both by aviation enthusiasts and commercial pilots as well as X-Plane's own interfaces. We did not observe any abrupt changes in the readings and observed a smooth takeover. The aircraft landed with an 18 m offset from the runway centerline for a spoofing offset of just 0.5°. Note that this is already close to the edge of the runway and potentially go undetected by both the air-traffic controllers as well as pilots onboard, especially in low visibility conditions. In the case of glideslope, a shift in the glide path angle by 0.1° i.e., 2.9° glide path angle instead of the recommended 3°, caused the aircraft to land almost 800 m beyond the safe touchdown zone of the runway. We have uploaded a video demonstration of the attack for reference (https://youtu.be/Wp4CpyxYJq4).

4.3 Evaluation of Single-tone Attack

We evaluate the effectiveness and feasibility of the proposed single-tone attack using the experimental setup described in Section 4.1. Recall that in the single-tone attack, the attacker transmits only one of the sideband tones (either the $f_c \pm 90$ or the $f_c \pm 150$ Hz) to cause deflections in the course deviation indicator needle. We implemented the attack by configuring one of the USRPs (attacker) to transmit the sideband signals and observed its effect on the handheld navigation receiver. We observed that the spoofing signal caused the needle to deflect to the configured offset. However, the needle was not as stable as in the overshadow attack and displayed minor oscillations. This is because the specific attack is sensitive to carrier phase oscillations and therefore must be accounted for to avoid detection. A significant advantage of this attack is the power required to cause needle deflections as the attacker only transmits one of the sideband components without the carrier. This gives an almost 50% increase in power efficiency and therefore can act as a low-power last-minute denial of service attack in case the attacker is unable to establish full synchronization with the legitimate signal. In the following sections, we evaluate the effect of phase synchronization on the single-tone attack and develop a real-time amplitude scaling algorithm that can counter the phase oscillations.

4.3.1 Effect of Phase Synchronization

Recall that the single-tone attack signal is similar to a conventional double-sideband suppressed-carrier (DSB-SC) signal. It is well known that one of the drawbacks of a DSB-SC communication system is the complexity of recovering the carrier signal during demodulation. If the carrier signal used at the receiver is not synchronized with the carrier wave used in the generation of the DSB-SC signal, the demodulated signal will be distorted. In the scenario of the single-tone attack, this distortion can potentially result in changes in the difference in the depth of modulation estimates causing the needle to oscillate. We simulated the effect of phase synchronization on the single-tone attack effectiveness and present our results in Figure 14 and Figure 15. We generated the single-tone attack signal to cause full-scale deviation i.e., $\gtrsim 2.5°$ for localizer and $\gtrsim 2.5°$ for the glideslope while perfectly in sync with the legitimate carrier signal. We observe that the phase difference causes the resultant offset to change. We also noted an uncertainty region around the 90° and 270° phase difference region. This is due to the dependency in a DSB-SC system [26] between the carrier phase difference ϕ and the resulting distortion at the output which is directly proportional to the $cos\phi$. Therefore, at angles around 90° and 270°, there is an uncertainty region for the resulting offset. However, in our experiments on the handheld receiver, we noticed that

[3] Advanced Flight Planner https://www.foreflight.com

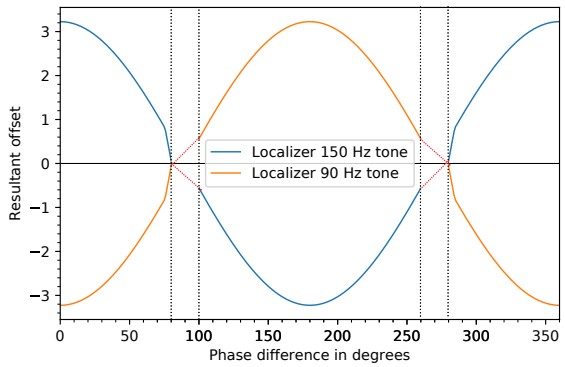


Figure 14: Comparison of calculated offset and the phase difference for localizer

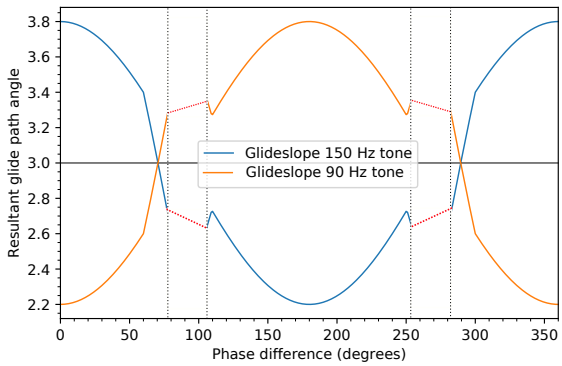


Figure 15: Comparison of calculated offset and the phase difference for glideslope

although the needle oscillated, it was not as pronounced as the simulation results indicate. One of the reasons is the rate at which the sensor measurements are being calculated and displayed on the screen. Additionally, the aircraft is in motion, therefore, causing the phase differences to cycle more rapidly than the display's refresh rate. A knowledgeable attacker can potentially leverage these properties to generate controlled spoofing signals and succeed with an optimized transmission power.

4.3.2 Real-time Amplitude Scaling

In the following, we propose and evaluate a strategy to counter the effect of phase synchronization on the single-tone attack. It is clear that the phase differences cause the output to be distorted. Besides the uncertainty region around the 90° and 270°, it is possible to predict the phase given sufficient knowledge such as aircraft speed, current location, and antenna positions. We assume such a motivated attacker for the single-tone attack evaluation in this section. It is also well known that tightly controlling the phase of a signal is not trivial and therefore our algorithm proposes to manipulate the amplitude of the attacker signal instead of the phase. Changing the amplitude of the attacker signal will compensate for the effect of phase on the signal at the receiver and we call this "real-time amplitude scaling" algorithm. The algorithm itself is inspired from prior works on amplitude scaling for DSB-SC systems [26]. We use the distance between the transmitter and the receiver to estimate the received phase of the signal by measuring complete and incomplete wave-cycles. In the simulation, we then create an ILS signal with the necessary phase shift. We also create the attacker's signal and add it to the legitimate signal to estimate the DDM. This allows us to assess the impact of phase on the transmitted signal and use this information to calculate the amplitude that will be required to counter the effects of phase. For example, if the predicted phase offset is zero, then to spoof a certain offset, the attacker needs to reduce the amplitude of its signal. We present the results of our amplitude scaling experiment in Figure 16 and Figure 17.

4.4 Comparison of Power Requirements

One of the major advantages of the single-tone attack is the improvement over the power required to execute the attack, given sufficient knowledge and environmental conditions. In this section, we evaluate and compare the power requirements of the overshadow and the single-tone attacks. We note that the absolute power profiles are specific for the handheld receivers used in the experiments. The goal of the power comparison is to verify whether there is indeed an improvement in terms of attacker's required transmission power. We present our results in Figure 18 and Figure 19. Our evaluations show the required signal strength to successfully cause 0.5° and 0.1° deviation in localizer and glideslope respectively. The received signal strength profile is shown in blue acts as a reference for the attacker based on which the attacker can compute its required power to transmit the spoofing signals. We performed the experiment by transmitting the signals to the handheld receiver and observing the success of the attack (needle indicating the intended offset). The values are a result of over 400 trials with 95% confidence interval and we find that on an average the difference in power required reaches close to 20.53 dB and 27.47 dB for the localizer and the glideslope respectively. Thus, given sufficient knowledge of the scenario, a motivated attacker can execute the single-tone attack successfully and with less power than the overshadow attack. We acknowledge that the single-tone attack has its drawbacks as described previously, however, we note that given the low power requirements, an attacker can exploit the single-tone attack to cause a low-power denial of service attack. Such an attack, especially in an aircraft's final moments before landing can be disastrous.

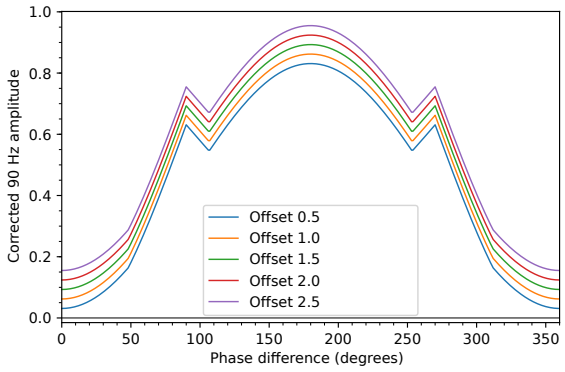


Figure 16: Amplitude scaling algorithm evaluation localizer. Amplitude required to compensate for the effect of phase

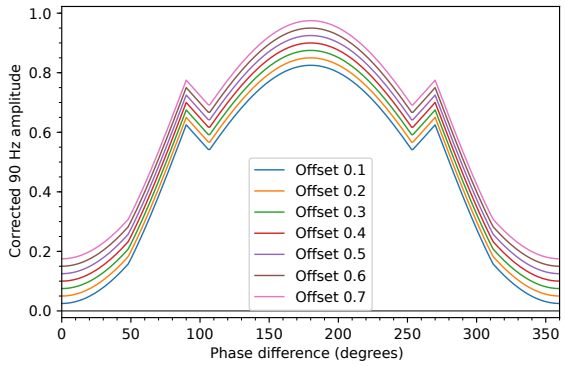


Figure 17: Amplitude scaling algorithm evaluation glideslope. Amplitude required to compensate for the effect of phase

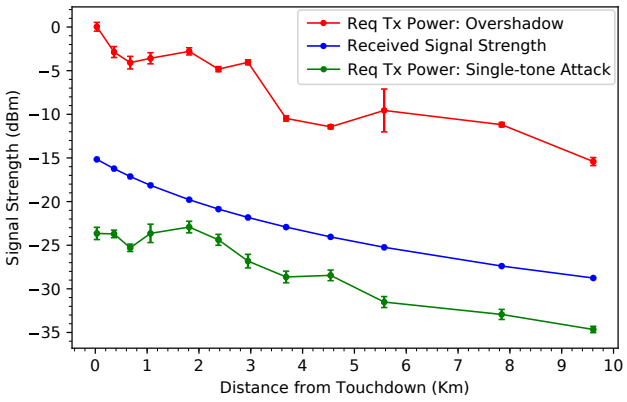


Figure 18: Comparison of required received signal strength for attack methodologies for the localizer

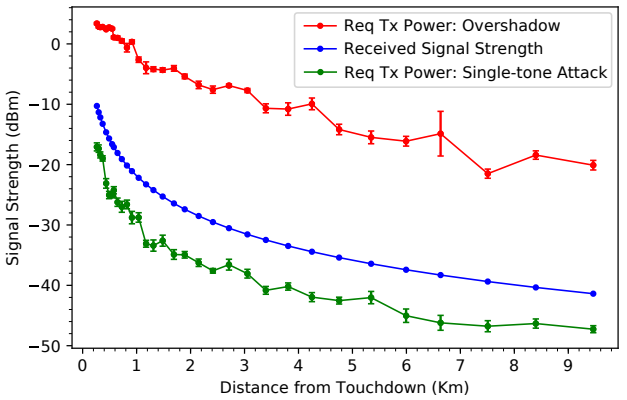


Figure 19: Comparison of required received signal strength for attack methodologies for glideslope

5 Discussion

Receiving antenna characteristics and location of the attacker. The receiver hardware and its characteristics[4] vary depending on the type of aircraft it is mounted on. For example, Cessna aircraft have their ILS antennas on the tail-fin or the vertical stabilizer. We note that the same antenna is typically used for a number of systems such as VOR, ILS, and DME; each signal arriving from a different direction. For commercial aircraft, the antennas are typically located on the nose of the plane with a forward-looking single broad lobe receiving beam pattern. Certain large aircraft, specifically those capable of landing with high nose attitude, the antennas are located either on the underside or on the landing gear of the aircraft itself [5]. The antenna equipment onboard plays an important role in determining the optimum location of the attacker to execute the attack. The ideal location of an on-ground attacker is at a point along the centerline of the runway that falls within the receiving lobe of the onboard antennas. Attackers inside the plane will have to deal with signal attenuation caused by the body of the aircraft itself and position the spoofing signal transmitter accordingly. A thorough investigation is required to fully understand the implications and feasibility of an on-board attacker and we intend to pursue the experiments as future work. The location of the attacker plays a more significant role in the scenario of the single-tone attacker since the attacker has to carefully predict the phase and accordingly manipulate the amplitude of the spoofing signal. The problem of identifying optimum locations for the attack is an open problem very similar to the group spoofing problem [56] proposed as a countermeasure for GPS spoofing attacks. In our context, the attacker has to identify locations on the ground such that the phase difference between the legitimate signal and the spoofing signal remains a constant along the line of approach. Recall that in the single-tone attack, the

[4] https://www.easa.europa.eu/certification-specifications/cs-23-normal-utility-aerobatic-and-commuter-aeroplanes

[5] https://www.casa.gov.au/sites/g/files/net351/f/_assets/main/pilots/download/ils.pdf

offset indicated by the cockpit is sensitive to phase changes and therefore locations that allow constant phase differences can result in a fixed spoofing offset and therefore minimal oscillations in the readings.

ILS Categories. The main advantage of ILS is that the pilot need not have visuals of the runway during the final approach as the ILS system is intended to guide the aircraft to a safe landing. The ILS categories are classified based on the maximum decision height at which a missed approach must be initiated if the pilot does not have a visual reference to continue the approach. In CAT I the decision height is at 60 m above the ground i.e., if the pilot does not have a visual reference at this height, a missed approach or go around must be initiated. The decision height for CAT III is as low as 15 m above the ground. The demonstrated attacks can cause severe consequences in CAT III systems due to the low decision height. It might potentially be too late to execute a missed approach in case of an attack. The consequences of the attack on CAT I and CAT II systems are less catastrophic. However, they can still cause major air traffic disruptions. Note that CAT I approach is mostly used by smaller flights. Commercial flights typically fly a CAT II or CAT III approach.

Alternative technologies and potential countermeasures. Many navigation technologies such as HF Omnidirectional Range, Non-directional Beacons, Distance Measurement Equipment and GPS provide guidance to the pilot during the different phases of an aircraft's flight. All the mentioned navigation aids use unauthenticated wireless signals and therefore vulnerable to some form of a spoofing attack. Furthermore, it is worth mentioning that only ILS and GPS are capable of providing precision guidance during the final approach. Also, ILS is the only technology today that provides both lateral and vertical approach guidance and is suitable for CAT III ILS approaches.

Most security issues faced by aviation technologies like ADS-B, ACARS and TCAS can be fixed by implementing cryptographic solutions [50] [52]. However, cryptographic solutions are not sufficient to prevent localization attacks. For example, cryptographically securing GPS signals [24, 33] similar to military navigation can only prevent spoofing attacks to an extent. It would still be possible for an attacker to relay the GPS signals with appropriate timing delays and succeed in a GPS location or time spoofing attack. One can derive inspiration from existing literature on mitigating GPS spoofing attacks [30, 31, 34, 35, 46, 56] and build similar systems that are deployed at the receiver end. An alternative is to implement a wide-area secure localization system based on distance bounding [19] and secure proximity verification techniques [45]. However, this would require bidirectional communication and warrant further investigation with respect to scalability, deployability etc.

Experiment Limitations. Our experimental setup described in Section 4 was carefully constructed in consultation with aviation experts. Since we use an FAA accredited flight simulator, we sent our configuration files and scripts to a licensed pilot for them to perform final approaches using the instruments and give us feedback. We were mainly concerned whether there was any other indicator on the cockpit that raises suspicion about the attack. We conducted our attack evaluations in both wired and controlled wireless settings. Note that it is illegal to transmit ILS signals over the air in a public space. Effects due to aircraft's motion such as Doppler shift do not affect the attacker signal as these are receiver end problems and the receiver hardware already accounts for such effects for the legitimate signal. Note that the attacker closely imitates the legitimate signals in frequency and amplitude. In short, we made the best effort to replicate a real-world approach. However our setup has its limitations. We did not perform the experiments on a real aircraft which would give us more insights on the effects of aircraft's construction, antenna placements, cockpit display sensitivity, etc. One of the factors that will get affected is the power required by the attacker. Note that commercial ILS transmitters use a 25 watts transmitter for localizer signals and a 5 W power for the glideslope signals. To put things in perspective, a standard 12 V 10 Ah battery can power a 24 Watts amplifier for about 5 hours. Furthermore, we are in touch with a leading aircraft manufacturer for access to such an experiment. We also note that we are in the process of acquiring IRB approval to recruit commercial pilots and studying their response to the attack proposed in this paper.

6 Related Work

Over the years, the aviation industry has largely invested and succeeded in making flying safer. Security was never considered by design as historically the ability to transmit and receive wireless signals required considerable resources and knowledge. However, the widespread availability of powerful and low-cost software-defined radio platforms has altered the threat landscape. In fact, today the majority of wireless systems employed in modern aviation have been shown to be vulnerable to some form of cyber-physical attacks. In this section, we will briefly describe the various attacks demonstrated in prior work. Strohmeier et al. [53] provide a comprehensive analysis of the vulnerabilities and attacks against the various wireless technologies that modern aviation depends on. Voice communication over VHF is primarily used to transfer information between the air traffic controller and the aircraft. There have already been incidents [51] related to spoofed VHF communications and several efforts [23] to design a secure radio communication system. Primary surveillance radars have been shown to be vulnerable to signal jamming attacks [40]. Secondary surveillance radars [6] leverage the ability of the aircraft to respond to ground-based interrogations for aircraft localization. Due to the unauthenticated nature of these messages, it is possible for an attacker to use publicly available implementations for software-defined radio platforms to mod-

ify, inject and jam messages creating a false picture of the airspace. Such attacks where even demonstrated to be low-power, targeted, and stealthy against sophisticated wireless systems such as Wi-Fi [59], and WPA-Enterprise [21]. The ADS-B protocol used by aircraft to transmit key information such as position, velocity and any emergency codes also face the same challenges of active and passive attacks due to the unauthenticated nature of the signals. Several works have repeatedly demonstrated the vulnerabilities of ADS-B signals [7, 18, 22, 38, 47, 48, 52, 54, 60]. ACARS [5], the data link communications system between aircraft and ground stations was found to leak a significant amount of private data [36, 50, 55] e.g., passenger information, medical data and sometimes even credit card details were transferred. Furthermore, an attacker can spoof TCAS messages [42, 48] creating false resolution advisories and forcing the pilot to initiate avoidance maneuvers. For navigation, the aviation industry relies on a number of systems such as ILS, GPS, VOR, and DME. Although the use of VOR and DME are rapidly decreasing, ILS and GPS will be in use for a very long time and are the only technologies available today for enabling autonomous landing. It is also well established that GPS is vulnerable to signal spoofing attacks [11, 13, 32, 39, 41, 56, 61]. Researchers have also demonstrated [43, 44] the feasibility of signal manipulation in the context of data communication systems. However, there has been no prior work on the security guarantees of ILS and this paper is a work in that direction. It is important to note that although many of the security issues in the aviation industry can be fixed by implementing some sort of cryptographic authentication, they are ineffective against the ILS attacks demonstrated in this paper.

7 Conclusion

In this work, we presented a first security evaluation of aircraft instrument landing system against wireless attacks. Through both simulations and experiments using aviation grade commercial ILS receivers and FAA recommended flight simulator, we showed that an attacker can precisely control the approach path of an aircraft without alerting the pilots, especially during low-visibility conditions. We discussed potential countermeasures including failsafe systems such as GPS and showed that these systems do not provide sufficient security guarantees and there are unique challenges to realizing a scalable and secure aircraft landing system.

Acknowledgements

This work was partially supported by NSF grants 1850264, 502481, and 502494. We thank civil air patrol volunteer Vaibhav Sharma for his valuable feedback.

References

[1] Air Traffic Activity System (ATADS). https://aspm.faa.gov/opsnet/sys/Airport.asp.

[2] Ettus research llc. http://www.ettus.com/.

[3] Sporty's SP-400 Handheld NAV/COM Aviation Radio.

[4] Aircraft serious incident report occurances number 00/2518 b767-319er zk-ncj, Civil Aviation Authority of New Zealand, 2002.

[5] Introduction to ACARS Messaging Services, International Communications Group, April 2006. https://www.icao.int/safety/acp/inactive%20working%20groups%20library/acp-wg-m-iridium-7/ird-swg07-wp08%20-%20acars%20app%20note.pdf.

[6] Aeronautical Telecommunications - Surveillance and Collision Avoidance Systems, International Civil Aviation Organization, 2007. https://store.icao.int/.

[7] Forget any security concern and welcome Air Force One on Flightradar24!, 2011. https://theaviationist.com/2011/11/24/af1-adsb.

[8] Status Report BFU EX010-11, German Federal Bureau of Aircraft Accident Investigation, 2011.

[9] Acceptable Means of Compliance and Guidance Material to Part-SERA, European Aviation Safety Agency, Sep 2012. https://www.easa.europa.eu/sites/default/files/dfu/NPA%202012-14.pdf.

[10] Yaesu FTA-750L, 2012. https://www.yaesu.com/airband/indexVS.cfm?cmd=DisplayProducts&DivisionID=2&ProdCatID=204&ProdID=1777.

[11] UT Austin Researchers Successfully Spoof an $80 million Yacht at Sea, 2013. http://news.utexas.edu/2013/07/29/ut-austin-researchers-successfully-spoof-an-80-million-yacht-at-sea.

[12] Stick shaker warning on ILS final, June 2014. https://www.onderzoeksraad.nl/en/onderzoek/1949/stick-shaker-warning-on-ils-final.

[13] Hacking A Phone's GPS May Have Just Got Easier, 2015. http://www.forbes.com/sites/parmyolson/2015/08/07/gps-spoofing-hackers-defcon/.

[14] Air Traffic By The Numbers, Nov 2017. https://www.faa.gov/air_traffic/by_the_numbers.

[15] Hawker Siddeley HS121 Trident, 2017. https://www.baesystems.com/en/heritage/hawker-siddeley-hs121-trident.

[16] Statistical Summary of Commercial Jet Airplane Accidents Worldwide Operations | 1959 – 2016, Boeing, 2017. www.boeing.com/news/techissues/pdf/statsum.pdf.

[17] Aeronautical Telecommunications - Radio Navigational Aids, Volume 1, 2018. https://store.icao.int/.

[18] Paul Berthier, José M Fernandez, and Jean-Marc Robert. Sat: Security in the air using tesla. In *Proceedings of the IEEE/AIAA 36th Digital Avionics Systems Conference (DASC)*, 2017.

[19] Stefan Brands and David Chaum. Distance-bounding protocols. In *Workshop on the theory and application of cryptographic techniques on Advances in cryptology*, 1993.

[20] Capt. Dennis M. McCollum. Evaluation of Instrument Landing System DDM Calibration Accuracies, 1983. http://www.dtic.mil/dtic/tr/fulltext/u2/a138301.pdf.

[21] Aldo Cassola, William Robertson, Engin Kirda, and Guevara Noubir. A practical, targeted, and stealthy attack against WPA-Enterprise authentication. In *Proceedings of the 20th Annual Network & Distributed System Security Symposium, NDSS'13*, 2013.

[22] Andrei Costin and Aurélien Francillon. Ghost in the Air (Traffic): On insecurity of ADS-B protocol and practical attacks on ADS-B devices. *BlackHat USA 2012*.

[23] Romano Fantacci, Simone Menci, Luigia Micciullo, and Laura Pierucci. A secure radio communication system based on an efficient speech watermarking approach. *Proceedings of the Security and Communication Networks*, 2009.

[24] Ignacio Fernández-Hernández, Vincent Rijmen, Gonzalo Seco-Granados, Javier Simon, Irma Rodríguez, and J David Calle. A Navigation Message Authentication Proposal for the Galileo Open Service. *Navigation*, 2016.

[25] Booz Allen Hamilton. ASRS - Aviation Safety Reporting System. https://asrs.arc.nasa.gov.

[26] Simon Haykin. *Communication systems*. 2008.

[27] Kai Hormann and Alexander Agathos. The point in polygon problem for arbitrary polygons. *Computational Geometry*, 2001.

[28] International Telecommunication Union. *Radio Regulations*. 2012.

[29] ITU-R - Radiocommunications Sector for ITU. Reception of automatic dependent surveillance broadcast via satellite and compatibility studies with incumbent systems in the frequency band 1 087.7-1 092.3 mhz. 2017.

[30] Kai Jansen, Matthias Schäfer, Daniel Moser, Vincent Lenders, Christina Pöpper, and Jens Schmitt. Crowd-GPS-sec: Leveraging crowdsourcing to detect and localize GPS spoofing attacks. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2018.

[31] Kai Jansen, Nils Ole Tippenhauer, and Christina Pöpper. Multi-receiver GPS spoofing detection: Error models and realization. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016.

[32] Roger G. Johnston Jon S. Warner. A Simple Demonstration that the Global Positioning System (GPS) is Vulnerable to Spoofing, 2003. https://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-03-2384.

[33] Andrew J Kerns, Kyle D Wesson, and Todd E Humphreys. A blueprint for civil GPS navigation message authentication. In *Proceedings of the IEEE/ION Symposium on Position, Location and Navigation Symposium (PLANS)*, 2014.

[34] Samer Khanafseh, Naeem Roshan, Steven Langel, Fang-Cheng Chan, Mathieu Joerger, and Boris Pervan. GPS spoofing detection using RAIM with INS coupling. In *Proceedings of the IEEE/ION Symposium on Position, Location and Navigation Symposium (PLANS)*, 2014.

[35] Brent M Ledvina, William J Bencze, Bryan Galusha, and Issac Miller. An in-line anti-spoofing device for legacy civil GPS receivers. In *Proceedings of the International Technical Meeting of the Institute of Navigation*, 2010.

[36] Frank Leipold. Session 5: Views of airlines and pilots lufthansa airlines 2014-05-27, May 2014.

[37] Domenic Magazu III. Exploiting the automatic dependent surveillance-broadcast system via false target injection. Technical report, Air Force Inst of Tech Wright-Patterson AFB OH Dept of Electrical and Computer Engineering, 2012.

[38] Donald L McCallie. Exploring potential ads-b vulnerabilites in the faa's nextgen air transportation system. Technical report, Air Force Inst of Tech Wright-Patterson AFB OH Dept of Electrical and Computer Engineering, 2011.

[39] Sashank Narain, Aanjhan Ranganathan, and Guevara Noubir. Security of GPS/INS based on-road location tracking systems. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2019.

[40] Naval Air Warfare Center. Electronic warfare and radar systems engineering handbook, 2013. http://www.navair.navy.mil/nawcwd/ewssa/downloads/nawcwd%20tp%208347.pdf.

[41] Tyler Nighswander, Brent M. Ledvina, Jonathan Diamond, Robert Brumley, and David Brumley. GPS software attacks. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2012.

[42] Pietro Pierpaoli, Magnus Egerstedt, and Amir Rahmani. Altering uav flight path by threatening collision. In *Proceedings of the IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*, 2015.

[43] Christina Pöpper, Nils Ole Tippenhauer, Boris Danev, and Srdjan Capkun. Investigation of signal and message manipulations on the wireless channel. In *Proceedings of the European Symposium on Research in Computer Security*, 2011.

[44] HU Qiao, Yuanzhen Liu, Anjia Yang, and Gerhard Hancke. Preventing overshadowing attacks in self-jamming audio channels. *IEEE Transactions on Dependable and Secure Computing*, 2018.

[45] Aanjhan Ranganathan and Srdjan Capkun. Are we really close? Verifying proximity in wireless systems. *IEEE Security & Privacy*, 2017.

[46] Aanjhan Ranganathan, Hildur Ólafsdóttir, and Srdjan Capkun. SPREE: A spoofing resistant GPS receiver. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*. ACM, 2016.

[47] Krishna Sampigethaya, Radha Poovendran, and Linda Bushnell. Assessment and mitigation of cyber exploits in future aircraft surveillance. In *Proceedings of the IEEE Aerospace Conference*, 2010.

[48] Matthias Schäfer, Vincent Lenders, and Ivan Martinovic. Experimental analysis of attacks on next generation air traffic communication. In *Proceedings of the International Conference on Applied Cryptography and Network Security*, 2013.

[49] Diana Siegel and R John Hansman. Development of an autoland system for general aviation aircraft. Technical report, 2011.

[50] M. Smith, M. Strohmeier, V. Lenders, and I. Martinovic. On the security and privacy of acars. In *Proceedings of Integrated Communications Navigation and Surveillance (ICNS)*, 2016.

[51] Tim H Stelkens-Kobsch, Andreas Hasselberg, Thorsten Mühlhausen, Nils Carstengerdes, Michael Finke, and Constantijn Neeteson. Towards a more secure atc voice communications system. In *Proceedings of the IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*, 2015.

[52] M. Strohmeier, V. Lenders, and I. Martinovic. On the security of the automatic dependent surveillance-broadcast protocol. *IEEE Communications Surveys Tutorials*, 2015.

[53] Martin Strohmeier, Matthias Schäfer, Rui Pinheiro, Vincent Lenders, and Ivan Martinovic. On perception and reality in wireless air traffic communication security. *IEEE Transactions on Intelligent Transportation Systems*, 2017.

[54] Allan Tart and Tõnu Trump. Addressing security issues in ADS-B with robust two dimensional generalized sidelobe canceller. In *Proceedings of 22nd International Conference on Digital Signal Processing (DSP)*, 2017.

[55] Hugo Teso. Aircraft hacking: Practical aero series. In *Proceedings of HITB Security Conference*, 2013.

[56] Nils Ole Tippenhauer, Christina Pöpper, Kasper Bonne Rasmussen, and Srdjan Capkun. On the requirements for successful GPS spoofing attacks. In *Proceedings of the 18th ACM Conference on Computer and communications security*, 2011.

[57] U.S. Department of Transportation. *Nondirectional Beacon (NDB) Installation Standards Handbook*. 1981.

[58] U.S. Department of Transportation. *Instrument Flying Handbook*. 2012.

[59] Triet Dang Vo-Huu, Tien Dang Vo-Huu, and Guevara Noubir. Interleaving jamming in Wi-Fi networks. In *Proceedings of the 9th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2016.

[60] Linar Yusupov. ADSB-Out, 2017. https://github.com/lyusupov/ADSB-Out.

[61] Kexiong Curtis Zeng, Shinan Liu, Yuanchao Shu, Dong Wang, Haoyu Li, Yanzhi Dou, Gang Wang, and Yaling Yang. All your GPS are belong to us: Towards stealthy manipulation of road navigation systems. In *Proceedings of the 27th USENIX Security Symposium*, 2018.

Please Pay Inside: Evaluating Bluetooth-based Detection of Gas Pump Skimmers

Nishant Bhaskar, Maxwell Bland, Kirill Levchenko[†], and Aaron Schulman
University of California, San Diego [†]*University of Illinois Urbana-Champaign*

Abstract

Gas pump skimming is one of the most pervasive forms of payment card attacks in the U.S. today. Gas pump skimmers are easy to install and difficult to detect: criminals can open gas pump enclosures and hide a skimmer in internal payment wiring. As a result, officials have resorted to detecting skimmers by performing laborious manual inspections of the wiring inside gas pumps. In addition, criminals can also avoid being caught using skimmers: many gas pump skimmers have Bluetooth connectivity, allowing criminals to collect payment data safely from inside their car.

In this work, we evaluate if the use of Bluetooth in skimmers also creates an opportunity for officials to detect them without opening gas pumps. We performed a large-scale study where we collected Bluetooth scans at 1,185 gas stations in six states. We detected a total of 64 Bluetooth-based skimmers across four U.S. states—all of which were recovered by law enforcement. We discovered that these skimmers were clearly distinguishable from legitimate devices in Bluetooth scans at gas stations. We also observed the nature of gas station skimming: skimmers can be installed for months without detection, and MAC addresses of skimmers may reveal the criminal entity installing or manufacturing them.

1 Introduction

Payment card skimming attacks at gas pumps have reached alarming levels. In 2018, law enforcement officials recovered 972 skimmers from gas pumps in Florida [11] and 148 skimmers from Arizona [10] alone. Based on industry estimates, a single skimmer can capture 30–100 credit cards per day [5] and each card, based on estimates from law enforcement officials, nets the criminal an estimated $500 [53], resulting in a daily loss of $15,000–50,000 per day of operation for each skimmer.[1] Less is known about how long a skimmer remains in operation, but allowing for even one day of operation per skimmer, 2018 losses exceed $16 million across these two states.

Gas pumps are an ideal skimming target. Gas pumps have relatively weak security: their payment circuitry can be accessed with universal keys or crowbars, and reading payment data is as easy as tapping into a ribbon cable (Section 2.1). Gas pump skimmers can be hidden inside of a gas pump enclosure, making them difficult to detect. As a result, inspectors have resorted to manually opening the pumps to inspect their wiring for skimmers. Gas pump skimming has become so pervasive that the Arizona Department of Agriculture, Weights and Measures Division (AZWMSD) now checks for skimmers while doing routine inspections.[2] From 2016 to 2018, the AZWMSD looked for skimmers in 7,325 gas station inspections. Inspectors found skimmers in only 1.5% of these inspections.

Unfortunately, Law Enforcement (LE) rarely catch criminals while they are collecting payment data from gas pump skimmers. The reason is, many gas pump skimmers are equipped with Bluetooth connectivity [26, 27, 28, 29]. This allows criminals to remain in their car while wirelessly retrieving card payment data. While Bluetooth is a vital tool for criminals to exfiltrate data from gas pumps, it also could be an opportunity to make it easier to detect skimmers.

In this paper, we evaluate the effectiveness of detecting skimmers with Bluetooth scanning from a smartphone. Indeed, if a skimmer can be detected with a smartphone, then authorities can discover and remove skimmers passively and quickly while they visit a gas station for other reasons. We built a smartphone application to perform this study, called Bluetana. Bluetana collects all Bluetooth scan data that is available via the Android Bluetooth APIs. We equipped 44 volunteers in six U.S. states with smartphones running Bluetana. Our volunteers have collected scans at 1,185 gas stations, where they observed a total of 2,562 Bluetooth devices. In these scans, Bluetana detected a total of 64 skimmers installed at gas stations in Arizona, California, Nevada,

[1] In Section 2.2, we compare these quoted estimates to other sources, and find them to be in agreement.

[2] For example, the "Vapor Recovery Inspection Pre-Test Checklist" has a checkbox for "Checked for Skimmers".

and Maryland, and it was the sole source of information that led law enforcement to find 33 skimmers.

The primary result of this study is the first comprehensive look at how skimmers can appear in Bluetooth scans. Namely, we observe that it is feasible to differentiate skimmers from other common Bluetooth devices that appear in Bluetooth scans at gas stations (e.g., vehicle telemetry collectors). The main differentiating factor for the skimmers we observed, is that the Bluetooth Class-of-Device—a parameter not collected by any consumer Bluetooth scanning applications that we are aware of—is "Uncategorized". We also find that signal strength is a reliable way to determine if a Bluetooth device is located near a gas pump, and thus could be a skimmer.

Our study reveals several problems with consumer Bluetooth-based skimmer detection applications [46, 2, 51]: (1) there are many legitimate products that appear at gas stations that use the same Bluetooth modules as known skimmers; therefore, MAC address-prefix based detection may lead to false positives, (2) there are many Bluetooth modules used in skimmers that do not comply with IEEE MAC assignment requirements. We also debunk advice on how to find skimmers with Bluetooth scans from authorities [4] and viral information from social media [33]. For instance, none of the skimmers we found using Bluetooth scans have a name that is a long string of letters and numbers.

Performing this in-depth study brought to light several important operational lessons learned about the importance of detecting skimmers with Bluetooth. Using Bluetooth scans, officials detected skimmers while driving by gas stations that they otherwise would not have inspected. We also witnessed several instances where an inspector tried to find a skimmer, but could not find it on their first pass looking inside a gas pump. However they persisted and found it based on the knowledge that a suspected skimmer had appeared in Bluetooth scans. Surprisingly, we observed that there are skimmers installed in the same gas station, or city, that have very similar MAC addresses—indicating their source is a single criminal entity. We even found skimmers installed hundreds of miles away that had surprisingly close MAC addresses.

The rest of the paper is organized as follows: Section 2 provides background on internal gas pump skimming: their construction, monetary incentive, and prevalence in the wild. Section 3 is an overview of our large-scale Bluetooth scan collection methodology. In Section 4, we present the results of our study: what the skimmers we detected look like, how they compare to skimmers recovered independently by Law Enforcement, and whether they are well hidden in the Bluetooth environment. In Section 5, we present possible counter measures to the Bluetooth detection. In Section 6 we present the operational lessons we learned about skimming and criminal investigation procedure, while performing our large scale measurement study. Section 7 is related work, and we conclude in Section 8.



Figure 1: An internal Bluetooth-based skimmer wrapped in grey tubing to blend in with the cabling inside the fuel pump. This skimmer was detected by Bluetana in Tempe, AZ.

2 Background

Skimmers are illicit devices that capture credit card magnetic stripe data when a card is used at a point-of-sale (PoS) terminal or automatic teller machine (ATM). External skimmers use a magnetic head concealed in a false faceplate to read the magnetic stripe of a card as it is inserted into the real card reader. However, this paper is concerned with a newer class of skimmers, called *internal skimmers*, that are installed entirely inside a PoS terminal or ATM, leaving no visual evidence of its presence [47]. Internal skimmers are attached inline to the cable that connects the card reader to the main circuit board of the PoS terminal, tapping into the data and drawing power. To make data collection easier, many internal skimmers include a Bluetooth-to-serial module that allows the perpetrator to covertly collect the "skimmed" card data from a safe distance. These skimmers are built using commodity hardware with a total unit cost of $20 or less.

Fuel pumps with a built-in PoS terminal have become a very popular target for such internal skimmers: they are unattended, easy to access, and have poor physical security, which make it easy to install a skimmer without being noticed. In a typical installation scenario, an attacker positions a van at a fuel station to block the station attendant's view of the target pump (Excerpt in A.2), opens the fuel pump using a common master key or crowbar, and clips a discreet gumstick-sized skimmer to the ribbon cable between reader and main circuit board using a vampire clip (Figure 1). The entire process to install skimmer can take less than 10 seconds [1]. The perpetrator can then return to the station with a smartphone, and without leaving their vehicle, connect to the skimmer using Bluetooth and download the card data.

2.1 Internal Bluetooth Skimmers

The subject of our study are *internal, Bluetooth-based skimmers* that are installed in fuel pump PoS terminals. Figure 2 shows a typical Bluetooth skimmer, recovered from a fuel station in Southern California. This skimmer consists of a "Teensy" development board with an ARM Cortex-M4F micro-controller and a Roving Networks RN-42 Bluetooth-to-serial module. It also includes connectors for tapping into the wiring inside the pump (not shown).

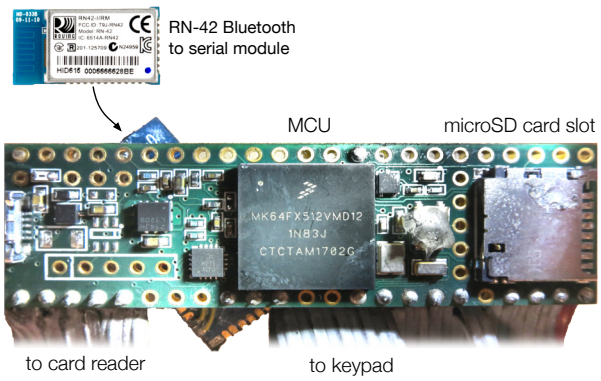


Figure 2: Parts of a typical internal Bluetooth-based fuel pump skimmer. This skimmer was detected by Bluetana.

Connections. In the figure, the ribbon cable on the left intercepts or replaces the ribbon cable that connects the magnetic stripe reader to the PoS terminal main board. The skimmer also uses this connection for power: the power and ground pins of the Teensy (on far left of board, not visible in Figure 2) are connected to power and ground on the card reader cable. The ribbon cable on the right intercepts or replaces the ribbon cable from the PoS keypad. This allows the perpetrator to capture additional card verification data, namely the debit card PIN or credit card billing ZIP Code. Availability of a PIN code with a stolen debit card in particular, can increase its value five-fold on the black market (Table 1). However, not all skimmers capture keypad data.

Most gas station skimmers read the unencrypted data pulled from magnetic stripe readers. Card issuers feel that removing sensitive data from the magnetic stripe on cards will help to solve the problem [42]. Newer literature has demonstrated attacks on chip payment systems [13, 15], and law enforcement in Latin America have begun to find EMV skimmers that are Bluetooth enabled [30, 3].

Controller board. The skimmer pictured in Figure 2 used a Teensy micro-controller development board equipped with a 120 MHz ARM Cortex-M4F micro-controller made by Freescale Semiconductor. By using a development board, a skimmer requires only rudimentary electronic assembly: soldering wires to the development board.

However, skimmers have also been found using what appeared to be fully custom-designed boards. These are compact, making them better for hiding in the dispenser. Examples of micro-controllers used in recovered skimmers include Microchip PIC18F4550 [2] and Atmel XMEGA128A4U [3].

Storage. The Teensy board also has a microSD card slot for additional data storage. Skimmers built on custom PCBs have also used flash and EEPROM ICs for storage. The storage capacities vary across designs, with examples using the PCT25VF032B (32-Mbit) [3] and M25P16VP (16-Mbit) [2].

Bluetooth module. The skimmer shown in Figure 2 uses a Roving Networks RN-42 module, an inexpensive Bluetooth-to-serial module found in many skimmers. In Section 3.1 we describe characteristics of popular Bluetooth-to-serial modules used in recovered skimmers for wireless data exfiltration. On the Bluetooth side, a Bluetooth-to-serial module provides a Serial Port Peripheral interface, which most operating systems recognize as a Bluetooth modem and instantiate a serial device for it. Operating systems will create a corresponding serial device, allowing user-space applications, namely a criminal's card dumping application, to communicate with the module. On the hardware side, a Bluetooth-to-serial module provides a TTL-level receive and transmit pin, allowing it to interface to any micro-controller UART. The module this allows even the simplest micro-controller to communicate via Bluetooth with a host device. The 2.4GHz Bluetooth antenna is included on the module's circuit board (exposed area to the left of the metal shield for the module shown in Figure 2), so the antenna is also hidden.

Bluetooth-to-serial modules generally require no configuration, however, most can be reconfigured using Hayes-style modem AT commands. In Section 4.1 we describe the configuration capabilities of popular modules. Notably, all of the Bluetooth-to-serial modules we found in skimmers support changing the device MAC address, Bluetooth device name, changing the pairing password, and the ability to become non-discoverable once paired.

2.2 Economics of Carding

Stealing and monetizing stolen credit and debit card data, called *carding* by its practitioners, is a well-studied form of financial fraud, however, reliable estimates of losses resulting from a single skimmer are difficult to find. To the criminal operating a skimmer, the expected revenue per skimmer breaks down as:

$$W = (\text{card value}) \times (\text{cards per day}) \times (\text{days deployed}).$$

Of these, we found published estimates for only the first two quantities, and very little about skimmer lifetimes. Here, we summarize the available data with the goal of estimating the losses incurred by a single skimmer.

Card value. To monetize stolen credit card data, skimmer installers have two options: sell the data on the black market, or cash out the cards on themselves. Based on our survey of sites selling stolen card data, black market prices for stolen cards fall in the $10–220 range, depending on whether the card is a debit or credit card, and whether it comes with a PIN (for debit) or billing ZIP code (for credit). Table 1 provides a summary of these prices with references.

Criminals can also cash out the cards themselves. Debit cards with a PIN are often cashed out by withdrawing money from an ATM, while credit cards are often cashed out by

Scheme	*Value*	*Reference*
Black market price		
Debit, no PIN	$20–30	[35, 49, 21, 44]
Debit with PIN	$110–220	[31, 49, 44]
Credit, no ZIP	$10–25	[35, 49, 21, 44]
Credit with ZIP	$25–60	[35, 49, 21, 44]
Cash-out value		
Credit or Debit (standard)	$400–800	[19, 40, 18, 56]
Credit (premium)	$1,000	[40, 45, 20]
Bank and merchant loss		
Credit	$1,003	[1]
Debit	$650	[12]
Consumer liability		
Debit (> 60 days)	unlimited	15 USC 1693g
Debit (< 60 days)	max $500	15 USC 1693g
Debit (< 2 days)	max $50	15 USC 1693g
Credit	max $50	15 USC 1643
Prosecuted loss		
Credit or debit	$500	[6]
Court documents		
Credit	$362–400	[36, 16, 8, 7]
Debit	$665–1132	[9, 50]

Table 1: Value of stolen credit and debit cards.

purchasing high-value merchandise (e.g. iPhones) and reselling them. Reported cash-out values for debit and credit cards range between $400 and $1,000, depending on credit limit associated with the card. We also conducted a survey of cash-out values reported in court documents involving skimmers.[3] Several cases reported specific cash-out values, rather than ranges. The debit card cash-out values were $1132 [36], $444 [16] $665 [8], $1354 [7]. The credit card cash-out values were $362 [50] and $400 [9].

Losses due to credit and debit card fraud are borne largely by banks and merchants. This is likely because consumer liability for fraud in the U.S. is limited to $50 for credit cards, and $50 or more for debit cards (depending on how quickly the consumer reports the fraud). Industry estimates for losses per-card incurred by banks are $650 for debit cards and, $1,003 for credit cards [1, 12]. The U.S. Sentencing Commission estimates per-card losses at $500 or more.

Cards per day. The number of cards a skimmer captures each day depends on the number of transactions at that pump, which will vary by station. Rippleshot, a payment fraud prevention service, states: "a single compromised pump can capture data from roughly 30–100 cards per day" [5]. The lower end Rippleshot's estimate agrees with the estimate of 20–50 cards per day we received from U.S. law enforcement agents. In addition, we found two court documents that report criminals captured 25 [9] and 30 [8] cards per day. We also studied 10 skimmers recovered from the field, which we were told were used and wiped daily. We found an average of 20 cards per skimmer, divided evenly between debit and credit cards.[4]

[3] We surveyed only documents available without fee from Court Listener.

Location & Year	*Recovered skimmers*	*Skimmed stations*	*Skimmers / station*	*Skimmers / 10^6 people*
San Diego				
FY 2018	42	11	3.2	11.9
Arizona				
2016	88	54	1.6	4.3
2017	57	46	1.2	2.7
2018	148	86	1.7	6.9
All	293	134	2.2	14.0
Florida				
2016	207	162	1.3	10.0
2017	650	432	1.5	31.1
2018	972	524	1.8	45.6
All	1,829	1,029	1.7	87.4

Table 2: Prevalence of skimming in three regions of the U.S.

Days deployed. Internal skimmers are not limited by battery life and can remain in operational indefinitely, because they draw power from the PoS circuitry, Skimmer lifetime, then, is limited only by how long they can remain undetected. Unfortunately, there is little reliable data on this. Our only direct experience is our discovery of a pair of skimmers that remained undetected for six months (Section 3.1). However, LE informed us that criminals may leave skimmers in gas pumps after only a few days of retrieving card data and moving on to another location. Given the very limited data available on skimmer lifetimes, we instead consider skimmer value *per day of operation*.

Cashout success rate. Our analysis of court documents revealed that criminals are often unsuccessful when trying to cashout a skimmed card. This may be due to a variety of reasons, such as the following: incorrectly reading card data, hitting daily withdrawal limits, and activating fraud alerts. Several cases mentioned that criminals were not successful in cashing all skimmed cards. One case mentions a specific cashout success rate of 47% [7].

Total skimmer value. Finally, we estimate the range of per-day revenue from a skimmer based on the prior figures. Our low end estimate is $4,253 (25 cards per day, cashout of $362 per card, and 47% cashout success rate), and our high end estimate is $63,638 (100 cards per day per day, $1,354 cashout per card, and cashout success rate of 47%).

[4] These skimmers were provided to us because they were removed by the station owner, rather than LE, making them unsuitable for use as evidence.

2.3 Skimmers Recovered in the Wild

To understand the prevalence of skimmers in the wild, we obtained data on recovered skimmers from three regions in the United States: San Diego and Imperial counties of California, with a combined population of 3.5 million; the state of Arizona, with a population of 7 million inhabitants; and the state of Florida, with a population of 21 million inhabitants. Table 2 summarizes the statistics. We note that these numbers do not represent *all* recovered skimmers. For San Diego and Imperial counties, our statistics represent the number of skimmers found by or reported to a U.S. federal law enforcement agency. For Arizona and Florida, our statistics represent skimmers found by or reported to the AZWMSD and the Florida Department of Agriculture and Consumer Services.

The number of recovered skimmers has increased from 2016 to 2018 in both Florida and Arizona. The total number of skimmers recovered in 2018 across the three geographic regions is significant: if each skimmer operated for just one day, we estimate their total monetary impact would be $17.43 million. Yet, as the skimmers-per-million people number shows, the possibility of an average consumer encountering a skimmer at a gas station is quite small.

3 Data Collection Methodology

Driven by the observation that skimmers are hard to find—few pumps in San Diego, Arizona, and Florida have been found to have skimmers installed in them (Table 2)—we created a tool, called Bluetana, to evaluate the effectiveness of Bluetooth-based skimmer detection. We begin by presenting an overview of the tool and the data it collects. Then we describe how Bluetana identifies suspicious devices and directs users to collect additional data. Finally, we discuss how we retroactively inspect data to find skimmers.

3.1 Crowdsourcing Bluetooth Scanning

We developed Bluetana, an Android-based measurement tool that officials and volunteers use to scan for skimmers at gas stations. Bluetana scans for nearby Bluetooth—both Classic and Bluetooth Low Energy (BLE)—devices every 5 seconds using Android's Bluetooth API. It collects the Bluetooth scans and geo-location data, and uploads this data to a secure database over a cellular link. Bluetana collects all of the Bluetooth scan data that Android makes available, including Device name, MAC Address, Class-of-Device[5], and signal strength (RSSI).

How we visited 1200 gas stations. We outfitted 44 volunteers and inspectors in six U.S. states (CA, AZ, MD, NC, NV, IL) with low-end smartphones running Bluetana in kiosk mode (they could not close the application). We selected officials who frequent gas stations as part of their daily job duties. Primarily, they were Weights and Measures inspectors.

[5] Class-of-Device is twenty four bits indicating the device's intended use, such as *smartphone* or *speaker*.

Indicating suspicious devices to inspire data collection

The Bluetana display shows a list of Bluetooth devices detected during scanning. When Bluetana detects a potential skimmer, it indicates this to the user by highlighting the device record (Figure 4). The Bluetooth scan profile of the modules that have been found in skimmers inform which devices we highlight in Bluetana.

Skimmers recovered by LE are often found to use CSR (Qualcomm) chip-set-based Bluetooth modules. Our highlighting procedure primarily looks for the default Bluetooth profile of these modules—with the exception of the Device Name which can be missing due to poor signal strength, and modified by criminals in an attempt to hide the device (Section 4). The factory default Bluetooth scan profile (i.e., MAC prefix, Device Name, and Class-of-Device) of these modules are as follows:

Mod.	MAC Prefix	Dev. Name	Class of Dev.
RN	`00:06:66`	"`RBNT-*`"	Uncategorized
HC	*Various*	"`HC-05/06`"	Uncategorized

Bluetana chooses a highlight color via a three-step decision process, depicted in Figure 3. First, the app checks the device's class. All skimmers studied within this work, whether discovered by Bluetana or not, had a device class of *Uncategorized*. If the device class is not uncategorized, the data is saved for later analysis. The device's MAC prefix is then compared against a "hitlist" of prefixes used in skimming devices recovered by law enforcement. If the device has a MAC that is not on this hitlist, it is unlikely to be a skimmer, and the app highlights the record yellow. Next, if the device name matches a common product using the same MAC prefix, the record highlights in orange. If all three fields (MAC prefix, Class-of-Device, and Device Name) indicate the device is likely to be a skimmer, Bluetana highlights the record in red. The highlighting procedure is the result of a year of refinements based on our experience finding skimmers in the field, and Bluetana includes a remote update procedure to account for these incremental changes.

This simple highlighting proved to be vital to our data collection. Red serves as a cue to perform signal strength localization: it directed our users to collect more samples of signal strength to determine if a device is located in the gas pump area—and is therefore likely to be a skimmer. In several cases, Bluetana highlighting a device in red was the only reason officials performed a manual skimmer inspections: out of the 64 skimmers we found, 33 were recovered because an official started an inspection only after noticing a device was highlighted in red in Bluetana.

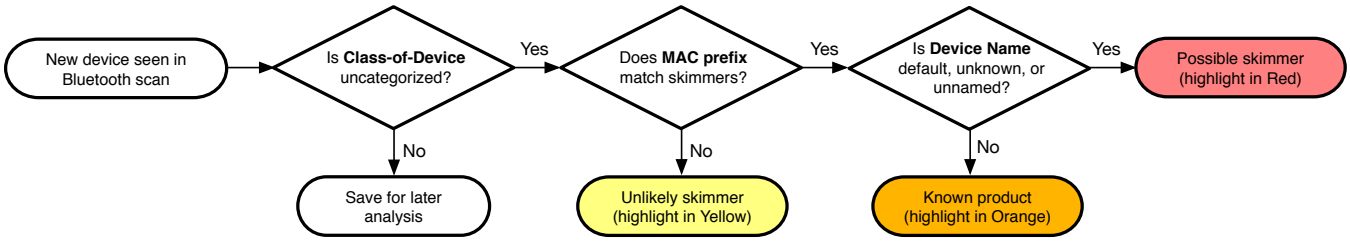


Figure 3: The procedure Bluetana uses for highlighting suspicious devices.

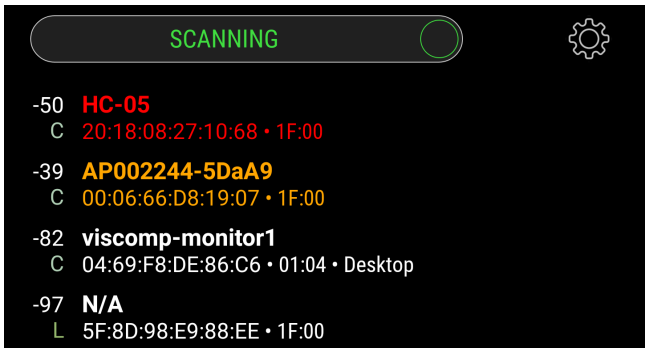


Figure 4: The Bluetana user interface. Bluetana highlights suspicious devices, inspiring users to collect more signal strength samples, and even perform inspections.

In one instance, an Arizona Weights and Measures inspector was driving by a gas station when two red highlighted devices appeared in Bluetana. He made an unscheduled stop at the gas station, performed a skimmer inspection, and discovered two skimmers. Figure 5 shows a portion of the official Arizona inspection report documenting this incident.

Bluetana's highlighting procedure is more comprehensive than other skimmer detection apps on the Play Store. Scaife et al. [46] investigated the behavior of these apps and found that they flag skimmers based on either MAC prefix or Device Name. These apps would miss skimmers with non-standard MAC prefixes or customized (missing) device names which Bluetana was able to find (Section 4.1). Bluetana also found legitimate devices that would be considered skimmers by these apps (Section 4.2).

Identifying skimmers after data collection

During the study, we manually examined every Classic Bluetooth device observed at a gas station visit in real time (as Bluetana users upload their scan data). At the beginning of our study, we relied primarily on the signal strength of the device to determine if it was a suspected skimmer. By the nature of being installed inside a gas pump, the Bluetooth signal of a skimmer is strongest in the pump area. Other devices that we suspected to be skimmers all had a low signal strength in the pump area, because aside from the cars parked at the pumps, the only places where a Bluetooth device would be located in the pump area would be inside the pump. Combining the signal strength and geo-location with satellite imagery of the gas station, we were able to easily detect when the signal was emanating from inside of a gas pump (example shown in Figure 6). While at a gas station, Bluetana users also noticed this by moving toward the pump area to see if the device's signal strength increases.

If we saw any suspicious devices in the dataset, we alerted officials that they should inspect the pumps at the station in question. Initially, we did not know which of these devices were skimmers: many initial inspections we requested turned up empty handed. However, as the study progressed, we improved our understanding of the profile of skimmers.

A natural experiment observing deployment duration

Having a database of all prior scans made it possible for us to look for skimmers that we may have missed in the past. In particular, looking back in at the database led to us to discover two skimmers that we had initially missed. A retroactive analysis of two stations discovered skimmers that were still operating even though we first detected them *six months* earlier. This natural experiment is likely the first concrete data on how long skimmers can be installed without being found in a routine or complaint-induced pump inspection.

3.2 Limitations

Selection bias

We designed our data collection to look for a specific type of gas pump skimmer: one that uses a Classic Bluetooth module, and is discoverable in Bluetooth scans. Our contacts in LE confirmed that this type of skimmer has been found in gas stations across the entire U.S. They also reported that these skimmers are particularly common in Arizona and California; therefore, these states were the focus of our study.

The results of our study may not be representative of the nature of gas pump skimming across the country. Criminals in other regions may evade Bluetooth-based detection by using alternate exfiltration methods (e.g., Bluetooth Low Energy and SMS), or configurations (e.g., non-discoverable mode). We outline these countermeasures in Section 5.

ARIZONA DEPARTMENT OF WEIGHTS AND MEASURES

1688 W. Adams St. Phoenix, AZ 85007 https://dwm.az.gov
Phone: 602-771-4920 or 1-800-277-6675 (Outside of Phoenix Metro)
Agency contact: Damien DeSantiago (602) 771-4948
State Ombudsman 602-277-7292

INSPECTION COMMENTS /NOTES
A.R.S. §41-1009(A)(7)

BMF # [redacted] INSPECTION # [redacted] TEST DATE [redacted] PAGE 1 OF 1

COMMENTS / NOTES

While using the "Bluetana" scanner two items showed up in red. I opened a fueling device skimmer inspection then announced myself to location staff. The scanner showed the strongest signal to the dispensers closet to [redacted] In dispensers 1/2 and 5/6 I found skimmers installed. For a total

Figure 5: Bluetooth scanning helps inspectors find more skimmers because they detect skimmers when driving by a gas station.

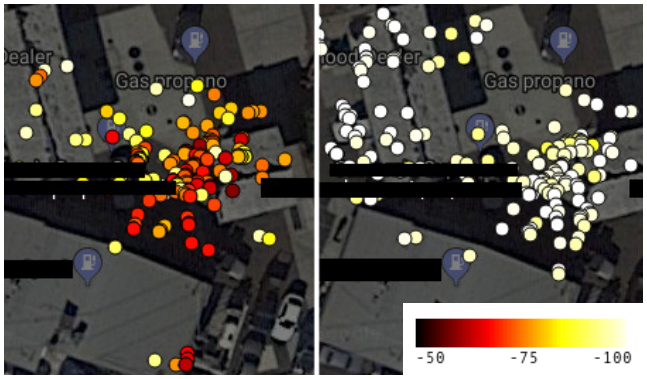


Figure 6: Combining RSSI data with satellite imagery reveals if a device is located in the pump area of a gas station.

Bluetana does not connect to devices

We could collect more data about Bluetooth devices by trying to connect to them. This could be useful for conclusively detecting a skimmer or collecting information about the type of Bluetooth device. By sending commands that skimmers are known to respond to, Bluetana would be able to see if the device responds equivalently to known skimmers. This is precisely what one of the current Bluetooth skimmer scanning applications on the Play Store does.

This practice may seem innocuous, but our discussions with law enforcement indicate that this could overwrite information critical to future investigations. The problem is, internal registers in many skimmer Bluetooth modules records the last-paired MAC address. This information can be used to link a suspect possessing a smartphone or laptop with their skimmers. The typical forensic evidence collection performed by law enforcement on skimmers includes collecting the last-paired MAC address [48].

4 Results

In this section, we present the results of our 19 month study of Bluetooth devices observed with Bluetana at 1,185 gas stations across six U.S. states (CA, AZ, NV, MD, IL, NC). During the course of this study, Bluetana detected 64 skimmers operating in 34 gas stations; all of the skimmers were removed from the pumps by local and federal law enforcement agents. Bluetooth scanning is a surprisingly effective way of detecting skimmers: in Arizona, Bluetana has detected skimmers at 1.58% of the 491 stations it scanned, and routine inspections by state inspectors had a similar detection rate of 1.5% from 2016 to 2018.

The primary result of this study is as follows: there are distinct characteristics of the 64 internal skimmers detected by Bluetana that differentiate them from the 2,562 other Bluetooth devices that Bluetana found at gas stations (e.g., car stereos). Namely, these skimmers were predominately using the default Bluetooth module configuration. Additionally, we discovered that some criminals use a custom Device Name in an apparent attempt to hide their skimmers from Bluetooth scans. These custom Device Names stand out, making them easier to differentiate from other devices.

4.1 What Do Skimmers Look Like in Scans?

We begin by presenting how skimmers we observed appear in Bluetooth scans. We describe the properties of two sets of skimmers: 64 skimmers that we detected in the field during the course of this study, as well as 23 skimmers that were independently recovered by two LE agencies. The 23 skimmers recovered independently by LE have similar characteristics to the 64 that Bluetana detected in the field. The Bluetooth characteristics of these skimmers are detailed in Table 3. We now analyze the following properties: Class-of-Device, MAC prefix, and Device Name.

All of the skimmers are "Uncategorized" Class-of-Device

Class-of-Device is primarily used to select the icon that indicates the category of a device in a Bluetooth scan (e.g., Headphones). Bluetooth modules used in skimmers analyzed in this study (i.e., HC and RN), have an "Uncategorized" Class-of-Device assigned by default. Changing Class-of-Device on these modules is trivial: the modules provide a serial command to set it. Despite this, criminals do not appear to be modifying the Class-of-Device on any of the skimmers we observed: all of the 87 skimmers detected by Bluetana and recovered independently by LE used the default "Uncategorized" device class.

	# of skimmers	
Bluetooth Scan Property	*Bluetana*	*LE*
Class-of-Device		
Uncategorized	64	23
Manufacturer (MAC prefix)		
Roving Networks		
`00:06:66`	45	13
Shenzhen Bolutek		
`98:D3:31`	1	
Unknown		
`20:13:04`	1	
`20:17:11`	1	
`20:18:01`	2	
`20:18:04`	1	
`20:18:07`	1	
`20:18:08`	4	10
`20:18:09`	4	
`20:18:10`	1	
`20:18:11`	2	
`98:D3:35`	1	
Device Name		
Default	36	23
[Law enforcement]	2	
[Mobile phone]	4	
[Indescript object]	2	
[Numerical]	2	
Unnamed	18	
Total	64	23

Table 3: Bluetooth scan properties of skimmers observed during our study. The exact Device Names are not shown, instead we describe the names we found.

MAC prefixes are often manufacturer defaults

Bluetooth module manufacturers burn a MAC address into the module's EEPROM. Although it is possible to change the MAC with a SPI-based reprogramming of the CSR chip's EEPROM, we have not observed any skimmers that have a modified MAC. The first three bytes (prefix) of the MAC address typically correspond to the manufacturer of the device.

Although MAC address prefixes are often assigned by IEEE (e.g., all of the RN Bluetooth modules have the same manufacturer MAC prefix) the HC modules have a wide variety of MAC prefixes. Of the HC modules we observed, only one has a MAC prefix assigned by the IEEE. This could make it significantly more difficult to detect an HC-equipped skimmer. However, looking at of the MAC prefixes of the skimmers that we observed, a clear pattern emerges: manufacturers appear to be burning module manufacture date into the first four bytes of the MAC address in the following format: `YY:YY:MM:(DD)`.

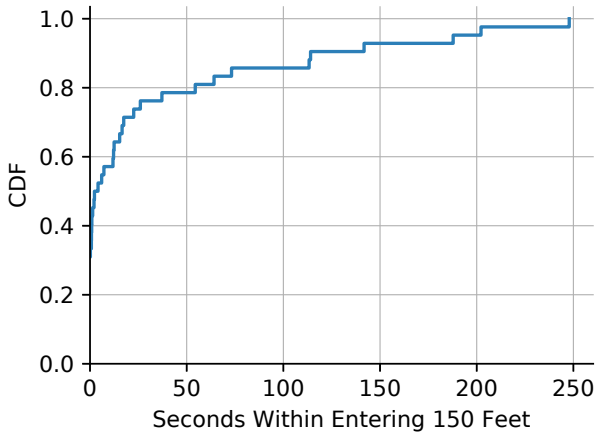


Figure 7: Skimmers are detected within a minute of passing near a gas station.

Device names are often default, occasionally customized

Device Names allow users to identify their devices in Bluetooth scans. They are assigned a factory default value by the manufacturers, and are modifiable by users. Most of the skimmers we observed had a default Device Name: namely, all of the skimmers provided by LE, and more than half the skimmers we detected in with Bluetana. A skimmer with a default Device Name looks innocuous, because some legitimate products using the same modules are also shipped with the default module name (Section 4.3). Occasionally, we found that criminals set a custom device name on their skimmers. This appears to be an attempt to make the skimmer look less suspicious. Bluetana detected custom-named skimmers with a variety of names. The custom names of skimmers discovered by Bluetana had variety: some were random strings of numbers, and others masqueraded as LE.

Bluetana did not detect a Device Name for several skimmers. This is expected because the device sends its MAC and Class-of-Device in the first scan response packet; it sends the device name in a subsequent packet (that may be missed).

Skimmers are detected within one minute

Bluetooth scanning has the benefit of detecting some skimmers without manually inspecting each of the pumps. However, attenuation from a gas pump's metal enclosure, may limit the range that Bluetooth scans are effective. We analyzed the scans from Bluetana to see how long an official had to spend at a gas station before they detected the skimmers installed there (Figure 7). The median time to detection was 3 seconds, and 80% of the skimmers were detected within one minute. This is a 99% decrease in search time compared to the average of 30 minutes that inspectors take

		Devices Observed				
State	*Stations*	*#*	*Avg.*	*Std.*	*Days*	*Skimmers*
CA	571	1148	2.01	1.94	152	22
AZ	491	1140	2.32	2.03	130	36
NV	38	93	2.45	3.44	21	4
MD	23	42	1.83	1.86	14	2
IL	18	37	2.06	2.01	13	0
NC	10	20	2	1.67	10	0

Table 4: On average there are two Classic Bluetooth devices seen at each gas station; infrequently, there are skimmers.

to check a gas station for skimmers.[6], This result indicates that inspectors can quickly stop at gas stations to check for Bluetooth-detectable internal skimmers.

4.2 Are Skimmers Distinguishable in Scans?

Next, we evaluate if the skimmers detected by Bluetana were clearly distinguishable from the other devices observed at gas stations. The primary result of this study is that these skimmers were not hidden well. Many of these skimmers use the default configuration of their Bluetooth modules. Legitimate devices using the same Bluetooth modules may have some default parameters, and a few have all of parameters set to the default. We conclude that by combining multiple characteristics: MAC prefix, Class-of-Device, and Device Name, there are only a small number of devices that could be confused with skimmers.

This study also reveals that when criminals creatively modify their skimmer's Device Name, it makes detection easier. We also found that criminals could improve how they hide skimmers in Bluetooth scans. For example, they could change the Class-of-Device to hide as a more popular device (e.g., a smartphone).

Dataset Overview

Over the course of the 19 month study, Bluetana users visited 1,185 gas stations across six states (Table 4). During these visits, Bluetana detected a total of 64 skimmers—all of which were recovered by officials. These skimmers were in the presence of 2,562 other devices. On average, Bluetana saw 2.2 devices per station ($\sigma = 2.05$). Given that there are only a small number of Bluetooth devices seen per station, it may seem likely that these devices are all skimmers. However, only a small fraction (4.25%) of these devices matched the characteristics of the skimmers we observed during the course of our study.

We performed this study on Classic Bluetooth devices only. We did not include BLE because we are not aware of any internal gas station skimmers using BLE modules. However, we observed a large number of BLE devices at gas stations; therefore, switching skimmers to BLE modules may make them more difficult to detect with scanning tools like Bluetana (Section 5.1).

[6]Source: discussions with inspectors.

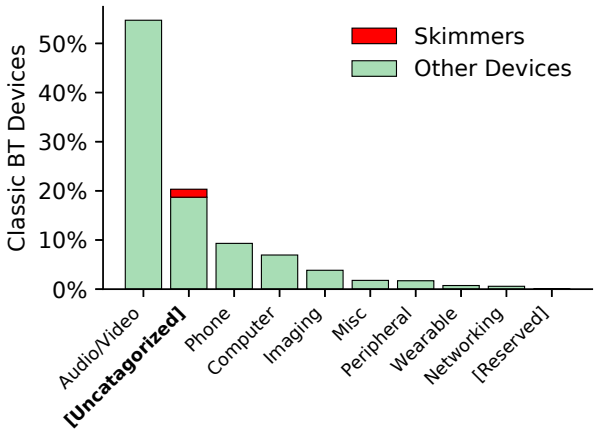


Figure 8: Skimmers appear in the second most common class of Bluetooth devices.

For this analysis, we only include the scan data that is collected the first time a Bluetana user visits a station. Restricting the dataset in this way ensures fairness in our results. Analyzing all inspections may bias our observation of what Bluetooth devices tend to be found at gas stations to those that were visited multiple times. Specifically, we only analyze scans performed the first time Bluetana is near a gas station (within 150 feet) for at least 30 seconds and up to 5 minutes. 22 out of 64 of the skimmers were detected on subsequent visits to gas stations, so they are not included in this analysis.

Skimmers are Uncategorized, but so are other devices

The only Bluetooth property that is common among all skimmers we observed is that they have an Uncategorized Class-of-Device. Figure 8 shows that Uncategorized devices are commonly seen at gas stations: they are 20.3% of devices found by Bluetana. Out of the 1,185 gas stations that Bluetana users visited, Uncategorized devices were only observed at 315 gas stations (26.6%).

Other devices use the same modules as skimmers

Within the set of Uncategorized devices, we next look at the distribution of their MAC prefixes (Figure 9). We find that the Bluetooth modules used in skimmers are also used in many other legitimate devices. Specifically, more than half of the RN modules seen at gas stations were in skimmers, but there were many other devices that had RN modules. This is

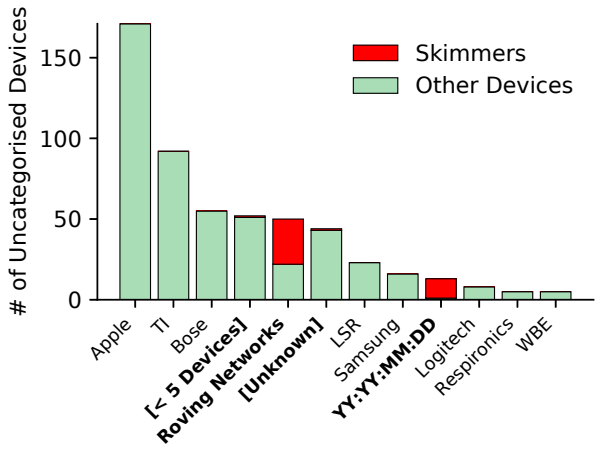


Figure 9: Many other devices appear to be using the same Bluetooth modules as skimmers.

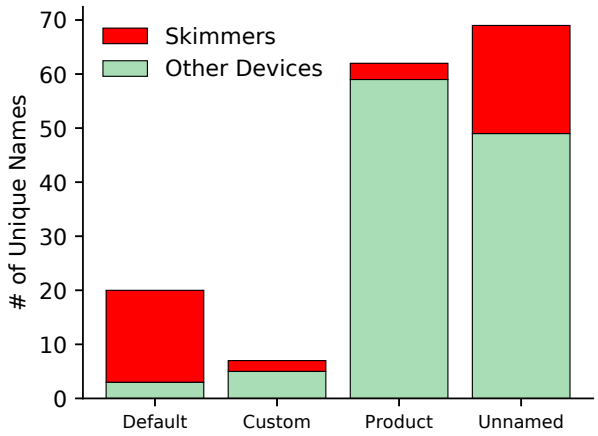


Figure 10: Default and custom names distinguish skimmers from legitimate devices.

an important observation because a popular detection application, SkimPlus [51], only flags skimmers based on a hitlist of MAC prefixes [46]; it may incorrectly flag legitimate devices as skimmers.

The devices observed with MAC prefixes that were in the `YY:YY:MM:DD` format (likely HC modules) were mostly skimmers. There were many devices that had IEEE assigned MAC prefixes that were infrequently seen at gas stations (< 5 Devices). Only one of these devices was a skimmer. Also, there were many devices with MAC prefixes unknown to the IEEE, but not in the date format, only one of these devices was a skimmer. Overall, 159 devices out of 353 Uncategorized devices matched the MAC prefixes of Bluetana-observed skimmers. This reduces the number of stations where Bluetana detected skimmers to 119 out of the 315 stations where it found Uncategorized devices.

Default- and custom-named modules are often skimmers

Finally, we investigate if skimmers can be differentiated from other devices by their Device Name. The remaining 159 devices are Uncategorized and their MAC prefixes are either: Roving Networks, `YY:YY:MM:DD`, Unknown, or seen on less than five devices. Only 42 of these devices were confirmed to be skimmers.[7] In Figure 10, we divide the remaining devices by their category of Device Name, including: *unnamed*, manufacturer *default*, known legitimate *product*, and *customized*. Devices observed by Bluetana with default names were often skimmers. Custom named devices were not common at gas stations but had a higher probability of being skimmers. Three skimmers were disguised as products, however all three were distinguishable because their names were popular smartphones, which should not have the MAC prefix of Bluetooth-to-Serial modules. Bluetana missed capturing the Device Name for many of the skimmers, as well as other devices that it detected.

[7] We do not include 22 of the Bluetana-detected skimmers in this analysis because they were not detected on the first visit to a gas station.

4.3 Accuracy of Bluetooth-based Detection

To evaluate the accuracy of Bluetooth-based detection, we analyze Bluetana scan data collected during inspections in Arizona. Specifically, there was a 7-month time period in which Bluetana was used by many of the Arizona inspectors (October 7, 2018 – May 7, 2019), and we compare the reports filed during these inspections with the scan data that Bluetana collected.

Missed skimmers

During this time period, there were 27 inspections where skimmers were found while an inspector was running Bluetana. A total of 42 skimmers were recovered during these inspections, of which Bluetana was able to detect 36. Therefore, Bluetana missed detecting 14.3% of the total skimmers recovered during these inspections.

We do not know exactly why Bluetooth-based scanning missed these skimmers. Half of the missed skimmers were from inspections where Bluetana detected other skimmers at the gas station. It is likely that these missed skimmers were not powered on due to improper installation. The remaining missing skimmers may have been built with alternate exfiltration methods, such as SMS [46], or even require physical recovery [47].

Incorrectly detected skimmers

Bluetana highlighted a device in red during 45 Arizona inspections where no skimmer was found. There were 757 total inspections where inspectors used Bluetana[8], Bluetana may have incorrectly detect skimmers in 5.9% of inspections.

Incorrectly identifying skimmers is likely due to the fact that RN and HC modules are used in a variety of legitimate products, some of which are seen in and around gas stations. We found RN and HC modules in radar-based speed limit signs, weather sensors [38] automotive diagnostic scanners, scales [37] and fleet tracking systems [52]. Some of these devices have Device Names that clearly indicate what product they are, but would be confused with skimmers if the Device Name is missing. Unfortunately, several of these products also use the default Device Names on their Bluetooth modules (*RNBT-xxxx* or *HC-05*). These legitimate devices will look exactly like skimmers. In such cases, inspectors will need to rely on RSSI localization to determine if these devices are located inside a gas pump.

5 Countermeasures and Responses

This work is a single snapshot in an evolving landscape of attacks on payment systems. While Bluetana has proven effective at finding Bluetooth skimmers, it by no means represents the last move in the cat-and-mouse game. In the remainder of this section, we discuss what the next few steps in this arms race might look like. That is, given that inspectors and volunteers are using Bluetana, what can be the skimmer installers' next move, its cost, and what our response might be. It is possible for a determined and resourceful criminal to implement the countermeasures that we will be describing (particularly non-discoverable mode).

5.1 Switching to Bluetooth Low Energy

We have observed that by switching to BLE, criminals have *many* more places to hide. Figure 11 shows the cumulative distribution of the number of BLE and Bluetooth devices we saw at each fuel station. Under the filtering of Section 4, over 8,000 unique BLE devices were seen, making the ratio of Classic to BLE approximately 1:4.

Cost to attacker. There is almost no cost to criminals in switching their Bluetooth modules to BLE. In fact, newer EMV skimmers discovered in other countries are BLE enabled [30]. However, none of our contacts in law enforcement have encountered BLE-based gas station skimmers. It is possible that there is simply no incentive to switch: the same reason criminals have not yet adapted to masking their Bluetooth device class.

[8]This includes both routine and complaint/prior knowledge triggered inspections

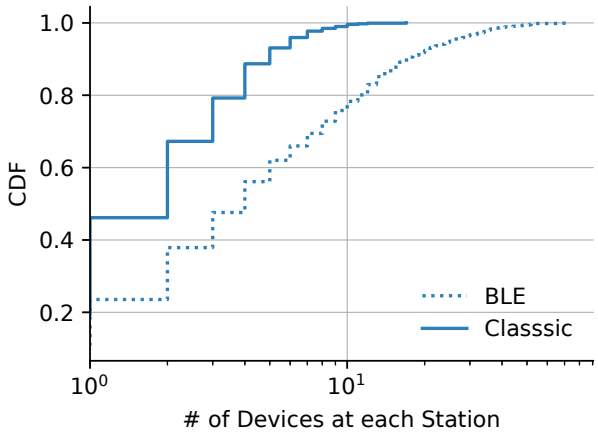


Figure 11: BLE devices are more common than Classic.

Response. BLE devices may be harder to differentiate due to the higher number of devices at each gas station and a lack of distinguishing features. 89% of BLE devices we saw had an uncategorized device class. With more sophisticated filtering techniques, it may still be possible to isolate BLE skimmers within this larger set of devices. One possibility is automated RSSI localization to the fuel dispenser location, a possible subject of future research.

5.2 Non-Discoverable Skimmers

The most natural way to evade discovery via Bluetooth would be to put the module in non-discoverable mode. When a Bluetooth device is non-discoverable, it does not respond to normal Bluetooth scans. Instead, it only responds to paging packets specifically addressed to its MAC address.

Cost to attacker. Non-discoverability would make exfiltration more difficult for criminals. One possibility is creating a pre-paired data collection device. However, we have been informed by law enforcement that the individual who installs the skimmer is often independent from the individual responsible for data recovery (called a "mule"). The criminal would not be able to send a mule to recover card data without first delivering them the device. Alternately the criminal could record the MAC address of the skimmer Bluetooth module. This would require careful bookkeeping and the use of tools that support the creation of a non-discoverable connection.

Response. It is still possible to discover a non-discoverable device. For a small set of target address ranges, e.g., `00:06:66` used by Roving Networks modules, we believe it would be practical to attempt to guess all 16.8 million possible addresses. Prior work has shown that it is possible to discover any non-discoverable device via brute force in 18.64 hours; knowledge of OUI would ideally allow us to reduce this search time [17]. Unfortunately, this requires specialized hardware, rather than an inexpensive Android phone.

5.3 Impersonating Common Benign Devices

Another natural response to Bluetana would be to change the MAC address and name of the device to that of a common benign device, such as a mobile phone or a Bluetooth-enabled car entertainment system. This would make the skimmer appear innocuous to Bluetana.

Cost to attacker. Reprogramming the MAC address on the CSR-based Bluetooth modules, which include the Roving Networks and HC-05 and HC-06 modules, cannot be done using the AT commands used to change device name and pairing. Instead, the skimmer installer would need to reflash the CSR firmware using a special programming cable. While, in principle, not difficult, it would require an additional degree of sophistication than programming a simple micro-controller development board. The skimmer installer could also change the device name but not the MAC address, say, to one of the known benign devices using the same module, something that us possible to do by issuing AT commands from the micro-controller to the module. While this may cause Bluetana to detect these as a skimmer, signal strength can still be used to identify location of the module.

Response. Because Bluetana collects all Bluetooth data, we can identify skimmers retroactively when we learn of a new MAC address and name used by known skimmers. Thus, if attacks switch to impersonating benign devices, we can update the Bluetana highlighting mechanism to identify those devices as suspicious. This would result in additional inspections, but would still provide significant gain over the state of the art.

5.4 Using Non-Bluetooth Communications

During discussions with law enforcement agencies tasked with identifying skimmers, we were told about skimmers that use GSM modems or WiFi as an alternative to Bluetooth. In the case of WiFi, we believe that the Bluetana methodology will still be effective. GSM poses a more serious challenge for detection.

Cost to attacker. While using GSM would avoid detection using Bluetana, it creates an additional trail of evidence linking the perpetrator to the skimmer. Law enforcement officers could obtain information about the SMS recipient through subpoenas, so receiving the SMS messages on another phone on a US carrier, for example, would be easy to trace.The perpetrator would need to use an SMS service that would not expose his/her identity.

Response. In addition to legal tools available to law enforcement to trace SMS messages, a GSM modem could be detected using a Software-Defined Radio.

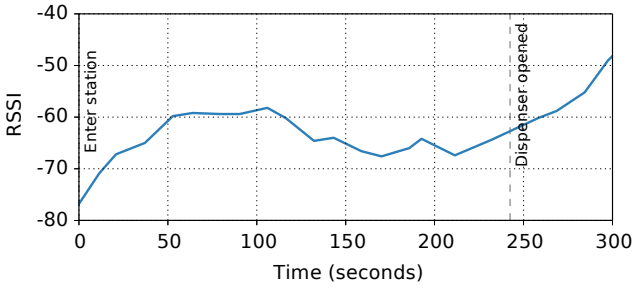


Figure 12: Opening of the gas pump enclosure results in a significant jump in observed Bluetooth signal strength from a skimmer.

5.5 Attacker Bottlenecks

The attacker (skimmer installer) has several practical ways to evade detection using Bluetana. Each of these, however , has an additional cost to the attacker in terms of effort, risk exposure, or expertise. We do not yet have a strong understanding to which of these costs attackers are most sensitive. Indeed, the very low price of stolen credit card numbers, compared to their potential cash out value (Table 1) suggests that the bottleneck in the carding value chain is *not* getting card information but cashing out cards. Thus, while Bluetana may raise the cost for attackers, we do not believe that it will raise it so much as to make fuel dispenser skimming unprofitable.

6 Operational Lessons Learned

While performing the Bluetana study, we learned several lessons about the operational use of Bluetooth scanning for skimmer detection. In this section, we provide an overview of two most important lessons we learned.

6.1 Bluetooth Helps During Inspections

Criminals hide skimmers in the crevices of gas pumps to avoid detection during inspections. We witnessed several instances where investigators were unable to locate skimmers via physical inspection alone. In one incident, Bluetana flagged four devices at a station; however, no skimmers were located. This result led officials more experienced in skimmer recovery to perform a second thorough inspection of the station. These officials located all four skimmers. The evidence provided by Bluetana forced them to continue the inspection, instead of abandoning it and leaving the devices in the field.

Figure 12 demonstrates an instance of how the signal strength measurements helped inspectors determine which pump had a skimmer. When the gas pump's metal door was opened, the signal strength increased significantly, prompting inspectors to look carefully for the skimmer in that pump.

	Group				
	1	**2**	**3**	**4**	**5**
Skimmers	3	5	6	4	3
Gas stations	2	2	5	4	2
Min. difference in MACs	1	4	9	10	4
Closest MAC distance (in miles)	0	17	59	203	448

Table 5: Several geographically separated skimmers had similar MAC addresses.

6.2 MAC Addresses May Indicate the Source

Network equipment vendors (e.g., Bluetooth module manufacturers) tend to allocate MAC addresses sequentially by production time [34]. Therefore, if two devices have similar MAC addresses, they are likely part of the same batch of devices sold. This information can be used to associate skimmer Bluetooth modules to the same board designer or crew.

We group the skimmers found by Bluetana with the same first 5 bytes of MAC address. Table 5 shows five such groups. We list the difference in MAC address and the geographic distance between the closest MACs in each group. Skimmers in group 1 and 2 were recovered at gas stations in the same county, separated by at most 17 miles. From LE sources, we know that criminals often plant skimmers across multiple stations in a given city/county, and the MAC address data collected indicates this. Groups 3-5 are the most interesting, as the closest MACs in the same group are in stations across different counties. The closest MACs in group 5 are at stations separated by 448 miles. This may seem surprising, but LE informs us that skimmer crews avoid detection by migrating from city to city.

7 Related Work

Skimmer Detection and Prevention. In recent work, Scaife et al. surveyed gas pump skimmer detection and prevention mechanisms [46]. They found that several popular Bluetooth-based skimmer detection applications use a only MAC prefix or device name matching. The results of our study show how the Bluetooth profile of skimmers in these applications can be improved to detect more skimmers, and to flag fewer legitimate devices as skimmers. We also find that Bluetooth-based scanning is an effective way to augment manual gas pump inspections. Scaife et al. also introduced SkimReaper [47], an effective tool for detecting external skimmers. SkimReaper is a credit-card shaped device that an official can swipe in a card reader to detect if the reader has an additional read head: indicating that the reader has an external skimmer attached to it. However, SkimReaper can not detect internal skimmers because they do not add an additional read head. Additionally, the PCI Security Standards Council have released guidelines for preventing external skimming [41]. Criminals may start using Bluetooth to retrieve card data from external skimmer. If they do, we demonstrate that Bluetooth scanning can augment these existing external skimmer detection and prevention methods.

Bluetooth. Prior work has evaluated the effectiveness of Bluetooth scanning for detecting and localizing Bluetooth devices. They found that Bluetooth signal strength (measured by an Android smartphone) is effective for localizing Bluetooth devices [32, 57]. This work inspired us to use signal strength to detect if a Bluetooth device appears to be installed inside of a gas pump. Previous studies also examined how long it takes to detect a Bluetooth device from stationary observers and moving vehicles. They found that Bluetooth devices are often detected in less time than the Bluetooth standard suggests [39, 43, 24]. This work supports our findings that skimmers are often discovered within the first few seconds of passing by a gas station.

Inventory Attacks. Prior work has demonstrated that user privacy can be violated by inspecting the characteristics of a user's device [58]. These so called *inventory attacks* have been demonstrated for Bluetooth Low-Energy, RFID, and even web browsers [54, 55, 23]. Our work demonstrates a Bluetooth-based inventory attack against malicious devices, can be used to protect the privacy of consumers.

8 Future Work and Conclusion

As new skimmer detection tools gain popularity, criminals will adapt skimming designs to evade detection. We expect future skimmers will use techniques such those described in Section 5. Similar to Bluetana, future work in this area should emphasize designing easy-to-deploy systems for detecting skimmers, and evaluating their effectiveness with large-scale studies.

Push-back from banks and card issuers has led to widescale adoption of EMV in retail PoS systems. However, EMV adoption in gas stations across the U.S. has been slow due to high costs. Therefore, Visa and Mastercard have pushed the EMV adoption deadline for gas stations from 2017 to October 2020 [22]. As gas stations begin migrating to EMV, skimmers targeting EMV will become more common. Future research should focus on the detection of EMV "shimmers" that are gaining in popularity.

Finally, we believe gas pump skimming is the harbinger of an era of attacks using wireless implants. For example, there is an internal Bluetooth-based implant for unlocking door access control systems [14]. Future work should also identify other systems that are vulnerable to using such implants.

In this paper, we presented results of a 19-month long measurement study of Bluetooth scanning as a mechanism to detect internal gas pump skimmers. Our evaluation showed

that Bluetooth characteristics of some internal skimmers can be distinguished from other Bluetooth devices commonly seen at gas stations. We detected, and LE recovered, 64 skimmers at 34 gas stations across four states in the U.S. For 33 of the detected skimmers, Bluetana was the only source of information that prompted investigators to conduct an inspection. In conclusion, crowdsourced Bluetooth scanning is an effective way to detect Bluetooth-based internal gas pump skimmers.

9 Acknowledgements

We would like to express our appreciation for the local and federal law enforcement agents who introduced us to gas pump skimming and guided us throughout this project. We also thank the Kevin Allen, and the field investigators at the Arizona Department of Agriculture, Weights and Measures Services Division, for their invaluable help in understanding and analyzing the skimming problem in Arizona. We also thank the Sacramento County Sheriff's Detectives Sean Smith and Matt Deaux, both are assigned to the Sacramento Valley Hi-Tech Crimes Task Force, for their help in understanding gas pump skimming in depth. We are also very grateful to the various individuals who drove to gas stations in several states and collected Bluetooth scan data. We also thank our shepherd Joseph Calandrino, and the anonymous reviewers for their insightful feedback and suggestions.

References

[1] Arizona Department of Agriculture, Weights and Measures Service Division . Data Skimmers in Motor Fuel Dispensers. https://agriculture.az.gov/sites/default/files/Skimmer%20Presentation%20%28Website%20Edition%29.pdf, Sept. 2017.

[2] Nate Seidle . Gas Pump Skimmers . https://learn.sparkfun.com/tutorials/gas-pump-skimmers, Sept. 2017.

[3] Nick Poole . Credit Card Skimmers Evolved: Shimming . https://www.sparkfun.com/sparkx/blog/2673, Apr. 2018.

[4] Office of Minnesota Attorney General Keith Ellison . ATM and Gas Pump Skimmers . https://www.ag.state.mn.us/Brochures/pubATMSkimmers.pdf.

[5] Rippleshot . State of Card Fraud: 2018. https://www.aba.com/Products/Endorsed/Documents/Rippleshot-State-of-Card-Fraud.pdf, 2018.

[6] United States Sentencing Commission . Guidelines Manual . https://guidelines.ussc.gov/gl/%C2%A72B1.1, 2018.

[7] Affidavit in Support of Criminal Complaints and Arrest Warrants, USA v. Khasanov et al, 1:18cr149. US District Court for the Eastern District of Virginia. https://www.courtlistener.com/recap/gov.uscourts.vaed.385830/gov.uscourts.vaed.385830.2.0.pdf, Jan. 2018.

[8] Appeal from the US District Court for the Eastern District of Oklahoma, USA v. Konstantinov et al, 6:13cr62. United States Court of Appeals for the Tenth Circuit. https://www.ca10.uscourts.gov/opinions/14/14-7050.pdf, June 2015.

[9] Application for Search Warrant, 2:18mj1277. US District Court for the Eastern District of Wisconsin. https://www.courtlistener.com/recap/gov.uscourts.wied.84529/gov.uscourts.wied.84529.1.0.pdf, July 2018.

[10] Arizona Department of Agriculture. Credit Card Skimmers. https://agriculture.az.gov/weights-measures/fueling/credit-card-skimmers, Feb. 2019.

[11] K. Arnold. Florida gas pump thefts rise as credit-card skimmers get more savvy. https://www.orlandosentinel.com/business/consumer/os-bz-credit-card-skimmers-20181108-story.html, Nov. 2018.

[12] ATM Industry Association. Global Fraud and Security Survey - 2017. https://www.ncr.com/company/blogs/financial/how-much-does-atm-crime-cost, Jan. 2018.

[13] H. Bar-El. White Paper: Known Attacks Against Smartcards. Technical report, Discretix Technologies Ltd., 2005.

[14] M. Bassegio and E. Evenchick. Breaking access controls with BLEKey. https://www.blackhat.com/docs/us-15/materials/us-15-Evenchick-Breaking-Access-Controls-With-BLEKey-wp.pdf, Aug. 2015.

[15] M. Bond, O. Choudary, S. J. Murdoch, S. Skorobogatov, and R. Anderson. Chip and Skim: Cloning EMV Cards with the Pre-play Attack. In *Proc. IEEE Symposium on Security and Privacy*. IEEE, 2014.

[16] Criminal Complaint, USA v Cristea et al, 4:16cr182. US District Court for the Southern District of Texas. https://www.courtlistener.com/recap/gov.uscourts.txsd.1357299.1.0.pdf, Apr. 2016.

[17] D. Cross, J. Hoeckle, M. Lavine, J. Rubin, and K. Snow. Detecting non-discoverable bluetooth devices. In *International Conference on Critical Infrastructure Protection*, pages 281–293. Springer, 2007.

[18] The Ultimate Instore Carding by n3d from Darknet. http://wickybay.com/2017/10/ultimate-instore-carding-n3d-darknet/.

[19] DbaseJob. Carding!!! How To Make Your First Money. https://prvtzone.ws/threads/carding-how-to-make-your-first-money.5052/#post-20315.

[20] Tutorial Carding with Dumps. https://honeymoney24cc.com/cardingwithdumps.

[21] CC Dumps Shop. https://dumps.to/, Feb. 2019.

[22] Electronic Transactions Association. ETA Statement on Visa and Mastercard's EMV Liability Shift Date Changes. https://www.electran.org/eta-statement-on-visa-and-mastercards-emv-liability-shift-date-changes/, 2016.

[23] K. Fawaz, K.-H. Kim, and K. G. Shin. Protecting Privacy of BLE Device Users. In *USENIX Security Symposium*, pages 1205–1221, 2016.

[24] J. Haartsen. Bluetooth—The universal radio interface for ad hoc, wireless connectivity. *Ericsson Review*, 3(1):110–117, 1998.

[25] Indictment, USA v. Rodriguez et al, 1:17cr417. US District Court for the Northern District of Ohio. https://www.courtlistener.com/recap/gov.uscourts.ohnd.237118.1.0.pdf, Oct. 2017.

[26] Krebs on Security. Skimmers Siphoning Card Data at the Pump. https://krebsonsecurity.com/2010/07/skimmers-siphoning-card-data-at-the-pump/, July 2010.

[27] Krebs on Security. Pro-Grade Point-of-Sale Skimmer. https://krebsonsecurity.com/2013/02/pro-grade-point-of-sale-skimmer/, Feb. 2013.

[28] Krebs on Security. Gang Rigged Pumps With Bluetooth Skimmers. https://krebsonsecurity.com/2014/01/gang-rigged-pumps-with-bluetooth-skimmers/, Jan. 2014.

[29] Krebs on Security. Tracking a Bluetooth Skimmer Gang in Mexico. https://krebsonsecurity.com/2015/09/tracking-a-bluetooth-skimmer-gang-in-mexico/, Sept. 2015.

[30] Krebs on Security. ATM 'Shimmers' Target Chip-Based Cards. https://krebsonsecurity.com/2017/01/atm-shimmers-target-chip-based-cards/, Jan. 2017.

[31] Legitshop. Trusted Dumps with PIN. https://legitshop.org/, Feb. 2019.

[32] S. Liu, Y. Jiang, and A. Striegel. Face-to-face proximity estimation using bluetooth on smartphones. *IEEE Transactions on Mobile Computing (TMC)*, 13(4):811–823, 2014.

[33] D. MacGuill. Can a Mobile Phone's Bluetooth Sensor Be Used to Detect Card Skimmers? https://www.snopes.com/fact-check/bluetooth-gas-pump-skimmers/, 2019.

[34] J. Martin, E. Rye, and R. Beverly. Decomposition of MAC address structure for granular device inference. In *Proc. Annual Computer Security Applications Conference (ACSAC)*, 2016.

[35] Meccadumps. Buy Dumps CVV online Fullz Verified seller . https://meccadumps.net/, Feb. 2019.

[36] Memorandum and Order, USA v. Hristov et al, 1:10cr10056. US District Court for the District of Massachussetts. https://www.courtlistener.com/recap/gov.uscourts.mad.127405/gov.uscourts.mad.127405.62.0.pdf, Apr. 2011.

[37] Mettler-Toledo. BC Shipping Scale Service Manual. https://thescalestore.com/manuals/Mettler-Toledo-BC-User-Manual-1.pdf, Aug. 2015.

[38] MH Corbin Highway Information Systems. Surface Scan. http://mhcorbin.com/Portals/0/MH%20Corbin%20Surface%20Scan%20User%20Manual%20v1.1%20(002)%20new%20cover.pdf, Jan. 2018.

[39] P. Murphy, E. Welsh, and P. Frantz. Using bluetooth for short-term ad-hoc connections between moving vehicles: A feasibility study. In *IEEE Vehicular Technology Conference (VTC)*, volume 1, 2002.

[40] Everything you need to know about instore carding. http://wickybay.com/2017/11/everything-need-know-instore-carding/, Nov. 2017.

[41] PCI Security Standards Council. Skimming Prevention: Overview of Best Practices for Merchants. https://www.pcisecuritystandards.org/documents/Skimming_Prevention_At-a-Glance_Sept2014.pdf, Sept. 2014.

[42] PCI Security Standards Council. PCI DSS Quick Reference Guide. https://www.pcisecuritystandards.org/documents/PCI_DSS-QRG-v3_2_1.pdf, 2018.

[43] B. S. Peterson, R. O. Baldwin, and J. P. Kharoufeh. Bluetooth inquiry time characterization and selection. *IEEE Transactions on Mobile Computing (TMC)*, 5, 2006.

[44] PRTSHIP. DUMPS. https://prtship.com/forums/dumps.6/.

[45] Santander Bank. What is my debit card spending/withdrawal limit? https://customerservice.santanderbank.com/app/answers/detail/a_id/3713/kw/atm%20withdraw/r_id/102441.

[46] N. Scaife, J. Bowers, C. Peeters, G. Hernandez, I. N. Sherman, P. Traynor, and L. Anthony. Kiss from a rogue: Evaluating detectability of pay-at-the-pump card skimmers. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1208–1222, Los Alamitos, CA, USA, may 2019. IEEE Computer Society.

[47] N. Scaife, C. Peeters, and P. Traynor. Fear the Reaper: Characterization and Fast Detection of Card Skimmers. In *Proc. USENIX Security*, 2018.

[48] Scientific Working Group on Digital Evidence. Best Practices for Examining Magnetic Card Readers. https://www.swgde.org/documents/Current%20Documents/SWGDE%20Best%20Practices%20for%20Examining%20Magnetic%20Card%20Readers.

[49] Sell CVV (CC). https://sellcvvdumps.shop/.

[50] Sentencing Memorandum of the United States, USA v. Aqel, 2:14cr270. US District Court for the Southern District of Ohio. https://www.courtlistener.com/recap/gov.uscourts.ohsd.178108/gov.uscourts.ohsd.178108.47.0.pdf, Nov. 2015.

[51] Skim Plus (Bluetooth Skimmer Detection). https://play.google.com/store/apps/details?id=com.rs.skimplus.beta, 2018.

[52] Teletrac. Teletrac Drive User Guide. http://community.teletrac.com/teletrac.com/assets/2014-04-23_android%20tablet%20user%20guide.pdf, Jan. 2014.

[53] The Newnan Times-Herald. Armenian skimmer leader pleads guilty. http://times-herald.com/news/2015/06/armenian-skimmer-leader-pleads-guilty, July 2017.

[54] T. van Deursen. 50 ways to break RFID privacy. In *IFIP PrimeLife International Summer School on Privacy and Identity Management for Life*, pages 192–205. Springer, 2010.

[55] A. Vastel, P. Laperdrix, W. Rudametkin, and R. Rouvoy. Fp-Scanner: The Privacy Implications of Browser Fingerprint Inconsistencies. In *Proc. USENIX Security*. USENIX Association, 2018.

[56] VICE. Gangs on the Dark Web: Credit Card Scammers. https://www.youtube.com/watch?v=jT-jmq8KBw0, June 2018.

[57] Y. Wang, X. Yang, Y. Zhao, Y. Liu, and L. Cuthbert. Bluetooth positioning using RSSI and triangulation methods. In *Consumer Communications and Networking Conference (CCNC), 2013 IEEE*, pages 837–842. IEEE, 2013.

[58] J. H. Ziegeldorf, O. G. Morchon, and K. Wehrle. Privacy in the internet of things: threats and challenges. *Security and Communication Networks*, 7(12):2728–2742, 2014.

A Court Cases

The appendix contains excerpts from various public court documents related to cases of credit card skimming. These excerpts provide anecdotal data about the monetary impact of the skimmer problem.

A.1 Cashout Value

USA v. Hristov et al [36]

"... Bank of America suffered a loss of $33,000 with 36 compromised customer accounts. Citizens Bank suffered a loss of $91,580 with 74 compromised customer accounts ..."

USA v. Cristea et al [16]

"... Altogether, on February 21,2016, FBI surveillance observed Cristea, Co-conspirator #1, and Co-conspirator #2 go to approximately 12 different locations, where, according to CardTronic's records, they withdrew at least $7,000 from at least 18 First National Bank accounts ..."

USA v. Khasanov et al [7]

"... USPS agents thereafter conducted record checks on the purchased USPS money orders and discovered that 10 of the 57 money orders had been purchased with 5 payment numbers issued by Citibank ..."

Date	Location of USPS	Amount
Aug 4 2017	Waldorf, MD	$2,904.80
Aug 7 2017	Washington, DC	$1,492.80
Aug 7 2017	McLean, VA	$1,400.00
Aug 7 2017	Washington, DC	$1,803.20
Aug 7 2017	Hyattsville, MD	$792.05

USA v. Aqel [50]

"... the Probation Officer also notes that the actual loss to victims was $8,327.58. Id. Similarly, the Probation Officer notes that while Mr. Aqel possessed 120 stolen credit card numbers, only 23 of those numbers were used to make purchases ..."

USA v. Rodriguez et al [25]

"... Between on or about July 7, 2016, and on or about July 20, 2016, Defendant ... attempted to conduct approximately 133 retail transactions totaling in excess of $27,000 ... using approximately 90 counterfeit access devices re-encoded with credit/debit account information that were obtained by a skimming device placed on the point of sale terminal of a gas pump ..."

Application for Search Warrant, 2:18mj1277[9]

"... On April 14, 2016, a man (later identified as Estrada) used a fraudulent Visa credit card and a fraudulent MasterCard to purchase two $300.00 gift cards from the Kohl's store ..."

USA v. Konstantinov et al [8]

"... In total, the defendants compromised approximately 524 debit card accounts and made approximately 779 fraudulent withdrawals, totaling $348,376.80 ..."

A.2 Credit/Debit cards per skimmer per day

Application for Search Warrant, 2:18mj1277 [9]

"... On September 9, 2016, an employee at Jilly's Mobil ... reported to Detective Craig Meyer that he had found what appeared to be a skimmer on pump #8 ... Detective Meyer downloaded and exported the data stored on the skimmer taken from Jilly's Mobil pump #8. The results showed data for 221 victim credit card accounts ...
... Detective Meyer reviewed the video surveillance footage for Jilly's Mobil from September 1, 2016. At 1:38 PM on September 1st, a red Ford Explorer drove to pump 8. The Ford Explorer was positioned in a manner whereby the opened passenger door blocked the view of the gas pump by the store employee inside the Jilly's Mobil ..."

CANvas: Fast and Inexpensive Automotive Network Mapping

Sekar Kulandaivel
Carnegie Mellon University
skulanda@andrew.cmu.edu

Tushar Goyal
Carnegie Mellon University
tgoyal1@alumni.cmu.edu

Arnav Kumar Agrawal
Carnegie Mellon University
akagrawa@alumni.cmu.edu

Vyas Sekar
Carnegie Mellon University
vsekar@andrew.cmu.edu

Abstract

Modern vehicles contain tens of Electronic Control Units (ECUs), several of which communicate over the Controller Area Network (CAN) protocol. As such, in-vehicle networks have become a prime target for automotive network attacks. To understand the security of these networks, we argue that we need tools analogous to network mappers for traditional networks that provide an in-depth understanding of a network's structure. To this end, our goal is to develop an automotive network mapping tool that assists in identifying a vehicle's ECUs and their communication with each other. A significant challenge in designing this tool is the broadcast nature of the CAN protocol, as network messages contain no information about their sender or recipients. To address this challenge, we design and implement *CANvas*, an automotive network mapper that identifies transmitting ECUs with a pairwise clock offset tracking algorithm and identifies receiving ECUs with a forced ECU isolation technique. *CANvas* generates network maps in under an hour that identify a previously unknown ECU in a 2009 Toyota Prius and identify lenient message filters in a 2017 Ford Focus.

1 Introduction

Recent efforts have demonstrated numerous vulnerabilities in automotive networks, particularly those that employ the CAN communication protocol. Although CAN is the prevailing standard for intra-vehicular communication due to its low cost and robustness, its broadcast nature has enabled many exploits initially exposed by the early work of Koscher et al. [20]. These exploits target the intra-vehicular CAN bus via either direct physical access [9, 20] or the remote exploitation of an ECU with existing direct access [26]. For the purpose of planning their well-known exploit [26], Miller et al. [25] analyzed the intra-vehicular networks of several vehicles, which revealed that the 2014 Jeep Cherokee was the "most hackable" based on its layout of ECUs. Once the authors gained access to the CAN via an exploited ECU, they simply had to discover which ECUs and real physical functions react to injected messages.

From these anecdotes, we can see that the set of ECUs and their inter-ECU communication channels determine the vulnerability of a vehicle's ECU network. Consequently, we argue that the automotive security world needs tools similar to Nmap [21], which are used to map the structure of modern IP networks. Such mapping tools prove useful in both attack and defense scenarios, such as identifying potentially malicious servers, attesting server configurations, and auditing firewalls by identifying available network connections. Analogously, with such a tool for scanning a car's network, we could (1) discover potentially malicious ECUs inserted through an attacker, (2) attest to the network configuration of ECUs over time, and (3) identify potential ECUs that are vulnerable to a recent type of attack (§2).

To aid in these scenarios, an ideal network mapper would require three main outputs: (1) the transmitting ECU for each unique CAN message, (2) the set of receiving ECUs for each unique CAN message and (3) a list of all active ECUs in the vehicle. To ensure that our network mapper is practical for our envisioned use cases, we ideally want our tool to be (a) *fast* to permit analysis of multiple vehicles at a time and limit the time a vehicle must be running and (b) *inexpensive* to avoid requiring costly equipment such as an oscilloscope or logic analyzer.

Unfortunately, extracting the necessary information to map these communication channels requires an unreasonable amount of effort. In the work by Koscher et al. [20], the authors analyzed the security of a vehicle's components by manually extracting ECUs to isolate and interact with them. This type of analysis requires significant time and effort or access to limited or proprietary information [25]. Second, obtaining vehicles for extended time and with permission to disassemble is costly and expensive. Considering new model years and over-the-air update capabilities, the frequency of analyzing an intra-vehicular network will quickly increase in time and cost requirements.

A key challenge we face in realizing this vision in practice is the lack of source information in CAN messages. CAN messages are "contents-addressed," i.e. messages are labeled

based on their data and provide no indication to the message's sender. Another significant challenge in mapping a CAN bus is the broadcast nature of the CAN protocol; we cannot tell which ECUs have received a message. A CAN message is not explicitly addressed to its recipients, but a node can indicate it has correctly received a message (§3).

In this paper, we present *CANvas*, a system that demonstrates a *fast* and *inexpensive* automotive network mapper without resorting to vehicle disassembly (§4). Rather than require physically isolating each ECU, our key insight is to extract message information by re-purposing two observations from prior work:

- **Identifying message source** (§5): Prior work by Cho et al. [11] state that clock skew is a unique characteristic to a given ECU and thus build an intrusion detection system (IDS) that measures this skew from the timestamps of periodic CAN messages. Using this insight, we envision a mapper that computes clock skew per unique message and uses skew to group messages from the same sender. Unfortunately, due to shortcomings of their approach in our mapping context, we instead track the clock offset of two messages over time to determine their source.

- **Identifying message destination(s)** (§6): In another prior work [10], the authors propose a denial-of-service (DoS) attack that exploits CAN's error-handling protocol to disable a target ECU. Using this insight, the mapper could disable all but one ECU via this DoS attack and observe what messages are correctly received by the isolated ECU. However, due to shortcomings in their method w.r.t. our context, we develop a method to forcefully isolate each ECU and detect which messages the ECU receives despite the broadcast nature of CAN.

We implement the *CANvas* mapper on the open-source Arduino Due microcontroller with a clock speed of 84 MHz and an on-board CAN controller. We evaluate our mapper on five real vehicles (2009 Toyota Prius, 2017 Ford Focus, 2008 Ford Escape, 2010 Toyota Prius, and 2013 Ford Fiesta) and on extracted ECUs from three Ford vehicles. We show that *CANvas* accurately identifies ECUs in the network and the source and destinations of each unique CAN message in under an hour (§7).

Contributions and roadmap: In summary, this paper makes the following contributions:

- Designing an accurate message source identification algorithm that tracks a message's relative clock offset (§5);
- Engineering a reliable message destination identification method by isolating ECUs with a forced shutdown technique (§6);
- A real implementation that maps five real vehicles and extracted ECUs (§7) along with two real examples of motivating use cases for mapping (§2).

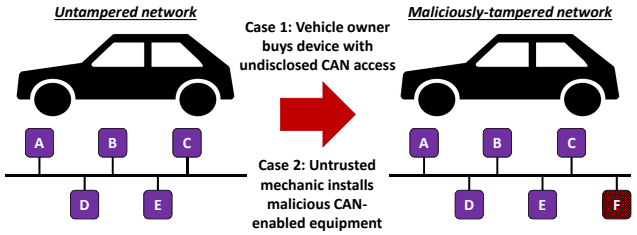


Figure 1: A network mapper could discover potentially malicious ECUs from an untrusted party.

After defining the automotive network mapping problem and describing typical CAN bus setups (§3), we highlight the challenges of identifying message information via the CAN protocol and provide an overview of our approach (§4). Finally, we discuss open issues and limitations (§8) and related work (§9) before concluding the paper (§10).

2 Motivation

In this section, we discuss motivating scenarios for mapping in the context of intra-vehicular networks and describe characteristics of an ideal version of this security tool. To guide our design, we draw an analogy to Nmap [21], a popular network scanning tool that discovers hosts, services, and their interconnections in traditional computer networks. We identify a number of automotive-specific scenarios to illustrate the potential benefits of mapping, although this is not meant to be a comprehensive list.

Malicious ECU discovery: One main feature of Nmap is its ability to discover hosts, i.e. enumerate devices on the network. In the context of automotive networks, these "devices" are equivalent to a vehicle's ECUs. One major automotive cybersecurity concern (depicted in Figure 1) is the potential for an attacker to gain access to a physical network and add a new device [26], which could be a malicious ECU installed by an untrusted party or even by a vehicle owner who installs a CAN-enabled device purchased from an untrusted source. For an attacker that aims to insert this ECU into the network under the guise of a new equipment installation, the ECU could connect to the existing CAN bus and gain unfettered access to the CAN. If a defender performs a mapping through the vehicle's lifetime, they could verify changes to the network's ECUs. We provide an example of this scenario in §7 where we discover a previously unknown ECU that was installed in a modified 2009 Toyota Prius.

Continuous network attestation: Another popular use for Nmap is performing security audits to identify changes to a network [21]. Where such audits would identify new servers or a modification in a server's open ports, an audit in an automotive context could identify changes to the ECUs and their communication channels. With future over-the-air update capabilities, automakers will install new firmware or activate

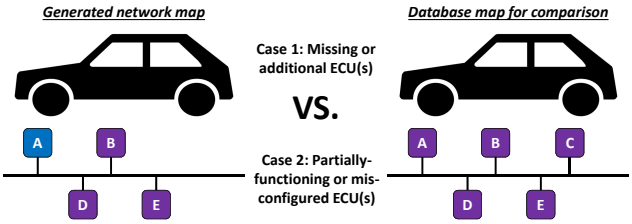


Figure 2: A network mapper could compare a generated map to one found on an online database. Differences between these maps could identify changes in ECUs (Case 1) and/or their message configurations (Case 2).

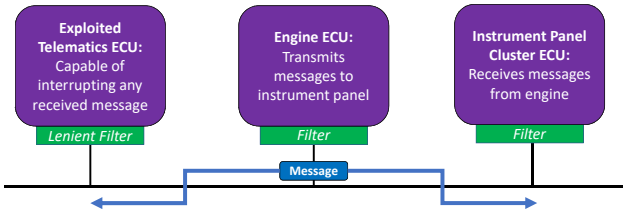


Figure 3: Assume that only the instrument cluster should receive messages from the engine. If the exploited telematics ECU is able to receive engine messages, then an attacker [10] could shutdown the engine ECU via the exploited telematics ECU.

different features in an existing vehicle. As the configuration of the network can change over time, it is necessary for vehicle owners to attest to the vehicle's expected configuration. If a user does not own the vehicle over its lifetime as in the *malicious ECU discovery* scenario, we could implement an online database where vehicle owners could upload the outputs of their network maps for comparison against maps generated from brand-new vehicles. Any differences from the expected maps could indicate malicious or accidental network changes.

Lenient filter identification: Nmap is often used to perform port scanning to identify open ports [21], which are potential vulnerabilities. These "open ports" are analogous to the set of CAN messages that an ECU is able to correctly receive, which we refer to as the ECU's message-receive filter. Now consider an attacker who aims to target a safety-critical ECU (e.g. engine ECU) as depicted in Figure 3. If gaining direct access to the engine ECU proves infeasible, the attacker could access an ECU that is less critical and potentially has access to remote networks (e.g. telematics ECU). Using the ECU shutdown attack as discussed in recent work [10], our attacker can shutdown the engine ECU by gaining control of the telematics ECU and reprogramming it; the attacker simply needs to receive a message from the victim ECU to target it. To combat this, a defender could perform a similar analysis via network mapping and implement filters that prevent the message from being received to limit the damage from a potential shutdown attack. We provide an example of this scenario in §7 where we discover lenient message-receive filters in a 2017 Ford Focus.

Goals: In designing a useful automotive network mapper, we must consider a few requirements that we impose to ensure practicality in the context of our motivating scenarios:

Fast: First, we want to limit the amount of time a vehicle (and its ECUs) are turned on. Also, a fast mapping process will make it more practical for a user to verify the state of their vehicle's network after a repair. Considering these reasons, we aim to achieve a mapping time of under one hour.

Inexpensive: To permit greater access to the mapper, the mapper should consist of relatively inexpensive components and should avoid expensive tools, such as oscilloscopes and logic analyzers. We aim to limit costs to under $100; a low-cost approach to network mapping will permit more users for our system.

Vehicle-agnostic: Every vehicle has a different setup of ECUs on the CAN bus and can employ additional features of the CAN protocol. For our mapper to be practical, it must work on many makes and models of vehicles as well as rely on only standard CAN features.

Minimally-intrusive and non-destructive: One extreme approach for mapping a vehicle requires physical disassembly, which is a very intrusive process and requires a great deal of access to the target vehicle. We should limit this access to simply connecting to a diagnostics port on the vehicle. If a CAN bus is not exposed on this port, we describe a method of getting access to these buses with minimal disassembly in §8. Additionally, the mapper must not cause any permanent damage to the vehicle or its network. Any of our methods can put the network into a non-ideal state (warning lights on, gear shift disabled, etc.), but as long as restarting the vehicle undoes any imposed errors, we satisfy this constraint.

3 Problem Overview

In this section, we give a concrete problem formulation for the network mapper and discuss technical challenges. We preface with some necessary background on CAN to understand the overall problem and mapping challenges.

3.1 CAN basics

To better understand the message information we hope to gain using a network mapper and the associated challenges in acquiring that information, we first discuss some necessary background on how the CAN protocol works.

CAN in modern vehicles: All vehicles produced for the U.S. market in 2008 and after are required to implement the CAN protocol for diagnostics purposes [4]. Many vehicles will often employ either one, two or three CAN buses. In the event of three CAN buses, it is likely that the vehicle has one bus for powertrain components (engine, transmission, etc.),

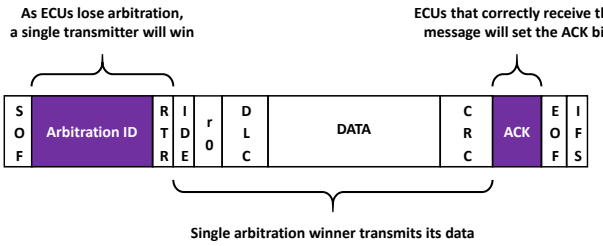


Figure 4: Each CAN frame is transmitted on the bus bit-by-bit. A single transmitter wins arbitration and will listen to receiving ECUs during the ACK slot.

one bus for infotainment components (radio, etc.) and another for body components (door controller, headlights, etc.). These CAN buses are usually exposed through a vehicle's On-Board Diagnostics (OBD-II) port as detailed in §8.

Message broadcast bus: The CAN protocol [13–15, 32, 33] is defined as a message broadcast bus, which means that ECUs in the network communicate with each other via *messages*. These ECUs are connected to a shared network where all ECUs can receive all transmissions. Due to the nature of this broadcast bus, it is not possible to send a message to a specific ECU. In the CAN protocol, after a message is broadcast to the network, devices that correctly receive this message will acknowledge their reception.

Typical CAN setup: A typical CAN setup for a vehicle will grant each ECU with a unique set of IDs and each message will be labeled with an ID, which is then transmitted onto the bus. An ECU will be responsible for a subset of the message IDs seen in the network, and each message ID will only be sent by a single ECU. Each message is queued by a software task, process or interrupt handler on the ECU, and each ECU will queue a message when the message's associated event occurs.

CAN frame format: Each CAN message from an ECU uses its assigned message ID (interchangeably referred to as the ID or the arbitration ID), which determines its priority on the CAN bus and may serve as an identifier for the message's contents. These messages are transmitted and received at the physical layer by an ECU's CAN controller as CAN data frames in the format depicted in Figure 4. The key fields in the CAN data frame, as relevant to our work, are: the start-of-frame (SOF) bit, the arbitration/message ID field, the acknowledgement (ACK) slot and the end-of-frame (EOF) bits.

All ECUs in the network with a queued message simultaneously start to transmit their message at the same time. During the arbitration ID field, all but one ECU will eventually stop transmitting based on CAN's arbitration resolution. Once an ECU has won arbitration on the bus, it will be the only sender and transmit the remainder of the CAN data frame until the ACK slot. During the ACK slot, the transmitter now becomes a receiver on the bus and all other ECUs in the network that correctly receive a message will simultaneously send a dominant bit on the network. This slot is then followed by the EOF and the inter-frame space (IFS).

Scenario	*Enum.*	*Src. map*	*Dest. map*
Malicious ECU discovery	✓	✓	
Continuous network attestation	✓	✓	✓
Lenient filter identification	✓		✓

Table 1: Mapping requirements for motivating scenarios

Message arbitration: To understand how ECUs communicate on the CAN bus, it is necessary to discuss the CAN message arbitration process [13–15, 33]. CAN is designed to support collision detection and bit-wise arbitration on message priority to allow higher-priority messages to dominate the network. The arbitration of these messages is performed on the message ID field of a data frame, where a lower ID indicates a higher priority. This priority-based arbitration process sets a 0-bit as dominant and a 1-bit as recessive. Since a 0-bit is dominant, a message with a lower ID will get priority on the CAN bus and will be sent before a message with a higher ID that is queued at the same time.

3.2 Mapping requirements

Unlike most traditional packet-switched networks, CAN messages do not have fields that identify the message's source and destination(s), which makes the mapping problem difficult. To develop a mapper that will aid in the motivating scenarios of §2, we formulate three required outputs for *CANvas*:

ECU enumeration: The importance of enumerating ECUs is evident in all of our provided scenarios as seen in Table 1; enumeration highlights new or absent ECUs. Note that in all of these scenarios, it is *not* necessary to know an ECU's type (engine, transmission, etc.) or its functionality (fan speed control, tire pressure sensing, etc.).

Formally, let E_i denote ECU i in a given vehicle that contains n total ECUs that are CAN-enabled. For each E_i in a vehicle's set of ECUs, $E_{1:n}$, the ECU is responsible for sending a specific set of m messages labeled with a unique arbitration ID from the set, $I_{E_i,1:m}$. This set of IDs is unique to E_i and no other ECU in the network should send the same ID. Given a CAN traffic dump from a vehicle, *CANvas'* enumerator should determine the number of ECUs, n, and differentiate between them to determine the set of ECUs $E_{1:n}$ for that particular vehicle.

Message source identification (§5): In the *malicious ECU discovery* and *continuous network attestation* scenarios, changes to the set of transmitted messages for each ECU can pinpoint a potentially malicious reconfiguration. This means that a goal for our mapper is to map each message ID to its source ECU.

Formally, given a CAN traffic dump from which we extract the set of uniquely-ID'd messages where l is the number of total unique message/arbitration IDs and $I_{1:l}$ is the set of

unique IDs, we should be able to determine which ECU E_i sent each unique message. This step is very closely related to ECU enumeration; once we know which ECU E_i that an arbitrary ID I_j originates from, we can produce a mapping of the ID to its source ECU, $I_j \in E_i$. Using this mapping, we can group the IDs with a common source ECU and complete our enumeration.

Message destination identification (§6): For the *continuous network attestation* scenario, we want to look for changes in what messages an ECU correctly receives as this could also indicate a potentially malicious reconfiguration. This component plays an important role in the *lenient filter identification* scenario, where an attacker could shutdown an ECU from an unintended message recipient.

We assume that at least one ECU in the network will correctly receive each message in the network. Formally, given the set of l unique IDs, $I_{1:l}$, from a traffic dump, we should be able to determine the set of ECUs, $E_{1:k}$, that correctly receive a message labeled with an arbitrary I_j. The expected output of this component should be a mapping of an ID to its destination ECUs, $I_{j,E_{1:k}}$.

3.3 Challenges in an automotive context

However, to achieve these mapping goals, we encounter two major challenges to determining the source and destination ECUs for CAN messages: (a) CAN lacks identifying source information and (b) CAN implements a broadcast protocol, which naturally implies that all nodes receive all messages. We discuss how we approach and solve these challenges in §5 and §6.

Lack of source information: If a message sent from ECU E_i has no identifying information, then it is non-trivial to determine that E_i sent the message. Since CAN messages are considered to be "contents-addressed" [13–15, 33], the value of the message ID is only related to the message's data and priority. In practice, the source ECU has no weight in determining the chosen arbitration ID for a particular message.

Broadcast protocol: We define destination as an ECU that correctly receives a message at the CAN controller level. Unfortunately, determining which ECUs correctly receive a message is non-trivial as an ECU connected to the CAN bus cannot detect which of its messages are received by certain ECUs. The ACK bit itself only indicates that some ECU has received the message, not which particular ECU(s) have received it. As multiple ECUs will set the ACK bit when a message is received, we cannot simply use this ACK bit to determine the set of ECUs $E_{1:k}$ that receive an arbitrary I_j.

4 System Overview

In this section, we provide a high-level overview of the *CANvas* network mapper.

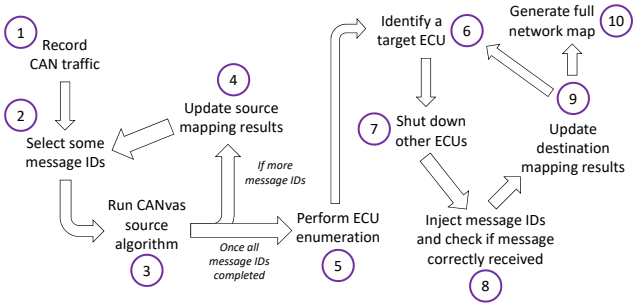


Figure 5: *CANvas* obtains source mapping results by step 4. Then, it will enumerate the ECUs in step 5. *CANvas* then performs destination mapping and generates the full map at step 10.

4.1 High-level idea

***CANvas* mapping overview:** We split *CANvas* into two main components: (1) a source mapper and (2) a destination mapper. As detailed in §3, we satisfy our ECU enumeration requirement by simply using the output of source mapping. For (1), we passively collect several minutes of CAN traffic. After an offline data collection, the source mapper uses the data to produce a mapping of each unique CAN ID to its source ECU and subsequently, by grouping IDs with a shared source, a list of all active source ECUs on the bus. For (2), we interact with the network directly and perform an online analysis to determine message destination. *CANvas* systematically isolates each ECU, which will most likely cause the vehicle to enter a temporary error state that the user can reset.

User capabilities: We assume that the user has access to the OBD-II port of the vehicle and can connect the *CANvas* mapper directly to the CAN bus with the ability to read and write to the bus. We also assume that the vehicle even has a CAN bus and that the standard CAN protocol is implemented, which most vehicles will reflect [11]. The user should also be able to transition the vehicle's ignition switch between the LOCK, ACC and ON positions as the user will have to reset the vehicle after each iteration to exit the error state.

Scope and evasion: We assume that the vehicle does not implement countermeasures that will alter timing of message transmissions, potentially to prevent intruders from identifying transmitting ECUs. We also assume that the vehicle cannot identify a maliciously-triggered error and prevent intruders from abusing CAN's error-handling protocol to shutdown an ECU. The vehicle should not employ an intrusion detection system capable of preventing an ECU suspension. We further discuss adversarial evasion and other scenarios for bus configurations in §8.

4.2 *CANvas* workflow

The workflow of *CANvas* involves four major steps seen in Figure 5:

1. *Data collection:* The CAN pins of the OBD-II port provide access to the frame-level signals and the message-level data. *CANvas* will read this traffic for several minutes and timestamp each received message. From this traffic, we will obtain the set of unique message IDs observed in the network and a set of timestamped data for each ID.
2. *Source mapping:* With the list of all unique message IDs, the source mapper will extract the timestamped CAN traffic for each ID and determine which IDs share the same source as detailed in §5. To do this, we select two message IDs and run their CAN traffic through our comparison algorithm, which will determine if the two IDs originate from the same ECU.
3. *ECU enumeration:* Using the set of matching ID pairs from source mapping, the enumerator will simply group pairs that originate from the same ECU. The output of this step will be a list of ECUs and associated source IDs.
4. *Destination mapping:* Using the ECU enumeration output, the destination mapper will identify the ECUs that correctly receive a given message ID. *CANvas* will isolate a target ECU by performing a shutdown on all other ECUs, which we discuss in §6. Once an ECU is isolated, we inject all unique observed message IDs and determine which ECUs receive the message.

5 ID Source Mapping

In this section, we describe an approach to map each CAN message to its source.

Intuition: Due to the absence of source information in a CAN message, we must rely on some uniquely identifying characteristic that can be tied to a particular ECU. Following observations from prior work [11, 29] and CAN documentation [2, 14], we consider *clock skew* as a candidate fingerprinting mechanism. In particular, time instants for in-vehicle ECUs rely on a quartz crystal clock [14], and we can use the relationship between these clocks to identify a transmitting ECU. We first define the following terms considering two clocks, $\mathbb{C}_1$ and $\mathbb{C}_2$:

- **Clock frequency**: The number of cycles per true second, e.g. if $\mathbb{C}_1$ operates at 16kHz, then $\mathbb{C}_1$ cycles 16,000 times every one true second.
- **Relative clock offset**: The difference in time reported by $\mathbb{C}_1$ and $\mathbb{C}_2$, e.g. if $\mathbb{C}_1$ reports time t_1 of 4.1ms and $\mathbb{C}_2$ reports t_2 of 4.2ms, their offset $O_{\mathbb{C}_1,\mathbb{C}_2}$ is 0.1 ms. Where only one clock is denoted for relative offset, the other clock is the clock of the receiving node.
- **Relative clock skew**: The difference in clock frequencies of two clocks, or the first derivative of offset w.r.t. true time, e.g. if $\mathbb{C}_1$ operates at 16kHz and $\mathbb{C}_1$ operates at 16.1kHz, their skew $S_{\mathbb{C}_1,\mathbb{C}_2}$ is 100Hz. Where only one clock is denoted for relative skew, the other clock is the clock of the receiving node.

Two clocks with a relative clock offset of 0 are considered to be *synchronized*, and two clocks with a nonzero relative clock skew are said to "skew apart," or have an increasing relative offset over time [2]. Since the CAN protocol does not implement a global clock, it is considered to be unsynchronized as each ECU relies on its own local clock.

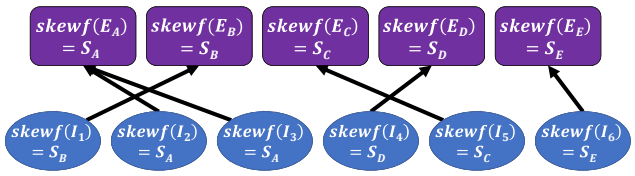


Figure 6: *CANvas* aims to cluster message IDs with a similar relative skew or offset.

Observation 1: ***The clock offset and skew of an ECU relative to any other ECU is distinct, thus providing us with a uniquely identifying characteristic for source mapping.***

High-level idea: To map each unique ID to its transmitting ECU, we break the module into two steps as Figure 6 illustrates: (1) computing either the skew $skewf(I_i)$ or offset $offsetf(I_i)$ of each ID I_i and (2) then clustering IDs with the same skew or offset where each cluster denotes a distinct source ECU, E_{src}. This module outputs a mapping of source ECUs to their set of source IDs. The main input to this module is a passively-logged CAN traffic dump, which contains entries in the form of $(I_i, t_{I_i,n})$ where I_i is the ID of the message and $t_{I_i,n}$ is the timestamp of the nth occurrence of I_i.

5.1 Prior work and limitations

Cho et al. [11] use clock skew as a means of building an intrusion detection mechanism to identify an attack by a malicious ECU. Specifically, this work uses timestamps of periodically-received message IDs and posit that IDs with the same skew originate from the same ECU.

To compute the clock skew of an ID I_i over time, Cho et al. [11] perform the following steps: (1) compute I_i's expected period, μ_{T_i}, (2) compute the offset, O_i, by subtracting the expected timestamp (using μ_{T_i} from the actual timestamp), (3) take the average of O_i over a batch of N messages, (4) add $O_{i_{avg}}$ to an accumulated offset, O_{acc}, and (5) then compute the skew, S_{I_i}, by taking the slope of O_{acc} versus time. This work uses the Recursive Least Squares algorithm to minimize the errors. After every batch of N messages, O_{acc} increases by O_i, where k is the kth batch. From this plot, since O_i should be constant, their formula for skew w.r.t. batch size sets S_{I_i} to:

$$skewf_i^{Cho}(N) = \frac{kO_i}{kN} = \frac{O_i}{N} \quad (1)$$

As an extension to this work, Sagong et al. [29] note that the skew of Equation 1 varies significantly based on N and use

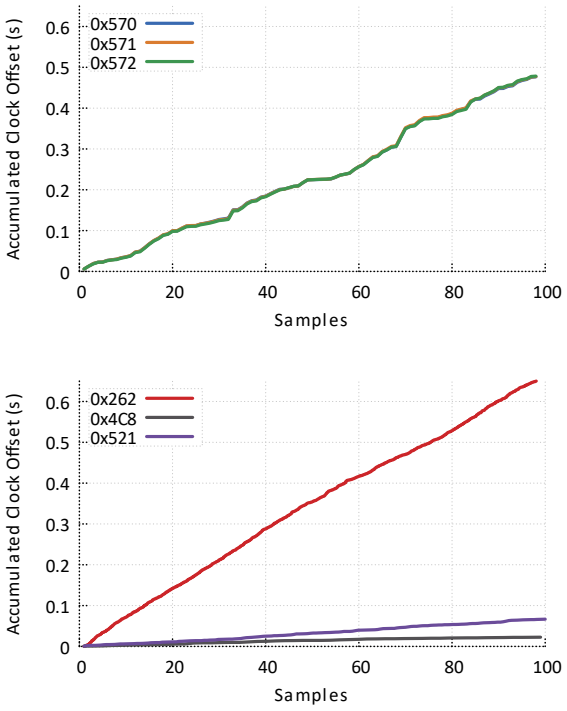


Figure 7: E_A transmits IDs 0x570, 0x571 and 0x572 at the same period and E_B transmits IDs 0x262, 0x4C8 and 0x521 at different periods. Above are plots of accumulated clock offset vs. samples for E_A and E_B using the algorithm by Cho et al. [11].

an updated formula for S_{I_i} w.r.t. batch size:

$$skewf_i^{Sagong}(N) = N \cdot \frac{kO_i}{kN} = O_i \tag{2}$$

Using data from a real vehicle, we now highlight a key limitation of Equations 1 and 2. Consider Figure 7: (1) E_A is the source of IDs 0x570, 0x571 and 0x572, which share the same period and (2) E_B is the source of IDs 0x262, 0x4C8 and 0x521, which each have different periods. In Figure 7, we use $skewf_i^{Cho}$ with $N = 20$ to plot the skew of all six IDs; $skewf_i^{Sagong}$ produces similar results. We can correctly conclude from Figure 7 that the IDs of E_A originate from a single ECU. However, from Figure 7, we will incorrectly conclude that IDs 0x262, 0x4C8 and 0x521 originate from three *separate* ECUs. Our analysis and experiments shed light on why these approaches fail–the skew value they compute is *period-dependent*.

As such, we update Equations 1 and 2 w.r.t. period T and batch size N:

$$skewf_i^{Cho}(N,T) = \frac{kO_i}{kTN} = \frac{O_i}{TN} \tag{3}$$

$$skewf_i^{Sagong}(N,T) = N \cdot \frac{kO_i}{kTN} = \frac{O_i}{T} \tag{4}$$

To potentially fix this issue, we can attempt a strawman that is not dependent on period or batch size.

$$skewf_i^{Straw}(N,T) = TN \cdot \frac{kO_i}{kTN} = O_i \tag{5}$$

Ideally, accounting for both batch-size and message-period (essentially batch-period, NT) using Equation 5 should give us a unique value that is common only among IDs from the same ECU. We apply Equation 5 for all I_i of a vehicle, and we attempt to establish distinct groupings of the computed skew for each ID, S_{I_i}, which would identify which I_i share the same E_{src}.

Unfortunately, this is a difficult task as I_i from the same E_{src} still do not have similar skews. This issue is further demonstrated as S_{I_i} varies across different data dumps or even segments of a given dump. Upon further inspection, we find that the measured S_{I_i} is affected by the deviation in an ID's period. This deviation in the period, σ_{p_i}, is attributed to sources of "noise", i.e. the period of a given message varies due to scheduling, queuing and arbitration delay. We also find that some I_i produce S_{I_i} with more deviation than others and produce widely-varying skew values, thus making our straw-man solution an unlikely candidate for source mapping.

Observation 2: ***We need a method of extracting the clock skew invariant that is: (a) independent of the period of I_i and (b) robust to noise in the period.***

5.2 Pairwise offset tracking

Issue with straw-man: In Equation 5, it is clear that, relative to the receiver, this "skew" function computes offset rather than true skew. Following our definitions in §5, a plot of relative offset over time should either be linearly increasing or decreasing if there is a nonzero skew between two clocks. In other words, if the relative skew between an E_{src} and the receiver is non-zero, then we should observe a gradual change in the offset. However, previous work [11, 29] fails to capture this change in offset over time.

Relative offset as a unique identifier: As mentioned in §5, clock offset and skew of an ECU relative to another ECU is distinct. We must note that the clock offset measured from one ID, I_1, of an E_{src} may not be the same as the offset of another ID, I_2, from E_{src}. If the initial transmission time of I_1 differs from that of I_2, the O_{I_1} could not equal O_{I_2}. Rather, the invariant here is the *change in relative offset*, ΔO_{I_i}; as the skew of E_{src} relative to the receiver is a constant nonzero value, the ΔO_{I_i} will be a constant nonzero as well (the derivative of offset is skew).

By measuring this change in offset, we can uniquely identify an E_{src}, but we must ensure our method of extracting this change in offset is (a) robust to a noisy period and (b) period-independent. To address the issue of noise in the period of I_i, p_{I_i}, we compute the relative offset between a pair

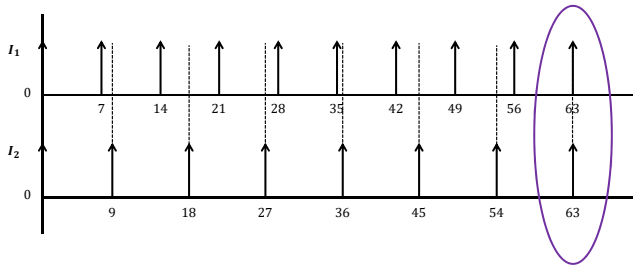


Figure 8: Timeline of two message IDs, I_1 and I_2, that have periods, p_1 = 7ms and p_2 = 9ms. Their hyper-period occurs every 63ms.

of two different IDs denoted by O_{I_1,I_2}. By performing this computation pair-wise, we expect O_{I_1,I_2} to have a deviation of approximately 0 if $I_1, I_2 \in E_{src}$ as the sources of noise for I_1, I_2 should mostly be shared. In reality, this deviation is very close but not exactly equal to 0; we define a practical threshold for this deviation in §7.

With a pairwise approach to computing O_{I_1,I_2} and the requirement for a period-independent approach, we face a new challenge: determining at what point in time to observe this relative offset regardless of the period of I_1 or I_2.

Observation 3: ***Compute offset at the* hyper-period *of* I_1 *and* I_2*, or the least common multiple of their periods.***

Measuring offset at the hyper-period: To guide our algorithm design for computing $\Delta O_{I_1,I_2}$ over time, we first model two periodically-transmitted IDs observed on the CAN bus. Consider two IDs, I_1 and I_2, from the same E_{src} which transmit at a period of p_1 and p_2, respectively. For example, let p_1 be 7ms and p_2 be 9ms. For now, we assume that the relative offset between I_1 and I_2 is 0. This offset should not change over time as they originate from the same E_{src}. To accurately compute the relative offset of these two IDs, O_{I_1,I_2}, we must select a time instant when the expected offset should also be 0: the hyper-period of I_1 and I_2, or the least common multiple of p_1 and p_2. As seen in Figure 8, this time instant occurs at 63ms, or the $lcm(7,9)$. Therefore, by computing the difference between the times reported from I_1 and I_2 every 63ms, or the hyper-period of I_1 and I_2, we can track the value of relative offset over time. If this relative offset is a nonzero constant, then the two IDs originate from the same ECU.

With an input of several minutes of timestamped CAN data to Algorithm 1, we can track relative offset over the timeline of two message IDs. Note that each timestamp has a noise component that stems from scheduling, queuing and arbitration delay. To compare whether two message IDs originate from the same ECU, we first assume that they are sent by separate ECUs. The two message IDs, I_1 and I_2, have periods, p_1 and p_2, and they have relative offsets, O_{I_1} and O_{I_2}. We draw the following relationships between these variables:

Algorithm 1 Pairwise offset tracking

```
1:  function PairwiseCompare(I1, I2, log_I1, log_I2)
2:      p1 = ⌊ComputeAveragePeriod(log_I1)⌋
3:      p2 = ⌊ComputeAveragePeriod(log_I2)⌋
4:      n = lcm(p1, p2)/p1
5:      m = lcm(p1, p2)/p2
6:      pos_I1 = 0, pos_I2 = 0
7:      Δ_I1,I2 = [ ]
8:      while pos_I1 < len(log_I1) and pos_I2 < len(log_I2) do
9:          Δ_I1,I2.append(log_I1[pos_I1] − log_I2[pos_I2])
10:         pos_I1 += n
11:         pos_I2 += m
12:     return true if σ(Δ_I1,I2) < threshold else false
13:     end function
```

- $p_2 = lp_1$, where l is the ratio of the periods.
- $O_{I_2} = jO_{I_1}$, where if j=1, then both IDs sent by same ECU; otherwise, they were sent by different ECUs.
- $n = ml$, where $LCM(n,m) = l$ as depicted in Figure 8.

By computing the difference between every n occurrences of I_1 and every m occurrences of I_2, which occurs at the *hyper-period* of I_1 and I_2, we produce the following equation:

$$O_{I_1,I_2} = (mp_2 + O_{I_2} + i_2) - (np_1 + O_{I_1} + i_1)$$

We find that when we average the result of the above equation across the entire data log, the expected value is 0 if I_1 and I_2 originate from the same ECU. In reality, this value is close to 0 due to the deviation of a message's period. From experimental data, we define a threshold of 1ms for the change in relative offset, where a value under the threshold will classify the two IDs with the same source ECU. Using this approach to revisit the setup described in Figure 7, we correctly conclude that IDs 0x262, 0x4C8 and 0x521 originate from the same ECU.

Practical challenges: While the above approach is correct, there are a number of other practical challenges we need to address to ensure accurate mapping:

1. *Large hyper-period*: Consider a hyper-period that is "large", or on the scale of several minutes, e.g. the hyper-period of $p_1 = 980$ms and $p_2 = 5008$ms is over 20 minutes. Since we only extract one relative offset value per hyper-period, we would need hours of CAN traffic to produce a valid result. To ensure that our mapper is fast, this length of traffic log is unreasonable; we want to produce a full network map in under an hour. Fortunately, with a pairwise approach, we can choose to *not* attempt a comparison when the hyper-period is large; for example, if we assume that the E_{src} of I_1 also transmits another ID, I_3, where the hyper-period of I_1 and I_3 is small, we can still determine that $I_1, I_3 \in E_{src}$.
2. *Large period deviation,* σ_{p_i}: In early experiments, we discovered messages that had a large measured σ_{p_i} (we define

large as $\sigma_{p_i} \geq 0.1 p_i$) and, at first, assumed that these messages were either aperiodic or sporadic (aperiodic with a hard deadline). However, upon closer inspection, we noticed that these messages appeared to be periodic in nature. We observed three different patterns that altered the measured σ_{p_i}: (1) the period simply had a large σ_{p_i}, (2) periodic messages would occasionally stop transmitting for some time, and (3) periodic messages were missing their deadlines. With a large enough σ_{p_i}, the deviation would conceal an inconstant ΔO_{I_i} and make it difficult to detect a mismatch. We experimentally find that a σ_{p_i} greater than 8% of p_i results in incorrect outputs. Therefore, *CANvas* will choose to test I_i on the following cases when its σ_{p_i} is under a defined threshold, which we set to $\sigma_{p_i} \leq 0.08 p_i$ from our experiments.

3. *Periodic messages that occasionally stop:* We find that some I_i are periodic and will stop transmitting for some time, causing a measured σ_{p_i} to be large. To combat this issue, we only perform pairwise offset tracking when the given message was actively transmitting. In the event we compare two I_i that both occasionally stop and there is no overlap of active transmissions, we then rely on our pairwise approach to match the I_i to another ID from the same E_{src}.
4. *Messages that miss deadlines*: For some I_i with a large σ_{p_i}, we observe two different inter-arrival times: p_i and $2p_i$. When a task on one of the ECUs misses its deadline and cannot produce a message on time, it will skip that cycle and transmit during the next cycle [2]. Thus, when a deadline is missed, we will observe an inter-arrival time of $2p_i$. In this situation, there are two options: (1) perform relative offset tracking on portions of the log when deadlines are not missed or (2) interpolate the missed inter-arrival times. If a message frequently misses its deadline, the first option is not viable. To interpolate a missed arrival time, we insert a psuedo-entry in the traffic log with a timestamp equal to the average of the preceding and the following timestamp.

Factors for mapping time: For source mapping, we experimentally find that 30 minutes of data provides enough samples for larger hyper-periods to map accurately. While this stage has static run-time, the variation in time requirements will be dependent on the number of observed messages IDs. The more message IDs that exist in the network, the longer the mapping time takes; vehicles with more message IDs take longer to complete mapping due to an increase in message-pairs. However, to further reduce mapping time, mapping messages with small periods requires much less traffic data. To save additional time if necessary, it is recommended to reduce the traffic log length for high-frequency messages. Also, if there are few large periodic messages or if those messages are not relevant for whatever reason, the length of the initial traffic log can be reduced as necessary instead of the recommended 30 minutes.

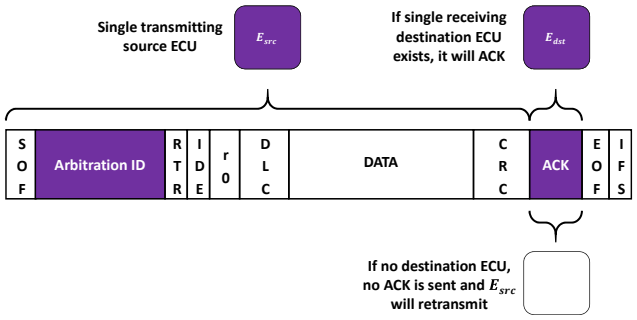


Figure 9: Observing ACK bit with single ECU in network.

6 ID Destination Mapping

The goal of the destination mapping module is to accurately associate each ID with its set of receiving ECUs. The key consideration here is to maximize the accuracy of our mappings within our defined time constraint. In this section, we describe an approach to map each CAN message to one or more destination ECUs as defined in §4 and then present a systematic procedure that reliably determines which messages an ECU correctly receives.

6.1 Problem formulation

Intuition: As defined in §4, the destination(s) of a particular CAN message are those ECUs who correctly receive a given message. Despite the broadcast nature of CAN, if an ECU does not correctly receive a message, it will not set the ACK bit; however, if other ECUs receive this message, they will set the dominant ACK bit. Unfortunately, an ACK observed by the transmitting ECU only means that *some* active ECU correctly received the message. Therefore, with multiple active ECUs in the network, we cannot identify which ECUs were the destination for a given message.

Consider the scenario in Figure 9 where there was only one active destination ECU, E_{dst}, in the network other than the transmitting source ECU, E_{src}. For each message sent by E_{src}, a set ACK bit (performed only by E_{dst}) would indicate that only one ECU received the message: E_{dst}. Thus, in this scenario, E_{src} could simply inject all possible I_i and detect which messages have a set ACK bit. The major challenge here is identifying a method of isolating an E_{dst} and "removing" all other ECUs from the network. We define the bare minimum of "removal" as preventing an ECU from participating in the acknowledgement process.

Observation 4: ***Our idea for performing this removal is to transition an ECU into an error-state that prevents it from setting the ACK bit for any message.***

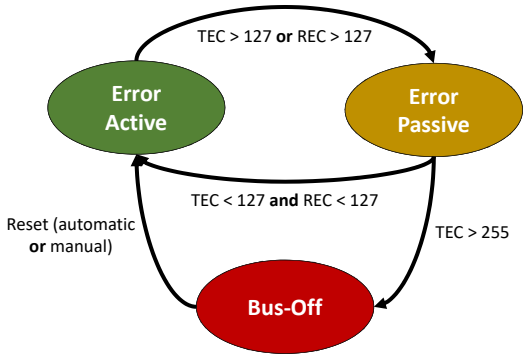


Figure 10: CAN transitions between three error states: error-active, error-passive and bus-off.

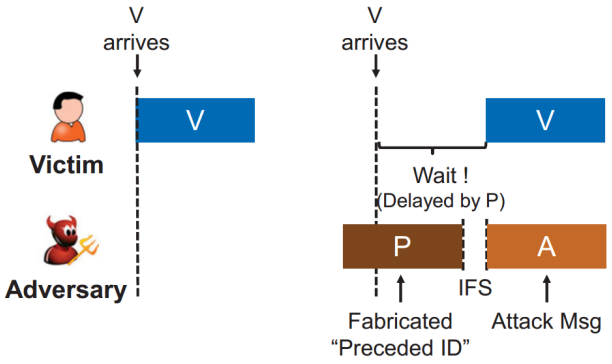


Figure 11: Injecting a fabricated message to impose a bus-off [10].

We now introduce the error-handling mechanism for CAN [2, 10], which follows the state diagram in Figure 10. Each ECU has two error counters: one for errors detected as a receiver (the Receive Error Counter, or REC), and another for errors detected as a transmitter (the Transmit Error Counter, or TEC). The TEC increments much faster than the REC as the transmitter is more likely to be at fault; the TEC increments by 8 while the REC increments by 1. If a message is received correctly, the error counter will decrease by 1. We describe the three CAN error-states and, under what conditions, the ECU will transition:

- **Error-active:** When an error is detected by an ECU in error-active, it will transmit an active error flag, or 6 dominant bits, that destroy the bus traffic. When either the TEC or REC increments past 127, the ECU transitions to error-passive.
- **Error-passive:** When an error is detected in error-passive, the ECU transmits a passive error flag, or 6 recessive bits, that do *not* destroy the bus traffic. Once the TEC or REC increases above 255, the ECU goes to bus-off.
- **Bus-off:** In this state, the ECU effectively removes itself from the network; it will not transmit anything onto the bus, including setting the ACK bit.

Thus, it is evident that we can isolate an ECU by transitioning all other ECUs to the bus-off state.

6.2 Limitations of prior work

Imposing bus-off state: The challenge in transitioning an ECU to bus-off is to determine what kind of error to produce and how to produce it. We look to previous work [10] that aims to shutdown an ECU for the purpose of an attack. The authors aim to shutdown an ECU by causing an error in the target ECU. By exploiting the error-handling protocol in CAN, where bus-off effectively removes an ECU from the network, they choose to increment the error counter of a target by causing a bit error. This error occurs when a transmitting ECU reads back each bit it writes; when the actual bit is different, the ECU invokes an error.

Since only one ECU is expected to win the bus arbitration as detailed in §2, the authors point out that two winners would potentially cause a bit error. For example, suppose that the victim ECU transmits a message with ID 0x262. If the attacker ECU also transmits ID 0x262 at the exact same time as the victim, both ECUs will win arbitration. However, to ensure that the victim has a bit error, the attacker's message will set its DLC, or data length count, to 0 (most practical messages contain at least some data). After a sufficient number of these attack messages, the victim ECU will transition into the bus-off state.

The main challenge here is synchronizing the attack message with the victim message so they both enter arbitration simultaneously. Their insight as depicted in Figure 11 is to inject a message of higher priority around the time when the victim should transmit. The higher priority message will block the victim until the bus is idle, where it will then transmit. The attacker will load its attack message immediately after the higher priority message is transmitted, thus allowing both the victim and attack message to arbitrate simultaneously. Since there is noise in the true transmission time of the victim's first attempt at transmitting, there is a chance that the attacker will need to make multiple attempts to cause an error. The number of injection attempts needed to cause a single bit error, κ, is defined as the following where $\mathbb{I}$ is a confidence attack parameter (high parameter value means higher confidence in attack), σ_{p_v} is the jitter deviation of the victim's period, and S_{bus} is the speed of the bus in Kbps:

$$\kappa = \left\lceil \frac{2\sqrt{2}\mathbb{I}\sigma_{p_v}S_{bus}}{124} \right\rceil \tag{6}$$

The authors state that only one of these injections is needed to cause a bit error if setting $\mathbb{I} = 3$ and at most 2 if setting $\mathbb{I} = 4$, given that the period deviation is 0.025ms.

Straw-man limitations: Suppose we used the above ap-

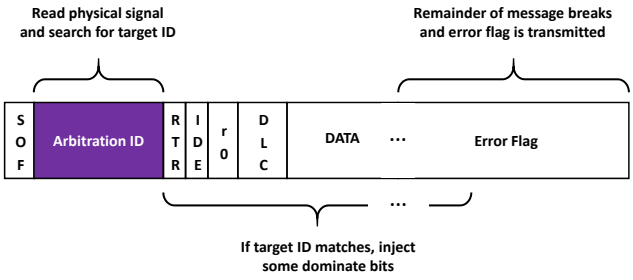


Figure 12: *CANvas* identifies target message by end of ID field and injects dominant bits during the DATA field.

proach to cause a bus-off in a real vehicle. Unfortunately, in sample traffic dumps from two real vehicles, the smallest deviation that we observed was approximately 0.15ms. Using the equation given by Cho et al. [10], the number of preceded message injections per error is 8 when the period deviates by at least 0.205ms; if 8 injections are required, any successful bit error would be undone by successful message transmissions. We look at available traffic logs used in the works by Miller et al. [24]. For this traffic log, the majority of the messages have a period deviation over 0.205ms. In other words, assuming the best case scenario of 0.15ms, we would need to inject at least 6 higher-priority messages, or preceded messages, for a bus speed of 500Kbps. Considering that each successful transmission by the victim ECU decrements the TEC by 1, we would effectively only increase the TEC by 2 with each successful attack (instead of the expected 8). Since the majority of messages have a period deviation greater than 0.205ms, it is highly unlikely to use this method for isolating an ECU.

Observation 5: ***We need a method of transitioning an ECU into the bus-off state that is reliable and robust even when the period deviates by more than 0.025ms.***

6.3 Forced ECU isolation

High-level idea: To map each unique ID to its set of destination ECUs, we break the module into two steps. We repeat these two steps for all n ECUs in the network. The first step is to isolate the target ECU and shut off all others by transitioning the non-target ECUs to the bus-off state. As there are n ECUs in the network, we will need to "bus-off" $n-1$ ECUs for each ECU, i.e. we will need to perform the bus-off at least $n(n-1)$ times. Once we isolate an ECU, we then inject the set of all I_i and observe which messages have a set ACK bit, thus identifying the set of I_i where the target ECU is an E_{dst}.

Inducing a direct bit-error: Isolating an ECU via the bus-off method requires a quick and effective approach. Since we are not limited to operating through the interface of a CAN controller, we can directly view the CAN frames in real-time via digital I/O pins. However, since we are using a microcontroller that operates at the same voltage of the CAN controller, we do not operate at the true CAN voltage. Instead, we tap directly between the interface of the Arduino's CAN controller and the CAN transceiver, where we can safely access the bus data. At this junction, we observe that the data on the line is within the Arduino's voltage and contains the full data frame, including SOF, ACK and EOF bits. With this access to the full data frame rather than just the components of the CAN message, we can directly induce an error on the bus and thus achieve the bus-off attack as seen in Figure 12.

Observation 6: ***By reading the ID of the message in real-time, we can choose to attack any ID by simply driving a dominant bit to the CAN transceiver.***

Note that the bus-off method requires attacking a message ID every time it occurs until the ECU enters the bus-off state. However, in the event that a message has a very long period, the time to perform the bus-off will not satisfy our speed requirement. As such, we can employ the result of *CANvas'* source mapping component by identifying the ID with the smallest period per ECU and attacking just that ID. In practice, we have found that every real ECU we have encountered has at least one ID that operates under 100ms. Thus, this approach makes the destination mapping component of *CANvas* fast.

Determining message receive filter: Now that we can isolate a single ECU in the network, we can simply inject all messages in the observed ID space and determine which messages are correctly received by the ECU. However, to view the ACK bit at the network level, which is not visible to the user, the obvious option is to use a logic analyzer. As this does not satisfy our requirement for low-cost mapping, we seek an alternative. We observed that if a message is sent to a single ECU and it does not correctly receive the message, the transmitter will re-attempt to send the message until it is received correctly. As such, if we transmit a message and see a continuous stream of the same ID from our transmitter, then we may conclude that the message ID is not received by the isolated ECU.

Practical challenges of mapping a real vehicle: Since our approach to destination mapping involves shutting off multiple ECUs at a time, we encounter a couple of challenges in a real vehicle setting: (1) ECUs that auto-recover and (2) ECUs that are persistently active. We now define these scenarios and provide a detailed approach to addressing these practical challenges:

1. *ECUs that auto-recover*: In our earlier experiments, we performed a simple experiment to verify the potential of an isolation method. We attempted to transition all ECUs in the network to the bus-off state by shorting the CAN bus pins, which would effectively cause a transmit error for all ECUs and force them into bus-off. However, after removing the short, we saw that some CAN messages were still transmitted onto the network, clearly indicating that some

ECUs left the bus-off state. We find that these ECUs would wait a predefined amount of time before re-transmitting again as these ECUs were critical to the vehicle's powertrain (engine, hybrid, etc.) [14]. In this situation, we would transmit a portion of the injected messages onto the bus and then re-isolate our target ECU when a non-target starts to transmit again. This approach is only reasonable for recovery times on the scale of seconds.

2. *ECUs that are persistently active*: Out of the set of ECUs that did auto-recover, we also noticed that one ECU seemed to be persistently active. In other words, there appeared to be no delay between a transition into the bus-off state and the next transmission from the ECU. Upon closer inspection, we found that this ECU would auto-recover only after 128 occurrences of 11 recessive bits [27]. In this situation, we must "hold" the bus open by constantly transmitting false messages from our device to trick the recovering ECU into thinking that the bus is still active.

Factors for mapping time: For destination mapping, the run-time is dependent on the number of ECUs and increases with more ECUs. We acknowledge the potential of long run-times for vehicles with 70+ ECUs if all were CAN-enabled. To combat this, we suggest performing the bus-off on the ID with the smallest period per ECU to reduce the time attributed to achieving ECU isolation. Also, for our two vehicles, all observed IDs were active when the vehicle was simply in ACC rather than ON so there may be no need to crank the engine per ECU.

7 Evaluation

In this section, we show that *CANvas*:

1. identifies an unexpected ECU in a '09 Toyota Prius,
2. identifies lenient message-receive filters in a '17 Ford Focus,
3. produces a sound source mapping of two real vehicles and accurately identifies the source of approximately 95% of all I_i in the network and a complete destination mapping with an isolation technique that is 100% reliable,
4. successfully demonstrate our forced ECU isolation on three extracted ECUs,
5. and produces source mapping of three additional vehicles.

Setup and methodology: Our experimental setup includes five real vehicles and several synthetic networks to demonstrate the above benefits. Below is a brief description of the *CANvas* hardware implementation, five real vehicles and our synthetic network of real ECUs:

- *Mapping device:* To interface with a CAN bus, our mapping device consists of three components: an Arduino Due microcontroller with an 84 MHz clock and an on-board CAN controller, a TI VP232 CAN transceiver, and a 120Ω resistor. To gain direct write access to the bus for destination mapping, we connect a digital I/O pin to the driver input pin of the transceiver.
- *'09 Toyota Prius and '17 Ford Focus:* The Prius contains eight original ECUs that transmit on a single CAN bus at 500 kbps. The Focus contains eleven original ECUs that transmit on three CAN buses at varying speeds; as our model of the Focus is the standard edition, only the high-speed 500 kbps bus has more than one active ECU. We obtain ground truth for our experiments by physically taking apart the car and gaining direct access to the ECUs by splicing directly into the CAN wires as seen in Figure 13. We use a paid subscription to both Toyota and Ford's mechanics' manuals [3, 6] for guidance on disassembly of vehicle components. Due to the non-destructive design of *CANvas*, our interaction does not impose any permanent errors to the vehicle.
- *'08 Ford Escape, '10 Toyota Prius and '15 Ford Fiesta* We obtain CAN traffic from three additional vehicles for testing only our source mapper, as we did not have permission to inject data. We use data from the '09 Prius and '17 Focus to partially confirm our source mapping output without disassembling these vehicles.
- *Synthetic networks:* To further validate the capability of our mapper, we perform additional experiments on three real engine ECUs extracted from a '12 Ford Focus, '13 Ford Escape and '14 Ford Escape.

7.1 Discovering an unexpected ECU

We now describe a real scenario where, in the process of designing *CANvas*, we discovered an unexpected ECU in our Prius. Using the results of our source mapping on the '09 Prius as seen in Table 2, we noticed that there were a total of nine ECUs when only eight were expected. Even after manually disconnecting all eight known ECUs, we still observed CAN traffic, specifically IDs $I_{570-572}$, coming from a single ECU. By looking at the history of the vehicle and systematically disconnecting various systems, we discovered that this ECU was installed as part of a modification from several years ago. The Prius had an additional battery installed to grant it all-electric capabilities, and with the use of the network mapper, we now know that a new CAN-enabled device was added. If we took a network map of the vehicle when first purchased or used an online database as mentioned in §2, we could easily compare our results with published results and identify the unexpected ECU. We confirm that these IDs are new by comparing our IDs to a same-generation Prius [23].

7.2 Identifying lenient filters

As detailed in §2, a real concern for network security is the ability to shut-down an ECU by simply receiving the target's CAN messages. Using the results of *CANvas'* destination mapping, we can identify several instances where an ECU is expected to only receive messages from a subset of other ECUs but still receives all other messages. We have found that all ECUs in the Focus and Prius do not employ any filter on the receipt of incoming messages. In Ford's Motorcraft TechInfo





Figure 13: Images of the vehicles we used for ground truth: the 2009 Toyota Prius and the 2017 Ford Focus.

ECU #	*Source message IDs*	*Actual ECU*
A	020, 030, 0B1, 0B3, 0B4, 230, 4C3, 591	Skid control ECU
B	022, 023	Yaw rate sensor
C	025, 4C6	Steering sensor
D	038, 03A, 03E, 120, 244, 348, 527, 528, 529, 540, 5B2, 5C8, 5EC, 602	Hybrid vehicle control ECU
E	039, 3C8, 3CF, 526, 52C, 5CC, 5D4, 5F8	Engine control module
F	262, 4C8, 521	Power steering ECU
G	3C9, 3CB, 3CD	Battery ECU
H	553, 554, 57F, 5B6	Gateway ECU
I	570, 571, 572	*Unknown ECU*

Table 2: 2009 Toyota Prius source mapping output

Service [3], we can see simple diagrams of how the ECUs communicate as part of the vehicle's systems. For example, the Focus' braking system involves communication between the instrument panel cluster, the transmission ECU, the body control ECU and the engine ECU. Now suppose an attacker takes over the infotainment unit of the Focus, has complete access to rewrite the ECU's code and gains the ability to inject CAN messages as described in §2. The attacker can launch a bus-off attack and shut-down the transmission ECU simply because the infotainment ECU receives its messages. It is evident that these devices need filters on what messages are received by their CAN controllers.

7.3 Mapping our test vehicles

We now present results and observations from mapping both the Prius and Focus.

Source mapping results: Using a threshold of 1ms and 30 minutes of traffic collection, we get a false positive rate of 0% for both vehicles, permitting us to get a sound source mapping output. Out of a total of 59 unique message IDs, our pairwise timing comparison resulted in 102 matching pairs for the Prius. By performing a simple grouping of these pairs as detailed in §5, we get the output as seen in Table 2. While the majority of the IDs observed on the Prius have a strong periodic characteristic, we discuss some special cases we encountered. Most of the messages were under five seconds except for I_{57F} with a period of 5 seconds and I_{602} with a period of 60 seconds. The majority of our messages matched with multiple IDs from the same ECUs but due to the large period of I_{57F} and I_{602}, they only had a single match. However, due to our pairwise approach, we can still map these two IDs using a shared matching pair as discussed in §5. We also encounter a few examples of messages that miss their deadline and wait until the next cycle to re-transmit. For the Focus, we observe messages that miss their deadlines and either transmit two messages on the next cycle or drop the missed message and wait for the next cycle. In these cases, we simply remove the inter-arrival times that exceed two standard deviations from the average period and interpolate for the removed timestamps as discussed in §5.

Destination mapping results: With a CAN bus running at 500 kbps, we discover that all of the ECUs in the Prius do not implement any filtering between the network and the CAN controller. When each ECU is isolated, we see that all IDs are properly acknowledged by the receiving ECU. We do observe two ECUs that recover quickly from the bus-off method, specifically the engine control module and the skid control ECU. With the other ECUs in the vehicle, it was sufficient to perform our bus-off once and the ECU would stop transmitting. For these two ECUs, we selected the smallest period ID and held the bus open by injecting false messages to keep the two ECUs from auto-recovering. Additionally, we discovered that the Focus also do not implement any sort of filtering for the IDs we observe on the CAN. From these

findings, we can conclude that attacking via the reception of a message for these vehicles could prove trivial due to the lack of filtering between the network and the controller. In general, the maximum number of manual transitions of the ignition switch is equal to the number of detected CAN-enabled ECUs in the vehicle. For the keyless ignition of the 2009 Prius, we transition the ignition 7 times as two ECUs recover on their own (the Prius has 9 total CAN-enabled ECUs). For the keyed ignition of the 2017 Focus, we transition the ignition 7 times as two ECUs recover on their own (the Focus has 9 total CAN-enabled ECUs).

7.4 Mapping additional vehicles

Mapping real extracted Ford ECUs: We also obtained three Ford engine ECUs from a '12 Focus, '13 Escape and '14 Escape. By collecting data from these three ECUs, we found that they shared the many of the same message IDs and conclude that they are based off of the same engine controller configuration. As they all auto-recover, they were prime candidates for testing our forced ECU isolation technique.

We use *CANvas* on three other vehicles to look for data that seems logical to our findings from the test cars. For the Ford vehicles, we look for similarities with our extracted engine ECUs. For the '08 Escape, we found a set of IDs that we believe is the engine ECU and only has a subset of those found on our extracted ECU. For the '15 Fiesta, we also found a likely candidate for an engine ECU that has more IDs than our extracted ECUs. Since these vehicles range over three different Ford generations, it seems logical that the newer engine ECUs transmit more IDs. Additionally, we find a few similarities between the '09 and '10 Prius. We found an ECU on the '10 that is likely to be the skid control ECU, which has similar IDs to the '09 Prius. These findings potentially demonstrate *CANvas'* source mapping capabilities.

8 Discussion

Adversarial evasion: For *CANvas'* source mapping, an adversary could attempt to modify the timestamps to trick *CANvas* into thinking that a pair of IDs originate from the same ECU when in fact the opposite is true, and vice versa. We acknowledge that an attacker who aims to spoof IDs from an implanted or compromised ECU breaks the assumption for message-source analysis. If the attacker performs an active attack (i.e. attack occurs during data capture) or simultaneously transmits with the spoofed ECU, then IDSes from several previous works could detect such an attack and thus we did not perform such experiments. *CANvas* instead could discover ECUs that do not actively inject messages but rather change the ID-ECU source mapping (a new ECU or existing ECU that sends different IDs). We also make the assumption that ECUs do not intentionally alter their timing due to the challenges that arise from scheduling real-time embedded systems. There are numerous challenges that automakers already face in achieving reliable and robust scheduling for their vehicles and any modification to the timing of CAN messages would add a great amount of complexity to the already complex challenge of scheduling. Additionally, as our destination mapping approach deals with the error-handling mechanism, it would also not be practical to change these basics of CAN.

Avoiding permanent damage: We take care to avoid any damage to our test vehicles. Even with our active interaction with the bus in destination mapping, most dash lights that turn on are simply reset by power cycling the car; it may sometimes be necessary to drive the car for a few minutes so the ECUs can identify the absence of a real error. After mapping, all of our vehicles operate with no error codes once the above steps have been followed. Sometimes, a persistent Diagnostic Trouble Code may exist in the network as indicated by the Malfunction Indicator Light (MIL, commonly known as a "check engine light"). To remedy this, a simple OBD-II scan tool can be used to reset these lights with no harm to the vehicle. In the event of network communication failure (e.g. bus-off), manufacturers implement a "limp-home" mode where ECUs will default to secondary programming and allow the vehicle to operate with limited capabilities [7]. It is possible for the CAN bus to be shorted (effectively causing a bus-off on all ECUs) during faults, repairs, etc. so this mode protects the vehicle from our methods. In our experiments, the engine did not need to be running as all ECUs became active with the ignition at ACC. However, this may not apply to all vehicles so it is possible that the ignition will need to be ON.

Multiple CAN buses: For the typical OBD-II port, the CAN bus uses pins 6 and 14 on the connector. While many vehicles only have one CAN bus using these pins, it is possible for additional CANs to exist. These CAN buses may not be connected and they may employ different bus speeds. Sometimes, vehicles may also employ a gateway which handles how and which messages are passed between the various buses for reasons of fault confinement and network security. These CAN buses are often accessible at the OBD-II port but on different pins that are vendor optional: pins 3 and 11 and pins 1 and 8/9. In the case that a CAN bus is not exposed to the OBD-II, it is possible to access this bus by simply removing the door panel of a car and accessing the connector between the door assembly and the car body. This connector will likely contain the unexposed bus, which can be discovered as suggested by others [30].

Message acceptance filtering: CAN controllers have the option to employ a programmable acceptance filter where a message that is received by the controller can either be sent to the application layer or dropped after the message is received. It is possible to define message destination as a message that is "accepted" by an ECU rather than correctly received. This definition provides finer granularity on message destination and can prove useful for many other security scenarios; however, to identify what messages are accepted by an ECU, this

may require vendor-specific methods. For example, in our experimental setup, we enable a CAN protocol feature called the overload frame [32]. If a vendor chooses to enable this feature, an accepted message can be determined by flooding the bus as fast as possible with a given message ID. When the receiving ECU gets behind on processing these messages, it will transmit an overload frame, indicating its acceptance filter allows the injected message ID; if the ID is dropped, then no overload frame will be present.

Non-transmitting ECUs: *CANvas* expects ECUs to transmit their messages periodically, but it is possible for ECUs to only activate under certain conditions or simply read from the network. As all ECUs that receive messages but have the ability to write to the network must participate in the ACK process, *CANvas'* forced ECU isolation technique can be used to identify the presence of a non-transmitting ECU. *CANvas* should detect these ECUs prior to starting to ensure that the detected ECUs do not interfere with destination mapping.

9 Related Work

We already discussed several of the key related work with respect to source and destination mapping. We discuss other related efforts here.

Automotive attacks: There have been a number of efforts at demonstrating vulnerabilities of automotive networks, including work on injecting messages [20], attacking keyless entry systems [8, 16, 28], and specific components such as TPMS [17, 18]. Our work can better inform such attack efforts and defenses by proactively identifying possible attack channels.

Intrusion detection for automotive: Given the growing security concerns, related work has also developed intrusion detection and firewall capabilities akin to traditional networks (e.g., [11, 19, 22, 29, 31]). Some of these may interfere with mapping efforts. More generally, however, these may have blind spots that a network mapper can highlight.

Alternative source identification: We acknowledge previous efforts that aim to identify message sources [12, 27]. While these efforts may prove valid, they either require many hours of data or require physical access to the bus for just source mapping. *CANvas* permits source mapping using a passively-recorded timestamped traffic log.

Authentication in CAN: We acknowledge that authentication for CAN devices may implicitly solve the source mapping problem. However, proposed authentication methods are rarely employed in real vehicles due to either the permanent addition of new devices or changes to the existing CAN protocol. Prior work, such as the TCAN system [5], requires the addition of a new device, access to two locations on the bus and a static authentication table. *CANvas*, however, acknowledges that timing characteristics can and will change due to clock drift. By comparing clock offsets, *CANvas* does not rely on static timing characteristics. CANvas does not even need physical access to the bus for source mapping as we only require a hardware-timestamped traffic log, and we operate solely from the OBD-II port without an additional permanent device.

Other work on ECU fingerprinting: Following initial efforts on fingerprinting [14, 27], other work has improved on their basic approach by identifying potential pitfalls [11, 12, 29]. As we show in our work, all of these still suffer from the same limitations in our context as they still assume either active access to the bus or very long traffic dumps.

10 Conclusions

In this work, we develop *CANvas*, a fast and inexpensive automotive network mapper. We have released our code and data under open-source licenses to enable further work in this area. A natural direction of future work is to add richer functionality, e.g. identifying the function of an ECU (transmission ECU, engine ECU, etc.), identifying gateway ECUs that potentially bridge multiple CAN buses and identifying vendor-specific message acceptance filters. Future work should also investigate network mapping on other automotive protocols, e.g. automotive Ethernet.

Acknowledgements

This work was funded in part by the PITAXVIII PITA award and the CNS-1564009 NSF IoT award. We gratefully acknowledge support from Technologies for Safe and Efficient Transportation (T-SET) University Transportation Center. This work was also supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. We thank the anonymous reviewers and our shepherd Konrad Rieck for their helpful suggestions.

Availability

This work is made available [1] to encourage the community to add richer functionality and use *CANvas* to further the creation of automotive security tools.

References

[1] Canvas. https://github.com/sekarkulandaivel/canvas.

[2] Introduction to can. http://www.ti.com/lit/an/sloa101b/sloa101b.pdf.

[3] Motorcraft info service. https://www.motorcraftservice.com/.

[4] Obd-ii background information. http://www.obdii.com/background.html.

[5] Tcan: Authentication without cryptography on a can bus based on nodes location on the bus. https://autosec.se/wp-content/uploads/2019/03/3.-ESCAR-EU-2018.pdf.

[6] Toyota techinfo service. https://techinfo.toyota.com.

[7] What limp mode is, and why cars use it. https://repairpal.com/symptoms/what-is-limp-mode-why-cars-use-it.

[8] Ansaf Ibrahem Alrabady and Syed Masud Mahmud. Analysis of attacks against the security of keyless-entry systems for vehicles and suggestions for improved designs. *IEEE transactions on vehicular technology*, 54(1):41–50, 2005. https://ieeexplore.ieee.org/iel5/25/30186/01386610.pdf.

[9] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, Tadayoshi Kohno, et al. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security Symposium*, pages 77–92. San Francisco, 2011. http://www.autosec.org/pubs/cars-usenixsec2011.pdf.

[10] Kyong-Tak Cho and Kang G Shin. Error handling of in-vehicle networks makes them vulnerable. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1044–1055. ACM, 2016. https://dl.acm.org/citation.cfm?id=2978302.

[11] Kyong-Tak Cho and Kang G Shin. Fingerprinting electronic control units for vehicle intrusion detection. In *USENIX Security Symposium*, pages 911–927, 2016. https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_cho.pdf.

[12] Wonsuk Choi, Hyo Jin Jo, Samuel Woo, Ji Young Chun, Jooyoung Park, and Dong Hoon Lee. Identifying ecus using inimitable characteristics of signals in controller area networks. *IEEE Transactions on Vehicular Technology*, 67(6):4757–4770, 2018. https://ieeexplore.ieee.org/iel7/25/4356907/08303766.pdf.

[13] Robert I Davis, Alan Burns, Reinder J Bril, and Johan J Lukkien. Controller area network (can) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, 2007. https://link.springer.com/article/10.1007/s11241-007-9012-7.

[14] Marco Di Natale, Haibo Zeng, Paolo Giusto, and Arkadeb Ghosal. *Understanding and using the controller area network communication protocol: theory and practice*. Springer Science & Business Media, 2012. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.512.5543&rep=rep1&type=pdf.

[15] Mohammad Farsi, Karl Ratcliff, and Manuel Barbosa. An overview of controller area network. *Computing & Control Engineering Journal*, 10(3):113–120, 1999. https://ieeexplore.ieee.org/iel5/2218/17068/00788104.pdf.

[16] Aurélien Francillon, Boris Danev, and Srdjan Capkun. Relay attacks on passive keyless entry and start systems in modern cars. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. Eidgenössische Technische Hochschule Zürich, Department of Computer Science, 2011. https://www.research-collection.ethz.ch/bitstream/handle/20.500.11850/42365/eth-4572-01.pdf.

[17] Abdulmalik Humayed and Bo Luo. Cyber-physical security for smart cars: taxonomy of vulnerabilities, threats, and attacks. In *Proceedings of the ACM/IEEE Sixth International Conference on Cyber-Physical Systems*, pages 252–253. ACM, 2015. https://dl.acm.org/citation.cfm?id=2735992.

[18] Rob Millerb Ishtiaq Roufa, Hossen Mustafaa, Sangho Ohb Travis Taylora, Wenyuan Xua, Marco Gruteserb, Wade Trappeb, and Ivan Seskarb. Security and privacy vulnerabilities of in-car wireless networks: A tire pressure monitoring system case study. In *19th USENIX Security Symposium, Washington DC*, pages 11–13, 2010. https://www.usenix.org/legacy/event/sec10/tech/full_papers/Rouf.pdf.

[19] Min-Joo Kang and Je-Won Kang. Intrusion detection system using deep neural network for in-vehicle network security. *PloS one*, 11(6):e0155781, 2016. https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0155781.

[20] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, et al. Experimental security analysis of a modern automobile. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 447–462. IEEE, 2010. http://www.autosec.org/pubs/cars-oakland2010.pdf.

[21] Gordon Fyodor Lyon. *Nmap network scanning: The official Nmap project guide to network discovery and security scanning*. Insecure, 2009. https://dl.acm.org/citation.cfm?id=1538595.

[22] Tsutomu Matsumoto, Masato Hata, Masato Tanabe, Katsunari Yoshioka, and Kazuomi Oishi. A method of preventing unauthorized data transmission in controller area network. In *2012 IEEE 75th Vehicular Technology Conference (VTC Spring)*, pages 1–5.

IEEE, 2012. https://ieeexplore.ieee.org/iel5/6238551/6239848/06240294.pdf.

[23] Jérôme Maye and Mario Krucker. Communication with a toyota prius. https://attachments.priuschat.com/attachment-files/2017/04/122809_Communication_with_a_Toyota_Prius.pdf.

[24] Charlie Miller and Chris Valasek. Adventures in automotive networks and control units. *Def Con*, 21:260–264, 2013. http://illmatics.com/car_hacking.pdf.

[25] Charlie Miller and Chris Valasek. A survey of remote automotive attack surfaces. *black hat USA*, 2014:94, 2014. http://illmatics.com/remote%20attack%20surfaces.pdf.

[26] Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015:91, 2015. http://illmatics.com/Remote%20Car%20Hacking.pdf.

[27] Pal-Stefan Murvay and Bogdan Groza. Source identification using signal characteristics in controller area networks. *IEEE Signal Processing Letters*, 21(4):395–399, 2014. https://ieeexplore.ieee.org/iel7/97/4358004/06730667.pdf.

[28] Irving S Reed, Xiaowei Yin, and Xuemin Chen. Keyless entry system using a rolling code, February 4 1997. https://patentimages.storage.googleapis.com/c3/02/da/89f0cef9c2a9ea/US5600324.pdf.

[29] Sang Uk Sagong, Xuhang Ying, Andrew Clark, Linda Bushnell, and Radha Poovendran. Cloaking the clock: emulating clock skew in controller area networks. In *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*, pages 32–42. IEEE Press, 2018. https://dl.acm.org/citation.cfm?id=3207896.3207901.

[30] Craig Smith. *The Car Hacker's Handbook: A Guide for the Penetration Tester*. No Starch Press, 2016. http://opengarages.org/handbook/.

[31] Hyun Min Song, Ha Rang Kim, and Huy Kang Kim. Intrusion detection system based on the analysis of time intervals of can messages for in-vehicle network. In *2016 international conference on information networking (ICOIN)*, pages 63–68. IEEE, 2016. https://ieeexplore.ieee.org/abstract/document/7427089/.

[32] CAN Specification. Bosch. 1991. http://esd.cs.ucr.edu/webres/can20.pdf.

[33] Ken Tindell, H Hanssmon, and Andy J Wellings. Analysing real-time communications: Controller area network (can). In *RTSS*, pages 259–263. Citeseer, 1994. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.57.5047&rep=rep1&type=pdf.

Losing the Car Keys: Wireless PHY-Layer Insecurity in EV Charging

Richard Baker
University of Oxford
richard.baker@cs.ox.ac.uk

Ivan Martinovic
University of Oxford
ivan.martinovic@cs.ox.ac.uk

Abstract

Electric vehicles (EVs) are proliferating quickly, along with the charging infrastructure for them. A new generation of charger technologies is emerging, handling more sensitive data and undertaking more complex interactions, while using the charging cable as the communication channel. This channel is used not only for charging control, but will soon handle billing, vehicle-to-grid operation, internet access and provide a platform for third-party apps — all with a public interface to the world.

We highlight the threat posed by wireless attacks on the physical-layer of the Combined Charging System (CCS), a major standard for EV charging that is deployed in many thousands of locations worldwide and used by seven of the ten largest auto manufacturers globally. We show that design choices in the use of power-line communication (PLC) make the system particularly prone to popular electromagnetic side-channel attacks. We implement the first wireless eavesdropping tool for PLC networks and use it to observe the ISO 15118 network implementation underlying CCS, in a measurement campaign of 54 real charging sessions, using modern electric vehicles and state-of-the-art CCS chargers. We find that the unintentional wireless channel is sufficient to recover messages in the vast majority of cases, with traffic intercepted from an adjacent parking bay showing 91.8% of messages validating their CRC32 checksum.

By examining the recovered traffic, we further find a host of privacy and security issues in existing charging infrastructure including plaintext MAC-layer traffic recovery, widespread absence of TLS in public locations and leakage of private information, including long-term unique identifiers. Of particular concern, elements of the recovered data are being used to authorise billing in existing charging implementations.

We discuss the implications of pervasive susceptibility to known electromagnetic eavesdropping techniques, extract lessons learnt for future development and propose specific improvements to mitigate the problems in existing chargers.

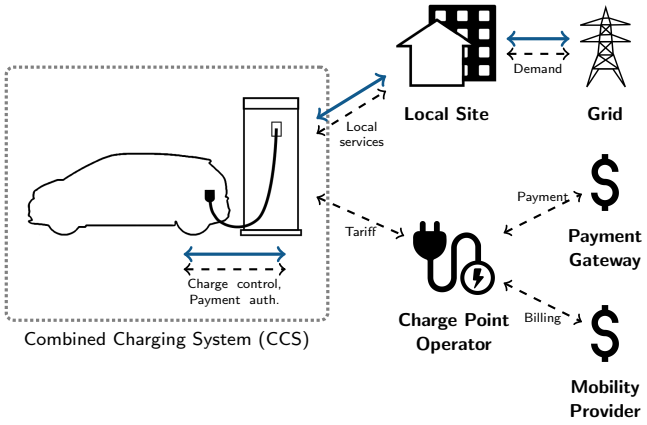


Figure 1: Overview of EV charging with V2G and payment options shown. Solid, blue lines indicate power flow whilst dashed, black lines indicate communication.

1 Introduction

The rise of electric vehicles (EVs) as a contemporary and future transport mechanism has been swift in recent years and continues to accelerate, helped by prevailing attitudes, technological advances and notable personalities contributing in the area. There are already widespread government plans to eradicate fossil-fuel vehicles in cities [61], states [28] and countries [9] in the coming years.

As EV technology advances rapidly, the availability of charging infrastructure has become a challenge for users, who require access both to private charging points at home and public ones on longer journeys. The lack of sufficient charging points is noted as a slowing influence on adoption of electric mobility [62] and this has prompted endeavours to expand the infrastructure, both from governments recognising the potential public good and from competing EV manufacturers who understand that having the best infrastructure makes their vehicles more appealing to purchasers. There are already multi-billion dollar pulibc deployment plans in

progress [18] and predictions of worldwide numbers exceed 50 million chargers by 2025 if private systems are included [2].

With several major charging standards in existence, the race to become the dominant one has reached a fervour in recent years and a new generation of high-power charging systems has emerged. But the pressure to achieve rapid expansion has so often been seen to inhibit secure implementation. Users demand charging systems that are consistent and convenient, but with such drive for the adoption of electric mobility, it is critical that they are also secure. The security community has raised concerns in the past that standards do not fully address security and privacy issues [4, 8, 72], as well as noting vulnerabilities in back-end and payment systems of earlier charging system deployments [35, 19].

Meanwhile the complexity of developing all the infrastructure required for a secured charging network is enormous. As Figure 1 shows, vehicle charging involves interaction between the vehicle, the owner, the charger operator, a payment gateway and the grid regulator. This requires establishing communication links capable of supporting the higher-level protocols for this interaction, within a dynamic and untrusted environment, where many thousands of users come and go. It also necessitates trust relationships between all the participants to ensure each is acting legitimately.

In light of the challenges this infrastructure development faces and the acknowledged side-channel vulnerabilities that exposed cabling presents, we undertook to investigate the security of the charging cable communication.

We make the following specific contributions:

1. *Demonstrate that the use of powerline communication, and its specific configuration in CCS, makes systems particularly vulnerable to EM eavesdropping*
2. *Develop an eavesdropping system for HomePlug GP and the ISO15118 PHY-layer*
3. *Conduct a real-world measurement campaign, demonstrating the widespread nature of the problem*
4. *Highlight the potential for privacy violation and user tracking with existing systems*
5. *Propose countermeasures to mitigate the capabilities of an eavesdropper*

Our findings are relevant to thousands of chargers across Europe and North America [29, 67], along with having implications for ongoing deployments both in public locations and private homes.

2 Background

The availability of EV charging infrastructure is growing enormously. Early, simple alternating-current chargers are being superseded by a new generation of charging technologies that provide greater charging power and advanced functionality. The greater power is provided by the use of direct-current (DC) charging, allowing an enormous increase in current delivery over previous alternating-current designs. Public DC charging stations currently well exceed the 3kW power levels commonly available in a home, with 50kW supplies plentiful and those providing up to 350kW soon to appear [30][38]. But the improvements in *power* are only part of the benefit of this new generation of technologies. The *communication* capabilities are also vital to enable a host of new uses:

Reactive charging allows a vehicle to vary its charging process based on electricity price or expected time of departure.

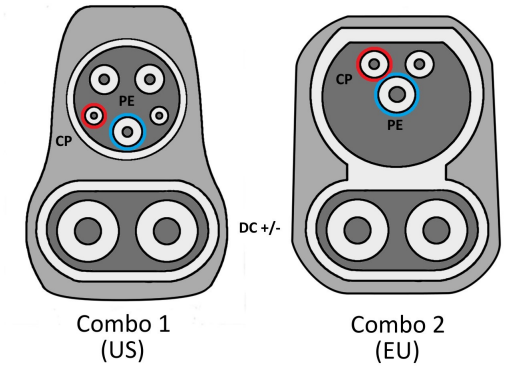
Automatic billing or "plug-and-charge" allows a vehicle to authorise billing of its owner for charging, without the owner explicitly interacting with it. Aside from the obvious convenience benefit, the same capability also allows the user to 'roam' between charging providers with a seamless experience as cross-provider billing is handled automatically as well.

Vehicle-to-Grid (V2G) makes use of bidirectional power flow to allow the vehicle to deliver energy as well as consume it. As energy prices fluctuate with demand, the vehicle can either act as a storage battery for a user's home or sell energy back to the grid on demand. This can bring economic benefits for the user and stability improvement for the grid operator.

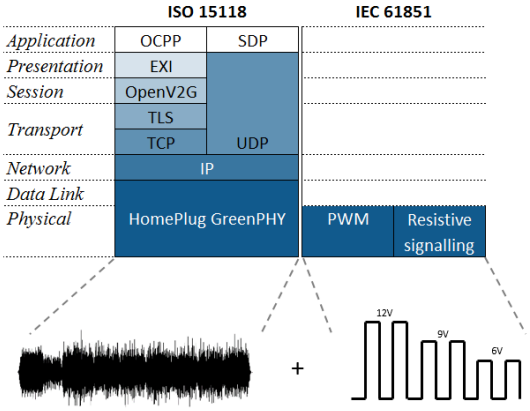
External payment is commonly provided by RFID cards [19], apps that communicate with the charger separately or card payment terminals. Additional systems exist though, for payment through separate providers or via a blockchain network [64, 74, 6, 26].

Additional services that operate in conjunction with charging are proposed [47]. In a private environment this might comprise access to the local network to communicate with smart-home devices or make use of domestic internet service and avoid mobile network charges. At public charging stations site-specific services such as loyalty schemes, to-vehicle delivery, parking charges or 'where-have-I-parked' reminders can operate, with middleware layers to support an app ecosystem in commercial development [22]. Internet access can also be made available for connected vehicles in areas without mobile network coverage, such as underground parking complexes.

Examples of each are in production use and deployment is



(a) Two charging cables are used by CCS. The Combo 1 and Combo 2 plugs are dominant in the US and Europe respectively, while other locations adopt one or the other. DC power is delivered by the large conductors at the bottom of the plug, meanwhile communication happens over the Control Pilot and Protective Earth lines (red and blue, respectively).



(b) CCS high-level and low-level signalling share the same communication lines. The corresponding ISO 15118 PLC and IEC 61851 systems have their signals superposed at the physical layer. The PLC provides a standard IP stack for use by charging traffic and other services.

Figure 2: Illustrations of the physical connectors for CCS charging, along with the network stack used for communication.

becoming more widespread. The underpinning communication mechanisms go beyond indicating presence and readiness to charge, also providing a general-purpose channel for software operating in the vehicle and charger. Figure 1 shows the potential extent of communication during charging. The vehicle can demand current flow, the charger can provide tariff information for reactive charging or reverse current demands for vehicle-to-grid, and the two can interact with external parties for automatic billing or to provide additional services.

Four major next-generation charging systems exist: CHAdeMO[1], Supercharger[2], GB/T 20234[3] and the Combined Charging System (CCS)[4]. Each uses the charging cable for primary communication: CHAdeMO, Supercharger and GB/T 20234 make use of CAN-Bus, whilst CCS makes use of powerline communication (PLC).

We examine the CCS standard as it has the most extensive, current functionality (supporting reactive charging, automatic billing and additional services) and has been adopted by seven out of the ten largest automobile manufacturers by production numbers [57]. In addition it is being integrated by competing manufacturers, such as Tesla [42].

2.1 Combined Charging System (CCS)

The Combined Charging System (CCS) is an amalgamation of standards governing all physical and logical elements of the charging infrastructure; from the physical connector to the protocols for automated billing. Figure 2a shows the charging plug, while Figure 2b illustrates the communications undertaken. The communication between vehicle and charger is standardised as ISO 15118. This uses powerline communication (PLC) over the Control Pilot (CP) and Protective Earth (PE) lines of the charging cable. The PLC shares the lines with the older IEC 61851 signalling system for backwards-compatibility reasons, with the signals superposed at the physical layer. The specific PLC implementation is HomePlug GreenPHY (HPGP) [5], a derivative of the commonplace broadband LAN technologies sold to consumers, that has been modified to support pairing between devices with no pre-shared key, and to be more robust to noise. Atop the PLC, ISO 15118 communication provides a full IP stack to act as the general-purpose channel. The same standard also defines interactions for identification, authorisation, tariff provision and control. Communication persists throughout the duration of charging and allows charge parameters to be varied quickly.

CCS provides reactive charging by allowing a charger to present current and future tariff information to the vehicle, which can then make charging requests based on a user's settings. The user may have a price preference or timing constraints for when the vehicle should be charged. Contract-based automated billing is implemented by having a user's contract with a charging provider represented by a public-key certificate stored on the vehicle. A complex public-key infrastructure (PKI) then allows the vehicle to authenticate the charger, the charger to validate the charging contract and the provider to produce verifiable metering receipts. The same PKI is used to underpin the TLS tunnel for protecting traffic.

[1] An open standard developed by Nissan and dominant in Japan
[2] A proprietary standard developed by Tesla Motors
[3] A nationwide standard in China
[4] An open standard backed by the European Union

Competing automated billing approaches do exist however, that do not use the contract-based approach, nor rely on the PKI. Blockchain-based payment systems, seeking to protect the user's privacy from charging operators, simply use the communication channel as a building block for their own service [6, 26]. A system named "AutoCharge" [58] is also used in some networks [33, 56] to enable automated billing for even those users whose vehicles do not support the required certificates. The AutoCharge system is based on a simplified ISO 15118 use-case [52] that uses only vehicle-provided identifiers to match the vehicle to a customer record at the provider.

As there is a general-purpose channel, any IP communication is supported for additional functionality. Fast internet access is suggested in the ISO 15118 standard and a selection of data collection, targeted marketing, on-demand entertainment and third-party app platforms are emerging to take advantage of this [6, 22].

2.2 CCS Security

Communication security is considered in many of the systems making up CCS standard; with traffic encryption available at the PHY layer, TLS at the Transport layer and XML Security at the Application layer [55, 48].

At the PHY layer, the HPGP PLC network maintains a shared secret key called the Network Membership Key (NMK), with ephemeral Network Encryption Keys (NEKs) rotated periodically. All MAC-layer traffic is encrypted via AES-128 using the NEK. However, HPGP security is based upon a private-network model, while EV charging is fundamentally a public-network model. To adapt the technology to the use case, additions were made to HPGP to incorporate an initial association protocol[5], during which the vehicle and charger verify that they are connected to each other and are not communicating with the wrong party due to crosstalk on their communication cable. The determination is known as Signal-Level Attenuation Characterisation (SLAC) and is illustrated in Figure 3. The protocol involves the vehicle sending a series of sounding messages, for which the charger reports the measured attenuation. If multiple chargers respond due to crosstalk, the one reporting the least attenuation is selected and communication commences. Once a charger is selected, a Network Membership Key is created by the charger and used to establish a private network. The key is then sent to the vehicle in the final `CM_SLAC_MATCH.CNF` message of the protocol. The SLAC protocol can operate in a secure mode, with mutual authentication and encrypted communication, but this capability is optional if supported by both parties. Indeed, despite the availability of this mechanism, the ISO 15118 standard specifies that SLAC only operates in its plaintext mode, leaving message security to TLS.

[5]The comprehensively-named GreenPPEA, or "GreenPHY Plug-in-electric-vehicle Electric-vehicle-supply-equipment Association"

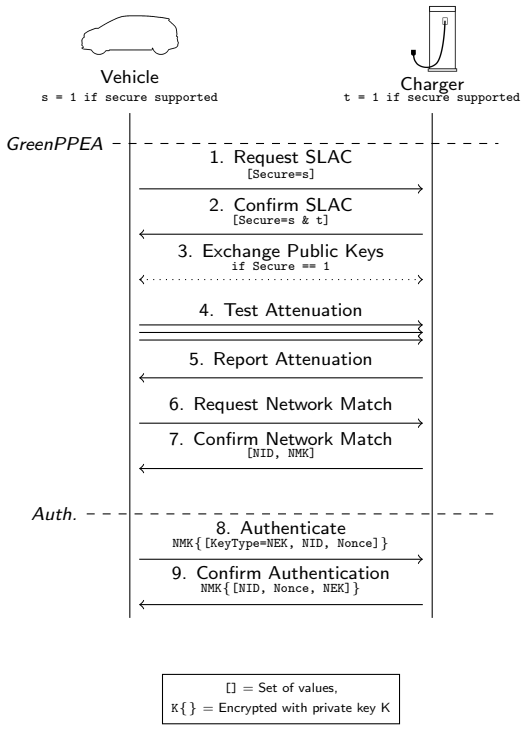


Figure 3: An overview of vehicle-to-charger network establishment in HomePlug GreenPHY. If the secure mode is supported by both parties and enabled in initialisation then step 3 occurs, allowing the messages in steps 5–7 to be signed and the one in step 7 also encrypted.

Once a network is established, a TLS connection is only created under certain conditions. If contract-based automated billing is used then TLS is required, similarly for the discovery of additional services, but only when they are ones defined in the ISO 15118 use cases. When charging is externally authorised, no TLS is required for the control traffic [48]. The options for external authorisation are open; including RFID cards, mobile-app networks, manual authorisation by a charger operator or some other service operating on the charger. The method need not be external to the charger, only external to the ISO 15118 scope. For all other traffic not managed through a standard use case, security is left to the implementer. In the alternative payment example of [26], independent IP communication is undertaken completely outside the scope of ISO 15118 (although secured in an SSH tunnel in that case).

3 Related Work

The privacy and security issues surrounding EV charging are the subject of ongoing work; with attempts to devise architectures that protect each stakeholder [32, 40] and analyses of the security of upcoming standards [4, 10, 8]. These works

are theoretical in nature, however, and leave aside implementation issues. They also assume a wireline threat model for attacks on the vehicle-to-charger communication, discussing where an attacker must use "a modified cable or an adapter plug installed on the [charger]" [8]. By contrast, we consider a wireless threat model that permits deniability on the part of the attacker.

Practical attacks have been demonstrated on previous-generation infrastructure, particularly against RFID authorisation [35, 19], but require the attacker to clone a user's physical token or access debug ports on an unlocked charger.

Since electromagnetic emissions security was brought from a military discipline into academic study by van Eck's work on eavesdropping video displays [68], efforts have been devoted to studying a wide range of systems [7]. Recent work has focused primarily on extracting secrets from operating devices [3, 13], although the emissions security of digital communication systems have been studied in the context of eavesdropping on RS232 serial devices [66] and 100BaseT ethernet [63], along with use as a covert channel for USB [39]. While radiated emissions from powerline communication have been studied from an electromagnetic compatibility perspective [71], we demonstrate the first practical wiretap attack using these emissions.

Vehicle tracking using unique identifiers has been studied in the context of electronic license plates [41], tire-pressure monitoring systems [46] and vehicular ad-hoc networks [49], highlighting the impact upon individuals' location privacy and inspiring this work on new charging technologies. Practical attacks have also been demonstrated to wirelessly compromise in-vehicle systems [17], to unlock vehicles for theft via remote keys [70] or passive entry [34, 37] and to misdirect drivers to unwanted locations [73]. These attacks consider an active attacker with different goals to those studied here and as such could be considered orthogonal to our work.

Energy monitoring has been shown to enable the tracking of individuals [54] and this has prompted proposals to mask energy signatures, such as by using rechargeable vehicle batteries [69], which assumes that data about vehicle power flow cannot be monitored.

4 A Near-Ideal Side-Channel

The underlying principles of electromagnetic (EM) side-channels are very well-explored and their study has informed modern security design [7]. Despite this, we describe here how the use of PLC and its specific arrangement within CCS exacerbates the vulnerability to EM attacks.

The design of PLC technologies assumes differential signalling; wherein two identical transmission lines that are located in close proximity are driven with equal but opposite signals, such that those fields largely cancel and no residual electric field exists. Practical challenges often break these underlying assumptions for in-home PLC deployments, leading to EM interference and susceptibility thereto [71]. Despite EV charging requiring simpler and more constrained wiring than domestic electrics, these assumptions are still broken in CCS. A design choice to incorporate backwards compatibility with an earlier low-power charging standard led to a PLC circuit design that connects one transmission line to ground (see Fig. 2b and App. A). This renders the signalling *single-ended instead of differential*. With no inverse field, the charging circuit functions as a suitable antenna for emissions or interference.

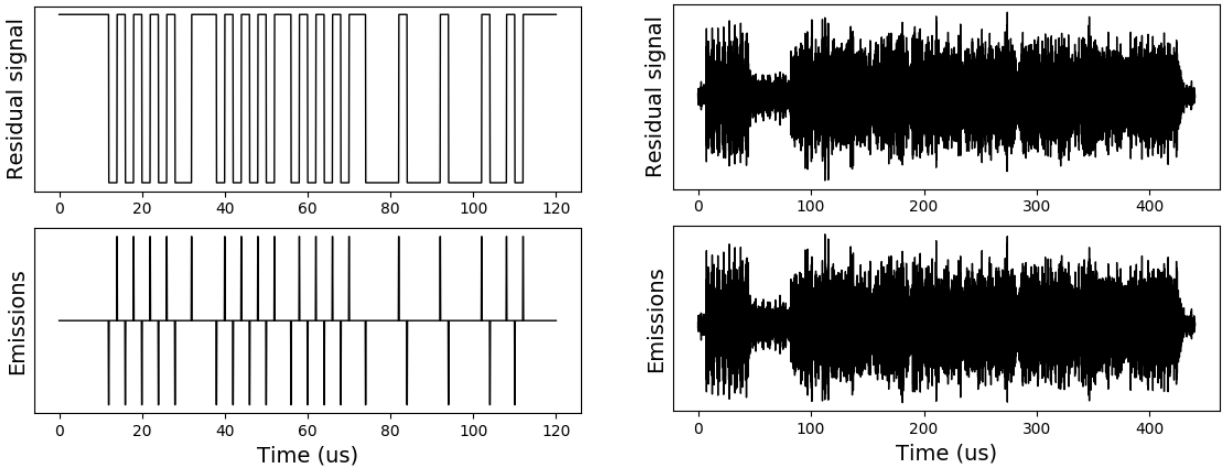
The nature of the PLC waveform itself, however, makes it ideal for wireless observation and interaction. It can be seen in Figure 4, operating as a single-ended system alongside single-ended CAN-Bus communications for comparison. The radiated signal represents the gradient of the original signal: only the changes in voltage. This introduces a minor problem for an attacker whenever they wish to observe and a major one when they wish to inject signals with constant voltage levels, most notably the square waves used ubiquitously in digital communication (and in other EV charging communication based on CAN-Bus). In observation the static voltage produces no response, so only state transitions are detectable. The attacker uses these where they can or hopes for the signal to leak elsewhere in the circuit and be modulated onto a more easily-observable one [7]. In injection the attacker cannot directly induce the desired static voltage level and instead must exploit nonlinearities in components or undersampling effects in order to synthesize the signal at the victim [51]. The absence of components to subvert, or the presence of filtering in the target circuit, limit the attacker's opportunities.

Broadband PLC technologies predominantly use orthogonal frequency division multiplexing (OFDM); in which the data are modulated in the frequency domain before constructing a time-domain waveform using an inverse Fourier transform. The resulting, transmitted waveform is a finite sum of sinusoids and does not exhibit any non-zero static voltage levels. *The observed emissions simply form a phase-shifted replica of the original signal.* The attacker therefore does not need to make inferences to determine the original signal from eavesdropped observations, nor predict what transformations an injected signal will undergo in the receiver. They need only contend with the characteristics of the channel itself.

5 Threat Model

While we discuss the channel properties in a bidirectional sense above, we focus our further investigation and practical attacks on passive eavesdropping. Testing on deployed infrastructure restricts us to only passive operation.

The attacker listens to the unintended electromagnetic radiation of the EV charging communication. Their goal is to eavesdrop on the general-purpose channel established be-



(a) CAN-Bus. (b) PLC.

Figure 4: Example single-ended signals, with the radiated emissions that result. As the emissions are the gradient of the signal, the square wave produces only impulses while the OFDM waveform is all but unchanged.

tween the vehicle and the charger; such that they obtain access to private data it carries. The attacker can approach close to the target vehicle and charger but cannot modify or interfere with the equipment. They perform their attack either *in-person* from a nearby location, or by *situating a device* at the site and leaving it unattended.

We justify this model on the basis of deniability and access. Interfering with a vehicle or charger is an immediately suspicious activity that would draw attention from the owner, people nearby and operators reviewing CCTV footage. The charging equipment is also handled regularly by drivers, so a cable modification or plug insert is more likely to be noticed. By contrast parking near another vehicle at a public station or briefly visiting a private property appear to be benign actions.

6 PLC Eavesdropping Tool

Given the properties described in Section 4, the passive attacker's task is the same as that of a legitimate receiver; to maximise the signal-to-noise ratio (SNR) and bandwidth (BW) of the received signal. In a real setting, additional complicating factors exist. While the exposed components are the most obvious targets, any element of the communication circuit (i.e., charging plug, cabling, vehicle, charger), or indeed multiple elements, could act as an unintentional antenna(s). The size of the equipment makes potential antennas physically distant from one another, so it can be difficult to predict the location that optimises the SNR and BW for each target. Similarly, electric vehicles and chargers are powerful electrical devices and even minor imperfections can introduce significant interference levels, which must be suitably mitigated by careful positioning or filtering.

Exploiting the properties and design choices of CCS, we developed a tool for wireless eavesdropping of the underlying physical layer; a HomePlug GreenPHY (HPGP) network. The tool is applicable to monitoring any HPGP network as well as network management traffic in HomePlug AV and AV2 networks, although the vehicle charging scenario is particularly beneficial for the reasons discussed above. The tool is available open-sourced under the MIT licence[6].

The eavesdropping tool broadly resembles a normal HPGP receiver. While the HPGP standard is public, all compatible implementations are proprietary and implemented as integrated circuits. Our pure-software implementation allowed far greater insight and flexibility during captures however, particularly for experimenting with different preprocessing steps to improve reception and collecting partial data that would be discarded by a black-box implementation. The receiver architecture can be seen in Figure 5. Given that Wi-Fi shares the same OFDM underpinnings, the overall structure bears many similarities to a Wi-Fi receiver, albeit distinct in details to match the HPGP protocol specification.

As the signal processing chain is complicated we describe it briefly here but elide full details from the main text, providing them in Appendix B instead. The signal is captured and digitally filtered to suppress local interference. Messages, known as PHY-layer Protocol Data Units (PPDUs), are identified using a power detector and correlation of the signal preamble against the known preamble structure. As an OFDM technology, data are represented in individual *symbols* throughout the Frame Control and Payload sections of

[6] `https://gitlab.com/rbaker/hpgp-emis-rx`

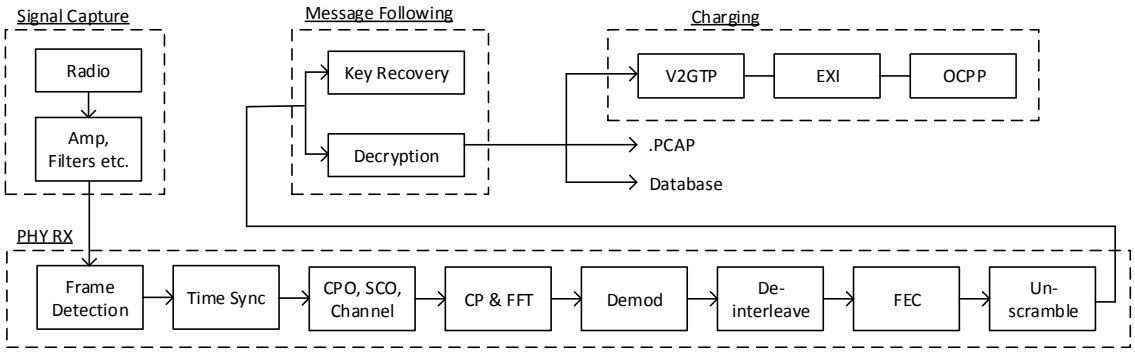


Figure 5: Architecture of PLC monitoring tool. The signal is captured and prefiltered, before moving through a software receiver chain to recover messages. The message following behaviour extracts security-relevant data and stores all messages. Charging traffic can be further processed, while traffic using other protocols will need separate onwards processing.

the PPDU. Once the receiver is time synchronised to the PPDU, each symbol is processed in turn; with channel estimation and frequency offset correction applied before demodulation. With complete messages the Turbo Code error correction is processed to reduce errors and the Cyclic-Redundancy Check checksums are calculated (a CRC24 for the Frame Control and a CRC32 for the Payload). The application of the Turbo Code decoder is limited in our tool, owing primarily to the computational cost of the process. A Turbo Code is intended to be decoded by iterating a probabilisitic decoder over various rearrangements of the received bits. We use only a single pass of the decoder and its application already dominates the message reception time; exceeding the rest of the software processing chain. As such we suffer from reduced error-correction performance compared with an arrangement using multiple repetitions. Such an arrangement could be expected to receive more messages correctly in all circumstances.

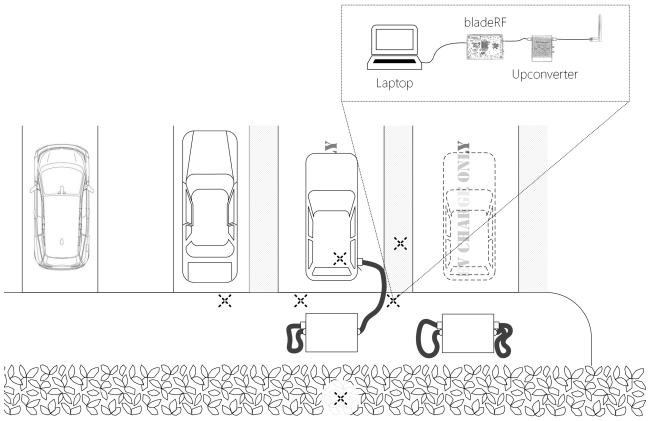


Figure 6: A composite diagram showing the experiment layout. The five antenna locations are denoted with a dashed $\times$ symbol.

7 Real-World Measurement Campaign

To explore the accessibility of the wireless side channel, we undertook a data collection campaign with three fully-electric vehicles: a BMW i3, a Jaguar I-PACE and a Volkswagen e-Golf. The campaign comprised over 800 miles of driving and spanned six major administrative regions of the UK. A total of 54 unique charging sessions were conducted, at locations including service stations, highway rest stops, superstores and hotels.

During charging sessions, we monitored radiated emissions to measure the extent of signal leakage and the ability of an attacker to eavesdrop it. Where we were able to receive sufficient emissions we used the tool detailed in Section 6 to recover the original transmissions and examine the communication itself. For the majority of our testing we monitored one vehicle at a time, although we did conduct testing with multiple vehicles to examine the effects of cross-traffic. Further details of the locations and installed hardware are given in Table 1, while examples can be seen in Figures 7 and 8. All of the chargers are state-of-the-art at the time of writing. We tested only public chargers due to their availability, but equivalent chargers for private use are also on sale [25]. As the chargers were public, we did not modify or interfere with the equipment in any way. The vehicle, charger and associated cabling remained entirely untouched. While this prevented us from injecting messages or capturing ground-truth via a directly-connected receiver, it was necessary to conduct a widespread survey of existing infrastructure.

At each site, the vehicle was parked and connected to the charger for a series of charging sessions[7]. The receiving an-

[7]Care was taken to ensure we only observed signals from our own vehi-

Site	Location	Type	Charger (Operator)	Vehicle			Charge Sessions
				i3	I-PACE	e-Golf	
A	Oxford Belfry, Oxon.	Hotel	DBT Dual DC [20] (Polar [14])	✓			1
B	Abingdon, Oxon.	Superstore	DBT Dual DC [20] (Polar [14])	✓			1
C	Maldon, Essex	Superstore	ABB Terra 53 CJG [1] (POD Point Open [59])	✓			1
D	South Mimms, Herts.	Road services	DBT Dual DC [20] (Ecotricity [23])	✓			1
E	Bishops Stortford, Herts.	Road services	DBT Dual DC [20] (Ecotricity [23])	✓			1
F	Hythe, Kent	Road services	DBT Dual DC [20] (Ecotricity [23])	✓	✓	✓	9
G	Dover, Kent	Superstore	ABB Terra 53 CJG [1] (POD Point Open [59])	✓			10
H	Marden, Kent	Local garage	Chargepoint CPE200 [43] (InstaVolt [44])	✓	✓	✓	15
I	Chatham, Kent	Racetrack	Chargemaster Ultracharge 500S [12] (Polar [14])			✓	1
J	Ticehurst, Kent	Golf club	Chargemaster Ultracharge 500S [12] (Polar [14])		✓	✓	4
K	Hawkhurst, Kent	Local garage	EVTronic QUICKCHARGER [31] (GeniePoint [15])		✓		2
L	Tunbridge Wells, Kent	Local garage	Efacec QC45 [24] (Shell Recharge [65])			✓	2
M	Hastings, Sussex	Local garage	EVTronic QUICKCHARGER [31] (GeniePoint [15])			✓	1
N	Milton Keynes, Bucks.	Public car park	Efacec QC45 [24] (Polar [14])			✓	5

Table 1: Details of all tested charging locations, across the southern United Kingdom. There were a total of 54 unique charging sessions. Multiple signal captures were taken during each session; at initialisation, during charging and at shutdown. At sites **F** and **H**, two vehicles were charged and monitored simultaneously.

tenna was placed at various locations to investigate the reception capabilities. As noted in Section 6, deriving an optimal attack location beforehand is challenging, so this placement was exploratory. The locations are illustrated with a dashed $\times$ symbol in Figure 6. Locations near the cable itself, on the outside of the vehicle, within the vehicle, hidden in a nearby hedge and on a nearby car were all tested. As each site had a different layout, Figure 6 is a composite to show the arrangements, rather than a meticulous depiction of any one site.

The data were collected using a bladeRF software-defined radio, an RF Explorer Upconverter and a GNU Radio flowgraph running on a Lenovo Thinkpad X1 Carbon laptop. We made use of an electrically-short monopole antenna to collect the signal. Owing to the long wavelengths involved, testing with a suitably-tuned directed antenna was not possible. The equipment for our experiments cost approximately $800, although equivalent setups are available for less than $300. The collected signal was passed through 25dB amplification and upconversion (+530MHz) to bring it into the tunable range of the bladeRF. Initial filtering and packet detection was performed with further GNURadio flowgraphs, while subsequent processing was implemented using Python and numPy libraries. We tuned the receiver's interference-rejection filter by observation at each site, but left all other reception parameters constant throughout.

8 Results

In this section we examine the results of our testing in real environments, both in terms of raw observable signal and message recovery.

cles. Upon arrival we waited for any other users to leave before capturing traffic and aborted immediately if another arrived.

8.1 Eavesdropped Communications

Table 2 details the observations for each site. It indicates the peak signal-to-noise ratio (SNR) over all the sessions, along with the widest bandwidth (BW) with a positive SNR. It then lists the count of all PPDUs detected, the number of data PPDUs, the rate at which messages were well-formed and the rate at which messages had a correct CRC32 checksum.

Every site displayed some form of unintentional wireless channel from the PLC communication, with properties that exceeded our expectations. The weakest signal showed 9dB from the peak to the background and spanned a bandwidth of 4.5MHz. In the best case 25MHz could be seen, up to a peak of 35dB. This was true irrespective of charger manufacturer, indeed varying notably between sites with the same charger hardware antenna location. This would seem to confirm the expectation that the site layout and variations in parking have a substantial impact upon reception.

Figure 9 shows spectrograms of the captured signal at a selection of sites, covering each tested antenna location. Overlaid on each subfigure is the utilised HPGP spectrum, showing the regions of the band in which transmission occurs. A transmission will originally have a frequency-domain representation that matches the spectral mask, with a peak power of -50dBm in utilised regions. Apparent power levels up to approximately -70dBm we observed, although the receiver was not calibrated against a reference scale so this value is uncertain. The degradation of signal across the band is clear in every case; the flat-topped spectral usage of the transmission is observable as a jagged range with many subcarriers severely attenuated, particularly at lower frequencies. This correlates well with studies of the wireline channel that legitimate receivers (with a conductive connection) experience, albeit with a different noise profile [53].

Site	Antenna	Peak SNR (dB)	BW (MHz)	Total PPDUs	Data PPDUs	Bi-direc.?	Start?	RX% Mean	CRC32% Min	CRC32% Mean	CRC32% Max
A	In car	15	6	526	272	✓		99.3	1.1	1.8	3.3
B	In car	18	12	1063	567	✓		29.8	0.5	3.3	5.3
C	In car	25	14	2976	1819	✓		99.9	46.6	48.1	50.3
D	In car	10	12	556	293	✓		88.2	1.4	2.3	3.0
E	In car	9	4.5	569	306			100	11.0	11.1	11.2
F	In car	21	12	3660	2009	✓	✓	99.3	27.8	36.8	45.8
	Bay behind	15	8	1434	1430	✓		99.3	**43.5**	43.5	**43.5**
	Outside car	10	10	12987	8255	✓		76.2	34.9	46.6	89.5
	Two cars	14	11	2449	2274			99.1	**24.3**	47.5	70.8
G	In car	19	12	5837	3670	✓	✓	99.0	51.1	60.3	71.4
	Next bay	15	13	4157	2749	✓		99.7	**91.8**	91.8	**91.8**
	By cable	29	23	23984	17246	✓	✓	80.2	52.9	74.0	**99.8**
H	In car	16	12.5	15052	9362	✓		99.2	69.9	71.0	72.8
	Outside car	20	11	16243	10407	✓		99.5	27.7	61.6	**80.6**
	By cable	35	25	19535	14717	✓	✓	92.1	**34.2**	70.0	92.8
	Two cars	15	12	24121	21006			99.6	42.2	71.9	**94.8**
I	In car	20	12	1501	1193	✓	✓	98.0	94.8	97.4	**100.0**
J	In car	20	7	14231	10291	✓	✓	81.0	1.0	33.6	67.9
	Outside car	23	7	1084	935	✓	✓	96.0	49.2	49.2	49.2
K	In car	8	5	1971	1278	✓		92.5	**0.0**†	22.0	38.3
L	Outside car	8	7	3004	1849		✓	25.8	**0.0**	0.0	0.0
M	In car	20	12	13631	9743	✓	✓	98.8	42.4	64.9	82.5
N	In car	24	14	4317	3364	✓	✓	68.3	**0.0**†	44.5	72.6

Table 2: Eavesdropping results, from all sites and antenna locations. Raw signal properties are quantified as Peak SNR and Bandwidth. PPDU counts are given and the observance of bidirectional traffic and session startup is indicated. The rates of well-formed messages are then shown, along with the rates of CRC32 checksum validations. The worst and best performance for each antenna location is highlighted in **bold** († indicates joint-worst).

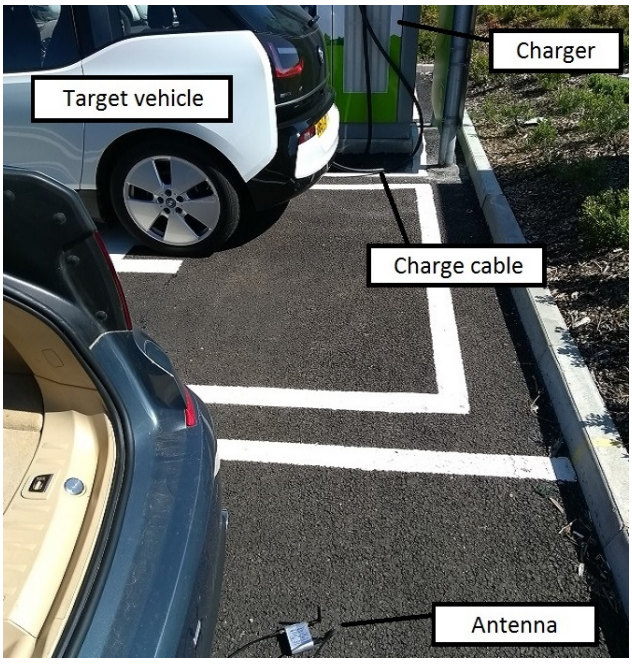
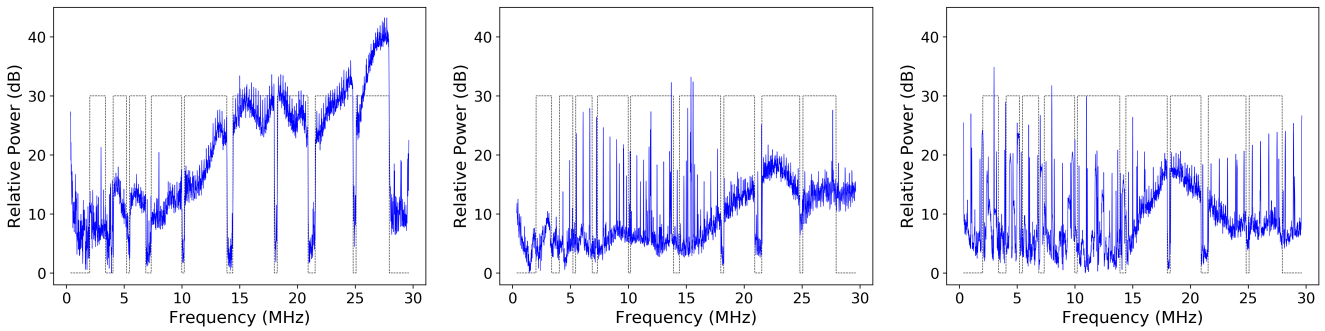


Figure 7: Eavesdropping from the next parking bay (site G), more than 4 metres away on the other side to the charging cable. In this arrangement 91.8% of messages were received successfully.



Figure 8: Two vehicles charging simultaneously. With the eavesdropper between the two vehicles 42.5% of messages were received successfully, including the NMK key establishment for both vehicles.



(a) Antenna by cable (site H)

(b) Antenna in bay behind (site F)

(c) Antenna in next bay (site G)

Figure 9: Observed signal across the HPGP bandwidth, at each antenna location. The HPGP spectral mask is overlaid to indicate the regions in which transmission occurs, although no valid comparison can be made with its power value as the measurement was not calibrated. Signal degradation and noise ingress is visible in every case, although far more prominently in (b) and (c).

8.2 Effects of Location

While systematic examination of performance by location was not our goal, we were able to observe trends across tested antenna positions, with the fidelity of the wireless channel varying substantially. The closest representation of the transmitted signal is that shown in Figure 9a, obtained approximately 0.5m from the charging cable. At other antenna positions the signal loss was more pronounced, both inside and outside of the vehicle, and in isolated cases the signal was swamped by interference more than a short distance from the cabling. Making general predictions about the channel gain at specific distances is not feasible due to the low frequencies at which the PLC operates (2 – 28MHz). Even at 28MHz the wavelength is still 10.7m and so all observations were taken well within the near field of the transmitter. In this region, common path loss calculations like the Friis equation [36] are not defined and near-field effects can change the channel gain drastically from position to position. Nevertheless, Figures 9b and 9c show the results of tests at the greatest distances; 4.2m in the latter case when the antenna was positioned by a vehicle in an adjacent parking bay (shown in Figure 7). Interference is still substantial at these distances (e.g., everything below 15MHz in Figs. 9b and 9c), but in the higher reaches of the band signal still easily visible.

The consistency of observed leakage across different charger hardware indicates that the issue is not isolated to a single implementation; supporting the claim that the design choices in CCS make a wireless side-channel for the PLC communication a systemic problem.

8.3 Message Recovery

With such a clear channel, message recovery proved highly successful, with hundreds of complete messages captured even in short sessions. In the best case, at site **I**, 100.0% of received messages had correct CRC32 checksums, more surprisingly 91.8% were still received when the antenna was located in the next parking bay. Reception rates were broadly correlated with raw SNR and BW, with improvements to either benefiting the performance. However this was not universal, as the very poor performance at sites **B** and **K** shows. Site **B** showed poor results despite far higher SNR and BW than Site **K**. Reception performance is broken down by location in Table 2, with the lowest minimum and highest maximum for each location highlighted in bold. Without ground-truth for the number of messages sent by each party, we cannot determine the number of messages missed entirely (only those received with errors), although the only unreported messages would be those that did not even trigger the packet detection algorithm (see Appendix B). Examining Frame Control headers showed that traffic was observed bidirectionally between vehicle and charger in all but two cases.

As charging stations, at least in public, are busy venues, we tested whether multiple simultaneous charging sessions caused interference that affected the wireless channel quality. Two vehicles (a Jaguar I-PACE and a VW e-Golf) charged simultaneously in 5 charging sessions at 2 locations, one of which is shown in Figure 8. In each case, one vehicle initiated charging first and then the second did so. The eavesdropper's antenna was located between the two vehicles and attempted to listen to both. In all cases, the eavesdropper was able to listen to traffic from both vehicles, albeit with varying success. At worst, 24.3% of messages were received with correct CRC32, at best 94.8% (mean 59.7%).

9 Security Analysis

In this section we analyse the captured communications and their security implications.

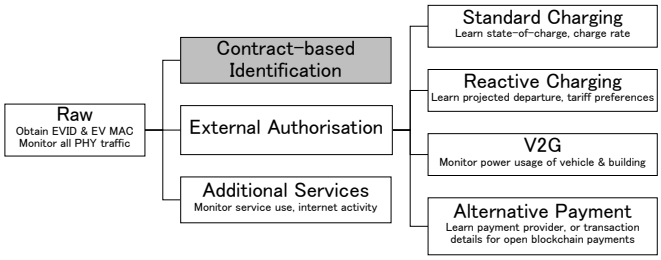


Figure 10: Tree diagram indicating the potential data available under a range of communication scenarios.

9.1 Unencrypted Communications

Where our testing campaign captured the initialisation of a charging session, we were able to examine the NMK exchange to form a network. In line with the ISO 15118 standard, every SLAC interaction we observed operated in insecure form. As such, the NMK was delivered in plaintext and the only barrier to acquiring it was receiving the message intact. We were able to intercept the `CM_SLAC_MATCH.CNF` message in 31 cases and acquire the NMK. Testing two vehicles side-by-side, in 4 sessions the attacker was able to extract an NMK value for one vehicle, meanwhile in one session both NMK values were extracted. In 9 cases, the subsequent `CM_GET_KEY.CNF` message was also recovered to obtain the ephemeral NEK and permit *passive decryption of physical-layer traffic*.

Examining compromised sessions, we saw the expected behaviour as the vehicle and charger established a network, the vehicle undertook the discovery protocol to find a charge controller and the two established a TCP connection. *No TLS tunnel was established in any charging session we observed*, leaving the high-level protocols exposed. Where external authorisation is employed, as it was in our testing, the use of TLS is optional under ISO 15118. Yet its complete absence from any vehicle or charger came as a surprise, especially given the charging locations were all public.

As a result we confirm that a passive attacker can wirelessly monitor all traffic at the PHY layer and that this ability *results from standards-compliant behaviour*, suggesting it is persistent. Likewise, the option to forego TLS means charging data is also left in the clear. We discuss this situation and its implications in Section 10.

9.2 Private Data

Figure 10 provides a breakdown of potential data available when eavesdropping, under various charging conditions or in the presence of different services. The PHY-layer traffic is always available and permits access to any higher-level communication, such as charging or internet access, that does not take additional steps to secure itself. Two unique identifiers for the vehicle are also available: its EV ID and its MAC address. These identifiers are persistent for the entire lifetime of the vehicle, including between owners, and are globally unique. They have been noted as personal data in previous privacy studies [40] and are covered by the European Union's GDPR as data that can be easily combined with other sources to identify an individual.

With contract-based billing, we do not expect charging traffic to be available, as TLS is always required in this case. However as we have seen, when it is optional to omit TLS, this has consistently been done. Currently, this leaves the majority of charging traffic in the clear at public locations, although these are likely to be the earliest adopters of contract-based billing (or some alternative). The long-term omission of TLS at private locations is of greater concern. Indeed it is in this case that there is more potential for behavioural profiling, due to the vehicle staying far longer at the user's home or workplace and with the emerging Reactive Charging and V2G systems far more beneficial to them there. The introduction of 'Vehicle-to-Home' capabilities, for instance, is prioritised for introduction as early as 2020 by the CCS standards body [16]. Resulting indicators of the user's day-to-day behaviour such as the vehicle's state-of-charge and projected departure time are contained within normal charging traffic, while reverse power flow data in a V2G system yields insights into the power usage of the building.

In addition to internet access for in-vehicle entertainment systems, third-party apps and alternative payment networks, the traffic of any local services would also be available at public locations, as would smart home integration traffic in private ones.

9.3 Charging Attacks

A reliable eavesdropping capability presents a range of opportunities for an attacker, both immediate and longer-term in their impact. We consider here a selection of potential attacks using these techniques. Although we did not perform the attacks against public chargers, we describe how they would be conducted.

AutoCharge Extant AutoCharge systems, such as one operating in production across a 60-location network in three European countries [33] are at particular risk from wireless eavesdropping. The use of the vehicle's charge-controller MAC address for billing identification [58, 56], while highly questionable from a purely-security standpoint, was undertaken for compatibility and convenience benefits (and has been lauded as such by customers). What may be an acceptable trade-off when physical interference is required to extract the values, is far less so when this can be done from another vehicle without any observable signs. The identifiers of the vehicles are shown partially-masked below (none is a customer of an AutoCharge system):

Vehicle	MAC
BMW i3	f0:7f:0c:02:●●:●●
VW e-Golf	00:7d:fa:01:●●:●●
Jaguar I-PACE	00:1a:37:70:●●:●●

We were able to obtain the identifiers in 41 cases (76%)[8] from a variety of locations including the two-car arrangement shown in Figure 8. Here the identifiers for both vehicles were acquired from the same antenna position, suggesting that an attacker could simply park next to a charging station and collect identifiers as other users arrive subsequently providing them[9] in order to obtain free charging on another user's account. As the charging spots are operated by a single provider, the attacker can be confident of targeting valid customers.

User Tracking In the simplest attack, charging sessions are linked by monitoring a number of busy public chargers for the appearance of vehicle identifiers. From time-of-day, charge duration and location information, behavioural profiles can be inferred. The invasiveness of the attack increases where the attacker is able to match a vehicle identity to other data. Popular charger-sharing schemes [60] allow anyone to register their home or business charger as a public site; any user booking to charge can then be associated with their vehicle identifier and tracked at any monitored station. Monitoring a charger near a sensitive event such as a union meeting, protest gathering or compromising night-spot would reveal more personal information about an individual's habits.

With a wireless attack, a wardriving approach also allows an attacker to associate a vehicle with a street address. This could easily be conducted by a delivery driver or postal worker as they visit properties regularly. Known MAC allocations to manufacturers provide a coarse-grained indication of the vehicle as well, such as identifying expensive vehicles and then determining when they have been left in a public car park, or indeed when their owner is out of the house.

10 Lessons Learnt

The refinement of EV charging systems is still ongoing. In light of our observations, we have distilled a set of security lessons that can improve existing and future designs.

10.1 Wireless Threats

The most notable finding here is that the design of CCS communication allows a wireless attacker to observe it at a distance without prior interaction or tampering. In this case the attack was entirely passive, but has similar implications for the potential of active attacks that would currently be far more invasive. As in-vehicle wireless systems have been plagued by attacks in recent years, our results indicate that a testing model which considers emissions security as well as unwanted interference is crucial in future development.

10.2 Reliance on a Non-Existent PKI

The ISO 15118 security model, and thus that of CCS, relies on the existence of a complex PKI, to underpin TLS at the Transport Layer and XML Security for external message values at the Application Layer. The merits of that infrastructure are an ongoing topic of academic study [8, 72, 52], but its complexity also presents a more practical problem. At the time of writing, no widespread ISO 15118 PKI is deployed. While small-scale pilots have been attempted, there is still open debate about provision of the infrastructure and the authors are aware of public proposals from three different commercial entities to provide transaction brokerage and act as the Root Certification Authority [27]. There is even disagreement about the model the PKI will take; whether it will derive from a single root of trust, a consortium of trusted entities or some more open model [50]. Meanwhile the competing pressures to provide new functionality remain, spurring alternative solutions such as AutoCharge and encouraging service development without underlying security provision.

Even once a PKI is operating for public chargers in large charging networks, it remains unclear to what extent private units in individual homes or offices will benefit. A capacity for self-signed contract certificates to be manually installed into vehicles by users does exist, but unless contract-based billing is used ISO 15118 exempts charging installations from any security requirements; instead relying on the physical security of the location and cabling — which we have demonstrated to be insufficient. Manufacturer choices (and indeed user willingness) will determine whether private chargers can enjoy these security benefits.

It is important therefore to provide at least some security implementation that is decoupled from the need for access to a PKI. We discuss such an approach in Section 11.

10.3 Available PHY Security Disabled

The HomePlug GreenPHY (HPGP) PLC technology supports a *Secure SLAC* mode that protects the pairing and NMK distribution process, but this is disabled by specification in the ISO 15118 standard, relying instead on TLS for all security properties. While this can meet the charging use cases outlined in that standard, it leaves an opportunity for a pervasive security baseline completely ignored, despite proposing the communication channel for general use. All too of-

[8] 31 cases from SLAC initialisation messages and 10 more from network management messages

[9] Typically updating the MAC setting using `open-plc-utils` [45] and a serial debug port over UART or SPI [21]

ten history has shown that leaving security to individual developers atop insecure platforms produces widespread security problems, even more so when the channel is considered physically private.

11 Countermeasures

To mitigate the unintended wireless channel, familiar emissions security mechanisms such as chokes or shielding can be applied to reduce leakage [7], although hardware modifications for existing systems are costly and time-consuming. Some proposals for future, high-power chargers include liquid-cooled charging cables and we would expect this to attenuate the signals if the cooling jacket wraps the communication lines as well as the power-delivery ones. This would not eliminate emissions from the vehicle or charger circuitry however, nor is it likely to exist in smaller, private chargers.

At a network level, we have argued for the use of the available HPGP security mechanisms above, but note that in their present form they are still reliant on a PKI to function. In addition the HPGP key distribution behaviour itself introduces an unnecessary risk of interception. Whether the SLAC protocol operates in its secure mode or not, it is still unilateral: the charger generates a network key and then provides it to the vehicle. However, the SLAC process is typically implemented in software by the same devices that undertake the higher-level ISO 15118 communication, including possible TLS sessions, and as such require the capabilities for an Elliptic Curve Diffie-Hellman key derivation for AES128 [48].

We propose additional steps in the SLAC initialisation, as a fallback to provide confidentiality from the MAC-layer upwards in the event that PKI access is unavailable. Figure 11 shows the modified protocol. Upon receiving a network match request, the charger generates an Elliptic-Curve keypair (d_C, Q_C) and instructs the vehicle to commence a key exchange, along with Q_C. If the vehicle also supports the protocol then it generates (d_V, Q_V) and responds with Q_V. The derived key becomes the new NMK and the charger blanks the NMK field in the subsequent `CM_SLAC_MATCH.CNF` message. If the vehicle does not support the protocol then the unrecognised message will be dropped. The charger maintains a timeout counter after step 6.1 and, upon expiry, falls back to the existing protocol's step 7.

While it cannot provide authentication and therefore cannot mitigate man-in-the-middle attacks, the threat of passive eavesdropping is eliminated using this approach. By building only on existing functionality, the protocol is deployable in existing vehicles as well as new ones.

12 Conclusion

We have demonstrated that use of PLC in EV charging and the design of the CCS standard lead to a uniquely high-

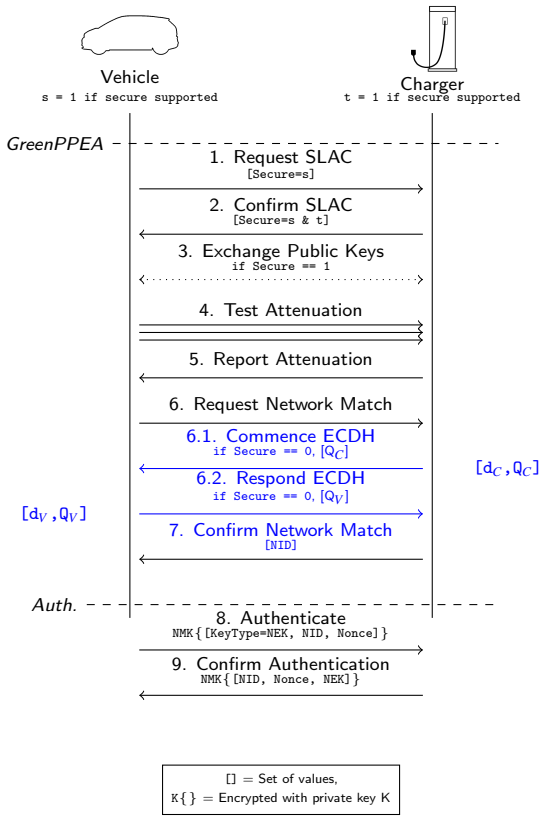


Figure 11: The modified SLAC network establishment. Steps 6.1 and 6.2 are new, while step 7 has been modified.

quality, unintentional wireless channel. We have evaluated the susceptibility of real-world chargers and found a reliable channel in every case. Although conditions vary substantially between sites, for eavesdropping we achieved a peak successful recovery rate of 100% in one case and could intercept traffic several metres from the target, in a different parking bay, with a rate of 91.8%. We showed how a series of further design choices allow recovery of network keys and passive monitoring of all traffic in plaintext. We presented lessons learnt and potential improvements to mitigate the problems so that they do not hinder the secure adoption of global EV charging infrastructure by the growing number of EV owners worldwide.

Acknowledgements

Richard Baker was supported by the EPSRC.

Disclosure Statement

We disclosed our findings to the tested vehicle and charger manufacturers, along with AutoCharge operators.

References

[1] ABB. Terra 53 Product Leaflet, 2017.

[2] International Energy Agency. Global EV Outlook 2018, 2018.

[3] Monjur Alam, Haider Adnan Khan, Moumita Dey, Nishith Sinha, Robert Callan, Alenka Zajic, and Milos Prvulovic. One&done: A single-decryption em-based attack on openssls constant-time blinded RSA. In *27th USENIX Security Symposium*, pages 585–602, 2018.

[4] Cristina Alcaraz, Javier Lopez, and Stephen Wolthusen. Ocpp protocol: Security threats and challenges. *IEEE Transactions on Smart Grid*, 8(5):2452–2459, 2017.

[5] HomePlug Powerline Alliance. HomePlug Green PHY Specification. *HomePlug, June*, 2010.

[6] AMO Labs. Amo labs preparing to enter the european market with gridwiz!, 2018.

[7] Ross Anderson. *Security engineering*. John Wiley & Sons, 2008.

[8] Kaibin Bao, Hristo Valev, Manuela Wagner, and Hartmut Schmeck. A threat analysis of the vehicle-to-grid charging protocol iso 15118. *Computer Science-Research and Development*, 33(1-2):3–12, 2018.

[9] BBC. Petrol and diesel ban: How will it work?, 2017. `https://www.bbc.co.uk/news/uk-40726868`.

[10] Cesar Bernardini, Muhammad Rizwan Asghar, and Bruno Crispo. Security and privacy in vehicular communications: Challenges and opportunities. *Vehicular Communications*, 2017.

[11] Bastian Bloessl, Michele Segata, Christoph Sommer, and Falko Dressler. An ieee 802.11 a/g/p ofdm receiver for gnu radio. In *Proceedings of the second workshop on Software radio implementation forum*, pages 9–16. ACM, 2013.

[12] BP Chargemaster. Chargemaster Ultracharge 500S Datasheet, 2019.

[13] Giovanni Camurati, Sebastian Poeplau, Marius Muench, Tom Hayes, and Aurélien Francillon. Screaming channels: When electromagnetic side channels meet radio transceivers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 163–177. ACM, 2018.

[14] Chargemaster Ltd. Polar network, 2018. `https://chargemasterplc.com/polar/`.

[15] Chargepoint Services. Geniepoint, 2019. `https://www.chargepointservices.co.uk`.

[16] CharIn. Target grid integration levels, 2019. `https://insideevs.com/ccs-combo-standard-v2g-2025/`.

[17] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, Tadayoshi Kohno, et al. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security Symposium*, volume 4, pages 447–462. San Francisco, 2011.

[18] Mark Chediak. Electrify america plans $200 million for california clean cars, 2019. `https://www.bloomberg.com/news/articles/2018-10-03/electrify-america-plans-200-million-for-california-clean-cars`.

[19] Matthias Dalheimer. Ladeinfrastruktur fr elektroautos: Ausbau statt sicherheit, 2017. `https://media.ccc.de/v/34c3-9092-ladeinfrastruktur_fur_elektroautos_ausbau_statt_sicherheit`.

[20] DBT. Quick Charger Dual DC Product Datasheet, 2014.

[21] devolo AG. dLAN Embedded PLC Module Datasheet, 2012. `https://www.codico.com/shop/media/datasheets/Devolo_dLAN_Green_PHY_Module_20130713_en_data_sheet_019.pdf`.

[22] EcoG. Providing a customized electric vehicle (ev) fast charging experience through a paas for value added services & shared revenue streams, 2019.

[23] Ecotricity. Electric highway, 2018. `https://www.ecotricity.co.uk/for-the-road`.

[24] Efacec. Efacec QC45 Datasheet, 2016.

[25] Efacec. QC45S Product Page, 2019. `https://electricmobility.efacec.com/ev-qc24s-quick-charger/`.

[26] ElaadNL. Iota charging station, 2018.

[27] ElaadNL. Update Global EV Charging Test: PKI Workshop, 2018.

[28] Engadget. California bill would ban new fossil fuel vehicles from 2040, 2018. `https://www.engadget.com/2018/01/04/california-bill-would-ban-new-fossil-fuel-vehicles-from-2040/`.

[29] European Alternative Fuels Observatory. Electric vehicle charging infrastructure, 2018.

[30] CharIN e.V. What is the combined charging system?, 2018. https://www.charinev.org/ccs-at-a-glance/what-is-the-ccs/.

[31] EVTRONIC. Quickcharger product datasheet, 2016.

[32] Rainer Falk and Steffen Fries. Electric vehicle charging infrastructure security considerations and approaches. *Proc. of INTERNET*, pages 58–64, 2012.

[33] Fastned. Autocharge, 2019. https://support.fastned.nl/hc/en-gb/articles/115012747127-Autocharge-.

[34] Aurélien Francillon, Boris Danev, and Srdjan Capkun. Relay attacks on passive keyless entry and start systems in modern cars. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2011.

[35] Achim Friedland. Security and privacy in the current e-mobility charging infrastructure, 2016. https://blog.deepsec.net/deepsec2016-talk-security-privacy-current-e-mobility-charging-infrastructure-achim-friedland/.

[36] Harald T Friis. A note on a simple transmission formula. *proc. IRE*, 34(5):254–256, 1946.

[37] Flavio D Garcia, David Oswald, Timo Kasper, and Pierre Pavlidès. Lock it and still lose iton the (in) security of automotive remote keyless entry systems. In *25th USENIX Security Symposium*, 2016.

[38] Jonathan M. Gitlin. Electrify america will deploy 2,000 350kw fast chargers by the end of 2019, 2018. https://arstechnica.com/cars/2018/04/electrify-america-will-deploy-2000-350kw-fast-chargers-by-the-end-of-2019/.

[39] Mordechai Guri, Matan Monitz, and Yuval Elovici. Usbee: Air-gap covert-channel via electromagnetic emission from usb. In *Privacy, Security and Trust (PST), 2016 14th Annual Conference on*, pages 264–268. IEEE, 2016.

[40] Christina Höfer, Jonathan Petit, Robert Schmidt, and Frank Kargl. Popcorn: Privacy-preserving charging for emobility. In *Proceedings of the 2013 ACM workshop on Security, privacy & dependability for cyber vehicles*, pages 37–48. ACM, 2013.

[41] Jean-Pierre Hubaux, Srdjan Capkun, and Jun Luo. The security and privacy of smart vehicles. *IEEE Security & Privacy*, (3):49–55, 2004.

[42] InsideEVs. Tesla model 3 with ccs combo inlet, s & x with ccs adaptor in europe, 2019. https://insideevs.com/tesla-model-3-ccs-combo-s-x-adaptor/.

[43] InstaVolt. Our technology, 2018. https://instavolt.co.uk/about-us/our-technology/.

[44] InstaVolt Ltd. About InstaVolt, 2018. https://instavolt.co.uk/.

[45] INSYS MICROELECTRONICS GmbH. INSYS Powerline GP Manual, 2017. https://256.insys-icom.com/bausteine.net/f/10637/HB_en_INSYS_Powerline_GP_1711.pdf?fd=0.

[46] Rob Millerb Ishtiaq Roufa, Hossen Mustafaa, Sangho Ohb Travis Taylora, Wenyuan Xua, Marco Gruteserb, Wade Trappeb, and Ivan Seskarb. Security and privacy vulnerabilities of in-car wireless networks: A tire pressure monitoring system case study. In *19th USENIX Security Symposium, Washington DC*, pages 11–13, 2010.

[47] Road vehicles Vehicle to grid communication interface Part 1: General information and use-case definition. Standard, International Organization for Standardization, Geneva, CH, 2013.

[48] Road vehicles Vehicle to grid communication interface Part 2: Network and application protocol requirements. Standard, International Organization for Standardization, Geneva, CH, 2014.

[49] Mohammad Khodaei, Hongyu Jin, and Panagiotis Papadimitratos. Secmace: Scalable and robust identity and credential management infrastructure in vehicular communication systems. *IEEE Transactions on Intelligent Transportation Systems*, 19(5):1430–1444, 2018.

[50] Paul Klapwijk and Lonneke Driessen-Mutters. Exploring the public key infrastructure for iso 15118 in the ev charging ecosystem, 2018.

[51] Denis Foo Kune, John Backes, Shane S Clark, Daniel Kramer, Matthew Reynolds, Kevin Fu, Yongdae Kim, and Wenyuan Xu. Ghost talk: Mitigating emi signal injection attacks against analog sensors. In *2013 IEEE Symposium on Security and Privacy*, pages 145–159. IEEE, 2013.

[52] Seokcheol Lee, Yongmin Park, Hyunwoo Lim, and Taeshik Shon. Study on analysis of security vulnerabilities and countermeasures in iso/iec 15118 based electric vehicle charging technology. In *IT Convergence and Security (ICITCS), 2014 International Conference on*, pages 1–4. IEEE, 2014.

[53] Michael Himmels. Devolo real world field tests, 2011. http://www.homeplug.org/media/filer_public/25/4f/254f6adb-096a-4913-842b-91e3775da045/devolo_presentation.pdf.

[54] Andrés Molina-Markham, Prashant Shenoy, Kevin Fu, Emmanuel Cecchet, and David Irwin. Private memoirs of a smart meter. In *Proceedings of the 2nd ACM workshop on embedded sensing systems for energy-efficiency in building*, pages 61–66. ACM, 2010.

[55] Marc Mültin. *Das Elektrofahrzeug als flexibler Verbraucher und Energiespeicher im Smart Home*. PhD thesis, KIT-Bibliothek, 2014.

[56] Open Fast Charging Alliance. Automatic charging start and authorization of electric vehicles, 2017.

[57] Organisation Internationale des Constructeurs d'Automobiles. World Motor Vehicle Production: World Ranking of Manufacturers, 2016.

[58] Johan Peeters. Fast charging just got faster. Presentation at eMove360 Conference 2017.

[59] POD Point. Open charge electric car charging stations, 2018. `https://pod-point.com/open-charge`.

[60] Recargo Inc. PlugShare, 2018. `https://www.plugshare.com/`.

[61] Reuters. Paris plans to banish all but electric cars by 2030, 2017. `https://www.reuters.com/article/us-france-paris-autos/paris-plans-to-banish-all-but-electric-cars-by-2030-idUSKBN1CH0SI`.

[62] Zeinab Rezvani, Johan Jansson, and Jan Bodin. Advances in consumer electric vehicle adoption research: A review and research agenda. *Transportation research part D: transport and environment*, 34:122–136, 2015.

[63] Matthias Schulz, Patrick Klapper, Matthias Hollick, Erik Tews, and Stefan Katzenbeisser. Trust the wire, they always told me!: On practical non-destructive wire-tap attacks against ethernet. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 43–48. ACM, 2016.

[64] Share&Charge. Share&charge, 2019. `https://shareandcharge.com/`.

[65] Shell Plc. Welcome to shell recharge, 2019. `https://www.shell.co.uk/motorist/welcome-to-shell-recharge.html`.

[66] Peter Smulders. The threat of information theft by reception of electromagnetic radiation from rs-232 cables. *Computers & Security*, 9(1):53–58, 1990.

[67] U.S. Department of Energy Alternative Fuels Data Centre. Alternative fueling stations, 2018. `https://www.afdc.energy.gov/stations/`.

[68] Wim Van Eck. Electromagnetic radiation from video display units: An eavesdropping risk? *Computers & Security*, 4(4):269–286, 1985.

[69] David Varodayan and Ashish Khisti. Smart meter privacy using a rechargeable battery: Minimizing the rate of information leakage. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 1932–1935. IEEE, 2011.

[70] Roel Verdult, Flavio D Garcia, and Josep Balasch. Gone in 360 seconds: Hijacking with hitag2. In *21st USENIX Security Symposium*, pages 237–252, 2012.

[71] Brad Zarikoff and David Malone. Experiments with radiated interference from in-home power line communication networks. In *Communications (ICC), 2012 IEEE International Conference on*, pages 3414–3418. IEEE, 2012.

[72] Daniel Zelle, Markus Springer, Maria Zhdanova, and Christoph Krauß. Anonymous charging and billing of electric vehicles. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, page 22. ACM, 2018.

[73] Kexiong (Curtis) Zeng, Shinan Liu, Yuanchao Shu, Dong Wang, Haoyu Li, Yanzhi Dou, Gang Wang, and Yaling Yang. All your GPS are belong to us: Towards stealthy manipulation of road navigation systems. In *27th USENIX Security Symposium*, pages 1527–1544, Baltimore, MD, 2018. USENIX Association.

[74] ZF Car eWallet GmbH. Car ewallet, 2019. `https://car-ewallet.de/index.php/what-we-do/`.

Appendices

A CCS Circuit Design

Figure 12 shows the communication circuit for PLC in CCS charging systems, including the connection of the circuit to the Control Pilot and Protective Earth lines, along with the additional components affecting the Control Pilot line due to the need for backwards-compatibility with the IEC 61851 communication that shares the lines.

B HomePlug GreenPHY Receiver

In this section we describe our eavesdropping tool in detail. As noted in Section 6, the tool is effectively a modified receiver design, although newly-implemented entirely in software. Since HomePlug GreenPHY (HPGP) [5] is an orthogonal frequency-division multiplexing (OFDM) technology, many elements of the tool structure are similar to a Wi-Fi receiver.

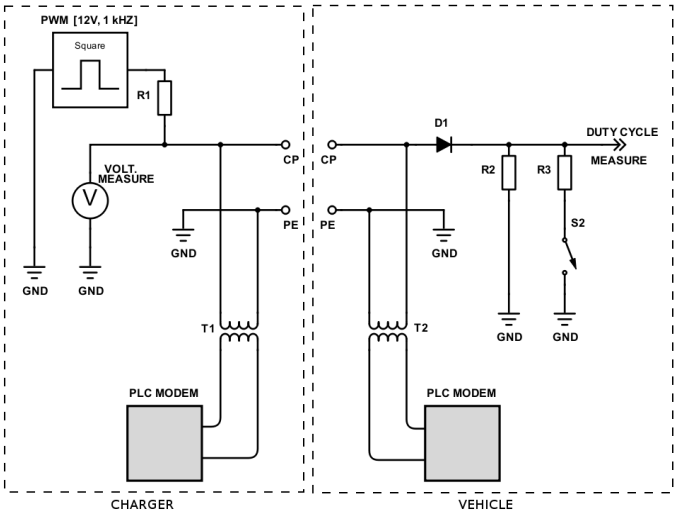


Figure 12: A diagram of the CCS communication circuit. The loads on each line connected to the PLC modem are not balanced. Resistors R2 & R3 alter the voltage in the low-level communication, but also vary the imbalance further.

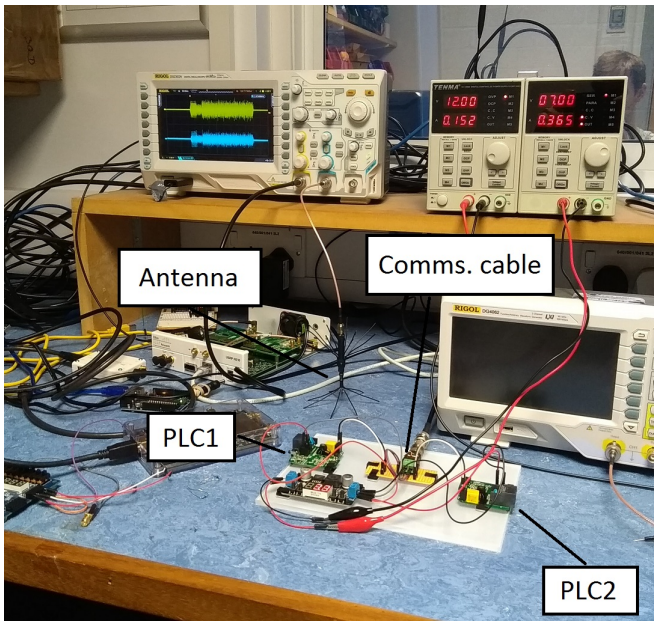


Figure 13: HomePlug AV adaptors communicating across a short wire. Conducted signals and radiated emissions can be seen on the oscilloscope (top in yellow and bottom in blue, respectively).

Raw signals are first collected using a suitable capture device. A Rigol DSA-2302A oscilloscope was used in our testbed arrangement, as can be seen in Figure 13. Even here the radiated emissions were easily observed; the yellow line in the figure represents the conducted signal, while the blue line is the radiated signal received by a short random-wire antenna. Although the distance shown here is very short, we were still able to observe the signal from the other side of the lab, several metres away. We later employed software-defined radios for signal capture, for their ability to receive and stream a captured signal in real time.

The captured signal is filtered in the frequency domain, benefiting from knowledge of the active regions of the HPGP band and the ability to survey initially an individual site's leakage before eavesdropping in earnest. A sharp-edged digital filter is used to remove regions with notable interference ingress or where channel gain is so low as to provide no useful information. The signal is then resampled into the HPGP native timebase of 75MHz.

Frame Detection and Time Alignment With the signal suitably pre-filtered and digitised, the PPDUs are detected using a *Double Sliding-Window* power detector; a design that accurately identifies the rise in power that accompanies the start of a packet. The detector calculates the power of the incoming signal and maintains two windows A and B of equal length L that are arranged with a time lag such that calculated power levels are included in window A at time t, subsequently passing out of window A and into window B at time $t+L$ and out of the detector entirely at time $t+2L$. At each sample, the power in each window is updated and the total power in A is divided by that in B. This configuration causes the output signal to spike quickly on increases in power levels, while remaining stable at equal power levels (i.e., prior to or during a frame). By selecting an appropriate value of L (based on the frame's structure), transient noise can be prevented from triggering a frame.

OFDM requires precise time synchronisation in order to demodulate correctly. We performed this by correlating the entire preamble against a template, which provided sample-accurate alignment.

CPO, SCO & Channel Estimation In practice, a transmitter and receiver in an OFDM system will have neither precisely-aligned oscillators nor synchronised sample clocks, leading to Carrier Phase Offset (CPO) and Sampling Clock Offset (SCO). CPO causes an apparent frequency offset for the entire received signal, meaning that the frequency-domain representation exhibits a phase rotation. Meanwhile, SCO leads to an apparent phase drift across subcarriers in the frequency domain. Both phenomena hamper demodulation and must be corrected beforehand. Channel estimation is also crucial to successful reception; assessing the gain and phase alterations that have been experienced by the signal due to the propagation environment.

Our receiver estimates the CPO using a method derived from Bloessel et. al.'s work; estimating the CPO using the seven full-amplitude SYNCP preamble symbols in place of the Wi-Fi short-training sequence [11] (omitting the initial

192 as they have been windowed in symbol shaping):

$$cpo_{est} = \frac{1}{384} Arg\left(\sum_{i=0}^{7\cdot 384} x[i]\bar{x}[i+384]\right)$$

where x is the received signal samples.

From the extracted section of the preamble, complex samples are multiplied with the conjugate of the same sample in the next SYNCP block. This produces an estimate of the phase progression introduced between those SYNCP blocks by the mismatch between transmitter and receiver (plus noise). Dividing through by the length of the SYNCP block gives an estimate of the phase offset per sample. The length of the sequence (2688) and the number of repetitions (7) permit an accurate CPO estimate. The per-sample CPO estimate can then be used to correct the remainder of the captured signal.

$$x[i] \leftarrow x[i] \cdot e^{-jcpo_{est}i}$$

As the estimated CPO will not precisely match the actual CPO, ongoing correction is applied to each received symbol by estimating the CPO between the cyclic prefix and the symbol tail, with a suitable correction being applied over that symbol.

$$cpo_{estcp} = \frac{1}{3072} Arg\left(\sum_{i=0}^{GI} x[i]\bar{x}[i+3072]\right)$$

where GI is the guard interval (with four values depending on the symbol and system settings).

The channel estimation is performed in the frequency domain, by comparing the received preamble symbols to a locally-computed template. HPGP provides no pilot symbols so all estimation must be performed from the preamble and maintained across the PPDU. The results for each preamble symbol are averaged and a channel estimate from the active preamble subcarriers computed. From this a channel estimate for the full channel is derived by interpolation, while the SCO is estimated from the slope of the phase differences in the channel estimate. As the CPO and SCO are due to hardware imperfections in the transmitter and receiver, rather than channel properties, estimates are maintained between received PPDUs by way of a moving average. The channel estimate, by contrast, is discarded after a PPDU has been received.

Demodulation Demodulation takes place in the frequency domain (via a 3072-point DFT), after the removal of the cyclic prefix for the symbol and correction for the channel effects at each subcarrier. As HPGP uses QPSK modulation, the receiver compares the measured value for the subcarrier in the in-phase and quadrature channels to the nominal values and estimates, under an additive white Gaussian noise assumption, the likelihood of the transmitted value having been a 0 or 1 bit. These probabilities are expressed as a ratio, the Log Likelihood Ratio (LLR) and then scaled according to the gain for the subcarrier in the channel estimate, such that the uncertainty inherent in weakly-received subcarriers is represented.

Post Processing Demodulated soft bits are combined by averaging to benefit from HPGP's redundancy schemes. They are then rearranged in read-by-row-write-by-column fashion to undo the channel interleaving process.

The FEC decoding is applied to produce hard decisions about the bit values. HPGP uses an unpunctured Turbo code with two systematic, rate $\frac{2}{3}$ constituent codes. Each pair of input bits (i, j) produces a codeword (i, j, p, q), where p and q are parity bits, p from the in-order input and q from an interleaved input.

Finally, the bits are unscrambled by XORing with the same generator polynomial used in the transmitter to recover the original sequence.

The CRC checks are computed over the received bits to determine if the contents have been received successfully, however the PHY-layer bits are delivered to the higher layers irrespective as even messages containing errors may provide useful information.

Each stage of the receiver is configurable with a wide range of parameters. In particular, the power threshold to trigger PPDU capture, the frequency-domain filtering, the initial CPO estimate and the estimated noise variance for demodulation all permit tailoring the receiver to a given scenario.

Considering the emissions as a wireless channel, the simple modulation and redundancy in HomePlug GreenPHY's robust ("ROBO") transmission modes mean the attacker need not match the channel characteristics of any particular receiver; they need only to receive the transmissions with enough of the signal intact. Specifically, the attacker requires a positive signal-to-noise ratio (SNR) over some fraction B of the transmitted bandwidth. The selection of B depends upon the transmissions mode in use and the effectiveness of any error-correction mechanisms, however for a rough estimate the level of redundancy can be used. Thus for `MINI_ROBO`, `STD_ROBO`, `HS_ROBO` B can be taken as 5.2MHz, 6.5MHz, 13MHz ($\frac{1}{5}$, $\frac{1}{4}$ and $\frac{1}{2}$ of the 26 MHz HPGP bandwidth) respectively.

RVFuzzer: Finding Input Validation Bugs in Robotic Vehicles Through Control-Guided Testing

Taegyu Kim[†], Chung Hwan Kim[*], Junghwan Rhee[*], Fan Fei[†], Zhan Tu[†], Gregory Walkup[†]
Xiangyu Zhang[†], Xinyan Deng[†], Dongyan Xu[†]
[†]*Purdue University, {tgkim, feif, tu17, gwalkup, xyzhang, xdeng, dxu}@purdue.edu*
[*]*NEC Laboratories America, {chungkim, rhee}@nec-labs.com*

Abstract

Robotic vehicles (RVs) are being adopted in a variety of application domains. Despite their increasing deployment, many security issues with RVs have emerged, limiting their wider deployment. In this paper, we address a new type of vulnerability in RV control programs, called input validation bugs, which involve missing or incorrect validation checks on control parameter inputs. Such bugs can be exploited to cause physical disruptions to RVs which may result in mission failures and vehicle damages or crashes. Furthermore, attacks exploiting such bugs have a very small footprint: just one innocent-looking ground control command, requiring no code injection, control flow hijacking or sensor spoofing. To prevent such attacks, we propose RVFuzzer, a vetting system for finding input validation bugs in RV control programs through control-guided input mutation. The key insight behind RVFuzzer is that the RV control model, which is the generic theoretical model for a broad range of RVs, provides helpful semantic guidance to improve bug-discovery accuracy and efficiency. Specifically, RVFuzzer involves a control instability detector that detects control program misbehavior, by observing (simulated) physical operations of the RV based on the control model. In addition, RVFuzzer steers the input generation for finding input validation bugs more efficiently, by leveraging results from the control instability detector as feedback. In our evaluation of RVFuzzer on two popular RV control programs, a total of 89 input validation bugs are found, with 87 of them being zero-day bugs.

1 Introduction

Robotic vehicles (RVs), such as commodity drones, are a type of cyber-physical system for autonomous transportation. They are typically equipped with a computing board with control hardware (e.g., micro-controller) and software (e.g., real-time control program). The on-board control program continuously senses the vehicle's physical state (e.g., position and velocity) and actuates the motors to control the vehicle's movement to accomplish a given mission. RVs have emerged in various application domains such as commercial, industrial, entertainment, and law enforcement. For instance, logistics companies (e.g., USPS, DHL, and Amazon) have introduced drone delivery services to meet the rapidly growing demand in e-commerce [6, 10, 13, 27].

With their increasing adoption in real-world applications, RVs are facing threats of cyber and cyber-physical attacks that exploit their attack surface. More specifically, an RV's attack surface spans multiple aspects, such as (1) physical vulnerabilities of its sensors that enable external sensor spoofing attacks [72, 77, 80]; (2) traditional "syntactic" bugs in its control program (e.g., memory corruption bugs) that enable remote or trojaned exploits [75]; and (3) control-semantic bugs in its control program that enable attacks via remote control commands. For attacks exploiting (1) and (2), there have been research efforts in defending against them [30, 38, 40, 50, 52, 70, 76]; whereas those exploiting (3) have not received sufficient attention. As a result, the RV's attack surface in the aspects of (1) and (2) is expected to get smaller, which may prompt attackers to increasingly look at the control-semantic vulnerabilities for new exploits.

In this paper, we focus on an important type of control-semantic bugs in RV control programs, called *input validation bugs*. An input validation bug involves an incorrect or missing validity check on a control parameter-change input. Such an input is provided to the control program via a remote control command, which could trigger RV controller malfunction and ultimately lead to physical impacts on the vehicle, such as mission disruption, vehicle instability, or even vehicle damage/crash. Finding input validation bugs is a new research problem because they are largely orthogonal to the traditional "syntactic" bugs (e.g., buffer overflow and use-after-free bugs) which can be detected by existing software testing/fuzzing techniques.

Input validation bugs, on the other hand, are created semantically via incorrect setting of control parameters. In an RV, the control program can be configured through *control parameters*, which are adjustable numerical inputs that de-

termine certain aspects of the control function's behavior (e.g., controller gain and default flight speed). We can further categorize input validation bugs into two sub-categories: (1) Incorrect or missing parameter range checks in the control program, which would accept illegitimate setting of control parameter values, are called *range implementation bugs*. (2) Incorrect specification of control parameter ranges, even if correctly implemented, may cause RV controller malfunction. We call such specification-level errors (implemented in the control program) *range specification bugs*.

Most RVs have a remote control interface [21] for operators to set or adjust control parameters during a flight. Unfortunately, such an interface can be leveraged by attackers [9, 55, 67, 68] to exploit input validation bugs and deliberately mis-configure certain control parameters. As an example (details in Section 6.3.3), an RV control program allows operators to dynamically adjust a parameter for the vehicle's angular control and suggests a range of valid values in its specification. However, the range is erroneously determined and implemented. Knowing this bug, an attacker can issue a malicious command to reset the parameter using an illegitimate value that falls into the "valid" range, consequently crashing the vehicle.

Testing RV control programs to find input validation bugs is challenging. Popular RV control software (e.g., ArduPilot [15], PX4 [24], and Paparazzi [23]) supports many different RV models (e.g., quadcopters and ground rovers) with a large number of hardware, software and control configuration options. Generating accurate control parameter value ranges requires thorough testing of hundreds of control parameters for each RV model. With the growing number of RV models supported by control software, such testing is increasingly difficult to scale and automate. To overcome this challenge, especially without assuming source code access, one might think of leveraging automated black-box software testing methods, such as fuzzing [14, 17, 19, 71]. However, traditional software fuzzing techniques are not directly applicable to RV control programs because: (1) With hundreds of configurable parameters, the control program has an extremely large input space to explore and (2) there is no uniform and obvious condition to automatically decide that a control program is malfunctioning. Many input validation bugs do not exhibit system-level symptoms until certain control and physical conditions are met at run-time.

Our solution to finding input validation bugs – without control program source code – is motivated by the following ideas: (1) The impacts of attacks exploiting input validation bugs can be manifested by the victim vehicle's control state; and (2) such state can be efficiently reproduced by combining the RV control program and a high-fidelity RV simulation framework, which is readily available [7, 8].

Based on these ideas, we develop RVFUZZER, an automated RV control program testing system to find input validation bugs. RVFUZZER supports input-driven testing of a subject control program's binary, which runs in an RV simulator – for safety and efficiency. Unlike a traditional program bug (e.g., a memory corruption or divide-by-zero bug) that can result in an obvious program execution failure, automatically determining if the control program is ill-behaving based on the simulated vehicle's physical state is not straightforward. To address the problem, RVFUZZER involves a *control instability detector* based on a standard control stability measurement formula [47] to detect vehicle control malfunction. More importantly, RVFUZZER leverages this detector to quantify control (in)stability as feedback to guide input mutation, so that bugs can be found more efficiently by covering a large portion of the input space in a reasonable number of test runs. Our control-guided input mutation method is based on the following control property: When RV control instability starts to be observed while increasing (decreasing) the value of a control parameter, further increase (decrease) of the parameter value will only maintain or intensify such instability (Section 4.3.2). Finally, RVFUZZER mutates environmental factors such as trajectory curve or wind condition during testing, as attackers may leverage predictable environmental factors as probabilistic attack-triggering conditions.

We have implemented a prototype of RVFUZZER and applied it to ArduPilot [15] and PX4 [24], which are two popular RV control software suites used in many commodity RVs [32, 45, 58, 69]. RVFUZZER finds a total of 89 input validation bugs that can cause RV control malfunction: Two of them are known input validation bugs that were previously patched by developers; whereas the remaining 87 bugs are zero-day bugs which we have reported to the developers. In response to our report, eight bugs have been confirmed and seven of them have been patched. The contributions of our work are as follows:

- We introduce input validation bugs, a new type of RV control-semantic vulnerability that can be exploited by attackers.
- We develop RVFUZZER, a control-guided program vetting system to discover input validation bugs with safety, efficiency, and automation.
- We apply RVFUZZER to two popular RV control software suites and find 89 input validation bugs including 87 zero-day bugs.

2 Background

RV Control Model The RV control model is the generic theoretical underpinnings that control the vehicle's movements and operations during its missions (e.g., flying in a trajectory with multiple waypoints). The RV's movements are along its six degrees of freedom (6DoF), which include the x, y, and z-axes for movement and the roll, pitch, and yaw for rotation (Fig. 1). The control model consists of multiple controllers, each for a specific degree of the 6DoF. For example, the x-axis

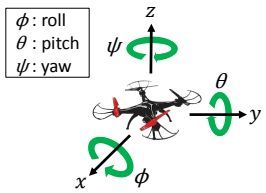


Figure 1: An RV's six degrees of freedom (6DoF).

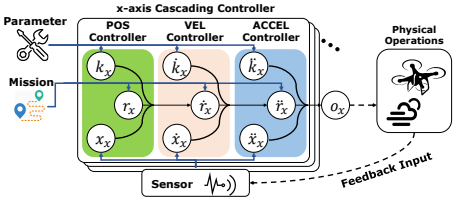


Figure 2: The x-axis controller (with three primitive controllers).

controller is shown in Fig. 2.

Inside the x-axis controller, there are three primitive controllers in a cascade, which are responsible for controlling the vehicle's position, velocity, and acceleration along the x-axis, respectively. Each primitive controller takes two state inputs: a reference state ($r(t)$) computed by its upstream primitive controller; and an observed state ($x(t)$) reported by sensors. The goal of the controller is to keep the observed state close to the reference state, via its core function of *control state stabilization*. The output of the function is the reference state for its downstream primitive controller. Each primitive controller has multiple adjustable parameters and accepts high-level mission directives (e.g., change of target location or speed).

Overall the RV control model involves complex dependencies between the 6DoF controllers, each having multiple parameters and accepting mission directives. Moreover, the controllers, sensors, and the vehicle's physical operations (e.g., those of motors) create a feedback loop, which enables the periodic, iterative working of the controllers.

RV Control Program An RV control program implements the RV control model. Correspondingly, it involves the following main modules: (1) a sensor module to collect sensor inputs (e.g., from GPS, inertial measurement unit, etc.) for periodic vehicle state observation, (2) a controller module to generate control output based on current mission, reference state, and sensor input, and (3) a mission module to interpret mission directives and execute them. These modules execute iteratively in the periodic control epochs.

During a flight, the RV communicates with a ground control station (GCS), which may issue a variety of GCS commands to the control program. Many of those commands allow RV operators to dynamically adjust the controller and mission parameters. We note that such a dynamic parameter change may be necessary to improve vehicle control performance (e.g., enhancing stability), in response to mission dynamics such as payload change and non-trivial external disturbances.

In addition to the control and communication functions, most RV control programs have a run-time control state logging function, for record-keeping and troubleshooting purposes. Real-world commodity RVs (e.g., Intel Aero [18], 3DR IRIS+ [12], and DJI drone series [16]), as well as their simulators, log in-flight control states in persistent storage. RVFUZZER leverages such logs for automatic determination of controller malfunction.

Control Parameters Because of the complexity and generality of RV control model and program, a large number (hundreds) of configurable parameters exist in the control program. Many of them are dynamically adjustable at run-time via the GCS command interface. For example, in the ArduPilot software suite [15], there are 247 configurable control parameters, including 111 parameters for the x-, y-axis controller, 119 for the z-axis controller, 29 for the roll controller, 29 for the pitch controller, 30 for the yaw controller, 103 for motor control, and 40 for mission specification. We note that, while the total number of the parameters is 247, some of the parameters are shared by multiple controllers. When receiving a GCS command to adjust one of these parameters, the control program is supposed to perform an input validity check to determine if the new value is within the safe range of that parameter. Unfortunately, such a check may be missing or based on an erroneous value range.

3 Attack Model

Attack Model and Assumptions Attacks that exploit input validation bugs are characterized as follows: Knowing an adjustable control parameter with incorrect or missing range check logic in the control program[1], the attacker concocts and issues a seemingly innocent – but actually malicious – parameter-change GCS command to the victim RV. Without correct input validation, the illegitimate parameter value will be accepted by the control program and cause at least one of the RV's 6DoF controllers to malfunction – either immediately or at a later juncture, inflicting physical impacts on the RV. When planning an attack, the attacker may also opportunistically exploit a certain environmental condition (e.g., strong wind) under which a parameter-change command would become dangerous. For example, he/she might wait for the right wind condition (e.g., by following weather forecast) to launch an attack with high success probability. Such a case will be presented in Section 6.3.3.

The attacker can be either an external attacker or an insider threat. In the case of an external attacker, we assume that he/she is able to perform GCS spoofing to issue the malicious command, which is justified by the known vulnerabilities in the wireless/radio communication protocols between RV and GCS [9, 55, 67, 68, 78]. In the case of an insider threat, we assume that the insider is a rogue RV operator (not a developer), who does not have access to control program

[1]The attacker may acquire such knowledge via a program vetting tool (such as RVFUZZER).

source code and cannot update the control program firmware.

Attack Model Justifications Our attack model is realistic (and attractive) to attackers for the following reasons: (1) Such an attack incurs a very small footprint – just one innocent-looking command, without requiring code injection/trojaning, memory corruption, or sensor/GPS spoofing; (2) The attack can still be launched even after the control program has been hardened against traditional software exploits [1,2,39,52]; (3) The attack looks like an innocent "accident" because the malicious parameter value passes the control program's validity check. In some cases (i.e., range specification bugs), it is even in the valid range set in the control program's specification.

Why would the attacker bother to manipulate control parameter values, instead of just taking control of, or crashing the vehicle? A key observation provides the answer: If the attacker is not aware of – and hence does not manipulate – illegitimate-but-accepted control parameter values, it would actually *not* be easy to disrupt or crash an RV with minimum footprint[2]. This is because both the RV control program and control model already achieve a level of robustness for the RV to resist being commanded into instability or danger: The control program can identify and reject many illegitimate commands; and the control model can filter or mitigate the impacts of some commands that escape the control program's check [11, 48]. Moreover, an internal attacker is also motivated to exploit illegitimate control parameter values that are erroneously considered normal in the RV's specification (i.e., range specification bugs), as the attacker could evade attack investigation by claiming that he/she was following RV control specification when issuing the command in question.

We do acknowledge that there exist scenarios where attackers can successfully launch attacks without exploiting input validation bugs. For example, an insider could hijack an RV by changing its trajectory, when working alone without a co-operator (who might otherwise catch the attack in action).

4 RVFuzzer Design

In this section, we present the design of RVFuzzer. We first give an overview of RVFuzzer's architecture (Section 4.1) and then present detailed design of two key components of RVFuzzer: (1) the control-guided instability detector that monitors the vehicle's control state to detect controller malfunction (Section 4.2) and (2) the control-guided input mutator that generates control program inputs for efficient program testing (Section 4.3).

4.1 Overview

RVFuzzer is designed to (1) detect physical instability of the RV during testing and (2) generate test inputs iteratively to achieve high testing efficiency and coverage. Fig. 3 presents an overview of RVFuzzer, which consists of four main components: a GCS program, the subject control program, a simulator, and a control-guided tester – the core component of RVFuzzer. The roles of the first three components are as follows: the GCS software is responsible for issuing RV control parameter-change commands; the subject control program, as the testing target, controls the operations of the (simulated) RV; and the simulator emulates the physical vehicle and its operating physical environment. We note that (1) the GCS and RV control programs are from real-world GCS and RV; and (2) our simulators [7, 8] are widely adopted for robotic vehicle design and testing.

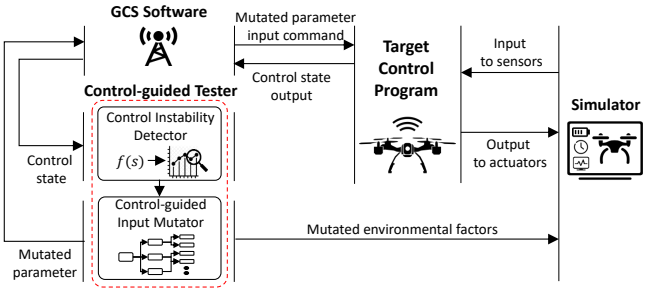


Figure 3: Overview of RVFuzzer.

RVFuzzer's control-guided tester consists of two sub-modules: (1) control instability detector and (2) control-guided input mutator. During testing, the control instability detector detects non-transient physical disturbances of the target RV (e.g., crash and deviation), as indication of control program execution anomaly caused by an input validation bug. The control-guided input mutator is a feedback-driven input mutator for efficient mutation of control parameter and environmental factor values. Using the results of the control instability detector as feedback, the mutator adaptively mutates control parameter values via a well-defined RV control interface (i.e., GCS commands created and issued by the GCS software). In addition, it mutates environmental factors (e.g., wind) by re-configuring the simulator.

4.2 Control Instability Detector

The goal of the control instability detector is to continuously monitor RV control state to determine if a specific GCS command has induced non-transient physical disturbance. Such a physical disturbance can be considered as an indication of an input validation bug. We note that input validation bugs may not lead to program crash, a common indicator of traditional bugs (e.g., memory corruption).

We first define a rule to detect physical disturbances, which is tailored for input validation bugs. We then describe the mechanism to monitor the RVFuzzer's 6DoF control states for detecting such a disturbance.

Indication of Control State Deviation Exploitation of an input validation bug will cause an RV's failure to stabilize

[2]The minimum footprint would help avoid detection before the attack succeeds.

its control states and/or complete its mission. To accurately detect bug-induced physical disturbance, RVFUZZER must be equipped with the capability of control state deviation detection. Among the possible physical disturbances experienced by an RV, there are two types of control state deviation: (1) observed state deviation and (2) reference state deviation. Accordingly, we define a detection rule to determine if one of the two types of control state deviation has occurred.

The first type – observed state deviation – is the case where a controller (e.g., the primitive x-axis velocity controller) fails to stabilize its observed state ($x(t)$) according to its reference state ($r(t)$). In the theoretical control model, a controller always tries to keep $x(t)$ close to $r(t)$ (Section 2). Consequently, if the difference between $x(t)$ and $r(t)$ keeps increasing and exceeds a certain threshold, the observed state will be considered deviating from the reference state. To quantify the observed state deviation, we leverage the integral absolute error (IAE) formula [47] which is widely used as a stability metric in control systems.

$$deviation(t) = \int_{t}^{t+w} \frac{|r(s) - x(s)|}{w} ds \qquad (1)$$

Given a time window w and starting from a certain time instance t, the formula quantifies the level of deviation ($deviation(t)$). If $deviation(t)$ is larger than a pre-determined threshold τ, our rule will determine that there is a control state deviation starting at t. We will describe how to experimentally determine w and τ for each 6DoF control state in Appendix A.

The second type – reference state deviation – is the case where an RV deviates from its given mission. A controller is expected to adjust its reference state to track its mission. If a controller fails to do that, it is considered malfunctioning. To detect such a deviation, our rule will check whether the difference between the reference state and the mission target becomes persistently greater than a threshold.

We note that our detection rule only considers non-transient control state deviation. An RV may experience transient control state deviation during normal operation but can effectively recover from it, thanks to the robustness features of the controllers such as the extended Kalman filter [46, 51, 60].

Control Instability Detection We now apply our "observed-reference" and "reference-mission" deviation determination rule to detect control instability. During a test mission, the control program readily logs all its 6DoF control states (Section 2). The log data can be retrieved by the GCS software, which will then be accessed by the Control Instability Detector and applied to the evaluation of the detection rule (Fig.3). Note that the control states include those of the three primitive controllers (for position, velocity, and acceleration control) inside each 6DoF controller; and each primitive controller logs its observed, reference and mission states. As such, the Control Instability Detector can apply the detection rule to detect control state deviation at any primitive controller.

4.3 Control-Guided Input Mutator

A software testing system needs to judiciously generate program inputs to achieve high bug coverage while reducing the number of the subject program's test runs. In other words, the set of generated testing inputs should be representative to produce the same or similar results when other untested inputs were provided to the program. We first define RVFUZZER's input mutation space (i.e., types and value ranges of dynamically adjustable control parameters). We then describe our control-guided input mutation strategy to generate representative testing inputs, with consideration of environmental factors that affect the RV operation and control.

Our input generation method considers both control parameters and environmental factors[3]. For control parameters, we first define their value mutation spaces (Section 4.3.1). We then present the feedback-driven input mutator which generates a reduced set of control parameter-change test inputs (Section 4.3.2). The mutator also mutates the external environmental factors and tests the control program under different combinations of input control parameter values and environment factor values.

4.3.1 Control Parameter Mutation Space

The input mutation space of the subject control program consists of: (1) the list of dynamically adjustable control parameters, (2) the range of all possible values for each parameter, and (3) the default value of each parameter.

The list of control parameters is obtained from the specification of control program and the GCS command interface. We note that this is public information even for a close-source control program. The three most popular control software suites (i.e., ArduPilot [15], PX4 [24], and Paparazzi [23]) all support a common parameter tuning interface defined in MAVLink [21], the de facto protocol for RV-GCS communications.

The value ranges of control parameters can be decided (1) by the data type of the control parameter and (2) by polling the control program itself. For a control parameter, its data type generically sets its value range. For example, the range of a 32-bit integer parameter is $[-2^{31}, 2^{31}-1]$. Interestingly, the ranges of many control parameters can be narrowed by polling the control program. This can be done by first sending a parameter-change command with a very large/small value; and then querying the actual value of that parameter, which now becomes the maximum/minimum value of the parameter defined in the control program. While the possibility of such a probe is specific to control program implementation, we do observe such implementation logic in ArduPilot and PX4.

The mutator also selects a default value within the range of each control parameter. Such a default value will be used

[3] Environmental factors are not program input but physical context in which the RV operates.

in the input space search during mutation (Section 4.3.2). We note that the set of default values of control parameters is normally made available by RV vendors (e.g., 3DR, DJI, and Intel), as a guidance to RV users when tuning the control parameters.

4.3.2 Feedback-Driven Parameter Input Mutator

RVFUZZER's input mutator accepts two inputs: the control parameter mutation space and the result of the Control Instability Detector from the previous run of the control program. The output of the mutator is the testing input for the program's next run. The efficiency of the control program vetting process depends on how well the mutated inputs are generated to trigger input validation bugs without launching too many program test runs with different inputs. To explain our mutation strategy and methods, we first introduce the underlying intuition of our strategy and then describe our feedback-driven testing process with two steps: one-dimensional mutation and multi-dimensional mutation.

Input Space Reduction Strategy The purpose of RVFUZZER is to find vulnerable – i.e., illegitimate but accepted – values for each dynamically adjustable control parameter. However, it is infeasible to test all possible values of a parameter. To improving testing efficiency, RVFUZZER must be able to selectively skip certain ranges of parameter values, if they lead to the same or similar outcome as the tested values. The value range-skipping idea is feasible thanks to the following observation: When control instability starts to be observed while increasing (decreasing) the value of a control parameter, further increase (decrease) of the parameter value will only maintain or intensify the instability.

We note that the aforementioned observation is generally valid. More specifically, in a control model, controllers and filters can be lumped together as part of its dynamics. Based on Root Locus [54], the trajectory of the loci always follows some asymptote. Hence, the change of a parameter will cause a *monotonic* change in stability. Sensor calibration can be considered as a constant disturbance, which will cause system response to degrade as the magnitude of the disturbance increases. Mission parameters will have different effects: Some can be grouped as part of the dynamics based on Root Locus; Some others, such as angle limitations, could cause an excessive response that introduces undesirable overshoot. This can be viewed as an integral windup, with a larger limit causing a larger overshoot.

Based on this observation, we propose two features for the mutator. (1) It will report valid/invalid value ranges — not individual values. Such a range will have a lower (minimum) and upper (maximum) bound. Any parameter value outside the range will cause control instability. (2) The mutator will be driven by feedback from the Control Instability Detector (Section 4.2) to determine the next testing input. Such feedback-driven mutation will be able to skip certain parameter value ranges for efficiency.

One-dimensional Mutation In the first step of control software vetting, RVFUZZER's input mutator determines the valid/invalid range for each control parameter *independently*. The mutator isolates the impact of the target parameter on the control state deviation by setting the values of all other parameters to their *default* values.

We present the one-dimensional mutation procedure in Algorithm 1. For each target control parameter, the mutator determines the upper and lower bounds of the valid value range by utilizing a mutation-based binary search method. We elaborate the method (Algorithm 1) to find the upper bound of the valid range as follows. We note that the mutator follows a syntactically similar method to find the lower bound of the valid range.

To find the upper bound, the mutator will iteratively launch test runs, using the binary search method to set the next run's *input value* and to update the *working range*. It will set the initial *min-limit* of the working range as the default value of the target parameter; the initial *max-limit* of the working range as the maximum possible value of the target parameter (Section 4.3.1); and the initial input value as the mid-point between the *min-limit* and *max-limit* values. Thereafter, in each run, the mutator obtains the output of the Control Instability Detector under the current input value, and updates the working range in the next run by considering the following two cases based on the detector's output (Line 14).

- **Case 1** (Line 17-18): If the mutator observes that the current input value does not cause any deviation, it skips the lower half of the working range in the next run and sets the new *min-limit* as the current input value. This decision is justified by our earlier observation on the monotonic property of control instability. For the next run, the mutator will again set the new input value as the mid-point between *min-limit* and *max-limit*.

- **Case 2** (Line 15-16): If the current input value leads to control state deviation, the mutator concludes that there are other values lower than the current input value which can also cause deviation. Hence, for the next run, the mutator will skip the upper half of the working range by setting *max-limit* as the current input value and the new input value as the mid-point between *min-limit* and *max-limit*.

We highlight that, after each run, the mutator skips the values corresponding to one half of the working range. This input space reduction strategy ensures that the mutator covers all possible values of the target control parameter efficiently. After determining the working range for the next run, the mutator sets the input value for the next run as the mid-point of the new working range (Line 19), following the binary search method. The mutator continues the (detector) feedback-driven

Algorithm 1 One-dimensional Mutation.

Input: Input mission (M), input parameter (P), test environmental factor (E), control state deviation threshold set for all primitive controllers (τ)
Output: An invalid range for a target parameter (R)

```
1: function OneDimensionalMutation(M, P, E, τ)                ▷ Main function
2:   Initialize R
3:   R.max ← OneMutation(M,P,E,τ,U)                           ▷ 'U': Upper-bound search
4:   R.min ← OneMutation(M,P,E,τ,L)                           ▷ 'L': Lower-bound search
5:   return R                                                 ▷ Return an invalid range of one parameter
6: function OneMutation(M, P, E, τ, bound)
7:   if bound = U then                                        ▷ 'U' indicates an upper-bound search
8:     {test,max-limit,min-limit} ← {(P.Max − P.Default)/2, P.Max, P.Default}
9:   else                                                     ▷ 'L' indicates a lower-bound search
10:    {test,max-limit,min-limit} ← {(P.Default − P.Min)/2, P.Default, P.Min}
11:  MinDiff ← 0
12:  do
13:    test' ← test                                           ▷ Store the testing value before mutation
14:    Dev ← RunAndDeviationCheck(M,P,test,E,τ)
15:    if (bound = U and Dev = True) or (bound = L and Dev = False) then
16:      max-limit ← test                                     ▷ Change the testing range
17:    else
18:      min-limit ← test                                     ▷ Change the testing range
19:    test ← (max-limit + min-limit)/2                       ▷ Mutate the testing value
20:  while |test' − test| > MinDiff                           ▷ Check the exit condition
21:  return GetInvalidRange(test,test',bound,Dev)
```

search method, until the difference between the input values in the current and the next runs is less than a pre-determined threshold $MinDiff$ (Line 20). Finally, the mutator determines the valid value range and the corresponding vulnerable value range (i.e., invalid range) for the target control parameter.

Multi-dimensional Mutation RVFuzzer also performs a more advanced form of input mutation: multi-dimensional mutation, which finds extra invalid parameter value ranges that one-dimensional mutation may not find. Such extra invalid parameter values are introduced because a target control parameter may have *dependencies* on other parameters. In other words, different (non-default) setting of such other parameters may expand the invalid range of the target parameter.

To test the impact of other parameters (P_{others}), RVFuzzer performs the multi-dimensional mutation for each target parameter (P_{target}) as described in Algorithm 2. In this algorithm, RVFuzzer utilizes the results from the one-dimensional mutation (Algorithm 1) of all control parameters (P_{all}) (i.e., the lower and upper bounds of their valid ranges). For the target parameter, RVFuzzer sets the initial working range as its valid value range obtained from one-dimensional mutation (Line 2). Thereafter, the mutation of the values of the other parameters (Line 8-15) and the target parameter (Line 18-21) are performed recursively.

In each recursion, the value of each of the other parameters is mutated among only three values: the default value, the lower bound of its valid value range and the corresponding upper bound (Line 11). We note that setting the values of one/more of the other parameters to their lower/upper bound values leads to an extreme scenario which can potentially exacerbate the impact of the target parameter on the control state deviation.

After setting the values of the other parameters (Line 18), the mutator follows a procedure similar to the one-

Algorithm 2 Multi-dimensional Mutation.

Input: Input mission (M), target testing input parameter (P_{target}), a set of all input parameters including one-dimensional search results (PS_{all}), test environmental factor (E), control state deviation threshold set for all primitive controllers (τ)
Output: An invalid range for a target parameter (R)

```
1: function MultidimensionalMutation(M, P_target, E, PS_all, τ)   ▷ Main function
2:   R ← GetInvalidRange(P_target)                                ▷ Results from the previous step
3:   PS_others ← PS_all − {P_target}                              ▷ A set of other parameters except for P_target
4:   PS_mut ← ∅                                                  ▷ Initialize the mutated parameter set
5:   R ← DepMutation(M, P_target, E, PS_others, PS_mut, R, τ)
6:   return R                                                    ▷ Return a new invalid range
7: function DepMutation(M, P_target, E, PS_others, PS_mut, R, τ)
8:   if PS_others ≠ ∅ then                                       ▷ Recursively mutate PS_others
9:     P_mut ← PS_others.Pop()
10:    PS_mut ← PS_mut ∪ P_mut
11:    for PV ∈ P_mut.Min, P_mut.Default, P_mut.max do
12:      PS_mut ← UpdateMutatedValue(PS_mut, P_mut, PV)
13:      R ← DepMutation(M, P_target, E, PS_others, PS_mut, τ)
14:  else                                                        ▷ Update the invalid range of P_target if all of PS_others are mutated
15:    R ← DepTest(M, P_target, E, PS_mut, R, τ)
16:  return R
17: function DepTest(M, P_target, E, PS_mut, R, τ)
18:  ParameterSet(PS_mut)                                       ▷ Configure parameters with values of PS_mut
19:  Upper ← OneMutation(M, P_target, E, τ, U)                  ▷ 'U': Upper-bound search
20:  Lower ← OneMutation(M, P_target, E, τ, L)                  ▷ 'L': Lower-bound search
21:  return UpdateInvalidRange(R, Upper, Lower)
```

dimensional mutation. It employs the mutation-based binary search method to determine and update the lower and upper bounds of the valid value range of the target parameter (Line 20-21). The new (in)valid range is then updated (Line 21).

In essence, as RVFuzzer mutates the values of multiple control parameters together, it can identify additional values of the target parameter that will cause control state deviation under specific value setting of the other parameters. If such invalid values lie outside the one-dimensional invalid value range, the multi-dimensional mutation will conditionally expand the invalid value range to include those values, subject to the setting of the other parameters. As such, the result of the multi-dimensional mutation can be considered as an incomplete set of constraints on the values of multiple control parameters.

4.3.3 Environmental Factors

In real-world missions, the RV interacts with the physical environment with external factors such as physical obstacles and wind. Such factors influence RV's control state and performance. We note that an external factor (e.g., wind) could make an otherwise valid parameter value cause control state deviation. This means that such values can be exploited by attackers. To detect such influence, RVFuzzer mutates and simulates the impact of environmental factors along with multi-dimensional mutation of parameter values. We categorize the environmental factors into two types: geography and disturbances.

Typical geographical factors of interest are obstacles encountered by an RV during its missions. The RV will need to take actions to avoid such an obstacle. The actions may entail changes in the parameter values to enable a change of trajectory. This may expand the invalid range of the parameter

values that will cause control state deviation. To expose such input validation bugs, RVFUZZER defines and simulates RV missions in which the RV needs to avoid obstacles via sudden, sharp trajectory changes. An attack case triggered by obstacle avoidance will be presented in Section 6.3.3.

External disturbances such as wind and turbulence may also disrupt the RV's operation. RVFUZZER simulates the wind gusts and mutates the wind speed and direction based on real-world wind conditions. Details of the wind factor setup are given in Section 6.2.2. The attack case presented in Section 6.3.3 also exploits the wind condition.

5 Implementation

To evaluate RVFUZZER experimentally, we have implemented a prototype of RVFUZZER. The implementation details of its main components are described as follows.

Subject Control Programs We choose the quadcopter as our subject vehicle as the quadcopter operates in all of the 6DoF and it is one of the most widely adopted types of RVs [49, 62, 64]. We point out that the implementation of RVFUZZER is not specific to a certain RV type or model as RVFUZZER only needs the physical quantities (e.g., weight and inertial parameters) and the corresponding simulator to support a vehicle. This means that RVFUZZER can be reconfigured to support other types of RVs, such as hexacopters and rovers.

We apply RVFUZZER to vet two control programs that both support the quadcopter: ArduPilot 3.5 and PX4 1.8. The default vehicle control model supported by both programs is that of the 3DR IRIS+ quadcopter [12]. All vetting experiments (on both ArduPilot and PX4) are performed using a desktop PC with quad-core 3.4 GHz Intel Core i7 CPU and 32 GB RAM running Ubuntu 64-bit.

Simulator To simulate the physical vehicle and environment, we utilized the APM simulator [8] and Gazebo [7, 42, 53] for ArduPilot and PX4, respectively. We note that RVFUZZER's control instability detection and input mutation functions can easily inter-operate with these simulators via the interfaces between the simulators and the control and GCS programs.

GCS Program We used QGroundControl [26] and MAVProxy [22] as the ground control station software for PX4 and ArduPilot, respectively.

Control-Guided Tester The control-guided tester is the core component of RVFUZZER. It is written in Python 2.7.6 with 5,722 lines of code. To implement the key functions in RVFUZZER, we leveraged the *Pymavlink* library [25], which provides APIs to remotely control the RV via the MAVLink communication protocol [21]. MAVLink is the de-facto communication protocol for robotic vehicles, which is used not only by ArduPilot and PX4, but also by other platforms such as Paparazzi [23], DJI [16], and LibrePilot [20]. MAVLink supports a wide range of GCS commands (e.g., for mission assignment, run-time control state monitoring, and parameter checking and adjustment) that are leveraged and tested by RVFUZZER.

To test the control performance of the subject vehicle, we adopted the *AVC2013* [5] mission which is an official mission provided by ArduPilot and used in autonomous vehicle competitions to test the control and mission execution capabilities of RVs. To improve the testing efficiency of RVFUZZER, we adjusted that mission by removing the overlapping flight courses, reducing the distance between each pair of waypoints, and increasing the vehicle's velocity.

To classify and generate the bug discovery results, we leverage a list of dynamically adjustable control parameters provided by ArduPilot and PX4 [28, 29]. Such a list is usually provided in the Extensible Markup Language (XML) format in the source code and can be easily parsed.

6 Evaluation

We now present evaluation results from our experiments with the RVFUZZER prototype. The three main questions that we want to answer are: (1) How effective is RVFUZZER at finding input validation bugs (Section 6.1); (2) How do different input mutation schemes of RVFUZZER contribute to the discovery of input validation bugs (Section 6.2); and (3) How can RVFUZZER be applied to discover input validation bugs that would otherwise be exploited to launch stealthy attacks (Section 6.3).

6.1 Finding Input Validation Bugs

We present a summary of the input validation bugs discovered by RVFUZZER from ArduPilot and PX4. These bugs are the result of a 8-day, non-stop testing session running RVFUZZER on the two control programs.

6.1.1 Classification of Input Validation Bugs

The validity of an input value of a control parameter is checked based on the *specified* range that has been determined and documented by developers during the development of the control program. Our subject control programs (ArduPilot and PX4) have the specified ranges of all the control parameters publicly available on their developer community websites [28, 29]. Leveraging these public range specifications, RVFUZZER found a number of input validation bugs through the 8-hour testing session. We classify these input validation bugs into two categories based on their root causes: range implementation bugs and range specification bugs.

Range Implementation Bugs Assuming that the specified valid range of a control parameter is correct, any value outside the specified range should be caught and rejected by the control program. If the implementation of the control program fails to enforce that, an out-of-range parameter value may

Table 1: Summary of input validation bugs found by RVFUZZER (RIB and RSB denote the number of range implementation and range specification bugs, respectively).

Module	Sub-module	ArduPilot		PX4	
		RIB	RSB	RIB	RSB
Controller	x, y-axis position	1	0	1	1
	x, y-axis velocity	2	1	1	1
	z-axis position	1	0	1	1
	z-axis velocity	1	0	1	0
	z-axis acceleration	3	0	0	0
	Roll angle	1	0	1	1
	Roll angular rate	5	0	3	3
	Pitch angle	1	0	1	1
	Pitch angular rate	5	0	3	3
	Yaw angle	1	0	2	2
	Yaw angular rate	6	0	3	3
	Motor	0	0	3	3
Sensor	Inertia sensor	3	3	0	0
Mission	x, y-axis velocity	1	1	2	0
	z-axis velocity	2	0	4	0
	z-axis acceleration	2	0	0	0
	Roll, Pitch	1	1	1	1
Total	-	36	6	27	20

be maliciously provided and accepted by the program, causing control state deviations. This is the nature of the range implementation bug which, based on our observation, arises from a lack of or an incorrect implementation of range check logic in the program. To discover range implementation bugs, RVFUZZER employs the one-dimensional mutation strategy. It mutates the value of each target parameter and issues the parameter-change GCS command with the mutated value to the control program. If the Control Instability Detector reports a control state deviation, RVFUZZER will report a range implementation bug associated with the target parameter.

Range Specification Bugs Ideally, the specified valid range of a parameter should correctly scope the value of the parameter. Unfortunately, this turns out not always the case. To reveal such problems, RVFUZZER first performs one-dimensional mutation and then performs multi-dimensional mutation on each target parameter, determining its invalid value range that will cause control state deviation. We observe that for some control parameters, their valid value ranges are erroneously specified by developers, allowing dangerous values in the specified – and subsequently implemented – ranges. This is the nature of the range specification bug. Based on our analysis, such bugs exist because a control program enforces a *fixed* valid value range for a control parameter, without considering three critical factors: (1) the difference between hardware models and configurations, (2) inter-dependencies between control parameters, and (3) impact of environmental factors. RVFUZZER reveals that the range of the valid input values of a target parameter tends to "shrink" under these factors, giving rise to range specification bugs.

6.1.2 Detection of Input Validation Bugs

Table 1 summarizes the range implementation bugs (RIB) and range specification bugs (RSB) discovered by RVFUZZER in ArduPilot and PX4. The detailed list of the 63 control parameters that are affected by these bugs is presented in Appendix B. For coherent presentation in Table 1, the control parameters in each of the two control programs are categorized into three modules (i.e., controller, sensor, and mission) and further into their sub-modules. Table 1 shows that RVFUZZER detected a total of 89 input validation bugs (42 bugs in ArduPilot and 47 bugs in PX4). We note that some of the control parameters are associated with *both* range implementation and the range specification bugs. Hence, the total number of input validation bugs (89) is higher than the total number of affected control parameters (63).

We highlight that only two of the 89 bugs discovered by RVFUZZER were detected and correctly patched by the developers *before* we reported our results to them. Out of the remaining 87 bugs, the developers have so far independently confirmed 8 bugs and patched 7 of them. The remaining bugs are under review. The delayed response of the developers brings forth an important point: Compared to the traditional "syntactic" bugs (e.g., buffer overflow), discovering, validating and patching input validation bugs require more time and effort. This is because the exploitability of each input validation bug must be fully verified under a spectrum of vehicle configurations and operating environments. In such a scenario, RVFUZZER can be utilized by developers as a helpful tool to automate the discovery and confirmation of input validation bugs.

6.1.3 Impact of Input Validation Bugs

We now detail the physical impacts (on the vehicle's operation) of the attacks that exploit the bugs found by RVFUZZER. We consider four levels of physical impact: crash, trajectory deviation, unstable movement, and frozen control states. Appendix B presents possible physical impact(s) of attacks that exploit each of the vulnerable control parameters. Here, we summarize the results by analyzing the impact on the modules of the control program. Specifically, we present the causality of the bugs in a bottom-up fashion and assess its impact on the control state deviation which is detected by RVFUZZER's Control Instability Detector.

Controller Module Among the control parameters related to the controller module, RVFUZZER discovered 27 range implementation bugs and 1 range specification bug in ArduPilot, and 20 range implementation bugs and 19 range specification bugs in PX4 (Table 1). These bugs can be used to maliciously set invalid parameter values or exploit environmental factors, which would directly affect the primitive controllers and corrupt the control states in the 6DoF. For example, if one of the control parameters related to the z-axis velocity is set to

a value in the invalid range due to an input validation bug, the manipulated parameter will corrupt the reference state of the (downstream) z-axis acceleration. As a result, the z-axis acceleration controller will attempt to bring its observed state closer to the corrupted reference state, which will cause control instability of the vehicle. Such instability may eventually lead to a crash.

Sensor Module For this module, while RVFUZZER found 3 range implementation bugs and 3 range specification bugs in ArduPilot, it did not find any input validation bug in PX4 (Table 1). We note that the vulnerable control parameters of the sensor module are related to either a sensor calibrator or a sensor filter for noise/disturbance. While the calibrator compensates for manufacturing errors in sensors and adjusts the observed state accordingly, the filter smooths out the sensor values and helps the controllers in robustly responding to physical interactions [73]. Hence, if an invalid value is assigned to a control parameter related to a sensor calibrator/-filter due to an input validation bug, the primitive controller that consumes the sensor values will compute a corrupted observed state. Such corruption will also propagate to its output reference state, and from there to other dependent primitive controllers, leading to unstable movement of the vehicle.

Mission Module For this module, RVFUZZER discovered 6 range implementation bugs and 2 range specification bugs in ArduPilot, and 7 range implementation bugs and 1 range specification bug in PX4 (Table 1). Recall that this module is responsible for setting the mission parameters (e.g., speed and tilting angles) which define or adjust the vehicle's mission. However, if a parameter related to the mission module is manipulated with an invalid value by exploiting an input validation bug, the corresponding controllers will generate misguided reference states. Such mission corruption will mislead one or more of the 6DoF controllers and prevent the vehicle from fulfilling its intended mission (e.g., not moving to the intended destination or at the intended speed), even if the vehicle does not experience any immediate danger.

6.2 Effectiveness of Input Mutation

RVFUZZER employs the control-guided input mutation strategy to generate control parameter value inputs and set environmental factors. We evaluate the effectiveness of this mutation strategy in enabling efficient discovery of input validation bugs.

6.2.1 Control Parameter Mutation

RVFUZZER discovers the range implementation bugs using the one-dimensional mutation strategy which detects the erroneous implementation of the parameter's range check logic. Through the extensive black-box-based (i.e., without source code) testing of the control parameters, RVFUZZER discovered a total of 63 range implementation bugs: 36 bugs in ArduPilot and 27 bugs in PX4.

To detect the incorrectly specified ranges of the parameters and find the range specification bugs, RVFUZZER employs one-dimensional mutation followed by the multi-dimensional mutation strategy. We demonstrate the effectiveness of RVFUZZER's mutation strategies in discovering the range specification bugs in Fig. 4, which presents the valid and invalid value ranges (detected using one-dimensional and multi-dimensional mutation) for the affected control parameters.

One-dimensional Mutation RVFUZZER discovered a total of 26 range specification bugs using one-dimensional mutation: 6 bugs in ArduPilot and 20 bugs in PX4 (Fig. 4). For example, for parameter `MC_TPA_RATE_P` in PX4, the specified range was between 0 and 1, and the default value was 0. However, RVFUZZER detected control state deviations with values between 0.1 and 1, and hence found 90% of the values in the specified range belonging to the invalid range. We note that almost 100% of the values in the specified range of the three parameters, `MC_PITCHRATE_FF`, `MC_ROLLRATE_FF` and `MC_YAWRATE_FF`, in PX4 are invalid. This is because, while each of these parameters can be independently configured with a wide range of input values, there is a smaller range of values that are valid when the other parameters take their default values.

Multi-dimensional Mutation Recall that the multi-dimensional mutation further expands the invalid range of the target parameter to include the additional values that cause control state deviation under specific, non-default settings of the other parameters. In Fig. 4, we observe that the multi-dimensional mutation expands the invalid ranges of 10 out of 26 range specification bugs found using one-dimensional mutation. For instance, RVFUZZER found that the invalid range of the `MC_ROLL_P` parameter in PX4 was expanded from 1.7% to 51.7% when multi-dimensional mutation was employed. We highlight that for some parameters, RVFUZZER reported a significant increase of invalid range with multi-dimensional mutation. In particular, compared to the invalid ranges detected using one-dimensional mutation, the invalid ranges of the `MC_PITCHRATE_MAX` and `MC_ROLLRATE_MAX` parameters in PX4 increased from 0.4% to 88.1% and from 0.1% to 87.9%, respectively. These results demonstrate that the multi-dimensional mutation strategy can discover invalid values of control parameters with stronger awareness of the inter-parameter dependencies (discussed further in Section 8).

6.2.2 Environmental Factor Mutation

RVFUZZER further found that the invalid ranges of some control parameters expand when environmental conditions are taken into account. This is important because the developers may not completely consider the impact of various environmental conditions when specifying the valid range of a pa-

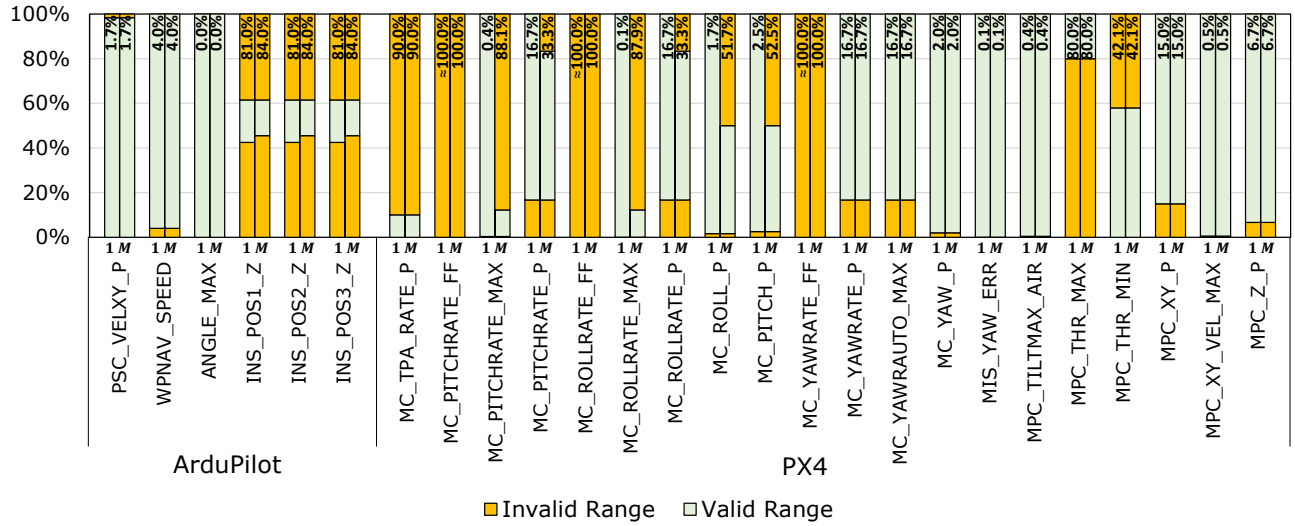


Figure 4: Invalid control parameter ranges discovered by RVFUZZER, normalized to the specified value ranges (**1**: One-dimensional mutation, ***M***: Multi-dimensional mutation). Percentage of invalid ranges (%) within the specified value ranges are noted at the top of the bars.

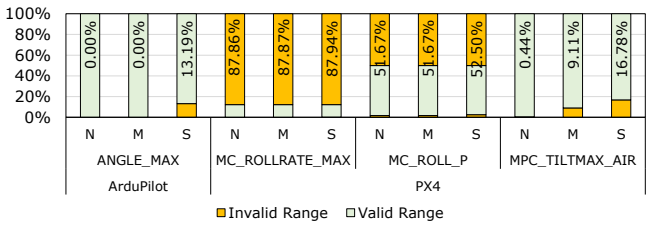


Figure 5: Normalized invalid ranges within the specified value ranges under different wind conditions (N: No wind, M: Medium wind, S: Strong wind).

rameter. Based on our observation, two factors may widen the invalid ranges: (1) geographical factor and (2) external disturbance (e.g., strong wind), as described in Section 4.3.3. RVFUZZER found four cases which can be exploited with realistic environmental factors.

We performed tests based on existing wind analysis statistics [33, 41, 59] and simulated various wind conditions. The wind conditions were divided into three categories: no wind, medium wind (with a horizontal wind component of 5 m/s or a vertical wind component of 1 m/s), and strong wind (with a horizontal wind component of 10 m/s or a vertical wind component of 3 m/s). For each wind condition, the wind gust was simulated from 0 to 360 degrees with 30-degree increments. Simulations were also performed where the wind gust was designed to come in at every 30-degree angle between the horizontal tests and the vertical tests, such that the tested wind vectors approximately formed an ellipsoid. These wind settings enrich our standard test mission (Section 5), which already reflects geographical factors as it emulates flight paths with sharp turns for obstacle avoidance.

Fig. 5 presents the impact of three different wind conditions on the four parameters which cause control state deviations. RVFUZZER discovered these four input validation bugs using multi-dimensional mutation over the four parameters. We observe that the impact of environmental factors expands the invalid ranges of those parameters. In particular, when the wind conditions were not considered, `ANGLE_MAX` did not have any invalid range under both one-dimensional and multi-dimensional mutations. However, with wind conditions, RVFUZZER reveals that this parameter can be exploited when strong wind is present.

Such an input validation bug is exploitable because a large angular change is required to alter the direction of the vehicle. Specifically, if the maximum allowed angle or angular speed is not large enough (even within the specified value ranges), the vehicle's motors cannot generate enough force to change the direction or resist the wind gusts. As a result, the vehicle may fail to change its direction at sharp turns or it might drift in the wind's direction in the worst case.

We note that the results with environmental factor mutation may be affected by other factors, such as the control model, configuration, and physical attributes (e.g., motor power and the size of the vehicle). For example, if the vehicle is capable of turning with a larger roll angle, has a smaller size, or has stronger motors, it may be able to resist wind gusts when changing its flight direction. Hence, these conditions need to be tested by RVFUZZER for each specific type of vehicle.

6.3 Case Studies

We present three representative case studies of input validation bugs. We also discuss how an attacker can exploit these bugs, and how RVFUZZER can proactively discover them. The three cases cover different affected controllers, cause different impacts on the RV, and require different components of RVFUZZER's testing techniques to detect. Specifically, the bug discussed in Case I (Section 6.3.1) affects the x and

```
1 #define WPNAV_WP_SPEED_MIN 100  //Buggy code 2
2 #define WPNAV_WP_SPEED_MIN 20   //Patched code 2
3 ...
4 void AC_WPNav::set_speed_xy(float speed_cms){
5 -if(_wp_speed_cms>=WPNAV_WP_SPEED_MIN){  //Buggy code 1
6 +if(speed_cms>=WPNAV_WP_SPEED_MIN){     //Patched code 1
7     _wp_speed_cms = speed_cms;
8     _pos_control.set_speed_xy(_wp_speed_cms);
9     ...
```

Listing 1: Input validation bug case on x, y-axis mission velocity. The parameter can be dynamically changed by either a mission speed-change command or a speed parameter-change command.

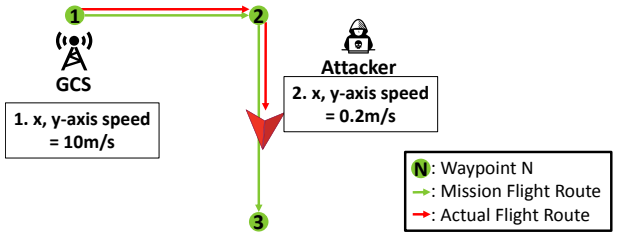


Figure 6: Illustration of Case Study I: An RV cannot recover its normal speed for the segment from Waypoint 2 to Waypoint 3.

y-axes controllers and causes unrecoverable slowdown, but can be discovered by RVFUZZER using the one-dimensional mutation technique. Case II (Section 6.3.2) presents a bug that affects the pitch controller, leads to a crash, and can only be found via multi-dimensional mutation strategy. Finally, the bug in Case III (Section 6.3.3) adversely affects the roll controller and causes significant deviation from the assigned mission, but can be discovered by mutating an environmental factor (wind force).

6.3.1 Case Study I: Bug Causing "Unrecoverable Vehicle Slowdown" Discovered by One-Dimensional Mutation

Attack We consider an RV that is assigned the mission of express package delivery (Fig. 6). Because of the urgency, the operator sets the RV's mission speed to 10 m/s at Waypoint 1. During the mission, while the RV slows down to make a turn at Waypoint 2, the attacker sends a seemingly innocent, but malicious, command to the RV to change its mission speed to 0.2 m/s (the minimum *specified* speed is 0.2 m/s). After the turn, however, the operator will not be able to resume the 10 m/s mission speed by issuing speed-change commands. This attack exploits an input validation bug in ArduPilot, illustrated in Listing 1.

Root Cause Listing 1 presents the code that runs in the RV when it receives a new speed-change input (denoted by `speed_cms`) during its mission. The specified minimum speed (in cm/s) is denoted by the `WPNAV_WP_SPEED_MIN` parameter (Line 1). We note that the *current* mission speed (denoted by `_wp_speed_cms`) is compared with the minimum mission speed (Line 5). This means that if (and only if) the current mission speed is equal to or higher than the minimum mission speed, it can be replaced by the new mission speed in the input command; If the current mission speed is lower than the minimum mission speed, it cannot be changed. Hence, this is the bug which can be exploited by the attacker, by sending a speed-change command with a value lower than the minimum mission speed while the current mission speed is higher than the minimum mission speed. This bug has been patched recently by the developers by correcting the value of the minimum mission speed (Line 2) and setting the comparison of the minimum mission speed with the input speed (Line 6).

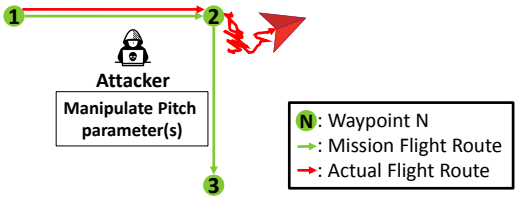


Figure 7: Illustration of Case Study II: The attack launched at Waypoint 2 causes an RV to oscillate due to failing control of the pitch angle.

Bug Discovery This bug was discovered by RVFUZZER while performing one-dimensional mutation of the input mission speed parameter. For input mission values above 1 m/s, the RV successfully changed its current mission speed. However, if the current mission speed dropped below 1 m/s, RVFUZZER can no longer change the current mission speed by setting the input mission speed parameter. The failure to change the current mission speed led to the incorrect execution of the mission, resulting in control state deviation, simulated and detected by RVFUZZER. Hence, RVFUZZER reported this deviation-triggering parameter as an input validation bug, which is confirmed by the related source code in Listing 1 (as ground truth of our evaluation).

6.3.2 Case Study II: Bug Causing "Oscillating Route and Crash" Discovered by Multi-Dimensional Mutation

Attack We consider an RV that is assigned the same mission as in Case Study I. As shown in Fig. 7, at Waypoint 2 of the mission, the attacker sends a malicious command to the RV to change one of the four pitch control parameters: `MC_PITCH_P`, `MC_PITCHRATE_P`, `MC_PITCHRATE_P`, and `MC_PITCHRATE_FF`. Because of the inter-dependency between these parameters, such a malicious command, which looks innocent, can cause the RV to fail to stabilize its pitch angle, resulting in unrecoverable oscillation and deviation from its route.

Root Cause The unrecoverable oscillation on the RV's route is caused by the failure of its pitch controller to track the reference state of the pitch. The pitch controller utilizes four inter-dependent parameters: the P con-

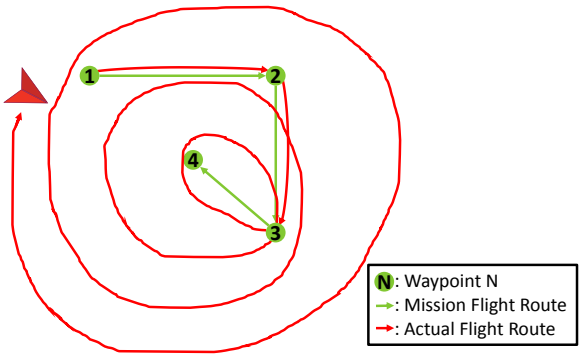


Figure 8: Illustration of Case Study III: An RV fails to complete a simple mission from Waypoint 1 to Waypoint 4 due to the impact of environmental factors.

trol gain of pitch angle (MC_PITCH_P), the P control gain of the pitch angular speed (MC_PITCHRATE_P), the maximum pitch rate (MC_PITCHRATE_MAX), and the feed-forward pitch rate (MC_PITCHRATE_FF). For example, a high value of MC_PITCHRATE_FF helps track the reference state of the pitch when MC_PITCH_P is low. When both MC_PITCHRATE_FF and MC_PITCH_P have high values, the RV may perform overly aggressive stabilization operations. In that case, a low value of the maximum pitch rate (MC_PITCHRATE_MAX) is desirable to mitigate the impact of such operations.

We point out that such dependencies can be exploited by an attacker to affect the RV's operations by corrupting the value of just *one* parameter. Let us assume that the RV is already configured with high values of MC_PITCHRATE_FF and MC_PITCH_P. If the attacker sets MC_PITCHRATE_MAX to a high value, the pitch controller will start to respond to the minuscule difference between the reference state and the observed state of the pitch angle with extreme sensitivity. As a result, the RV will not be able to strictly follow its flight path. We note that this type of bug can only be discovered when the dependencies between multiple parameters are considered in the test.

Bug Discovery This bug was found by RVFUZZER while performing multi-dimensional mutation (Algorithm 2) of the parameters related to the pitch controller. RVFUZZER mutated the target parameter (MC_PITCHRATE_MAX), while setting high values for MC_PITCH_P and MC_PITCHRATE_FF parameters. Unlike the one-dimensional mutation, which determined the parameter's valid range to be between 6.7 and 1800, the multi-dimensional mutation determined that the valid range of MC_PITCHRATE_MAX is to be between 6.7 and 220.1. RVFUZZER detected and reported the expanded invalid range of MC_PITCHRATE_MAX as an input validation bug.

6.3.3 Case Study III: Bug Causing "Diverging Route" Discovered by Wind Force Mutation

Attack In this case study, we consider an RV assigned a mission to deliver a food item to a customer via the path presented in Fig. 8. The RV is required to follow the path around tall buildings on a windy day with the wind direction towards the west. Since the item (e.g., soup) might spill if the RV changes its attitude drastically, the operator tries to prevent sudden changes in the roll angle by limiting the maximum angular-change speed (MC_ROLLRATE_MAX) to a small value. When the vehicle is approaching Waypoint 2, the attacker sends a command to set the maximum tilting angle (MPC_TILTMAX_AIR) to a low value. We note that the RV is supposed to make a 120-degree turn to avoid a tall building at Waypoint 3. However, the RV fails to make the correct turn at Waypoint 3 and hence cannot reach the destination (Waypoint 4) after multiple attempts to correct the diverging path. We note that the value of the maximum tilting/roll angle parameter is accepted by the control program because it is within the *specified* valid range, yet the value causes control state deviation due to the strong wind condition.

Root Cause There are three causes that induce the vehicle's unexpected flight path divergence: (1) the mission route with sharp turns, (2) the roll controller's parameter value that is not responsive enough to change the direction in time, and (3) the strong wind that expands the invalid ranges of the roll controller's parameters. In this case study, the combination of these three factors disrupts the vehicle's maneuver and trajectory, resulting in a failed mission (and a hungry customer).

Bug Discovery RVFUZZER discovered this bug in PX4 by mutating the wind condition during the AVC2013 mission (Section 5) which involves many sharp turns of the vehicle. As the input values of the roll controller parameters were mutated under a strong wind condition, RVFUZZER detected control state deviation between the reference state and the mission (Fig. 5). Hence, RVFUZZER reported this as an input validation bug contingent upon the influence of an external factor (wind).

7 Related Work

Control Semantics-Driven RV Protection There exists a body of work that leverages control semantics to protect RVs from attacks during flights and missions [38, 40, 50]. BlueBox [40] detects abnormal behaviors of an RV controller by running a shadow controller in a separate microprocessor that monitors the correctness of the primary controller, based on the same control model. CI [38] extracts control-level invariants of an RV controller to detect physical attacks. Similarly, Heredia et al. [50] propose using a fault detection and isolation model extracted from a target RV controller and enforces the model to detect anomalies during flights.

Another line of work focuses on deriving finite state models to detect abnormal controller behaviors [37, 61]. Orpheus [37] automatically derives state transition models using program analysis for run-time anomaly detection. Bruids [61] relies on

a manual specification of RV behaviors to derive a behavioral model to detect run-time anomalies.

Other approaches utilize machine learning techniques to derive benign behavioral models of an RV controller. Abbaspour et al. [31] apply adaptive neural network techniques to detect fault data injection attacks during flight. Samy et al. [70] use neural network techniques to detect sensor faults. Two related efforts [30, 76] leverage a similar approach but detect both sensor and actuator faults.

Complementing the prior efforts, RVFUZZER leverages control semantics to *proactively* find input validation bugs that may be exploited by RV attackers. Unlike the previous works that aim to detect abnormal behaviors *during* flights, our work focuses on identifying input validation bugs in RV control programs *before* flights via off-line RV simulation and program vetting. Control semantics are leveraged to reduce the input value mutation space and simulators are adopted to render the impacts of control parameter and external factor changes on control states.

Feedback-directed Testing RVFUZZER is inspired by many existing feedback-driven testing/fuzzing systems for conventional program testing [14, 17, 19, 34–36, 43, 44, 56, 57, 63, 66, 71, 74, 79]. These solutions leverage different mutation strategies to increase the coverage of testing/fuzzing. Several systems [14, 17, 19, 71] mutate input values with varying granularity (e.g., bit, byte-level) driven by the tested code's coverage achieved during each test run, using the code coverage as feedback. Another line of work [63, 74] adopts a hybrid approach to increase code coverage using both dynamic and symbolic execution. Finally, many efforts leverage taint analysis [36, 43, 56, 57, 79] or a combination of taint analysis and symbolic execution [34, 35, 44, 66] for high testing coverage. Such approaches mutate inputs with awareness of the dependencies between program input and logic.

Testing techniques for conventional, non-cyber-physical programs rely on well-established mechanisms for (1) bug detection and (2) input mutation. Specifically, these testing techniques leverage generic, easy-to-detect symptoms of program failures (e.g., segmentation faults) as indication of a triggered bug and mutate program input following information (e.g., code coverage) agnostic to domain semantics. Compared with conventional software testing, RVFUZZER addresses new problems and opportunities when finding (semantic) input validation bugs in RV control. Many such bugs do not cause an immediate, easy-to-detect crash of the control program, especially when running with an RV simulator. Meanwhile, control-theoretical properties offer hints to reduce the input value mutation space.

8 Discussion

Control Parameter Inter-dependencies As revealed by multi-dimensional mutation, the control parameters may have dependencies on one another. A specific value of one parameter can increase or decrease the (in)valid value ranges of other parameters. The ground truth on such inter-parameter dependencies can only be derived from full knowledge about the underlying control model and the control program implementation, given the large number of control variables (including hundreds of parameters), the wide ranges of their values, and the influence from various environmental factors. As a result, it is challenging to fully and accurately capture the control parameter inter-dependencies, with only the binary of a control program. In this work, we consider the subject control program binary as a black box and take a pragmatic approach by only revealing *part of* such inter-dependencies. A more generic approach to control parameter dependency derivation – possibly based on source code and a formal control model – is left as future work.

Standard Safety Testing and Certification For the safety of avionics software for airborne systems, there exist standard safety tests and software certifications such as DO-178B/C [4] and ISO/IEC 15408 [3]. To the best of our knowledge, however, there has been no standard safety testing framework created for RVs. We believe that RVFUZZER's post-production, black-box-based (i.e., without source code) vetting will serve as a useful complement to standardized safety testing during RV design and production.

9 Conclusion

Robotic vehicles (RVs) are facing cyber and cyber-physical attacks launched via various attack vectors. In this paper, we identify a new, small-footprint attack against RVs, where an attacker remotely issues a control parameter-change command with an illegitimate parameter value to disrupt the RV's control and mission. Such a value is erroneously accepted by the RV control program because of an input validation bug associated with the control parameter. The attack requires no code injection, control flow hijacking, or sensor spoofing hence cannot be defeated by existing RV security solutions. To proactively discover input validation bugs in a control program binary, we develop RVFUZZER, a control program testing system that reveals illegitimate-yet-accepted value ranges of dynamically adjustable control parameters. RVFUZZER adaptively mutates the input control parameter values to determine the (in)valid value ranges, driven by the detection of control state deviations in the simulated RV. Furthermore, it considers the impact of external factors by mutating their values and presence. RVFUZZER has discovered 89 real input validation bugs in two of the most popular RV control software suites, with mutation efficiency and automation.

10 Acknowledgment

We thank our shepherd, Nolen Scaife and the anonymous reviewers for their valuable comments and suggestions. We also thank Vireshwar Kumar for his detailed feedback and edits which have improved the quality of the paper. This work was supported in part by ONR under Grant N00014-17-1-2045. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the ONR.

References

[1] *Address space layout randomization*, 2001. http://pax.grsecurity.net/docs/aslr.txt.

[2] *Exec shield*, 2005. https://static.redhat.com/legacy/f/pdf/rhel/WHP0006US_Execshield.pdf.

[3] *ISO/IEC 15408-1:2009*, 2009. https://www.iso.org/standard/50341.html.

[4] *RTCA/DO-178C*, 2011. Software Considerations in Airborne Systems and Equipment Certification.

[5] SparkFun Autonomous Vehicle Competition 2013, 2013. https://avc.sparkfun.com/2013.

[6] *DHL parcelcopter launches initial operations for research purposes*, 2014. http://www.dhl.com/en/press/releases/releases_2014/group/dhl_parcelcopter_launches_initial_operations_for_research_purposes.html.

[7] *Gazebo - A dynamic multi-robot simulator*, 2014. http://gazebosim.org.

[8] *SITL Simulator (ArduPilot Dev Team)*, 2014. http://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html.

[9] Hijacking drones with a MAVLink exploit, 2015. http://diydrones.com/profiles/blogs/hijacking-quadcopters-with-a-mavlink-exploit.

[10] *USPS Drone Delivery | CNBC*, 2015. https://www.youtube.com/watch?v=V9GXiXgaK34&list=PLL3t5xY2V44xOxvTIxS4AHuUhFE_bMwhz&index=36.

[11] *Inertial Navigation Estimation Library*, 2016. https://github.com/priseborough/InertialNav.

[12] *3DR iris+*, 2018. https://3dr.com/support/articles/iris.

[13] *Amazone Prime Air*, 2018. https://www.amazon.com/b?node=8037720011.

[14] *American Fuzzy Lop*, 2018. http://lcamtuf.coredump.cx/afl.

[15] *ArduPilot*, 2018. http://ardupilot.org.

[16] *DJI Phantom 4 Advanced*, 2018. https://www.dji.com/phantom-4-adv.

[17] *Honggfuzz*, 2018. https://google.github.io/honggfuzz/.

[18] *Intel Aero*, 2018. https://software.intel.com/en-us/aero.

[19] *libFuzzer*, 2018. https://llvm.org/docs/LibFuzzer.html.

[20] *LibrePilot*, 2018. https://www.librepilot.org.

[21] *MAVLink — Micro Air Vehicle Communication Protocol*, 2018. https://mavlink.io.

[22] *MAVProxy - A UAV ground station software package for MAVLink based systems*, 2018. https://ardupilot.github.io/MAVProxy.

[23] *Paparazzi UAV - an open-source drone hardware and software project*, 2018. http://wiki.paparazziuav.org/wiki/Main_Page.

[24] *PX4 Pro Open Source Autopilot - Open Source for Drones*, 2018. http://px4.io.

[25] *Pymavlink - A python implementation of the MAVLink protocol*, 2018. https://github.com/ArduPilot/pymavlink.

[26] *QGroundControl - Intuitive and Powerful Ground Control Station for PX4 and ArduPilot UAVs*, 2018. http://qgroundcontrol.com.

[27] *Wing - Google X*, 2018. https://x.company/projects/wing.

[28] *ArduPilot Parameter List*, 2019. http://ardupilot.org/copter/docs/parameters.html.

[29] *PX4 Parameter List*, 2019. https://dev.px4.io/en/advanced/parameter_reference.html.

[30] Alireza Abbaspour, Payam Aboutalebi, Kang K Yen, and Arman Sargolzaei. Neural adaptive observer-based sensor and actuator fault detection in nonlinear systems: Application in uav. *ISA transactions*, 67:317–329, 2017.

[31] Alireza Abbaspour, Kang K Yen, Shirin Noei, and Arman Sargolzaei. Detection of fault data injection attack on uav using adaptive neural network. *Procedia computer science*, 95:193–200, 2016.

[32] Luis Afonso, Nuno Souto, Pedro Sebastiao, Marco Ribeiro, Tiago Tavares, and Rui Marinheiro. Cellular for the skies: Exploiting mobile network infrastructure for low altitude air-to-ground communications. *IEEE Aerospace and Electronic Systems Magazine*, 31(8), 2016.

[33] JC André, G De Moor, P Lacarrere, and R Du Vachat. Modeling the 24-hour evolution of the mean and turbulent structures of the planetary boundary layer. *Journal of the Atmospheric Sciences*, 35(10):1861–1883, 1978.

[34] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (IEEE S&P)*, IEEE S&P '12, 2012.

[35] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (IEEE S&P)*, IEEE S&P '15, 2015.

[36] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing.

[37] Long Cheng, Ke Tian, and Danfeng Daphne Yao. Orpheus: Enforcing cyber-physical execution semantics to defend against data-oriented attacks. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC)*, ACSAC '17, 2017.

[38] Hongjun Choi, Wen-Chuan Lee, Yousra Aafer, Fan Fei, Zhan Tu, Xiangyu Zhang, Dongyan Xu, and Xinyan Deng. Detecting attacks against robotic vehicles: A control invariant approach. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, CCS '18, 2018.

[39] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. Aces: Automatic compartments for embedded systems. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[40] Fan Fei, Zhan Tu, Ruikun Yu, Taegyu Kim, Xiangyu Zhang, Dongyan Xu, and Xinyan Deng. Cross-layer retrofitting of uavs against cyber-physical attacks. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, ICRA '18, 2018.

[41] Rod Frehlich, Yannick Meillier, Michael L Jensen, Ben Balsley, and Robert Sharman. Measurements of boundary layer profiles in an urban environment. *Journal of applied meteorology and climatology*, 45(6):821–837, 2006.

[42] Fadri Furrer, Michael Burri, Markus Achtelik, and Roland Siegwart. Rotors—a modular gazebo mav simulator framework. In *Robot Operating System (ROS): The Complete Reference (Volume 1)*, pages 595–625. 2016.

[43] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, ICSE '09, 2009.

[44] Vijay Ganesh, Tim Leek, and Martin Rinard. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security)*, USENIX Security '13, 2013.

[45] Balazs Gati. Open source autopilot for academic research-the paparazzi system. In *Proceedings of the American Control Conference (ACC)*, ACC '13, 2013.

[46] Demoz Gebre-Egziabher, Roger C Hayward, and J David Powell. Design of multi-sensor attitude determination systems. *IEEE Transactions on aerospace and electronic systems*, 40(2):627–649, 2004.

[47] Dunstan Graham and Richard C Lathrop. The synthesis of optimum transient response: criteria and standard forms. *Transactions of the American Institute of Electrical Engineers, Part II: Applications and Industry*, 72(5):273–288, 1953.

[48] Saeid Habibi. The smooth variable structure filter. *Proceedings of the IEEE*, 95(5):1026–1059, 2007.

[49] Zhijian He, Yanming Chen, Zhaoyan Shen, Enyan Huang, Shuai Li, Zili Shao, and Qixin Wang. Ard-mu-copter: A simple open source quadcopter platform. In *Proceedings of the 2015 11th International Conference on Mobile Ad-hoc and Sensor Networks (MSN)*, MSN '15, 2015.

[50] G Heredia, A Ollero, M Bejar, and R Mahtani. Sensor and actuator fault detection in small autonomous helicopters. volume 18, pages 90–99. Elsevier, 2008.

[51] Myungsoo Jun, Stergios I Roumeliotis, and Gaurav S Sukhatme. State estimation of an autonomous helicopter using kalman filtering. In *Proceedings of 1999 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IROS '99, 1999.

[52] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. Securing real-time microcontroller systems through customized memory view switching. In *Proceedings of the 27th Annual Symposium on Network and Distributed System Security (NDSS)*, 2018.

[53] Nathan P Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2004.

[54] Benjamin C Kuo. *Automatic control systems*. Prentice Hall PTR, 1987.

[55] Y. Kwon, J. Yu, B. Cho, Y. Eun, and K. Park. Empirical analysis of mavlink protocol vulnerability for attacking unmanned aerial vehicles. *IEEE Access*, 6:43203–43212, 2018.

[56] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Angora: Efficient fuzzing by principled search. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, ESEC/FSE '17, 2017.

[57] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, ESEC/FSE '17, 2017.

[58] Renju Liu and Mani Srivastava. Protc: Protecting drone's peripherals through arm trustzone. In *Proceedings of the 3rd Workshop on Micro Aerial Vehicle Networks, Systems, and Applications (DroNet)*, DroNet '17, 2017.

[59] Marie Lothon, Donald H Lenschow, and Shane D Mayor. Doppler lidar measurements of vertical velocity spectra in the convective planetary boundary layer. *Boundary-layer meteorology*, 132(2):205–226, 2009.

[60] F Landis Markley, John Crassidis, and Yang Cheng. Nonlinear attitude filtering methods. In *Proceedings of the AIAA Guidance, Navigation, and Control Conference and Exhibit (AIAA)*, AIAA '05, 2005.

[61] Robert Mitchell and Ray Chen. Adaptive intrusion detection of malicious unmanned air vehicles using behavior rule specifications. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 44(5):593–604, 2014.

[62] A. Nemati and M. Kumar. Modeling and control of a single axis tilting quadcopter. In *Proceedings of the American Control Conference (ACC)*, ACC '14, 2014.

[63] H. Peng, Y. Shoshitaishvili, and M. Payer. T-fuzz: Fuzzing by program transformation. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (IEEE S&P)*, IEEE S&P '18, 2018.

[64] Viswanadhapalli Praveen and S Pillai. A.,"modeling and simulation of quadcopter using pid controller". *International Journal of Control Theory and Applications (IJCTA)*, 9(15):7151–7158, 2016.

[65] Friedrich Pukelsheim. The three sigma rule. *The American Statistician*, 48(2):88–91, 1994.

[66] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the 24th Annual Symposium on Network and Distributed System Security (NDSS)*, NDSS '17, 2017.

[67] Nils Rodday. Hacking a professional drone. 2016.

[68] Nils Miro Rodday, Ricardo de O Schmidt, and Aiko Pras. Exploring security vulnerabilities of unmanned aerial vehicles. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS)*, NOMS '16, 2016.

[69] S Sabikan and SW Nawawi. Open-source project (osps) platform for outdoor quadcopter. *Journal of Advanced Research Design*, 24:13–27, 2016.

[70] Ihab Samy, Ian Postlethwaite, and Dawei Gu. Neural network based sensor validation scheme demonstrated on an unmanned air vehicle (uav) model. In *Proceedings of 47th IEEE Conference on Decision and Control (CDC)*, pages 1237–1242, 2008.

[71] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl: Hardware-assisted feedback fuzzing for os kernels. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, USENIX Security '17, 2017.

[72] Yun Mok Son, Ho Cheol Shin, Dong Kwan Kim, Young Seok Park, Ju Hwan Noh, Ki Bum Choi, Jung Woo Choi, and Yong Dae Kim. Rocking drones with intentional sound noise on gyroscopic sensors. In *Proceedings of 24th USENIX Security symposium (Usenix Security)*, Usenix Security '15, 2015.

[73] Yunmok Son, Juhwan Noh, Jaeyeong Choi, and Yongdae Kim. Gyrosfinger: Fingerprinting drones for location tracking based on the outputs of mems gyroscopes. *ACM Transactions on Privacy and Security (TOPS)*, 21(2):10, 2018.

[74] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 23rd Annual Symposium on Network and Distributed System Security (NDSS)*, NDSS '16, 2016.

[75] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (IEEE S&P)*, IEEE S&P '13, 2013.

[76] Heidar A Talebi, Khashayar Khorasani, and Siamak Tafazoli. A recurrent neural-network-based sensor and actuator fault detection and isolation for nonlinear systems with application to the satellite's attitude control subsystem. *IEEE Transactions on Neural Networks*, 20(1):45–60, 2009.

[77] T. Trippel, O. Weisse, W. Xu, P. Honeyman, and K. Fu. Walnut: Waging doubt on the integrity of mems accelerometers with acoustic injection attacks. In *Proceedings of 2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, EuroS&P '17, 2017.

[78] Mathy Vanhoef and Frank Piessens. Key reinstallation attacks: Forcing nonce reuse in wpa2. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, CCS '17, 2017.

[79] T. Wang, T. Wei, G. Gu, and W. Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (IEEE S&P)*, IEEE S&P '10, 2010.

[80] Chen Yan, Wenyuan Xu, and Jianhao Liu. Can you trust autonomous vehicles: Contactless attacks against sensors of self-driving vehicle. *DEF CON*, 24, 2016.

A Thresholds for Control State Deviation

We present how to determine the threshold values used by our control instability detector to detect control state deviation (Section 4.2). We use the AVC2013 mission (Section 5) and thirty other experimental missions in our experiments, similar to existing work [38]. Specifically, the thresholds are determined by applying the three-sigma rule [65] on the top deviation values. For the time window (w) in the IAE formula, we set it to the duration of each mission segment (i.e., flight segment between two consecutive waypoints) within a mission. The list of the threshold values that we use for each control state is presented in Table 2.

We note that we do not monitor control state deviation in the second derivative states of the 6DoF (i.e., acceleration of any of the 6DoF). This is because, if their observed states are oscillating, they can potentially cause false positives. In fact, for the same reason, some control programs do not control acceleration in some 6DoF controllers (e.g., ArduPilot does not control the angular acceleration of roll, pitch, and yaw). However, RVFUZZER can detect their control state deviation via the indirect impacts on the *dependent* states. The control state deviation in the second derivative states are propagated to their integral states (e.g., the first derivative states of the 6DoF), as their controls are intrinsically related.

Table 2: List of threshold values for each control state.

Control Program	ArduPilot	PX4
Latitude/Longitude Position	11.62 m	10.08 m
Latitude/Longitude Velocity	1.23 m/s	4.71 m/s
Altitude Position	2.06 m	3.43 m
Altitude Velocity	0.26 m/s	0.12 m/s
Roll	2.66 deg	1.98 deg
Roll Rate	2.83 deg/s	3.68 deg/s
Pitch	4.64 deg	3.94 deg
Pitch Rate	10.67 deg/s	15.35 deg/s
Yaw	4.13 deg	6.16 deg
Yaw Rate	16.24 deg/s	14.69 deg/s

Table 3: Input validation bugs in ArduPilot and the implications of the attacks exploiting them (C: Crash; D: Deviation from trajectory; U: Unstable movement; S: "Stuck" in certain location or speed).

Control Program Module	Parameter	Physical Impacts			
		C	D	U	S
Controller	PSC_POSXY_P	✓			✓
	PSC_VELXY_P	✓	✓	✓	
	PSC_VELXY_I		✓	✓	
	PSC_POSZ_P				✓
	PSC_VELZ_P	✓			
	PSC_ACCZ_P	✓			✓
	PSC_ACCZ_I	✓	✓	✓	
	PSC_ACCZ_D	✓	✓	✓	
	ATC_ANG_RLL_P	✓			
	ATC_RAT_RLL_I	✓			
	ATC_RAT_RLL_IMAX	✓			✓
	ATC_RAT_RLL_D	✓			
	ATC_RAT_RLL_P	✓		✓	
	ATC_RAT_RLL_FF	✓		✓	
	ATC_ANG_PIT_P	✓			
	ATC_RAT_PIT_P	✓		✓	
	ATC_RAT_PIT_I	✓			
	ATC_RAT_PIT_IMAX	✓			
	ATC_RAT_PIT_D	✓			✓
	ATC_RAT_PIT_FF	✓		✓	✓
	ATC_ANG_YAW_P	✓			
	ATC_SLEW_YAW			✓	
	ATC_RAT_YAW_P			✓	
	ATC_RAT_YAW_I			✓	
	ATC_RAT_YAW_IMAX				✓
	ATC_RAT_YAW_D	✓			✓
	ATC_RAT_YAW_FF	✓		✓	
Sensor	INS_POS1_Z	✓		✓	
	INS_POS2_Z	✓		✓	
	INS_POS3_Z	✓		✓	
Mission	WPNAV_SPEED				✓
	WPNAV_SPEED_UP				✓
	WPNAV_SPEED_DN				✓
	WPNAV_ACCEL	✓			✓
	WPNAV_ACCEL_Z	✓			✓
	ANGLE_MAX	✓			✓

B Physical Impacts Caused by Input Validation Bug Exploitation

We present more details about the input validation bugs found by RVFUZZER and the implications of the attacks that exploit them in Tables 3 (for ArduPilot) and Table 4 (for PX4). The columns of each table shows: (1) the control program modules where the bugs belong (Control Program Module), (2) the vulnerable control parameters (Parameter, i.e., with erroneous range specification or range implementation), and (3) the possible physical impacts caused by the attacks exploiting the bugs (Physical Impacts). While the two tables list a total of 63 parameters, some of the parameters are associated with *both* range implementation and specification bugs. This explains why the total number of bugs (89) is higher than the number of vulnerable parameters.

Depending on the specific (malicious) value of the control parameter, the impact of an attack may vary. Here the possible impacts are categorized into four types as shown in the four

sub-columns of the "Physical Impacts" column: "C" – vehicle crash; "D" – deviation from trajectory; "U" – unstable vehicle movement; and "S" – vehicle getting "stuck" at a certain location or speed. All of these impacts are non-transient and cannot be recovered by the controllers.

Table 4: Input validation bugs in PX4 and implications of attacks exploiting them.

Control Program Module	Parameter	Physical Impacts			
		C	D	U	S
Controller	MC_TPA_RATE_P	✓		✓	
	MC_PITCHRATE_FF	✓	✓	✓	
	MC_PITCHRATE_MAX	✓	✓		
	MC_PITCHRATE_P	✓	✓	✓	
	MC_PITCH_P	✓	✓	✓	✓
	MC_ROLLRATE_FF	✓	✓	✓	
	MC_ROLLRATE_MAX	✓	✓		
	MC_ROLLRATE_P	✓	✓	✓	
	MC_ROLL_P	✓	✓	✓	
	MC_YAWRATE_FF			✓	✓
	MC_YAWRATE_P			✓	✓
	MC_YAW_P			✓	✓
	MIS_YAW_ERR				✓
	MPC_TILTMAX_AIR		✓		✓
	MPC_THR_MAX	✓	✓	✓	
	MPC_THR_MIN	✓	✓	✓	
	MPC_XY_P	✓	✓		✓
	MPC_Z_P	✓	✓		✓
	MPC_XY_VEL_P	✓	✓	✓	✓
	MPC_Z_VEL_P	✓	✓		✓
Mission	MC_YAWRAUTO_MAX			✓	✓
	MPC_XY_VEL_MAX		✓		✓
	MPC_XY_CRUISE		✓		
	MPC_Z_VEL_MAX_DN		✓		✓
	MPC_Z_VEL_MAX_UP	✓	✓		✓
	MPC_TKO_SPEED				✓
	MPC_LAND_SPEED				✓

Seeing is Not Believing: Camouflage Attacks on Image Scaling Algorithms

Qixue Xiao* [1,4], Yufei Chen* [2,4], Chao Shen[2], Yu Chen[1,5], and Kang Li[3]
[1]*Department of Computer Science and Technology, Tsinghua University*
[2]*School of Electronic and Information Engineering, Xi'an Jiaotong University*
[3]*Department of Computer Science, University of Georgia*
[4]*360 Security Research Labs*
[5]*Peng Cheng Laboratory*

Abstract

Image scaling algorithms are intended to preserve the visual features before and after scaling, which is commonly used in numerous visual and image processing applications. In this paper, we demonstrate an automated attack against common scaling algorithms, i.e. to automatically generate camouflage images whose visual semantics change dramatically after scaling. To illustrate the threats from such camouflage attacks, we choose several computer vision applications as targeted victims, including multiple image classification applications based on popular deep learning frameworks, as well as mainstream web browsers. Our experimental results show that such attacks can cause different visual results after scaling and thus create evasion or data poisoning effect to these victim applications. We also present an algorithm that can successfully enable attacks against famous cloud-based image services (such as those from Microsoft Azure, Aliyun, Baidu, and Tencent) and cause obvious misclassification effects, even when the details of image processing (such as the exact scaling algorithm and scale dimension parameters) are hidden in the cloud. To defend against such attacks, this paper suggests a few potential countermeasures from attack prevention to detection.

1 Introduction

Image scaling refers to the resizing action on a digital image, while preserving its visual features. When scaling an image, the downscaling (or upscaling) process generates a new image with a smaller (or larger) number of pixels compared to the original one. Image scaling algorithms are widely adopted in various applications. For example, most deep learning computer vision applications use pre-trained convolutional neural network (CNN) models, which take data with a fixed size defined by the input layers of those models. Hence, input images have to get scaled in the data preprocessing procedure to meet with specific model input size. Popular deep learning frameworks, such as Caffe [17], TensorFlow [13] and Torch [26], all integrate various image scaling functions in their data preprocessing modules. The purpose of these built-in scaling functions is to allow the developers to use these frameworks to handle images that do not match the model's input size.

Although scaling algorithms are widely used and are effective to normal inputs, the design of common scaling algorithms does not consider malicious inputs that may intentionally cause different visual results after scaling and thus change the "semantic" meaning of images. In this paper, we will see that an attacker can utilize the "data under-sampling" phenomena occurring when a large image is resized to a smaller one, to cause "visual cognition contradiction" between human and machines for the same image. In this way, the attacker can achieve malicious goals like detection evasion and data poisoning. What's worse, unlike adversarial examples, this attack is independent from machine learning models. The attack indeed happens before models consume inputs, and hence this type of attacks affects a wide range of applications with various machine learning models.

This paper characterizes this security risk and presents a camouflage attack on image scaling algorithms (abbreviated as *scaling attack* in the rest of the paper). To successfully launch a scaling attack, attackers need to deal with two technical challenges: (a) First, an adversary needs to decide where to insert pixels with deceiving effects by analyzing the scaling algorithms. It is tedious and practically impossible to use manual efforts to determine exact pixel values to achieve a desired deceiving effect for realistic images. Therefore, a successful attack needs to explore an automatic and efficient camouflage image generation approach. (b) Second, for cloud-based computer vision services, the exact scaling algorithm and input size of their models are transparent to users. Attackers need to infer scaling-related parameters of the underneath algorithms in order to successfully launch such attacks.

To overcome these challenges, we first formalize the process of scaling attacks as a general optimization problem. Based on the generalized model, we propose an automatic

*Co-first authors. This work was completed during their internship program at 360 Security Research Labs.

generation approach that can craft camouflage images efficiently. Moreover, this work examines the feasibility of this attack in both the white-box and black-box scenarios, including applications based on open deep learning frameworks and commercial cloud services:

- In the white-box case (see Section 6.1 for more details), we analyze common scaling implementations in three popular deep learning frameworks: Caffe, TensorFlow and Torch. We find that nearly all default data scaling algorithms used by these frameworks are vulnerable to the scaling attack. With the presented attack, attackers can inject poisoning or deceiving pixels into input data, which are visible to users but get discarded by scaling functions, and eventually being omitted by deep learning systems.
- In the black-box case (see Section 6.2 for more details), we investigate the scaling attack against cloud-based computer vision services. Our results show that even when the whole image processing pipeline and design details are hidden from users, it is still possible to launch the scaling attack to most existing cloud-based computer vision services. Since image scaling modules are built upon open image processing libraries or open interpolation algorithms, possible ways of image scaling implementation are relatively limited. Attackers can design a brute-force testing strategy to infer the scaling algorithm and the target scale. In this paper, we exhibit a simple but efficient testing approach, with successful attack results on Microsoft Azure[1], Baidu[2], Aliyun[3] and Tencent[4].
- Interestingly, we also discover and discuss the range of the attacking influence extends to some computer-graphic applications, such as mainstream web browsers shown in Section 6.3.

We provide a video to demonstrate the attack effects, which is available at the following URL: https://youtu.be/Vm2N0mb14Ow.

This paper studies the commonly used scaling implementations, especially for image scaling algorithms employed in popular deep learning frameworks, and reveals potential threats to the image scaling process. Our contributions can be summarized as follows:

- This paper reveals a security risk in image scaling process in computer vision applications. We validate and testify the image scaling algorithms commonly used in popular deep learning (DL) frameworks, and our results indicate that the security risk affects almost all image applications based on DL frameworks.
- This paper formalizes the scaling attack into a constrained optimization problem, and presents the corresponding implementation to generate camouflage images automatically and efficiently.
- Moreover, we prove that the presented attack is still effective for cloud vision services, where the implementation details of image scaling algorithms and parameters are hidden from users.
- To eliminate the threats from the scaling attack, we suggest several potential defense strategies from two aspects: attack prevention and detection.

[1] https://azure.microsoft.com/en-us/services/cognitive-services/computer-vision/?v=18.05

[2] https://ai.baidu.com/tech/imagerecognition/fine_grained

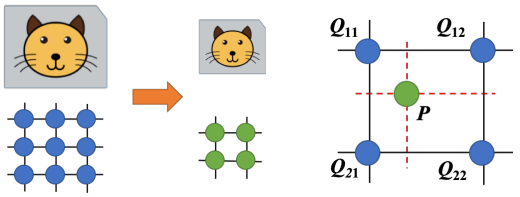
[3] https://data.aliyun.com/ai?spm=a2c0j.9189909.810797.11.4aae547aEqltqh#/image-tag

[4] https://ai.qq.com/product/visionimgidy.shtml#tag

2 Image Scaling Attack Concept and Attack Examples

In this section, we first present a high level overview of image scaling algorithms, followed by the concept of image scaling attacks. Then, we exhibit some examples of the image scaling attack, and finally we conduct an empirical study of the image scaling practices in deep learning based image applications.

2.1 An Overview of Image Scaling Algorithms



(a) Image scaling. (b) Interpolation in scaling.

Figure 1: The concept of image scaling.

Image scaling algorithms are designed to preserve the visual features of an image while adjusting its size. Fig.1 presents the general concept of a common image scaling process. A scaling algorithm infers value of each "missing point" by using interpolation methods. Fig.1b shows an example of constructing pixel P in the output image based on the pixels of Q_{11}, Q_{12}, Q_{21} and Q_{22} in the original image. A scaling algorithm defines which neighbor pixels to use in order to construct a pixel of the output image, determines the relative weight values assigned to each individual neighbor pixels. For example, for each pixel in the output image, a nearest neighbor algorithm only picks a single pixel (the nearest one) from the input to replace it. A bilinear algorithm considers a set of neighbor pixels surrounding the target pixel as the



Figure 2: An example showing deceiving effect of the scaling attack. (Left-side: what humans see; right-side: what DL models see)

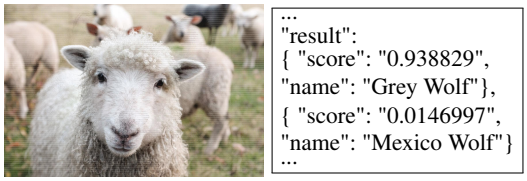


Figure 3: A scaling attack example against Baidu image classification service.

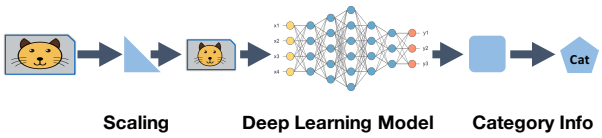


Figure 4: How data get processed in image classification systems.

basis. It then calculates a weighted average of the neighbor pixel values as the value assigned to the target pixel.

Such scaling algorithms often assume that pixel values in an image are results from natural settings, and they do not anticipate pixel-level manipulations with malicious intents. This paper demonstrates that an attacker can use scaling algorithms to alter an image's semantic meaning by carefully adjusting pixel-level information.

2.2 Attack Examples

The scaling attack potentially affects all applications that apply scaling algorithms to preprocess input images. To demonstrate potential risks and deceiving effects of the scaling attack, here we provide two attack examples of the scaling attack on practical applications.

Fig.2 presents the first attack example for a local image classification application *cppclassification* [16], a sample program released by the Caffe framework. For the classification model with an input size of 224*224, we specially craft input images of a different size (672*224). The image in the left column of Fig.2 is one input to the deep learning application, while the image in the right column is the output of the scaling function, i.e., the *effective image* fed into the deep learning model. While the input in the left column of Fig.2 visually presents a sheep-like figure, the deep learning model takes the image in the right column as the actual input and classifies it as an instance of "White Wolf".

To validate the deceiving effect on deep learning applications, we build one image classification demo based on the BAIR/BVLC GoogleNet model [8], which assumes the input data are of the scale of 224*224. When an image with a different size is provided, the application triggers the native resize() function built in the Caffe framework to rescale the image to fit the input size of the model (224*224). The exact classification setup details and the program outputs are presented in Appendix A.

Fig.3 exhibits one attack example against the Baidu image classification service. The attack image is crafted from a sheep image, with the aim to lead people to regard it as a sheep but the machine to regard it as a wolf. The results returned by the cloud service API[5] show that the attack image is classified as the "Grey Wolf" with a high confidence score (achieving 0.938829), indicating that our attack is effective. More examples of the scaling attack against more cloud-based computer vision services are presented in Table 3. In fact, image scaling algorithms are commonly used by a wide range of computer-graphic applications, rather than limited to deep-learning-based computer vision systems. Therefore, they are all potentially threatened by this type of security risk.

2.3 Empirical Study of Image Scaling Practices in Deep Learning Applications

Data scaling is actually a common action in deep learning applications. Fig.4 shows how the scaling process is involved in open-input applications' data processing pipelines, such as image classification as an Internet service. For design simplicity and manageable training process, a deep learning neural network model usually requires a fixed input scale. For image classification models, images are usually resizedd to 300*300 to ensure high-speed training and classification. As shown in Table 1, we examine nine popular deep learning models and all of them use a fixed input scale for their training and classification process.

For deep learning applications that receive input data from fixed input sources, such as video cameras, the input data formats are naturally determined. Even in such situation, we find that the image resizing is still needed in certain cases.

One common situation we observe is the use of pre-trained models. For example, NVIDIA offers multiple self-driving sample models [6], and all these models use a specific input

[5]The original API response is presented in Chinese. Here we translate it into English.

Table 1: Input sizes of various deep learning models.

Model	Size (pixels*pixels)
LeNet-5	32*32
VGG16, VGG19, ResNet, GoogleNet	224*224
AlexNet	227*227
Inception V3/V4	299*299
DAVE-2 Self-Driving	200*66

size 200*66. However, for the recommended camera [24] specification provided by NVIDIA, the size of generated images varies from 320*240 to 1928*1208. None of the recommended cameras produce output that matches the NVIDIA's model input size. Therefore, for system developers that do not want to redesign or retrain their models, they have to employ scaling functions in their data processing pipeline to fit the pre-trained model's input scale. Recent research work, such as the sample applications used in DeepXplore [25], also shows that the resizing operation is commonly used in self-driving applications to adjust original video frames' size to the input size of models.

Most deep learning frameworks provide data scaling functions, as shown in Table 2. Programmers can handle images with different sizes without calling scaling function explicitly. We examined several sample programs released by popular deep learning frameworks, such as Tensorflow, Caffe, and Torch, and we have found all of them implicitly trigger scaling functions in their data processing pipelines. Appendix B provides several examples.

3 Related Work

This section briefly reviews some related work and makes a comparison with our approach.

3.1 Information Hiding

Information hiding is a significant topic in information security [2,9,15,18,19,21,27,30,31]. Information hiding methods achieve reversible data hiding by image interpolation, but these are different from our attack method: First, the goals are different. The information hiding methods conceal data in a source image to make the secret information unnoticed by human, and image applications operate on the complete data. Our presented attack hides a target image in a source image to cause a visual cognition contradiction between human and image applications. The core components (such as deep learning classifiers) of the victim applications operate on a partial data (i.e. the scaling output). Second, information hiding efforts often impose a customized coding method (such as LSB and NIP [21]) in order to conceal and recover hidden information. This coding scheme is often kept as a secret known only to the designer of the specific information hiding method. In contrast, a scaling attack is based on the interpolation algorithm built within the victim application to achieve the deceiving effect. The main task for an attacker is to reverse engineering the scaling algorithms and design the pixel replacement policy.

3.2 Adversarial Examples

The research of adversarial examples attract growing public attentions with the reviving popularity of Artificial Intelligence. An adversarial image fools the Artificial Intelligence by inserting perturbations into the input image, which are hard to be noticed by human eyes. For example, Goodfellow *et al.* [14] presented a linear explanation of adversarial examples and revealed that such attack is effective for current sufficiently linear deep networks. In addition to the theoretical analysis, Alexey *et al.* [20] added the perturbations into the physical world and successfully launched the attack. It should be noted that the attack target of existing adversarial examples essentially aims at machine learning models, while our method focuses on the data preprocessing step, concretely, the image scaling action. Vulnerabilities in code implementation, such as control-flow hijacking, also could lead to recognition evasions [32]. However, we exploit the weakness of scaling algorithms in this work other than code implementation.

3.3 Invisible/Inaudible Attacks

Some researchers investigate potential attacks beyond the human sensing ability. Ian Markwood *et al.* [22] showed a content masking attack against the Adobe PDF standard. By tampering the font files, the attacker can insert secret information into PDF files without being noticed by human. They demonstrated such attack against state-of-the-art information-based services. Besides the attacks in vision fields, Zhang *et al.* [34] presented the *DolphinAttack* against speech recognition (SR) systems. They created secret voice commands on ultrasonic carriers that are inaudible for human beings, but can be captured and sensed by voice controllable systems. In this way, an attacker can control SR systems "silently". It has been proved that the widely used SR systems, like Apple Siri and Google Now, are vulnerable to such attack. Our attack is like a reverse of such invisible/inaudible attacks. The attacker leverage the difference between the input and output of the scaling function. Most part of the content visible to human is not consumed by the component that uses the scaling output.

4 Formalizing the Scaling Attack

This section describes the goal of a typical scaling attack and how we design a method to automatically create attack images with deceiving effects. The autonomous attack image crafting

Table 2: Scaling algorithms in deep learning frameworks.

DL Framework	Library	Interpolation Method	Order [a]	Validation [b]
Caffe	OpenCV [7]	Nearest	H→V	✓
		Bilinear(☆)	H→V	✓
		Bicubic	H→V	✓
		Area	H→V	†
		Lanczos	H→V	†
Tensorflow [c]	Python-OpenCV Pillow [10] Tensorflow.image	Nearest(☆Pillow)	H→V	✓
		Bilinear (☆Python-OpenCV, Tensorflow.image)	H→V	✓
		Bicubic	H→V	✓
		Area	H→V	†
		Lanczos	H→V	†
Torch	Torch-OpenCV	Nearest	H→V	✓
		Bilinear(☆)	H→V	✓
		Bicubic	H→V	✓
		Area	H→V	†
		Lanczos	H→V	†

☆ The default scaling algorithm.

[a] H→V means the algorithm first scales horizontally and then vertically.

[b] The validation is performed on attack with a constraint $\varepsilon = 0.01$. ✓ represents generate attack images successfully satisfying the constraints. † means we have not yet verified the attack effects because the algorithm is complex and rarely used in DL applications. More details please see Section 6.1.

[c] Tensorflow integrates multiple image processing packages, including Python-OpenCV, Pillow, and Tensorflow.image.

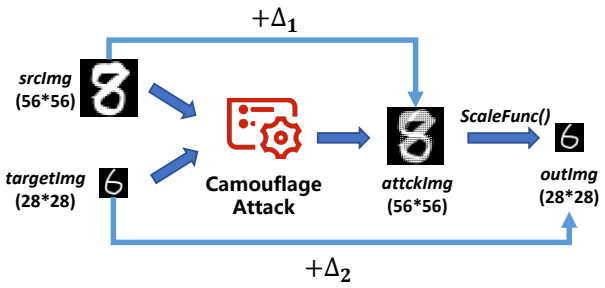


Figure 5: Automatic attack image crafting.

framework is shown in Fig.5, and details are presented in Section 4.2.

4.1 Attack Goals

The goal of the scaling attack is to create a deceiving effect with images. Here the deceiving effect refers to the case that an image partially or completely presents a different meaning to humans before and after a scaling action. In such case, we call the input file to the scaling action the *attack image*.

To describe the process of a scaling attack, we define the following four conceptual objects involved in one attack.

- source image (or *srcImg*) S_{m*n} – the image that an attacker wants the attack image to look like.
- attack image (or *attackImg*) A_{m*n} – the crafted image eventually created and fed to the scaling function.
- output image (or *outImg*) $D_{m'*n'}$ – the output image of the scaling function.
- target image (or *targetImg*) $T_{m'*n'}$ – the image that the attacker wants the *outImg* to look like.

In some cases, some of these objects are identical. For example, it is often possible for an attacker to generate an out image that is identical to the target image.

The process of performing a scaling attack is to craft an *attackImg* under visual similarity constraints with *srcImg* and *targetImg*. Based on the intent and constraints on source images, we define the image scaling attack into two attack modes.

The first attack mode is when both the source and target images are specified, i.e. the attacker wants to scale an image that looks like a specific source image to an image that looks like a specific target image. In this mode the attacker launches a **source-to-target attack**, where the semantics of *srcImg* and *targetImg* are controlled as he/she wants. However, posing constraints on the looks of both the source and target images makes this attack mode more challenging. We call this mode of attack the *strong attack form*.

The second attack mode is when only the target image is specified. In that case, the attacker just wants to cause a vision contradiction during image scaling, as long as it is related to a certain concept (such as any images of sheep). In some extreme cases, the image content could be meaningless, e.g., just to create a negative result to an image classifier. Without a specific source image, the attacker's goal is to increase the dissimilarity before and after image scaling as much as possible. In this mode, the similarity constraints get relaxed and we call this mode of attack the *weak attack form*.

4.2 An Autonomous Approach on Attack Image Generation

We are interested to develop a method to automatically create scaling attack images in both the strong and weak attack forms. In order to achieve such goal, we first formalize the description of the data transition process among the four conceptual objects, and then we seek an algorithmic solution to create attack images.

The relationship between the four conceptual objects can be described in the following formulas.

First, the transition between *srcImg* and *attackImg* can be represented by a perturbation matrix Δ_1, and so does the difference between *outImg* and *targetImg*. These transition can be represented by

$$A_{m*n} = S_{m*n} + \Delta_1 \tag{1}$$

For the transition between *attackImg* and *outImg*, we consider the scaling effect as a function ScaleFunc(), which converts an $m*n$ input image A_{m*n} to an $m'*n'$ output image $D_{m'*n'}$ [6].

$$\text{ScaleFunc}(A_{m*n}) = D_{m'*n'} \tag{2}$$

ScaleFunc() is a surjective function, i.e. there exist multiple possible inputs A_{m*n} that all result in the same output $D_{m'*n'}$.

To perform a scaling attack, attackers need to craft an attack image A_{m*n}, which is the source image S_{m*n} plus a perturbation matrix Δ_1. In the meanwhile, the scaling result of the attack image, i.e., the output image $D_{m'*n'}$, needs to be visually similar with the target image $T_{m'*n'}$. Here we use Δ_2 to evaluate the difference between $D_{m'*n'}$ and $T_{m'*n'}$.

$$\begin{aligned} &A_{m*n} = S_{m*n} + \Delta_1 \\ &\text{ScaleFunc}(A_{m*n}) = D_{m'*n'} \\ &D_{m'*n'} = T_{m'*n'} + \Delta_2 \end{aligned} \tag{3}$$

Let us consider the strong attack form of scaling attack as in Fig.5, where both source S_{m*n} and target image $T_{m'*n'}$ are specified. The attacker's task is to find an attack image A_{m*n} being able to cause deceiving effect. Considering Eq.3, we can find multiple possible candidate matrices as solutions for A_{m*n} that satisfy the whole set of formulas. This is due to the surjection effect of ScaleFunc(). What the attacker wants to find is the matrix that produces the best deceiving effect among all possible solutions for A_{m*n}. One strategy is to find an A that is the most similar with S, while limiting the difference between D and T within an upper bound.

To find the best deceiving effect, we theoretically define an objective function that compares all solutions of A_{m*n}. To seek an algorithmic solution, we choose the L-norm distance[7] to capture the pixel-level differences as an approximation for measuring how close two images are.

In the strong attack form, we want to minimize the difference between A_{m*n} and S_{m*n}, and limit the difference between $D_{m'*n'}$ and $T_{m'*n'}$ within a threshold. Consequently, when the source image S_{m*n} and target image $T_{m'*n'}$ are given, the best result can be found by solving the following objective function.

$$\begin{aligned} &A_{m*n} = S_{m*n} + \Delta_1 \\ &\text{ScaleFunc}(A_{m*n}) = D_{m'*n'} \\ &D_{m'*n'} = T_{m'*n'} + \Delta_2 \\ &||\Delta_2||_\infty \leq \varepsilon * IN_{max} \\ &\text{Objective function} : \min(||\Delta_1||^2) \end{aligned} \tag{4}$$

where IN_{max} is the maximum pixel value in the current image format.

For the weak attack form, i.e. only the target image $T_{m'*n'}$ is given, what an attacker wants to optimize is to pick the attack image that visually has the largest difference from the target image. Thus, the best result should be found by solving the following objective function:

$$\begin{aligned} &R_{m*n} = \text{ScaleFunc}(T_{m'*n'}) \\ &A_{m*n} = R_{m*n} + \Delta_1 \\ &\text{ScaleFunc}(A_{m*n}) = D_{m'*n'} \\ &D_{m'*n'} = T_{m'*n'} + \Delta_2 \\ &||\Delta_2||_\infty \leq \varepsilon * IN_{max} \\ &\text{Objective function} : \max(||\Delta_1||^2) \end{aligned} \tag{5}$$

Notice that in the above constraints, we apply ScaleFunc() twice. The first call of ScaleFunc() is actually scaling the target image to the size of the attack image, i.e., upscaling an image from dimension $m'*n'$ back to $m*n$.

5 Creating Scaling Attack Images

After building up the formalized model of the scaling attack, in this section we investigate how to generate attack images automatically.

[6] Conventionally, we say a matrix of $m*n$ dimension has m rows and n columns, while an image of $m*n$ size consists of m columns and n rows. For convenience sake, in this paper we use a matrix X_{m*n} of $m*n$ dimension to refer to "an $m*n$ image".

[7] In this paper, $||\cdot||$ denotes the L^2-norm, while $||\cdot||_\infty$ denotes the L^∞-norm.

5.1 Empirical Analysis of Scaling Functions

In our implementation, we first need to find an appropriate expression of ScaleFunc(). We studied the implementation details of commonly used image processing packages. All of the scaling functions we studied perform the interpolation in two steps, one direction in each step. We design our attack algorithm with the assumption that the target scaling algorithm first resizes inputs horizontally and then vertically. Empirically the popular algorithms take this order (see Table 2, and more detailed analysis and examples are provided in Appendix C.). In case the scaling algorithm takes vertical order first, the attack method just needs to change accordingly. Hence, the ScaleFunc() in Eq.2 can be presented as:

$$\text{ScaleFunc}(X_{m*n}) = CL_{m'*m} * X_{m*n} * CR_{n*n'} \tag{6}$$

In Eq.6, $CL_{m'*m}$ and $CR_{n*n'}$ are two constant coefficient matrices determined by the interpolation algorithm, related to horizontal scaling ($m*n \rightarrow m*n'$) and vertical scaling ($m*n' \rightarrow m'*n'$), respectively.

With Eq.4 and Eq.6, we eventually build a relationship among the source image, the target image, and the perturbation:

$$\begin{aligned} &CL_{m'*m} * (S_{m*n} + \Delta_1) * CR_{n*n'} = D_{m'*n'} \\ &D_{m'*n'} = T_{m'*n'} + \Delta_2 \end{aligned} \tag{7}$$

5.2 Attack Image Crafting: An Overview

The main idea of automated scaling attack generation is to craft the attack image by two steps. The first step is to obtain the coefficient matrices in Eq.7. Section 5.3 presents a practical solution to find CL and CR, implemented as GetCoefficient(). The second step is to find the perturbation matrix Δ_1 to craft the attack image. We perform the attack image generation along each direction, in **reverse** order that we assume ScaleFunc() uses. Further, we decompose the solution of the perturbation matrix into the solution of a few perturbation vectors. By this way, we can significantly reduce the computational complexity for large size images. Section 5.4 provides more details of the second step, based on which we implement GetPerturbation() to find the perturbation vectors. Algorithm 1 and Algorithm 2 illustrate the attack image generation in the weak attack form and the strong attack form, respectively.

Weak attack form (Algorithm 1) [8]. First, we obtain the coefficient matrices by calling GetCoefficient() (line 2), which receives the size of S_{m*n} and $T_{m'*n'}$, and returns coefficient matrices $CL_{m'*m}$ and $CR_{n*n'}$, and then generate an intermediate source image $S^*_{m*n'}$ from $T_{m'*n'}$. Then, we call GetPerturbation(), which receives the column vectors from $S^*_{m*n'}$ and $T_{m'*n'}$, with the coefficient matrix CL and the object option ('max'), and returns the optimized perturbation matrix Δ^v_1, to solve the perturbation matrix column-wisely and craft out

[8] To clarify, here we use $X[i,:]$ and $X[:,j]$ to represent the i-th row and j-th column of matrix X respectively.

Algorithm 1 Image Crafting of the Weak Attack Form

Input: scaling function ScaleFunc(), target image $T_{m'*n'}$, source image size $(width_s, height_s)$, target image size $(width_t, height_t)$
Output: attack image A_{m*n}

1: $m = height_s$, $n = width_s$, $m' = height_t$, $n' = width_t$
2: $CL_{m'*m}$, $CR_{n*n'}$ = GetCoefficient(m, n, m', n')
3: $\Delta^v_1 = \mathbf{0}_{m*n'}$ ▷ Perturbation matrix of vertical attack.
4: $S^*_{m*n'}$ = ScaleFunc($T_{m'*n'}$) ▷ Intermediate source image.
5: **for** $col = 0$ to $n'-1$ **do**
6: $\Delta^v_1[:,col]$ = GetPerturbation($S^*[:,col]$, $T[:,col]$, CL, obj='max') ▷ Launch the vertical scaling attack.
7: **end for**
8: $A^*_{m*n'}$ = unsigned int($S^* + \Delta^v_1$)
9: S_{m*n} = ScaleFunc($T_{m'*n'}$) ▷ Final source image.
10: $\Delta^h_1 = \mathbf{0}_{m*n}$ ▷ Perturbation matrix of horizontal attack.
11: **for** $row = 0$ to $m-1$ **do**
12: $\Delta^h_1[row,:]$ = GetPerturbation($S[row,:]$, $A^*[row,:]$, CR, obj='max') ▷ Launch the horizontal scaling attack.
13: **end for**
14: A_{m*n} = unsigned int($S + \Delta^h_1$)
15: **return** A_{m*n} ▷ Get the crafted attack image.

$A^*_{m*n'}$ (line 5 to 8). Similarly, we solve another perturbation matrix Δ^h_1 row-wisely and construct the final attack image A_{m*n} (line 9 to 15).

Strong attack form (Algorithm 2). The strong attack form follows a similar procedure, except of two parts different from the weak attack form: The first one is that the input in this form includes two independent images S_{m*n} and $T_{m'*n'}$, while the second one is that the optimization problem transforms from maximizing the object function into minimizing the object function (line 6 and line 11).

5.3 Coefficients Recovery

Here we investigate the design of GetCoefficient() function, i.e., how does an attacker obtain the coefficient matrix $CL_{m'*m}$ and $CR_{n*n'}$.

For public image preprocessing methods/libraries, the attacker is able to acquire the implementation details of ScaleFunc(). Hence, in theory, the attacker can compute each element in $CL_{m'*m}$ and $CR_{n*n'}$ precisely.

Eq.8 is a coefficient recovery result from the open-source package Pillow. In the bilinear interpolation algorithm, the coefficient matrices from 4*4 image to 2*2 image are:

$$CL_{m'*m} = \begin{bmatrix} \frac{3}{7} & \frac{3}{7} & \frac{1}{7} & 0 \\ 0 & \frac{3}{7} & \frac{3}{7} & \frac{1}{7} \end{bmatrix}, \quad CR_{n*n'} = \begin{bmatrix} \frac{3}{7} & 0 \\ \frac{3}{7} & \frac{1}{7} \\ \frac{1}{7} & \frac{3}{7} \\ 0 & \frac{3}{7} \end{bmatrix} \tag{8}$$

Though it is possible to retrieve coefficient matrices precisely, the pre-mentioned procedure may become challenging

Algorithm 2 Image Crafting of the Strong Attack Form

Input: scaling function ScaleFunc(), source image S_{m*n}, target image $T_{m'*n'}$, source image size $(width_s, height_s)$, target image size $(width_t, height_t)$
Output: attack image A_{m*n}

$m = height_s$, $n = width_s$, $m' = height_t$, $n' = width_t$
$CL_{m'*m}$, $CR_{n*n'}$ = GetCoefficient(m, n, m', n')
$\Delta_1^v = \mathbf{0}_{m*n'}$ ▷ Perturbation matrix of vertical attack.
$S^*_{m*n'}$ = ScaleFunc(S_{m*n}) ▷ Intermediate source image.
for $col = 0$ to $n' - 1$ **do**
  $\Delta_1^v[:, col]$ = GetPerturbation($S^*[:, col]$, $T[:, col]$, CL, obj='min') ▷ Launch the vertical scaling attack.
end for
$A^*_{m*n'}$ = unsigned int($S^* + \Delta_1^v$)
$\Delta_1^h = \mathbf{0}_{m*n}$ ▷ Perturbation matrix of horizontal attack.
for $row = 0$ to $m - 1$ **do**
  $\Delta_1^h[row, :]$ = GetPerturbation($S[row, :]$, $A^*[row, :]$, CR, obj='min') ▷ Launch the horizontal scaling attack.
end for
A_{m*n} = unsigned int($S + \Delta_1^h$)
return A_{m*n} ▷ Get the crafted attack image.

when the coefficient matrices grow large and the interpolation method becomes complex. To reduce the human effort for extracting the coefficient values, we introduce an easy approach to deduce the those matrices. The idea is to infer these coefficient matrices from input and output pairs.

First, we establish the following equation:

$$\begin{aligned} &CL_{m'*m} * (I_{m*m} * IN_{max}) = CL_{m'*m} * IN_{max} \\ &(I_{n*n} * IN_{max}) * CR_{n*n'} = CR_{n*n'} * IN_{max} \end{aligned} \tag{9}$$

where I_{m*m} and I_{n*n} are both identity matrices.

Then, if we set the source image $S = I_{m*m} * IN_{max}$ and scale it into an $m' * m$ image $D_{m'*m}$, we can obtain

$$\begin{aligned} &D = \text{ScaleFunc}(S) = \text{unsigned int}(CL_{m'*m} * IN_{max}) \\ &\rightarrow CL_{m'*m(appr)} \approx D / IN_{max} \end{aligned} \tag{10}$$

In the theoretical formulation, the sum of elements in each row of $CL_{m'*m}$ should be equal to one.

Finally, we do the normalization for each row (Eq.11). In fact, the type cast from float-point values to unsigned integers in Eq.10 will cause a slight precision loss. What we acquired is an approximation of $CL_{m'*m}$, but in practice our experimental results show that the precision loss can be ignored.

$$\begin{aligned} &CL_{m'*m(appr)}[i, :] = \frac{CL_{m'*m(appr)}[i, :]}{\sum_{j=0}^{m-1}(CL_{m'*m(appr)}[i, j])} \\ &(i = 0, 1, ..., m' - 1) \end{aligned} \tag{11}$$

The inference of $CR_{n*n'}$ follows a similar procedure. When scaling $S' = I_{n*n} * IN_{max}$ into $D'_{n*n'}$, we have

$$\begin{aligned} &D' = \text{ScaleFunc}(S') = \text{unsigned int}(IN_{max} * CR_{n*n'}) \\ &\rightarrow CR_{n*n'(appr)} \approx D' / IN_{max} \end{aligned} \tag{12}$$

Hence, we can obtain the estimated CR:

$$\begin{aligned} &CR_{n*n'(appr)}[:, j] = \frac{CR_{n*n'(appr)}[:, j]}{\sum_{i=0}^{n-1}(CR_{n*n'(appr)}[i, j])} \\ &(j = 0, 1, ..., n' - 1) \end{aligned} \tag{13}$$

So far, we have found a practical approach to recover coefficient matrices. In the next step, we focus on constructing the perturbation matrix Δ_1.

5.4 Perturbation Generation via Convex-Concave Programming

In the threat model established in Section 4.2, Δ_1 is a matrix with dimension $m * n$. The optimization problem tends to be complex when the attack image is large. This part illustrates how to simplify the original problem and find the perturbation matrix efficiently.

5.4.1 Problem Decomposition

Generally speaking, the complexity of an n-variable quadratic programming problem is no less than $O(n^2)$, as it contains complex computation operations, such as solving the Hessian matrix. The optimization is computationally expensive when the image size grows large. Here we simplify and accelerate the image crafting process by two feasible steps, only sacrificing the computing precision slightly.

Firstly, we separate the whole scaling attack into two subroutines. The image resizing in each direction is equivalent, because the resizing of S in the vertical direction can be regarded as the resizing of the source image's transpose S^T in the horizontal direction. Therefore, we only need to consider how to generate Δ_1 in one direction (here we choose the vertical resizing as the example). Suppose we have an input image S_{m*n} and an target image $T_{m'*n}$, and we have recovered the resizing coefficient matrix $CL_{m'*m}$, with the aim to craft the attack image $A_{m*n} = S_{m*n} + \Delta_1$.

Secondly, we decompose the calculation of the perturbation matrix into the solution of a few vectors. In fact, the image transformation can be rewritten as:

$$\begin{aligned} &CL_{m'*m} * A = \\ &\left[\ CL * A[:, 0]_{(m*1)} \quad ... \quad CL * A[:, n-1]_{(m*1)}\ \right] \end{aligned} \tag{14}$$

In this way, our original attack model has been simplified into several column-wise sub optimization problems:

$$\begin{aligned} &\textbf{obj: } \min/\max(||\Delta_1[:, j]||^2) \\ &\textbf{s.t. } CL * A[:, j]_{(m*1)} = T[:, j]_{(m'*1)} + \Delta_2 \\ &\quad ||\Delta_2||_\infty \leq \varepsilon * IN_{max} \\ &(j = 0, 1, ..., n - 1) \end{aligned} \tag{15}$$

5.4.2 Optimization Solution

We formulate our model in Eq.15 into a standard quadratic optimization problem.

Constraints. First there is a natural constraint that each element in the attack image A should be within $[0, IN_{max}]$. We have the following constraints:

$$\begin{aligned} &0 \leq A[:,j]_{m*1} \leq IN_{max} \\ &||CL*A[:,j]_{m*1} - T[:,j]_{m'*1}||_\infty \leq \varepsilon * IN_{max} \end{aligned} \tag{16}$$

Objective function. Our objective function is also equivalent to

$$\min/\max(\Delta_1[:,j]^T I_{m'*m'} \Delta_1[:,j]) \ (j = 0,1,...,n-1) \tag{17}$$

where $I_{m'*m'}$ is the identity matrix. Then, combining the objective function (Eq.17) and constraints (Eq.16), we finally obtain an m'-dimensional quadratic programming problem with inequality constraints.

Problem Solution. Back to the two attack models proposed in section 4.2, the strong attack model refers to a convex optimization problem while the weak model refers to a concave optimization problem. We adopt the Disciplined Convex-Concave Programming toolbox *DCCP* developed by Shen et al. [33] to solve the optimization problem. The results exhibited in Appendix D validate that this approach is feasible.

6 Experimental Results of Scaling Attack

In this section, we report attack results on three kinds of applications: local image classification applications, cloud computer vision services and web browsers.

6.1 White-box Attack Against Local Computer Vision Implementations

Many computer vision applications expose the model's input size and scaling algorithm to attackers. We regard this scenario as our white-box threat model.

White-box Threat Model. In our white-box threat model, we assume that the attacker has full knowledge of the target application's required input size and the scaling algorithm implementation. This can be achieved by inspecting the source codes, reverse engineering the application, or speculating based on open information. For instance, there is an image classification application claiming that it is built upon Caffe and uses the GoogleNet model. The attacker can ensure the input size is 224*224 (Table 1), and guess that the OpenCV.Bilinear (default for Caffe, see Table 2) is the scaling function with a high confidence. With the automatic attack image generation approach proposed in Section 5, the attacker can achieve the deceiving effect without much effort in the white-black threat model.

Results. We validate our attack image generation approach on the interpolation algorithms built within three popular deep learning frameworks: Caffe, Tensorflow, and Torch. For each framework, we write an image classification demo based on the BAIR/BVLC GoogleNet model, whose required input size is 224*224. We launch the attack with a 672*224 sheep-like image as the source image, and a 224*224 wolf-like image as the target image, under a tight constraint where we set $\varepsilon = 0.01$. If the generated attack image satisfies the constraints and deceives the application, we consider the attack is successful, and otherwise it fails. The results reported in Table 2 show that our attack method is effective for all the default scaling algorithms in these frameworks.

Notice that our approach does not generate successful attack images for some less commonly used algorithms. There are two factors affecting these attacks. First, some of these algorithms might pose more constraints during the scaling process. And because they are not popularly used, we have not yet studied the detail of these implementations. Second, in this paper, we only applied a tight constraint on our optimization task (Eq.16 and Eq.17), for the purpose of threat demonstration. There is a trade-off between the deceiving effect and image generation difficulty. Even if the automatic image generation process fails for some algorithms, by no means these algorithms should be considered as safe.

6.2 Black-box Attack Against Cloud Vision Services

Cloud-based computer vision services, provided by Microsoft, Baidu, Tencent and others, have arisen broadly, which simplify the deployment of computer vision systems. By sending queries to these powerful cloud vision APIs, users can obtain detailed image information, e.g., tags with confidence values and image descriptions. In this case, the pre-trained models are usually packaged as black boxes isolated from users, and users only are able to access these services through APIs. This section shows that the commercial cloud vision services are threatened by the scaling attack, even in the black-box scenario where the input size and scaling method are unknown.

Black-box Threat Model. In our black-box threat model, the goal of an attack is to deceive the image recognition service running on the cloud server, resulting in a misrecognition for input images. But the input scale and scaling method is unknown to the attacker, making the attack more challenging.

6.2.1 Attack Roadmap

The attack against black-box vision services mainly includes two steps. The first step is scaling parameter inference – the attacker estimates the input size and scaling algorithm used by the classification model. The second step is to craft attack images based on the inferred scaling parameters.

Scaling Parameter Inference. We design the scaling parameter inference strategy based on two empirical observations. First, from Table 1 we can see that for most commonly used CNN models, the input is a square-sized image with a side length in the range of [201,300]. Second, by comparing and analyzing the source codes of popular DL frameworks in Table 2, we found the most commonly used default scaling algorithm is Nearest, Bilinear, or Bicubic. Therefore, a naive approach by the adversary is to infer the scaling parameters via exhaustive tests. An adversary can send a series of m probing images $\{probeImg_i\}, (i = 1, 2, ..., m)$, crafted by the scaling attack method with various scaling parameters. The attacker can infer the scaling parameters by watching the classification results. If one query returns with the correct classification result, the corresponding scaling parameters are likely to be used by the target service. Then the attacker can try to launch the attack with the inferred parameters. This procedure is shown in Algorithm 3.

The inference efficiency can be increased by using a complex attack image involving several sub-probing images. These sub-probing images can only be recovered with their corresponding scaling parameters.

Here we show one simple approach to achieve this goal. First, the attacker collects n sub-probing images each belonging to different categories, and determines the input size range *SizeRange* and the scaling algorithms *AlgSet* to test. The search space $\mathcal{S}$ can be defined as $\mathcal{S} = \{\mathcal{S}_i\} = SizeRange \times AlgSet = \{size_i\} \times \{alg_i\}$. Second, the attacker chooses a large blank white image (with #FF as the pixel value) as the background, and divides it into n non-overlapped *probing regions*. Third, the attacker repeats the following procedure: he/she fills the j-th probing region of the blank image with the j-th sub-probing image respectively, next scales it with the scaling parameter $\mathcal{S}_i^j$, and then conducts the scaling attack with the original blank image as the *sourceImg* and the resized image as the *targetImg*. Finally, the attacker combines all the output images to create the $probeImg_i$. In this way, when $probeImg_i$ is resized, the j-th sub-probing image will be recovered if and only if the scaling parameter is set as $\mathcal{S}_i^j$. Fig.6 gives an example of the *probeImg*. Note that the larger n is, the fewer *probeImg*s are needed, but the recognition accuracy of sub-probing images might be reduced as the area of each probing region decreases.

Image Crafting. After retrieving the possible input size and scaling algorithm, the adversary can generate attack images as described in Section 5, and launch the scaling attack to cloud vision services .

Algorithm 3 Scaling Parameter Inference

Input: cloud vision API f, scaling algorithms $AlgSet = \{alg_1, alg_2, ...\}$, input size range $SizeRange = \{size_1, size_2, ...\}$, source image *sourceImg*, target image *targetImg*
Output: the inferred input size $size^*$ and scaling algorithm alg^*

```
for alg in AlgSet do
  for size in SizeRange do
    testImg = resize(targetImg, size)
    probeImg = ScalingAttack(alg, sourceImg, testImg) ▷ Can be recovered once resized into size by alg.
    if argmax(f(testImg)) == argmax(f(probeImg)) then
      Return size, alg                ▷ Get a feasible answer.
    end if
  end for
end for
Return NULL, NULL                     ▷ No match during the search.
```

6.2.2 Results

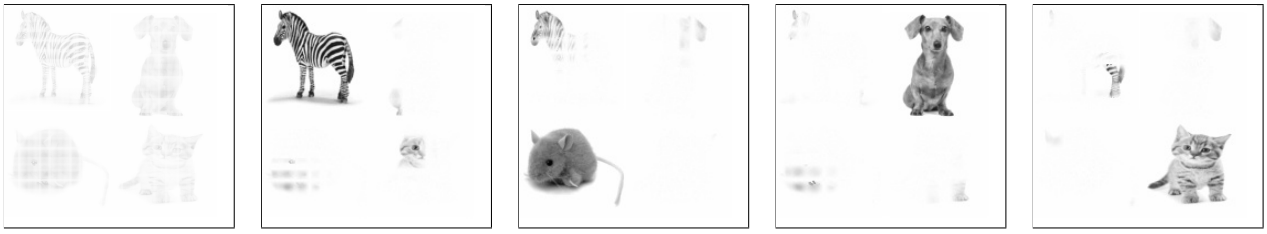
To show the feasibility of the scaling attack against black-box cloud vision services, we choose Microsoft Azure, Baidu, Aliyun, and Tencent cloud vision services as our test beds[9].

In our experiment, each *probeImg* contains four sub-images (classified as "zebra", "dog", "rat" and "cat") for different input parameters. For the input size, the *SizeRange* is set from 201 to 300, while the scaling algorithm options include two libraries OpenCV and Pillow with Nearest, Bilinear and Cubic interpolation methods. Considering the trade-off between efficiency and recognition accuracy, we set $n = 4$. Hence, the total amount of queries is 100 (#input size) * 2 (#scaling library) * 3 (#interpolation method) / 4 (#probing region) = 150 (#*probeImg*). As we can see, the searching space is extremely small and it consumes just up to several minutes to obtain the results. We provide the scaling parameter inference results and one scaling attack sample in Table 3.

Moreover, to verify the effectiveness of the proposed attack strategy, we collected 935 images from Internet, including 17 categories except of sheep or sheep-like animals, and cropped them into the 800*600 size holding the main object, as our *sourceImg* dataset. Then for each of the 935 *sourceImg*s, we generated one attack image with the same *targetImg* containing a sheep in the center, setting the scaling attack parameter $\varepsilon = 0.01$.

For Baidu, Aliyun and Tencent, all the attack images are classified as "sheep" or "goat" with the highest confidence value compared with other classes, while for Azure the result becomes more complex. In our experiment we requested the Azure cloud vision service API to respond with four fea-

[9] As part of the responsible disclosure etiquette, we have reported this issue and received replies from these companies. The latter three have confirmed this problem as are now in the process of fixing it. Microsoft Azure has also acknowledged the issue and is discussing with us about possible solutions.



(a) *probeImg* (1024*1024) (b) Scaled result (OpenCV.Bilinear, 201*201) (c) Scaled result (OpenCV.Bilinear, 202*202) (d) Scaled result (OpenCV.Bilinear, 203*203) (e) Scaled result (OpenCV.Bilinear, 204*204)

Figure 6: An example of the probeImg. (a) is a *probeImg* containing 4 subfigures, and (b) to (e) are the results when the *probeImg* is scaled under different scaling settings.

tures: "description", "tags", "categories", and "color". We find that for attack images, the word "sheep" may appear in the "description" or "tags" with a confidence value. Hence, we computed the CDF (cumulative distribution function) of attack images' confidence values of "tags" and "description" respectively, and plot the two CDF curves in Fig.7 (we assume the confidence value as 0 if "sheep" is absent in the API response). The result shows that for the "tags" feature, more than 60% attack images are classified as "sheep" with a confidence value higher than 0.9, which implies the effectiveness of our proposed attack.

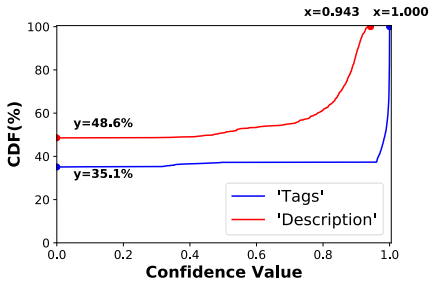


Figure 7: The CDF curve of responses from Azure.

6.3 Deceiving Effect on Web Browsers

Web browsers provide the page zooming function to scale the contents, including texts and images. Hence, an attacker may be able to utilize scaling functions in web browsers to achieve deceiving or phishing attacks.

We have evaluated such effect on several mainstream browsers running on different platforms. We generated an attack image (with a 672*224 sheep image as the source image, a 224*224 wolf image as the target image, using OpenCV.Bilinear as the scaling method, $\varepsilon = 0.01$), and used HTML tags to control its rendering size in browsers. The result is presented in Table 4, indicating the potential victims of scaling attack are beyond the scope of deep learning computer vision applications. One potential problem is that scaling attacks can cause inconsistency between different screen resolutions, when the browser's auto/adaptive-zooming function is enabled.

6.4 Factors that Might Interfere with Scaling Attacks

Image processing applications often contain a complex preprocessing pipeline. Besides scaling, an image processing applications might use cropping, filtering, and various other image transformation actions. If these additional image preprocessing actions occur prior to the scaling action, they might pose additional challenges to scaling attacks.

The following list presents an overview of common image preprocessing actions and discusses their potential impact on scaling attacks.

- **Cropping** – truncate certain regions of the input image, for the purpose of data augmentation or background removing. Cropping usually changes the source image aspect ratio, and if a scaling attack was designed under a wrong dimension, the automatic generated image would not scale to the right target image. Therefore, attackers need to know precisely which region in the input is expected to be cropped. Only under some special cases, such as the cropping preserves the aspect ratio and the underneath algorithm is Nearest, deceiving effect can be preserved. Certainly the degree of impact also depends on the relative size being cropped. If the pixels that are used to generate the targeted image are chopped, then the effect of scaling attack is definitely affected.

- **Filtering** – is to blur or sharpen an image, adjust its color palette. Image filtering changes the pixel values and thus directly interferes with scaling attacks, because the attack is based on the manipulation of "average" values of neighbor pixels used by the interpolation algorithms. For simple scaling algorithms, such as Nearest, the output image might still present deceiving effect as the result is

Table 3: Deceiving effect on four cloud vision services.

Service	Azure	Baidu	Aliyun	Tencent
Inferred Scale	227*227	256*256	224*224	224*224
Inferred Algorithm	OpenCV.Bilinear	OpenCV.Bicubic	OpenCV.Bilinear	OpenCV.Bilinear
Attack Image				
Response	"captions": { **"text": "a close up of a wolf", "confidence": 0.707954049** } } Tags: ... { **"name": "wolf", "confidence": 0.981169641**}...	... "result": { **"score": "0.938829", "name": "Grey Wolf"** }, { "score": "0.0146997", "name": "Mexico Wolf" }...	... "Object": { **"Grey Wolf": "49.37%"**, "White Wolf": "29.93%",...}	... "Tags": { **"Grey Wolf": "88%"**, "Eskimo": "15%"}...

Table 4: Proof-of-Concept sample image and the rendering effect under different browser settings. (The HTML file uses an IMG tag to specify the image rendering size.)

Browsers	Original Image	Firefox, Edge	IE11	Chrome	Safari
Image					
Size	672*224	224*224	224*224	224*224	224*224
Version	Firefox: 59.0.2, IE: 11.0.9600.18977 Chrome: 63.0.3239.84, Edge: 41.16299.371.0 Safari: 8.0 (10600.1.25.1)	Firefox: 59.0.2 Edge: 41.16299.371.0	IE: 11.0.9600.18977	Chrome: 63.0.3239.84	Safari: 8.0 (10600.1.25.1)

like the original target with a filtering effect. However, for complex scaling algorithms, such as Bilinear and Bicubic, the output image will likely not present as the intend target image.

- **Affine transformations** – is to rotate or mirror the input image. Rotation in an arbitrary degree likely breaks the calculation used by the automatic attack image crafting. However, flipping images in 180 degree, mirror images might have no impacts on the scaling attack which mainly depends on the size of the inputs and the scaling algorithms. Some scaling algorithms are orientation independent, i.e. the output is same regardless the scan of pixels is from left to right or the opposite order. In those cases, a flip or mirror action would not affect scaling attacks.

- **Color transformations** – to change the color space, like convert an RGB image to grayscale. Color transformation can be considered as a special type of filtering, and thus the impact to scaling attack is similar to filtering.

Although the above transformation actions all directly interfere with scaling attacks, the interference can be overcome by the attackers if they know these transformation details. In fact, each of these operations can be described by a transformation matrix. Once an attacker ensures the exact content of the transformation matrix and if there exists a corresponding reverse transformation matrix, the attacker can applies the reverse matrix to generated attack image before feeding it to the targeted application. In the black-box case, the attacker has to infer the transformation matrix. Therefore, these transformation actions would increase the attack difficulty.

The deceiving effect of scaling attacks is also subject to some native limitations, especially size and brightness, of source and target images. An attacker needs to find an appropriate pair of the source and target images to achieve a successful attack image.

- **Size:** Sizes of the source and target images decide how many redundant pixels can be leveraged to launch the attack. If the size differences between scaling input and output are very close, the information attenuation due to resampling may be insufficient to achieve a successful deceiving effect.

- **Brightness:** Brightness or color of the source and target images decides how tight the constraints are. In the worst case, it is hard to find a feasible solution given a full white source and a full black target. Even we generate an attack image successfully, it is hard to deceive human without noticing dark dots distributed in the white image.

6.5 Practical Attack Scenarios

This paper presents the risk of scaling attacks through a set of limited experiments with proof-of-concept images. We have shown these proof-of-concept images can achieve deceiving effect in deep-learning based image applications, web

browsers, as well as cloud-based visual recognition and classification services. Although these proof-of-concept images, such as the wolf-in-sheep set, do not cause any real damage, we believe the risks of scaling attacks are real. This section describes a few motivating scenarios to illustrate possible real life threats.

- **Data poisoning.** Many image based applications rely on label training sets and there are many large image datasets, such as ImageNet [12], on the Internet. Many deep learning developers rely on these datasets to train their models. Although data poisoning as a concept is known, developers and model trainers rarely consider data poisoning is a real threat on these public datasets since these datasets are public, and humans are expected to notice obvious genre mistakes and a large set of mislabels. However, with scaling attacks, people with malicious intent could conceal a hidden category of images (e.g. wolf) while providing mistaken labels as another category (e.g. sheep). We do not have evidence of such activities, but we envision that scaling attacks definitely make data poisoning more stealthy.

- **Detection evasion and Cloaking.** Content moderation is one of the most widely used computer vision applications. Many vendors provide content filtering services, such as Google [11], Amazon AWS [3] and Microsoft Azure [4]. ModerateContent claims that it is trusted by 1000's of sites to prevent offensive content [23]. An attacker may leverage the scaling attack to evade these content moderators to spread inappropriate images, which may raise serious problems in online communities. For example, suppose an attacker wants to advertise illegal drugs to users on the iPhone XS. The attacker can use the scaling attack to create a cloaking effect, so that the scaling result on the iPhone XS browser is the intended drug image while the image in the original size contains benign content. Certainly cloaking can also be achieved by using other approaches such as browser sensitive Javascript. However, scaling attacks create an alternative approach as no additional code is used to manage the rendering effect.

- **Fraud by Leveraging Inconsistencies between Displays.** An attacker can create a deceptive digital contract using the scaling attacks. An attacker can create an image document that contains a scanned contract but renders to different content when scaling to different ratios. The attacker can then get two parties to share the same document. If they each use different browsers, the content being displayed will be different. This inconsistency can become the basis of potential financial fraud activities.

7 Countermeasures

In this section, we discuss potential defense strategies to mitigate the threat from scaling attacks. First, we discuss possible countermeasures as the attack prevention in the image preprocessing procedure. Second, we discuss some approaches to detect scaling attacks.

7.1 Attack Prevention

A naive way to avoid the scaling attack is to omit inputs whose sizes are different from the input size used by the deep learning models. This approach is appropriate for applications that deal with the inputs collected by sensors in specific formats. However, this strategy is infeasible for many Internet services, since the input images uploaded by users are often in various sizes.

Another solution is that we can randomly remove some pixels (by line or by column) from the image before scaling it. This random cropping operation makes the scaling coefficient matrices unpredictable, and therefore, it can increase the attack difficulty effectively. However, we should carefully design the pixel removing policy to maintain the image quality.

7.2 Attack Detection

The scaling attack achieves the deceiving effect by causing dramatic changes in visual features before and after the scaling action. One potential solution is to detect such obvious changes of input features during the scaling process, such as the color histogram and the color scattering distribution.

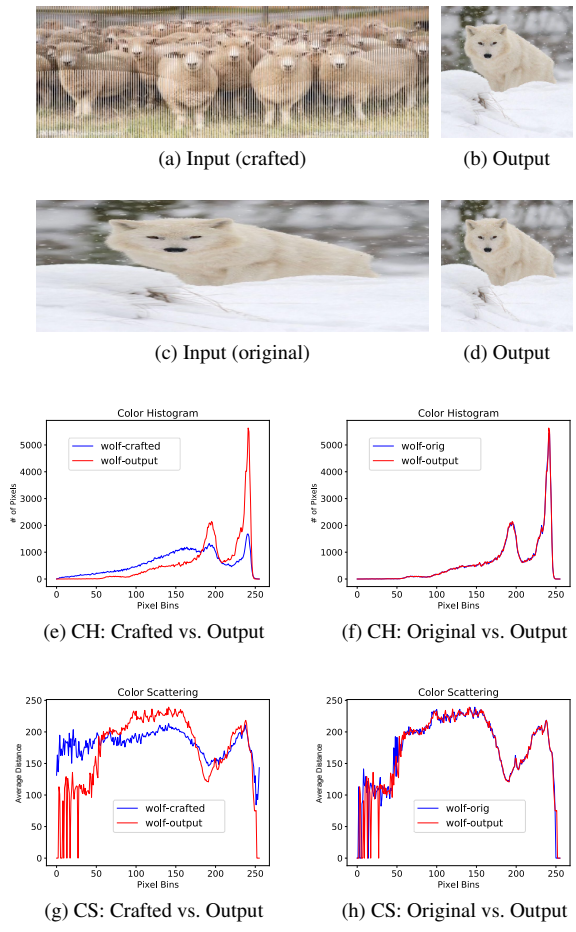
7.2.1 Color-histogram-based Detection

The color histogram counts the amount of pixels for color ranges of a digital image. It presents the color distribution in an image, and is commonly used as a measurement of image similarity. The main advantage of the color-histogram-based detection approach is that it can measure the color distribution change easily and quickly. It is a simple solution when the data processing speed is the main concern, especially when the system throughput is high. In our experiments, we convert the image into grayscale to examine the effectiveness of color-histogram-based detection, i.e., pixel values ranging from 0 to 255. Eventually, the color histogram of one image can be represented as a 256-dimension vector v^{his}, and we adopt the cosine similarity to measure the color-histogram similarity of two images $s^{his} = \cos(v_1^{his}, v_2^{his})$.

7.2.2 Color-scattering-based Detection

The color-histogram-based detection can only present a rough distribution of pixel values, disregarding the color spatial distribution information. The color scattering could become a

supplementary to the histogram, which presents the color distribution measured with the distance between pixels and the image center. In our experiments, we also convert the image to grayscale to evaluate the effectiveness the color-scattering-based detection approach. Specifically, we calculate the distance histogram as the color scattering measurement, and define a statistical metric to evaluate the similarity: First, we compute the average distance from pixels which belong to the same pixel value to the center of the image and we present the result with a 256-dimension color scattering vector v^{scat}. Second, we calculate the cosine similarity between vectors of two images as the color-scattering-based similarity $s = \cos(v_1^{scat}, v_2^{scat})$.



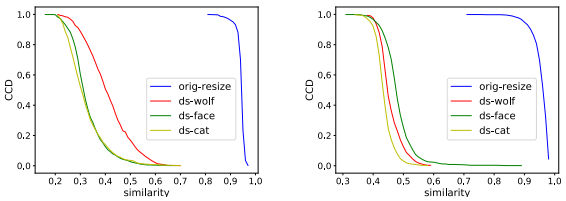
(a) Input (crafted) (b) Output

(c) Input (original) (d) Output

(e) CH: Crafted vs. Output (f) CH: Original vs. Output

(g) CS: Crafted vs. Output (h) CS: Original vs. Output

Figure 8: The color histograms and color scattering detection results of the scaling attack in the *wolf-in-sheep* example. (CH: color histogram, CS:color scattering)



(a) Color Histograms (b) Color Scattering

Figure 9: The CCD of color histogram similarity and color scattering similarity detection.

7.2.3 Evaluation

To evaluate the performance of two attack detection strategies, we have crafted three attack images for each *sourceImg* in the dataset established in Section 6.2, with three 224*224 target images belong to the *wolf*, *human face* and *cat* category. Before the similarity comparison, we resize the output to the same size with the input, in order to eliminate the difference in pixel amount. Fig.8 exhibits the detecting result of a *wolf-in-sheep* attack image.

Fig.8e and Fig.8f present the comparison of grayscale histograms between the input images and their scaled output. The x-axis refers to pixel values ranging from 0 to 255, while the y-axis refers to the number of pixels with the same value. From Fig.8f, we can see that the two curves of the original input and its scaling output almost coincide, where the similarity is 0.96. In the meanwhile, we can see an obvious difference between the color distribution of the attack image and its scaling output, where the similarity is 0.50.

Fig.8g and Fig.8h present the comparison of grayscale color scattering measurement. The x-axis refers to pixel values ranging from 0 to 255, while the y-axis refers to the average distance between the image center and pixels with the same value. Similarly, we can see an obvious difference in the color scattering measurement of the attack image and its scaling output.

Fig.9 reports the complementary cumulative distribution (CCD) of the detection results of our test set. The legend "original-resize", "ds-wolf", "ds-face" and "ds-cat" refer to the original-image, wolf-as-target, human-face-as-target and cat-as-target case, respectively. We can observe that for both two detection metrics, the similarity between original images and their scaling outputs is obviously higher than that between attack images and their scaling outputs. The result indicates the two attack detection strategies work well in most cases.

8 Conclusion

This paper presents a camouflage attack on image scaling algorithms, which is a potential threat to computer vision ap-

plications. By crafting attack images, the attack can cause the visual semantics of images change significantly during scaling. We studied popular deep learning frameworks and showed that most of their default scaling functions are vulnerable to such attack. Our results also exhibit that even though cloud services (such as Microsoft Azure, Baidu, Aliyun and Tencent) hide the scaling algorithms and input scales, attackers can still achieve the deceiving effect. The purpose of this work is to raise awareness of the security threats buried in the data processing pipeline in computer vision applications. Compared to the intense interests in adversarial examples, we believe that the scaling attack is more effective in creating misclassification because of the deceiving effect it can create.

Acknowledgments

We thank our shepherd Dr. David Wagner and all anonymous reviewers for their insightful suggestions and comments to improve the paper; Dr. Jian Wang and Dr. Yang Liu for feedback on early drafts; Deyue Zhang and Wei Yin for collecting the data. We also thank all members of 360 Security Research Labs for their support. Among all the contributors, Dr. Chao Shen (chaoshen@mail.xjtu.edu.cn) and Dr. Yu Chen (yuchen@mail.tsinghua.edu.cn) are the corresponding authors. Tsinghua University authors are supported in part by the National Natural Science Foundation of China (Grant 61772303), National Key R&D Program of China (Grant 2017YFB0802901). Xi'an Jiaotong University authors are supported in part by the National Natural Science Foundation of China (Grant 61822309, 61773310, and U1736205), the Natural Science Foundation of Shaanxi Province (Grant 2019JQ-084).

References

[1] adamlerer and soumith. ImageNet training in PyTorch. http://github.com/pytorch/examples/tree/master/imagenet, 2017.

[2] Adnan M. Alattar. Reversible watermark using the difference expansion of a generalized integer transform. *IEEE Transactions on Image Processing*, 13(8):1147–1156, Aug 2004.

[3] Amazon AWS. Detecting unsafe image. https://docs.aws.amazon.com/rekognition/latest/dg/procedure-moderate-images.html.

[4] Microsoft Azure. Content moderator. https://azure.microsoft.com/en-us/services/cognitive-services/content-moderator/.

[5] beniz. Deep Learning API and Server in C++11 with Python bindings and support for Caffe, Tensorflow, XGBoost and TSNE. https://github.com/beniz/deepdetect, 2017.

[6] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, L D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv: Computer Vision and Pattern Recognition*, 2016.

[7] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[8] BVLC. BAIR/BVLC GoogleNet Model. http://dl.caffe.berkeleyvision.org/bvlc_googlenet.caffemodel, 2017.

[9] M. U. Celik, G. Sharma, A. M. Tekalp, and E. Saber. Lossless generalized-lsb data embedding. *IEEE Transactions on Image Processing*, 14(2):253–266, Feb 2005.

[10] Alex Clark and Contricutors. Pillow: The friendly Python Imaging Library fork. https://python-pillow.org/, 2018.

[11] Google Cloud. Filtering inappropriate content with the cloud vision api. https://cloud.google.com/blog/products/gcp/filtering-inappropriate-content-with-the-cloud-vision-api.

[12] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009.

[13] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015.

[14] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and Harnessing Adversarial Examples. *ArXiv e-prints*, December 2014.

[15] Jie Hu and Tianrui Li. Reversible steganography using extended image interpolation technique. *Computers and Electrical Engineering*, 46:447–455, 2015.

[16] Yangqing Jia. Classifying ImageNet: using the C++ API. https://github.com/BVLC/caffe/tree/master/examples/cpp_classification, 2017.

[17] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[18] Ki Hyun Jung and Kee Young Yoo. Data hiding method using image interpolation. *Computer Standards and Interfaces*, 31(2):465–470, 2009.

[19] Devendra Kumar. REVERSIBLE DATA HIDING USING IMPROVED INTERPOLATION. pages 3037–3048, 2017.

[20] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *CoRR*, abs/1607.02533, 2016.

[21] Chin Feng Lee and Yu Lin Huang. An efficient image interpolation increasing payload in reversible data hiding. *Expert Systems with Applications*, 39(8):6712–6719, 2012.

[22] Ian Markwood, Dakun Shen, Yao Liu, and Zhuo Lu. PDF mirage: Content masking attack against information-based online services. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 833–847, Vancouver, BC, 2017. USENIX Association.

[23] ModerateContent. Realtime image moderation api to protect your community. https://www.moderatecontent.com/.

[24] NVIDIA developers. the latest products and services compatible with the DRIVE Platform of NVIDIA's ecosystem . https://developer.nvidia.com/drive/ecosystem, 2017.

[25] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, October 2017.

[26] Ronan, Clément, Koray, and Soumith. Torch: A SCIENTIFIC COMPUTING FRAMEWORK FOR LUAJIT. http://torch.ch/, 2017.

[27] Mingwei Tang, Jie Hu, Wen Song, and Shengke Zeng. Reversible and adaptive image steganographic method. *AEU - International Journal of Electronics and Communications*, 69(12):1745–1754, 2015.

[28] Tensorflow developers. TensorFlow C++ and Python Image Recognition Demo. https://www.github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/label_image, 2017.

[29] Torch developers. Tutorials for Torch7. http://github.com/torch/tutorials/tree/master/7_imagenet_classification, 2017.

[30] Xing-Tian Wang, Chin-Chen Chang, Thai-Son Nguyen, and Ming-Chu Li. Reversible data hiding for high quality images exploiting interpolation and direction order mechanism. *Digital Signal Processing*, 23(2):569 – 577, 2013.

[31] H. Wu, J. Dugelay, and Y. Shi. Reversible image data hiding with contrast enhancement. *IEEE Signal Processing Letters*, 22(1):81–85, Jan 2015.

[32] Qixue Xiao, Kang Li, Deyue Zhang, and Weilin Xu. Security risks in deep learning implementations. *2018 IEEE Security and Privacy Workshops (SPW)*, pages 123–128, 2018.

[33] Xinyue Shen, Steven Diamond, Yuantao Gu, and Stephen Boyd. DCCP source code. https://github.com/cvxgrp/dccp, 2017. Accessed 2017-09-03.

[34] Guoming Zhang, Chen Yan, Xiaoyu Ji, Tianchen Zhang, Taimin Zhang, and Wenyuan Xu. Dolphinattack: Inaudible voice commands. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 103–117, New York, NY, USA, 2017. ACM.

A Proof of Concept of the Scaling Attack

A.1 Software Version and Model Information for Attack Demonstration

Here we present the software setup for the attack demonstration. Although the example used here targets applications with Caffe, the risk is not limited to Caffe. We have tested the scaling functions in Caffe, TensorFlow and Torch. All of them are vulnerable to scaling attacks.

The Caffe package and the corresponding image classification examples were checked out directly from the official GitHub on October 25, 2017, and the OpenCV used was the latest stable version from the following URL: https://github.com/opencv/opencv/archive/2.4.13.4.zip

We used the BAIR/BVLC CaffeNet Model in our proof of concept exploitation. The model is the result of training based on the instructions provided by the original Caffe package. To avoid any mistakes in model setup, we download the model file directly from BVLC's official GitHub page. Detailed information about the model is provided in the list below.

Listing 1: Image classification model

```
name: BAIR/BVLC GoogleNet Model
caffemodel: bvlc_googlenet.caffemodel
caffemodel_url: http://dl.caffe.berkeleyvision.org/bvlc_googlenet.caffemodel
caffe_commit: bc614d1bd91896e3faceaf40b23b72dab47d44f5
```

A.2 Command Lines

The deceiving effect was demonstrated based on the official Caffe example *cppclassification*. The exact command line was shown in the list below.

Listing 2: Image classification command line

```
./classification.bin models/bvlc_googlenet/deploy.prototxt
```

```
models/bvlc_googlenet/bvlc_googlenet.caffemodel
data/ilsvrc12/imagenet_mean.binaryproto
data/ilsvrc12/synset_words.txt
IMAGE_FILE
```

A.3 Sample Output

The list below shows the classification results for the sample images used in the Section 2.2.

Listing 3: Sample classification results

```
# wolf-in-sheep.png [Image size: 672*224]
./classification.bin  models/bvlc_googlenet/deploy.prototxt
    models/bvlc_googlenet/bvlc_googlenet.caffemodel
    data/ilsvrc12/imagenet_mean.binaryproto
    data/ilsvrc12/synset_words.txt  /tmp/sample/wolf-in-sheep.png
---------- Prediction for /tmp/sample/wolf-in-sheep.png ----------
0.8890 - "n02114548 white wolf, Arctic wolf, Canis lupus tundrarum"
0.0855 - "n02120079 Arctic fox, white fox, Alopex lagopus"
0.0172 - "n02134084 ice bear, polar bear, Ursus Maritimus, Thalarctos maritimus"
0.0047 - "n02114367 timber wolf, grey wolf, gray wolf, Canis lupus"
0.0019 - "n02111889 Samoyed, Samoyede"


# wolf.png [Image size: 224*224]
./classification.bin  models/bvlc_googlenet/deploy.prototxt
    models/bvlc_googlenet/bvlc_googlenet.caffemodel
    data/ilsvrc12/imagenet_mean.binaryproto
    data/ilsvrc12/synset_words.txt  /tmp/sample/wolf.png
---------- Prediction for /tmp/sample/wolf.png ----------
0.8890 - "n02114548 white wolf, Arctic wolf, Canis lupus tundrarum"
0.0855 - "n02120079 Arctic fox, white fox, Alopex lagopus"
0.0172 - "n02134084 ice bear, polar bear, Ursus Maritimus, Thalarctos maritimus"
0.0047 - "n02114367 timber wolf, grey wolf, gray wolf, Canis lupus"
0.0019 - "n02111889 Samoyed, Samoyede"


# cat-in-sheep.png [Image size: 672*224]
./classification.bin  models/bvlc_googlenet/deploy.prototxt
    models/bvlc_googlenet/bvlc_googlenet.caffemodel
    data/ilsvrc12/imagenet_mean.binaryproto
    data/ilsvrc12/synset_words.txt  /tmp/sample/cat-in-sheep.png
---------- Prediction for /tmp/sample/cat-in-sheep.png ----------
0.1312 - "n02127052 lynx, catamount"
0.1103 - "n02441942 weasel"
0.1068 - "n02124075 Egyptian cat"
0.1000 - "n04493381 tub, vat"
0.0409 - "n04209133 shower cap"


# cat.png [Image size: 224*224]
./classification.bin  models/bvlc_googlenet/deploy.prototxt
    models/bvlc_googlenet/bvlc_googlenet.caffemodel
    data/ilsvrc12/imagenet_mean.binaryproto
    data/ilsvrc12/synset_words.txt  /tmp/sample/cat.png
---------- Prediction for /tmp/sample/cat.png ----------
0.1312 - "n02127052 lynx, catamount"
0.1103 - "n02441942 weasel"
0.1068 - "n02124075 Egyptian cat"
0.1000 - "n04493381 tub, vat"
0.0409 - "n04209133 shower cap"
```



(a) wolf-in-sheep.png (672*224)



(b) wolf.png (224*224)



(c) cat-in-sheep.png (672*224)



(d) cat.png (224*224)

Figure 10: Input pictures of the demo application.

B Code Samples Containing Image Scaling

This appendix provides code snippets of using data scaling procedure examples, from popular deep learning frameworks' released demos without change.

Listing 4: Preprocessing in image demo of Tensorflow [28]

```
def read_tensor_from_image_file(file_name, input_height=299, input_width=299,
                input_mean=0, input_std=255):
  input_name = "file_reader"
  output_name = "normalized"
  file_reader = tf.read_file(file_name, input_name)
  if file_name.endswith(".png"):
    image_reader = tf.image.decode_png(file_reader, channels = 3,
                                       name='png_reader')
  elif file_name.endswith(".gif"):
    image_reader = tf.squeeze(tf.image.decode_gif(file_reader,
                                          name='gif_reader'))
  elif file_name.endswith(".bmp"):
    image_reader = tf.image.decode_bmp(file_reader, name='bmp_reader')
  else:
    image_reader = tf.image.decode_jpeg(file_reader, channels = 3,
                                        name='jpeg_reader')
  float_caster = tf.cast(image_reader, tf.float32)
  dims_expander = tf.expand_dims(float_caster, 0);
  resized = tf.image.resize_bilinear(dims_expander, [input_height, input_width])
  normalized = tf.divide(tf.subtract(resized, [input_mean]), [input_std])
  sess = tf.Session()
  result = sess.run(normalized)

  return result
```

Listing 5: Preprocessing in *cppclassification* of Caffe [16]

```
189 void Classifier::Preprocess(const cv::Mat& img,
190                            std::vector<cv::Mat>* input_channels) {
191   /* Convert the input image to the input image format of the network. */
192   cv::Mat sample;
...
204   cv::Mat sample_resized;
205   if (sample.size() != input_geometry_)
206     cv::resize(sample, sample_resized, input_geometry_);
207   else
208     sample_resized = sample;
209
210   cv::Mat sample_float;
211   if (num_channels_ == 3)
212     sample_resized.convertTo(sample_float, CV_32FC3);
213   else
214     sample_resized.convertTo(sample_float, CV_32FC1);
...
224   CHECK(reinterpret_cast<float*>(input_channels->at(0).data)
225         == net_->input_blobs()[0]->cpu_data())
226     << "Input channels are not wrapping the input layer of the network.";
227 }
```

Listing 6: ImageNet classification with Torch7 [29]

```
function preprocess(im, img_mean)
  -- rescale the image
  local im3 = image.scale(im,224,224,'bilinear')
  -- subtract ImageNet mean and divide by std
  for i=1,3 do im3[i]:add(-img_mean.mean[i]):div(img_mean.std[i]) end
  return im3
end
```

Listing 7: ImageNet classification with PyTorch [1]

```
def main():
  global args, best_prec1
  args = parser.parse_args()
...
  # Data loading code
  traindir = os.path.join(args.data, 'train')
  valdir = os.path.join(args.data, 'val')
...
  val_loader = torch.utils.data.DataLoader(
    datasets.ImageFolder(valdir, transforms.Compose([
      transforms.Resize(256),
      transforms.CenterCrop(224),
      transforms.ToTensor(),
      normalize,
      ])),
    batch_size=args.batch_size, shuffle=False,
    num_workers=args.workers, pin_memory=True)
```

Listing 8: Code snippet in *deepdetect* based on Caffe [5]

```
  int read_file(const std::string &fname)
```

```
{
  cv::Mat img = cv::imread(fname,_bw ? CV_LOAD_IMAGE_GRAYSCALE :
                                       CV_LOAD_IMAGE_COLOR);
  if (img.empty())
{
  LOG(ERROR) << "empty image";
  return -1;
}
  _imgs_size.push_back(std::pair<int,int>(img.rows,img.cols));
  cv::Size size(_width,_height);
  cv::Mat rimg;
  cv::resize(img,rimg,size,0,0,CV_INTER_CUBIC);
  _imgs.push_back(rimg);
  return 0;
}
```

C Analysis and Examples of Popular Image Scaling Implentations

In this paper, we assume that the scaling algoritms first resize inputs horizontally and then vertically. This appendix provides examples of how we make our assumptions based on source code snippets of OpenCV and Pillow.

Here, Listing 9 shows one code snippet of OpenCV[10], where lines 3607-3700 are the main part of the resizing function implementation. From the loop condiction variables *dsize.width* (line 3607) and *dsize.height* (line 3674), we can infer that lines 3607-3662 present the horizontal scaling operation, and lines 3674-7300 show the vertical scaling operation.

Listing 9: Code snippet of OpenCV

```
...
3607  for( dx = 0; dx < dsize.width; dx++ )
3608  {
3609    if( !area_mode )
...
3662    }
3663
3664  for( dy = 0; dy < dsize.height; dy++ )
3665  {
3666    if( !area_mode )
...
3700  }
...
```

Listing 10 shows one code snippet of Pillow[11], which follows the same procesure (lines 635-681). The scaling direction can be inferred from the variables *need_horizontal* (line 636) and *need_vertical* (line 662).

Listing 10: Code snippet of Pillow

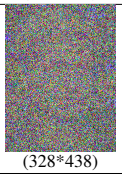
```
...
635  /* two-pass resize, horizontal pass */
636  if (need_horizontal) {
637     // Shift bounds for vertical pass
638     for (i = 0; i < ysize; i++) {
639       bounds_vert[i * 2] -= ybox_first;
640     }
...
659  }
660
661  /* vertical pass */
662  if (need_vertical) {
663    imOut = ImagingNewDirty(imIn->mode, imIn->xsize, ysize);
664    if (imOut) {
665      /* imIn can be the original image or horizontally resampled one */
666      ResampleVertical(imOut, imIn, 0,
667                       ksize_vert, bounds_vert, kk_vert);
668    }
...
681  }
```

[10] https://github.com/opencv/opencv/blob/master/modules/imgproc/src/resize.cpp

[11] https://github.com/python-pillow/Pillow/blob/master/src/libImaging/Resample.c

D Scaling Attack Examples

Table 5: Examples of two attack forms.

Style	Source Image	Target Image	Attack Image	Output Image
Strong Attack	 (328*438)	 (178*218)	 (328*438)	 (178*218)
Strong Attack	 (580*785)	 (128*128)	 (580*785)	 (128*128)
Strong Attack	 (580*785)	 (220*311)	 (580*785)	 (220*311)
Strong Attack◊	 (1280*720)	 (384*215)	 (1280*720)	 (384*215)
Strong Attack♦	 (922*692)	 (185*139)	 (922*692)	 (185*139)
Weak Attack	**None** (328*438)	 (109*145)	 (328*438)	 (109*145)

◊The car at the lower left corner of the attack image is removed after the attack image gets resized.

♦The "Prohibt left turn" sign in the attack image is changed into "Turn left" after the attack image gets resized.

CT-GAN: Malicious Tampering of 3D Medical Imagery using Deep Learning

Yisroel Mirsky[1], Tom Mahler[1], Ilan Shelef[2], and Yuval Elovici[1]
[1]Department of Information Systems Engineering, Ben-Gurion University, Israel
[2]Soroka University Medical Center, Beer-Sheva, Israel
yisroel@post.bgu.ac.il, mahlert@post.bgu.ac.il, shelef@bgu.ac.il, and elovici@bgu.ac.il

Abstract

In 2018, clinics and hospitals were hit with numerous attacks leading to significant data breaches and interruptions in medical services. An attacker with access to medical records can do much more than hold the data for ransom or sell it on the black market.

In this paper, we show how an attacker can use deep-learning to add or remove evidence of medical conditions from volumetric (3D) medical scans. An attacker may perform this act in order to stop a political candidate, sabotage research, commit insurance fraud, perform an act of terrorism, or even commit murder. We implement the attack using a 3D conditional GAN and show how the framework (CT-GAN) can be automated. Although the body is complex and 3D medical scans are very large, CT-GAN achieves realistic results which can be executed in milliseconds.

To evaluate the attack, we focused on injecting and removing lung cancer from CT scans. We show how three expert radiologists and a state-of-the-art deep learning AI are highly susceptible to the attack. We also explore the attack surface of a modern radiology network and demonstrate one attack vector: we intercepted and manipulated CT scans in an active hospital network with a covert penetration test.

1 Introduction

Medical imaging is the non-invasive process of producing internal visuals of a body for the purpose of medical examination, analysis, and treatment. In some cases, volumetric (3D) scans are required to diagnose certain conditions. The two most common techniques for producing detailed 3D medical imagery are Magnetic Resonance Imaging (MRI), and CT (Computed Tomography). Both MRI and CT scanner are essential tools in the medical domain. In 2016, there were approximately 38 million MRI scans and 79 million CT scans performed in the United States [1].[1]

MRI and CT scanners are similar in that they both create 3D images by taking many 2D scans of the body over the axial plane (from front to back) along the body. The difference between the two is that MRIs use powerful magnetic fields and CTs use X-Rays. As a result, the two modalities capture body tissues differently: MRIs are used to diagnose issues with bone, joint, ligament, cartilage, and herniated discs. CTs are used to diagnose cancer, heart disease, appendicitis, musculoskeletal disorders, trauma, and infectious diseases [2].

Today, CT and MRI scanners are managed though a picture archiving and communication system (PACS). A PACS is essentially an Ethernet-based network involving a central server which (1) receives scans from connected imaging devices, (2) stores the scans in a database for later retrieval, and (3) retrieves the scans for radiologists to analyze and annotate. The digital medical scans are sent and stored using the standardized DICOM format.[2]

1.1 The Vulnerability

The security of health-care systems has been lagging behind modern standards [3–6]. This is partially because health-care security policies mostly address data privacy (access-control) but not data security (availability/integrity) [7]. Some PACS are intentionally or accidentally exposed to the Internet via web access solutions. Some example products include Centricity PACS (GE Healthcare), IntelliSpace (Philips), Synapse Mobility (FujiFilm), and PowerServer (RamSoft). A quick search on Shodan.io reveals 1,849 medical image (DICOM) servers and 842 PACS servers exposed to the Internet. Recently, a researcher at McAfee demonstrated how these web portals can be exploited to view and modify a patient's 3D DICOM imagery [8]. PACS which are not directly connected to the Internet are indirectly connected via the facility's internal network [9]. They are also vulnerable to social engineering attacks, physical access, and insiders [10].

Therefore, a motivated attacker will likely be able to access a target PACS and the medical imagery within it. Later in section 4 we will discuss the attack vectors in greater detail.

1.2 The Threat

An attacker with access to medical imagery can alter the contents to cause a misdiagnosis. Concretely, the attacker can

[1]245 CT scans and 118 MRI scans per 1,000 inhabitants.

[2]https://www.dicomstandard.org/about/

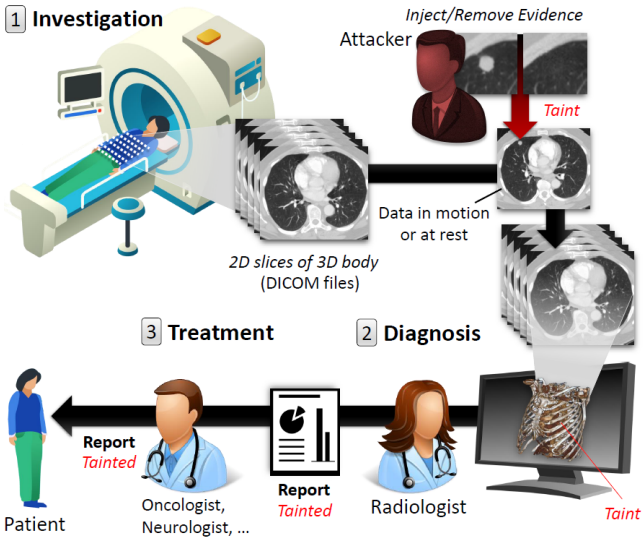


Figure 1: By tampering with the medical imagery between the investigation and diagnosis stages, both the radiologist and the reporting physician believe the fallacy set by the attacker.

add or remove evidence of some medical condition. Fig. 1 illustrates this process where an attacker injects/removes lung cancer from a scan.

Volumetric medical scans provide strong evidence of medical conditions. In many cases, a patient may be treated based on this evidence without the need to consider other medical tests. For example, some lesions are obvious or require immediate surgery. Moreover, some lesions will legitimately not show up on other medical tests (e.g., meniscus trauma and some breast cancers). Regardless, even if other tests aren't usually negative, ultimately, the evidence in the scan will be used to diagnose and treat the patient. As a result, an attacker with access to a scan has the power to change the outcome of the patient's diagnosis. For example, an attacker can add or remove evidence of aneurysms, heart disease, blood clots, infections, arthritis, cartilage problems, torn ligaments or tendons, tumors in the brain, heart, or spine, and other cancers.

There are many reasons why an attacker would want to alter medical imagery. Consider the following scenario: An individual or state adversary wants to affect the outcome of an election. To do so, the attacker adds cancer to a CT scan performed on a political candidate (the appointment/referral can be pre-existing, setup via social engineering, or part of a lung cancer screening program). After learning of the cancer, the candidate steps-down from his or her position. The same scenario can be applied to existing leadership.

Another scenario to consider is that of ransomware: An attacker seeks out monetary gain by holding the integrity of the medical imagery hostage. The attacker achieves this by altering a few scans and then by demanding payment for revealing which scans have been affected.

Furthermore, consider the case of insurance fraud: Somebody alters his or her own medical records in order to receive money directly from his or her insurance company, or receive

Table 1: Summary of an attacker's motivations and goals for injecting/removing evidence in 3D medical imagery.

+ : Add Evidence
− : Remove Evidence
± : Either

● : Target Effect
○ : Side Effect

			Goal: Steal Job Position	Affect Elections	Remove Leader	Sabotage Research	Falsify Research	Hold Data Hostage	Insurance Fraud	Murder	Terrorize
Motivation		Ideological			+						±
		Political		+	+					−	
		Money	+		+	+	−	±	+		
		Fame/Attn.	+		+		±				
		Revenge	+		+					−	+
Effect	Physical	Injury	○	○	○		○	○		○	●
		Death						○		●	●
	Mental	Trauma	○	○	○			○		○	●
		Life Course	●	●	●	○			○	○	●
	Monetary	Cause Loss	○		○	●		○		○	●
		Payout	●			○	●	●	●		
Attack Type		Untargeted				X		X			X
		Targeted	X	X	X	X	X		X	X	

handicap benefits (e.g., lower taxes etc.) In this case, there is no risk of physical injury to others, and the payout can be very large. For example, one can (1) sign up for disability/life insurance, then (2) fake a car accident or other incident, (3) complain of an inability to work, sense, or sleep, then (4) add a small brain hemorrhage or spinal fracture to his or her own scan during an investigation (this evidence is very hard to refute), and then (5) file a claim and receive cash from the insurance company.[3]

There are many more reasons why an attacker would want to tamper with the imagery. For example: falsifying research evidence, sabotaging another company's research, job theft, terrorism, assassination, and even murder.

Depending on the attacker's goal, the attack may be either untargeted or targeted:

Untargeted Attacks are where there is no specific target patient. In this case, the attacker targets a victim who is receiving a random voluntary cancer screening, is having an annual scan (e.g., BRACA patients, smokers...), or is being scanned due to an injury. These victims will either have an 'incidental finding' when the radiologist reviews the scan (injection) or are indeed sick but the evidence won't show (removal).

Targeted Attacks are where there is a specific target patient. In these attacks, the patient may be lured to the hospital for a scan. This can be accomplished by (1) adding an appointment in the system, (2) crafting a cancer screening invite, (3) spoofing the patient's doctor, or (4) tampering/appending the patient's routine lab tests. For

[3] For example, see products such as AIG's Quality of Life insurance.

example, high-PSA in blood indicates prostate cancer leading to an **abdominal MRI**, high thyrotropin in blood indicates a brain tumor leading to a **head MRI**, and metanephrine in urine of hypertensive patients indicates cancer/tumor leading to a **chest/abdominal CT**

In this paper we will focus on the injection and removal of lung cancer from CT scans. Table 1 summarizes attacker's motivations, goals, and effects by doing so. The reason we investigate this attack is because lung cancer is common and has the highest mortality rate [11]. Therefore, due its impact, an attacker is likely to manipulate lung cancer to achieve his or her goal. We note that the threat, attack, and countermeasures proposed in this paper also apply to MRIs and medical conditions other than those listed above.

1.3 The Attack

With the help of machine learning, the domain of image generation has advanced significantly over the last ten years [12]. In 2014, there was a breakthrough in the domain when Goodfellow et al. [13] introduced a special kind of deep neural network called a generative adversarial network (GAN). GANs consist of two neural networks which work against each other: the *generator* and the *discriminator*. The *generator* creates fake samples with the aim of fooling the *discriminator*, and the *discriminator* learns to differentiate between real and fake samples. When applied to images, the result of this game helps the *generator* create fake imagery which are photo realistic. While GANs have been used for positive tasks, researchers have also shown how they can be used for malicious tasks such as malware obfuscation [14, 15] and misinformation (e.g., deepfakes [16]).

In this paper, we show how an attacker can realistically inject and remove medical conditions with 3D CT scans. The framework, called CT-GAN, uses two conditional GANs (cGAN) to perform in-painting (image completion) [17] on 3D imagery. For injection, a cGAN is trained on unhealthy samples so that the *generator* will always complete the images accordingly. Conversely, for removal, another cGAN is trained on healthy samples only.

To make the process efficient and the output anatomically realistic, we perform the following steps: (1) locate where the evidence should be inject/removed, (2) cut out a rectangular cuboid from the location, (3) interpolate (scale) the cuboid, (4) modify the cuboid with the cGAN, (5) rescale, and (6) paste it back into the original scan. By dealing with a small portion of the scan, the problem complexity is reduced by focusing the GAN on the relevant area of the body (as opposed to the entire CT). Moreover, the algorithm complexity is reduced by processing fewer inputs[4] (pixels) and concepts (anatomical features). This results in fast execution and high anatomical realism. The interpolation step is necessary because the scale of a scan can be different between patients. To compensate for the resulting interpolation blur, we mask the relevant content according to water density in the tissue (Hounsfield units) and hide the smoothness by adding Gaussian white noise. In order to assist the GAN in generating realistic features, histogram equalization is performed on the input samples. We found that this transformation helps the 3D convolutional neural networks in the GAN learn how to generate the subtle features found in the human body. The entire process is automated, meanings that the attack can be deployed in an air gapped PACS.

To verify the threat of this attack, we trained CT-GAN to inject/remove lung cancer and hired three radiologists to diagnose a mix of 70 tampered and 30 authentic CT scans. The radiologists diagnosed 99% of the injected patients with malign cancer, and 94% of cancer removed patients as being healthy. After informing the radiologists of the attack, they still misdiagnosed 60% of those with injections, and 87% of those with removals. In addition to the radiologists, we also showed how CT-GAN is an effective adversarial machine learning attack. We found that the state-of-the-art lung cancer screening model misdiagnosed 100% of the tampered patients. Thus, cancer screening tools, used by some radiologists, are also vulnerable to this attack.

This attack is a concern because infiltration of healthcare networks has become common [3], and internal network security is often poor [18]. Moreover, for injection, the attacker is still likely to succeed even if medical treatment is not performed. This is because many goals rely on simply scaring the patient enough to affect his/her daily/professional life. For example, even if an immediate deletion surgery is not deemed necessary based on the scan and lab results, there will still be weekly/monthly follow-up scans to track the tumor's growth. This will affect the patient's life given the uncertainty of his or her future.

1.4 The Contribution

To the best of our knowledge, it has not been shown how an attacker can maliciously alter the content of a 3D medical image in a realistic and automated way. Therefore, this is the first comprehensive research which exposes, demonstrates, and verifies the threat of an attacker manipulating 3D medical imagery. In summary, the contributions of this paper are as follows:

The Attack Model We are the first to present how an attacker can infiltrate a PACS network and then use malware to autonomously tamper 3D medical imagery. We also provide a systematic overview of the attack, vulnerabilities, attack vectors, motivations, and attack goals. Finally, we demonstrate one possible attack vector through a penetration test performed on a hospital where we covertly connect a man-in-the-middle device to an actual CT scanner. By performing this pen-test, we provide insights into the security of a modern hospital's internal network.

Attack Implementation We are the first to demonstrate how GANs, with the proper preprocessing, can be used to efficiently, realistically, and automatically inject/remove lung cancer into/from large 3D CT scans. We also evaluate how well the algorithm can deceive both humans and

[4] A 3D CT scan can have over 157 million pixels whereas the latest advances in GANs can only handle about 2 million pixels (HD images).

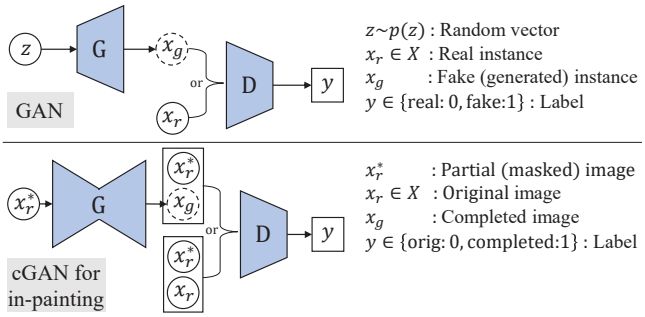


Figure 2: A schematic view of a classic GAN (top) and a cGAN setup for in-painting.

machines: radiologists and state-of-the-art AI. We also show how this implementation might be used by an attacker since it can be automated (in the case of an air gapped system) and is fast (in the case of an infected DICOM viewer).

Countermeasures We enumerate various countermeasures which can be used to mitigate the threat. We also provide the reader with best practices and configurations which can be implemented immediately to help prevent this attack.

For reproducibility and further investigation, we have published our tampered datasets and source code online[5] along with a pen-test video.[6]

The remainder of the paper is organized as follows: First we present a short background on GANs. Then, in section 3, we review related works and contrast them ours. In section 4 we present the attack model and demonstrate one of the attack vectors. In section 5, we present CT-GAN's neural architecture, its attack process, and some samples. In section 6 we evaluate the quality of the manipulations and asses the threat of the attack. Finally, in sections 7 and 8 we present countermeasures and our conclusion.

2 Background: GANs

The most basic GAN consists of two neural networks: the *generator* (G) and *discriminator* (D). The objective of the GAN is to generate new images which are visually similar to real images found in a sample data distribution X (i.e., a set of images). The input to G is the random noise vector z drawn from the prior distribution $p(z)$ (e.g., a Gaussian distribution). The output of G, denoted x_g, is an image which is expected to have visual similarity with those in X. Let the non-linear function learned by G parametrized by θ_g be denoted as $x_g = G(z;\theta_g)$. The input to D is either a real image $x_r \in X$ or a generated image $x_g \in G(Z;\theta_g)$. The output of D is the probability that x_g is real or fake. Let the non-linear function learned by D parametrized by θ_d be denoted as $x_d = D(x;\theta_d)$. The top of Fig. 2 illustrates the configuration of a classic GAN.

It can be seen that D and G are playing a zero-sum game where G is trying to find better (more realistic) samples to fool D, while D is learning to catch every fake sample generated by G. After training, D is discarded and G is used to generate new samples.

A cGAN is a GAN which has a *generator* and *discriminator* conditioned on an additional input (e.g., class labels). This input extends the latent space z with further information thus assisting the network to generate and discriminate images better. In [17] the authors propose an image-to-image translation framework using cGANs (a.k.a. pix2pix). There the authors showed how deep convolutional cGANs can be used to translate images from one domain to another. For example, converting casual photos to a Van Gogh paintings.

One application of the pix2pix framework is in-painting; the process of completing a missing part of an image. When using pix2pix to perform in-painting, the *generator* tries to fill in a missing part of an image based on the surrounding context, and its past experience (other images seen during training). Meanwhile, the *discriminator* tries to differentiate between completed images and original images, given the surrounding context. Concretely, the input to G is a copy of x_r where missing regions of the image are replaced with zeros. We denote this masked input as x_r^*. The output of G is the completed image, visually similar to those in X. The input to D is either the concatenation (x_r^*, x_r) or $(x_r^*, G(x_r^*;\theta_g))$. The bottom of Fig. 2 illustrates the described cGAN. The process for training this kind of GAN is as follows:

Training Procedure for cGAN In-painting

Repeat for k training iterations:

1. Pull a random batch of samples $x_r \in X$, and mask the samples with zeros to produce the respective x_r^*.
2. **Train** D:
   - 2.1. Forward propagate (x_r^*, x_r) through D, compute the error given the label $y = 0$, and back propagate the error through D to update θ_d.
   - 2.2. Forward propagate $(x_r^*, G(x_r^*;\theta_g))$ through D, compute the error given the label $y = 1$, and back propagate the error through D to update θ_d.
3. **Train** G:
   - 3.1. Forward propagate x_r^* through G and then $(x_r^*, G(x_r^*;\theta_g))$ through D, compute the error at the output of D given the label $y = 0$, back propagate the error through D to G without updating θ_d, and continue the back propagation through G while updating θ_g.

Although pix2pix does not use a latent random input z, it avoids deterministic outputs by performing random dropouts in the generator during training. this forces the network to learn multiple representations of the data.

We note that there is a GAN called a CycleGAN [19] that can directly translate images between two domains (e.g., benign $\leftrightarrow$ malign). However, we found that the CycleGAN was unable to inject realistic cancer into 3D samples. Therefore, we opted

[5] https://github.com/ymirsky/CT-GAN
[6] https://youtu.be/_mkRAArj-x0

to use the pix2pix model for in-painting because it produced much better results. This may be due to the complexity of the anatomy in the 3D samples and the fact that we had relatively few training samples. Since labeled datasets contain at most a few hundred scans, our approach is more likely to be used by an attacker. Another reason is that in-painting is arguably easier to perform than 'style transfer' when considering different bodies. Regardless, in-painting ensures that the final image can be seamlessly pasted back into the scan without border effects.

3 Related Work

The concept of tampering medical imagery, and the use of GANs on medical imagery, is not new. In this section we briefly review these subjects and compare prior results to our work.

3.1 Tampering with Medical Images

Many works have proposed methods for detecting forgeries in medical images [20], but none have focused on the attack itself. The most common methods of image forgery are: copying content from one image to another (image splicing), duplicating content within the same image to cover up or add something (copy-move), and enhancing an image to give it a different feel (image retouching) [21].

Copy-move attacks can be used to cover up evidence or duplicate existing evidence (e.g., a tumor). However, duplicating evidence will raise suspicion because radiologists closely analyze each discovered instance. Image-splicing can be used to copy evidence from one scan to another. However, CT scanners have distinct local noise patterns which are visually noticeable [22, 23]. The copied patterns would not fit the local pattern and thus raise suspicion. More importantly, both copy-move and image-splicing techniques are performed using 2D image editing software such as Photoshop. These tools require a digital artist to manually edit the scan. Even if the attacker has a digital artist, it is hard to accurately inject and remove cancer realistically. This is because human bodies are complex and diverse. For example, cancers and tumors are usually attached to nearby anatomy (lung walls, bronchi, etc.) which may be hard to alter accurately under the scrutiny of expert radiologists. Another consideration is that CT scans are 3D and not 2D, which adds to the difficulty. It is also important to note that an attacker will likely need to automate the entire process in a malware since (1) many PACS are not directly connected to the Internet and (2) the diagnosis may occur immediately after the scan is performed.

In contrast to the Photoshopping approach, CT-GAN (1) works on 3D medical imagery, which provide stronger evidence than a 2D scans, (2) realistically alters the contents of a 3D scan while considering nearby anatomy, and (3) can be completely automated. The latter point is important because (1) some PACS are not directly connected to the Internet, (2) diagnosis can happen right after the actual scan, (3) the malware may be inside the radiologist's viewing app.

3.2 GANs in Medical Imagery

Since 2016, over 100 papers relating to GANs and medical imaging have been published [24]. These publications mostly relate image reconstruction, denoising, image generation (synthesis), segmentation, detection, classification, and registration. We will focus on the use of GANs to generate medical images.

Due to privacy laws, it is hard to acquire medical scans for training models and students. As a result, the main focus of GANs in this domain has been towards augmenting (expanding) datasets. One approach is to convert imagery from one modality to another. For example, in [25] the authors used cGANs to convert 2D slices of CT images to Positron Emission Tomography (PET) images. In [26, 27] the authors demonstrated a similar concept using a fully convolutional network with a cGAN architecture. In [28], the authors converted MRI images to CT images using domain adaptation. In [29], the authors converted MRI to CT images and vice versa using a CycleGAN.

Another approach to augmenting medical datasets is the generation of new instances. In [30], the authors use a deep convolutional GAN (DCGAN) to generate 2D brain MRI images with a resolution of 220x172. In [31], the authors used a DCGAN to generate 2D liver lesions with a resolution of 64x64. In [32], the authors generated 3D blood vessels using a Wasserstien (WGAN). In [33], the authors use a Laplace GAN (LAPGAN) to generate skin lesion images with 256x256 resolution. In [34], the authors train two DCGANs for generating 2D chest X-rays (one for malign and the other for benign). However, in [34], the generated samples were down sampled to 128x128 in resolution since this approach could not be scaled to the original resolution of 2000x3000. In [35] the authors generated 2D images of pulmonary lung nodules (lung cancer) with 56x56 resolution. The author's motivation was to create realistic datasets for doctors to practice on. The samples were generated using a DCGAN and their realism was assessed with help of two radiologists. The authors found that the radiologists were unable to accurately differentiate between real and fake samples.

These works contrast to our work in the following ways:

1. We are the first to introduce the use of GANs as a way to tamper with 3D imagery. The other works focused on synthesizing cancer samples for boosting classifiers, experiments, and training students, but not for malicious attacks. We also provide an overview of how the attack can be accomplished in a modern medical system.

2. All of the above works either generate small regions of a scan without the context of a surrounding body or generate a full 2D scan with a very low resolution. Samples which are generated without a context cannot be realistically 'pasted' back into any arbitrary medical scan. We generate/remove content realistically within existing bodies. Moreover, very low-resolution images of full scans cannot replace existing ones without raising suspicion (especially if the body doesn't match the actual person).

Our approach can modify full resolution 3D scans,[7] and the approach can be easily extended to 2D as well.

3. We are the first to evaluate how well a GAN can fool expert radiologists and state-of-the-art AI in full 3D lung cancer screening. Moreover, in our evaluation, the radiologists and AI were able to consider how the cancer was attached and placed within the surrounding anatomy.

4 The Attack Model

In this section we explore the attack surface by first presenting the network topology and then by discussing the possible vulnerabilities and attack vectors. We also demonstrate one of the attack vectors on an actual CT scanner.

4.1 Network Topology

In order to discuss the attack vectors we must first present the PACS network topology. Fig. 3 presents the common network configuration of PACS used in hospitals. The topology is based on PACS literature [9, 36–38], PACS enterprise solutions (e.g., Carestream), and our own surveys conducted on various hospitals. We note that, private medical clinics may have a much simpler topology and are sometimes directly connected to the Internet [8].

The basic elements of a PACS are as follows:

PACS Server. The heart of the PACS system. It is responsible for storing, organizing, and retrieving DICOM imagery and reports commonly via SQL. Although the most facilities use local servers, a few hospitals have transitioned to cloud storage [39].

RIS Server. The radiology information system (RIS) is responsible for managing medical imagery and associated data. Its primary use is for tracking radiology imaging orders and the reports of the radiologists. Doctors in the hospital's internal network can interface with the RIS to order scans, receive the resulting reports, and to obtain the DICOM scans [40].

Modality Workstation. A PC (typically Windows) which is used to control an imaging modality such as a CT scanner. During an appointment, the attending technician configures and captures the imagery via the workstation. The workstation sends all imagery in DICOM format to the PACS server for storage.

Radiologist Workstation. A radiologist can retrieve and view scans stored on the PACS server from various locations. The most common location is a viewing workstation within the department. Other locations include the radiologist's personal PC (local or remote via VPN), and sometimes a mobile device (via the Internet or within the local network).

[7] A CT scan can have a resolution from 512x512x600 to 1024x1024x600 and larger.

Web Server. An optional feature which enables radiologists to view of DICOM scans (in the PACS server) over the Internet. The content may be viewed though a web browser (e.g., medDream and Orthanc [41]), an app on a mobile device (e.g., FujiFilm's Synapse Mobility), or accessed via a web API (e.g., Dicoogle [42]).

Administrative Assistant's PC. This workstation has both Internet access (e.g., for emails) and access to the PACS network. Access to the PACS is enabled so that the assistant can maintain the devices' schedules: When a patient arrives at the imaging modality, for safety reasons, the technician confirms the patient's identity with the details sent to the modality's workstation (entered by the assistant). This ensures that the scans are not accidentally mixed up between the patients.

Hospital Network. Other departments within the hospital usually have access to the PACS network. For example, Oncology, Cardiology, Pathology, and OR/Surgery. In these cases, various workstations around the hospital can load DICOM files from the server given the right credentials. Furthermore, it is common for a hospital to deploy Wi-Fi access points, which are connected to the internal network, for employee access.

4.2 Attack Scenario

The attack scenario is as follows: An attacker wants to achieve one of the goals listed in Table 1 by injecting/removing medical evidence. In order to cause the target effect, the attacker will alter the contents of the target's CT scan(s) before the radiologist performs his or her diagnosis. The attacker will achieve this by either targeting the data-at-rest or data-in-motion.

Thedata-at-rest refers to the DICOM files stored on the PACS Server, or on the radiologist's personal computer (saved for later viewing). In some cases, DICOM files are stored on DVDs and then transferred to the hospital by the patient or an external doctor. Although the DVD may be swapped by the attacker, it is more likely the interaction will be via the network. The data-in-motion refers to DICOM files being transferred across the network or loaded into volatile memory by an application (e.g., a DICOM viewer).

We note that this scenario does not apply to the case where the goal is to falsify or sabotage research. Moreover, for insurance fraud, an attacker will have a much easier time targeting a small medical clinic. For simplicity, we will assume that the target PACS is in a hospital.

4.3 Target Assets

To capture/modify a medical scan, an attacker must compromise at least one of the assets numbered in Fig. 3. By compromising one of (1-4), the attacker gains access to every scan. By compromising (5) or (6), the attacker only gains access to a subset of scans. The RIS (3) can give the attacker full control over the PACS server (2), but only if the attacker can obtain the right

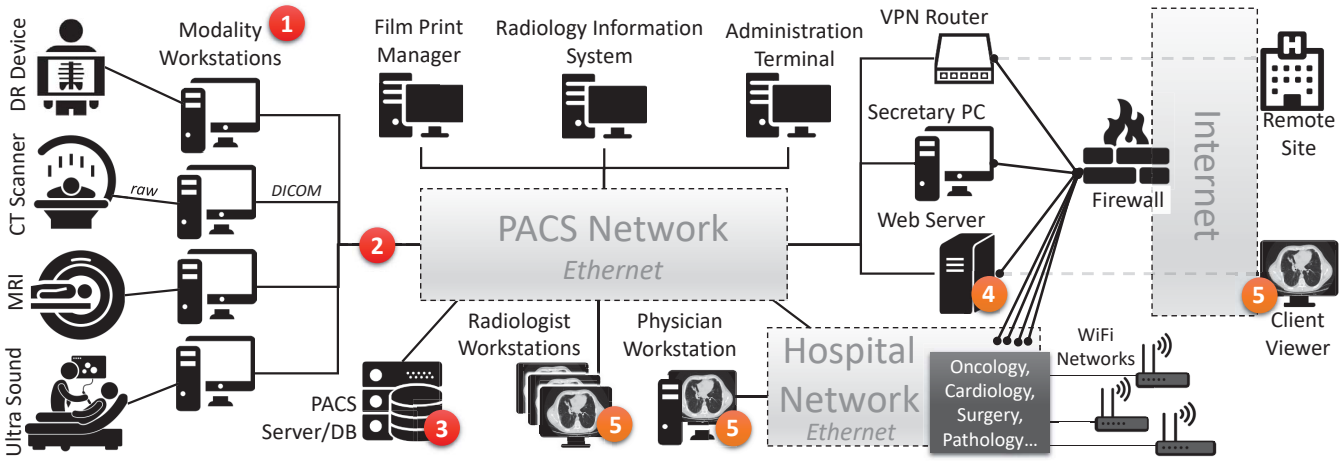


Figure 3: A network overview a PACS in a hospital. 1-3: points where an attacker can tamper with all scans. 4-5: points where an attacker can tamper with a subset of scans.

credentials or exploit the RIS software. The network wiring between the modalities and the PACS server (4) can be used to install a man-in-the-middle device. This device can modify data-in-motion if it is not encrypted (or if the protocol is flawed).

In all cases, it is likely that the attacker will infect the target asset with a custom malware, outlined in Fig. 4. This is because there may not be a direct route to the PACS via the Internet or because the diagnosis may take place immediately after the scan is taken.

4.4 Attack Vectors

There are many ways in which an attacker can reach the assets marked in Fig. 3. In general, the attack vectors involve either remote or local infiltration of the facility's network.

Remote Infiltration. The attacker may be able to exploit vulnerabilities in elements facing the Internet, providing the attacker with direct access to the PACS from the Internet (e.g., [8]). Another vector is to perform a social engineering attack. For example, a spear phishing attack on the department's administrative assistant to infect his/her workstation with a backdoor, or a phishing attack on the technician to have him install fraudulent updates.

If the PACS is not directly connected to the Internet, an alternative vector is to (1) infiltrate the hospital's internal network and then (2) perform lateral movement to the PACS. This is possible because PACS is usually connected to the internal network (using static routes and IPs), and the internal network is connected to the Internet (evident from the recent wave of

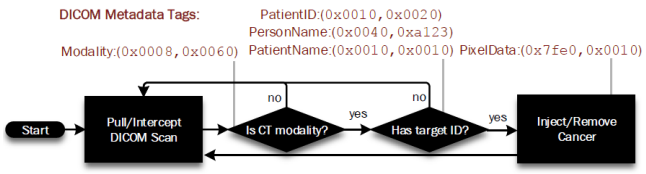


Figure 4: The tampering process of an autonomous malware.

cyber-attacks on medical facilities [3, 43–45]). The bridge between the internal network and the PACS is to enable doctors to view scans/reports and to enable the administrative assistant to manage patient referrals [9]. Another vector from the Internet is to compromise a remote site (e.g., a partnered hospital or clinic) which is linked to the hospital's internal network. Furthermore, the attacker may also try to infect a doctor's laptop or phone with a malware which will open a back door into the hospital.

If the attacker knows that radiologist analyzes scans on his or her personal computer, then the attacker can infect the radiologist's device or DICOM viewer remotely with the malware.

Local Infiltration. The attacker can gain physical access to the premises with a false pretext, such as being a technician from Philips who needs to run a diagnostic on the CT scanner. The attacker may also hire an insider or even be an insider. A recent report shows that 56% of cyber attacks on the healthcare industry come from internal threats [10].

Once inside, the attacker can plant the malware or a back door by (1) connecting a device to exposed network infrastructure (ports, wires, ...) [46] or (2) by accessing an unlocked workstation. Another vector which does not involve access to a restricted area, is to access to the internal network by hacking Wi-Fi access points. This can be accomplished using existing vulnerabilities such as 'Krack' [47] or the more recent 'Bleeding-Bit' vulnerabilities which have affected many hospitals [48].

Compromising the PACS. Once access to the PACS has been achieved, there are numerous ways an attacker can compromise a target asset. Aside from exploiting misconfigurations or default credentials, the attacker can exploit known software vulnerabilities. With regards to PACS servers, some already disclose private information/credentials which can be exploited remotely to create of admin accounts, and have hard-coded credentials.[8] A quick search on `exploit-db.com` reveals seven implemented exploits for PACS servers in 2018 alone. With regards to modality workstations, they too have been found to have significant vulnerabilities [49]. In 2018

[8] `CVE-2017-14008` and `CVE-2018-17906`

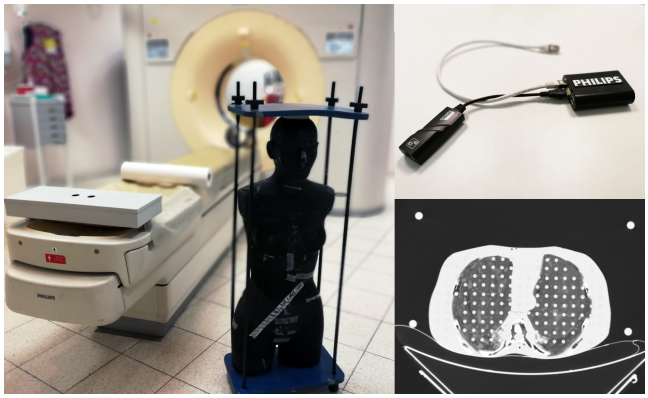


Figure 5: Left: The CT scanner and the medical dummy used to validate the attack. Top-right: the Pi-bridge used to intercept the scans. Bottom-right: one of the dummy's slices, sent by the CT scanner, and intercepted by the Pi-bridge.

the US Department of Homeland Security exposed 'low skill' vulnerabilities in Philips' Brilliance CT scanners [50]. For example, improper authentication, OS command injection, and hard-coded credentials.[9] Other recent vulnerabilities include hard-coded credentials.[10]

Given the state of health-care security, and that systems such as CT scanners are rarely given software updates [51], it is likely that these vulnerabilities and many more exist. Once the target asset in the PACS has been compromised, the attacker will be able to install the malware and manipulate the scans of target patients.

4.5 Attack Demonstration

To demonstrate how an attacker can access and manipulate CT scans, we performed a penetration test on a hospital's radiology department. The pen-test was performed with full permission of the participating hospital. To gain access to all CT scans, we performed a man-in-the-middle attack on the CT scanner using a Raspberry Pi 3B. The Pi was given a USB-to-Ethernet adapter, and was configured as a passive network bridge (without network identifiers). The Pi was also configured as a hidden Wi-Fi access point for backdoor access. We also printed a 3D logo of the CT scanner's manufacturer and glued it to the Pi to make it less conspicuous. The pen-test was performed as follows: First we waited at night until the cleaning staff opened the doors. Then we found the CT scanner's room and installed the Pi-bridge between the scanner's workstation and the PACs network (location #2 in Fig. 3). Finally, we hid the Pi-bridge under an access panel in the floor. The entire installation process took 30 seconds to complete. We were able to connect to the Pi wirelessly from the waiting room (approximately 20m away) to monitor the device's status, change the target identifier, etc.

At this point, an attacker could either intercept scans directly or perform lateral movement through the PACS to other subsystems and install the malware there. To verify that we could intercept and manipulate the scans, we scanned a medical dummy (Fig. 5). We found that the scan of the dummy was sent over the network twice: once in cleartext over TCP to an internal web viewing service, and again to the PACS storage server using TLSv1.2. However, to our surprise, the payload of the TLS transmission was also in cleartext. Moreover, within 10 minutes, we obtained the usernames and passwords of over 27 staff members and doctors due to multicasted Ethernet traffic containing `HTTP POST` messages sent in cleartext. A video of the pen-test can be found online.[11]

These vulnerabilities were disclosed to the hospital's IT staff and to their PACS software provider. Though inquiry, we found that it is not common practice for hospitals to encrypt their internal PACs traffic [52]. One reason is compatibility: hospitals often have old components (scanners, portals, databases, ...) which do not support encryption. Another reason is some PACS are not directly connected to the Internet, and thus is it erroneously thought that there is no need for encryption.

5 The CT-GAN Framework

In this section, we present the technique which an attacker can use to add/remove evidence in CT scans. First, we present the CT-GAN architecture and how to train it. Then, we will describe the entire tampering process and present some sample results. As a case study, we will focus on injecting/removing lung cancer.

It is important to note that there are many types of lung cancer. A common type of cancer forms a round mass of tissue called a solitary pulmonary nodule. Most nodules with a diameter less than 8mm are benign. However, nodules which are larger may indicate a malign cancerous growth. Moreover, if *numerous* nodules having a diameter $>$ 8mm are found, then the patient has an increased risk of primary cancer [53]. For this attack, we will focus on injecting and removing multiple solitary pulmonary nodules.

5.1 The Neural Architecture

A single slice in a CT scan has a resolution of *at least* 512x512 pixels. Each pixel in a slice measures the radiodensity at that location in Hounsfield units (HU). The CT scan of a human's lungs can have over 157 million voxels[12] (512x512x600). In order to train a GAN on an image of this size, we first locate a candidate location (voxel) and then cut out a small region around it (cuboid) for processing. The selected region is slightly larger than needed in order to provide the cGAN with context of the surrounding anatomy. This enables the cGAN to generate/remove lung cancers which connect to the body in a realistic manner.

To accurately capture the concepts of injection and removal, we use a framework consisting of two cGANs: one for injecting cancer (GAN_{inj}) and one for removing cancer (GAN_{rem}). Both GAN_{inj} and GAN_{rem} are deep 3D convolutional cGANs

[9] `CVE-2018-8853`, `CVE-2018-8857`, and `CVE-2018-8861`
[10] `CVE-2017-9656`
[11] `https://youtu.be/_mkRAArj-x0`
[12] A voxel is the three dimensional equivalent of a pixel.

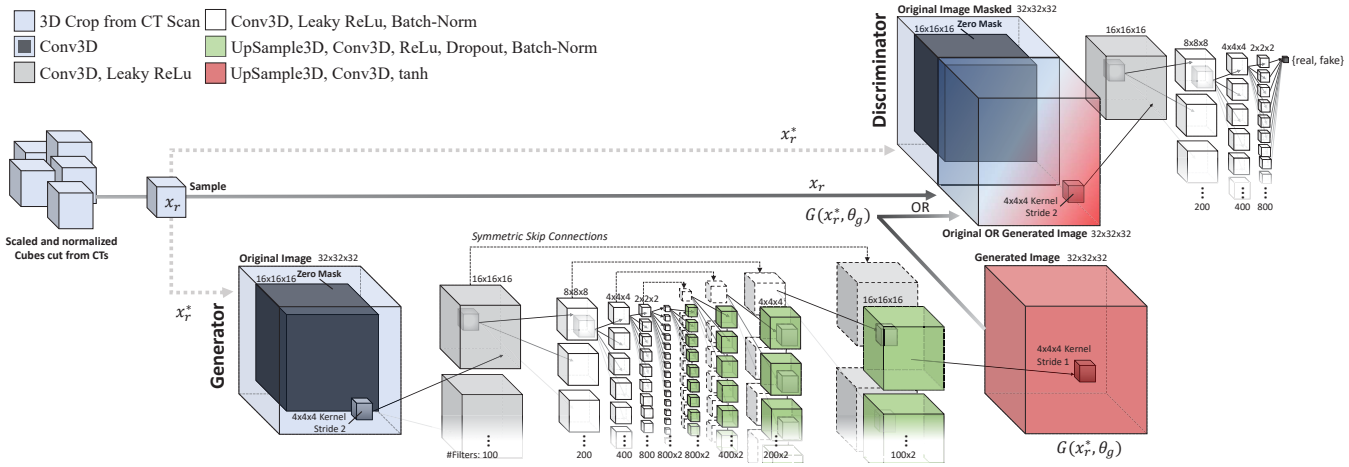


Figure 6: The network architecture, layers, and parameters used for both the injection (GAN_{inj}) and removal (GAN_{rem}) networks.

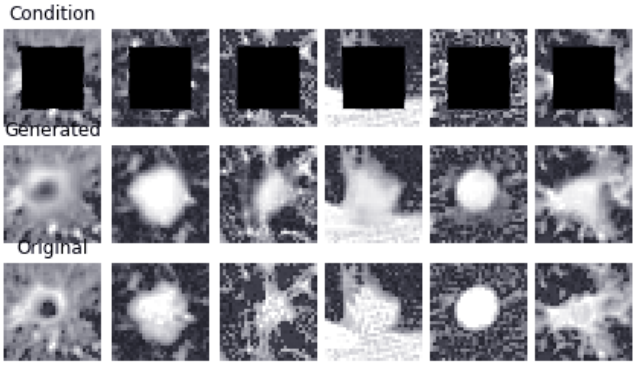


Figure 7: Training samples after 100 epochs showing the middle slice only. Top: the masked sample x_r^* given to both the *generator* G_{inj} and *discriminator* D_{inj}. Middle: The in-painted image x_g produced by the G_{inj}. Bottom: the ground-truth x_r. Note, D_{inj} sees either (x_r^*, x_r) or (x_r^*, x_g).

trained to perform in-painting on samples which are 32^3 voxels in dimension. For the completion mask, we zero-out a 16^3 cube in the center of the input sample. To inject a large pulmonary nodule into a CT scan, we train GAN_{inj} on cancer samples which have a diameter of least 10mm. As a result, the trained *generator* completes sample cuboids with similar sized nodules. To remove cancer, GAN_{rem} is trained using the same architecture, but with samples containing benign lung nodules only (having a diameter <3mm).

The model architecture (layers and configurations) used for both GAN_{inj} and GAN_{rem} is illustrated in Fig. 6. Overall, θ_g and θ_d had 162.6 million and 26.9 million trainable parameters respectively (189.5 million in total).

We note that follow up CT scans are usually ordered when a large nodule is found. This is because nodule growth is a strong indicator of cancer [53]. We found that an attacker is able to simulate this growth by conditioning each cancerous training sample on the nodule's diameter. However, the objective of this paper is to show how GANS can add/remove evidence realistically. Therefore, for the sake of simplicity, we have omitted this 'feature' from the above model.

5.2 Training CT-GAN

To train the GANs, we used a free dataset of 888 CT scans collected in the LIDC-IDRI lung cancer screening trial [54]. The dataset came with annotations from radiologists: the locations and diameters of pulmonary nodules having diameters greater than 3mm. In total there were 1186 nodules listed in the annotations.

To create the training set for GAN_{inj}, we extracted from the CT scans all nodules with a diameter between 10mm and 16mm (169 in total). To increase the number of training samples, we performed data augmentation: For each of the 169 cuboid samples, we (1) flipped the cuboid on the x, y, and xy planes, (2) shifted the cuboid by 4 pixels in each direction on the xy plane, and (3) rotated the cuboid 360 degrees at 6 degree intervals. This produced an additional 66 instances for each sample. The final training set had 11,323 training samples.

To create the training set for GAN_{rem}, we first selected clean CT scans which had no nodules detected by the radiologists. On these scans, we used the nodule detection algorithm from [55] (also provided in the dataset's annotations) to find benign micro nodules. Of the detected micro nodules, we selected 867 nodules at random and performed the same data augmentation as above. The final training set had 58,089 samples.

Prior to training the GANs, all of the samples were preprocessed with scaling, equalization, and normalization (described in the next section in detail). Both of the GANs were trained on their respective datasets for 200 epochs with a batch size of 50 samples. Each GAN took 26 hours to complete its training on an NVIDIA GeForce GTX TITAN X using all of the GPU's memory. Fig. 7 shows how well GAN_{inj} was able to in-paint cancer patterns after 150 epochs.

5.3 Execution: The Tampering Process

In order to inject/remove lung cancer, pre/post-processing steps are required. The following describes the entire

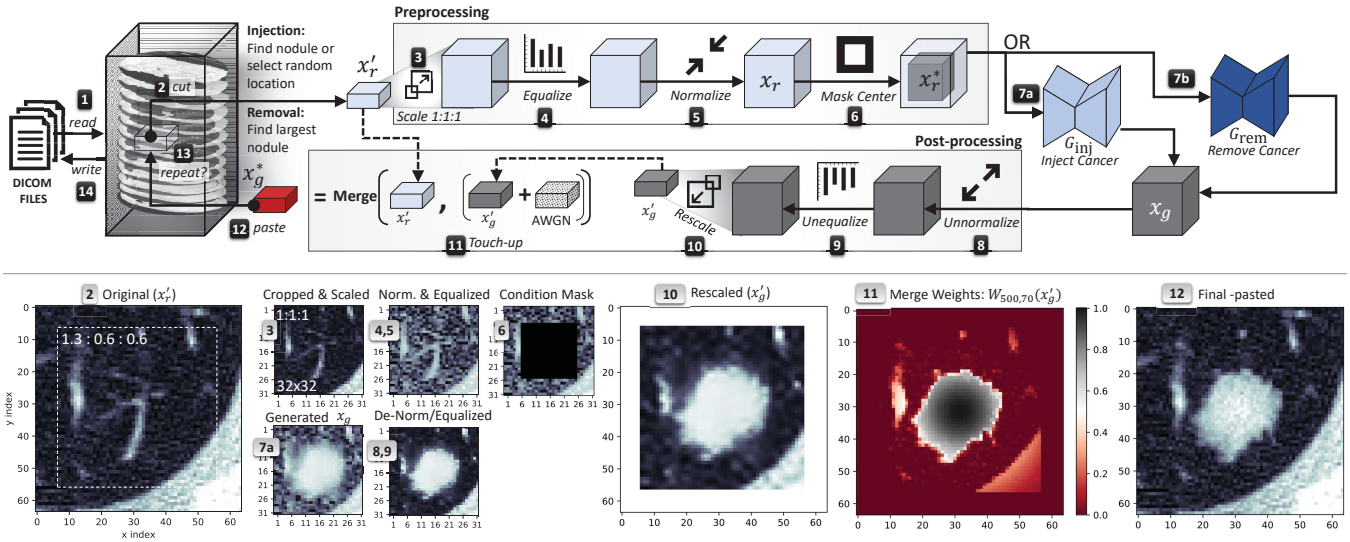


Figure 8: Top: the complete cancer injection/removal process. Bottom: sample images from the *injection* process. The grey numbers indicate from which step the image was taken. The sample 2D images are the middle slice of the respective 3D cuboid.

injection/removal process as illustrated in Fig. 8:

1. **Capture Data.** The CT scan is captured (as data-at-rest or data-in-motion) in either raw or DICOM format using one of the attack vectors from section 4.

2. **Localize & Cut.** A candidate location is selected where cancer will be injected/removed, and then the cuboid x'_r is cut out around it.

   Injection: An injection location can be selected in one of two ways. The fastest way is to take one of the middle slices of the CT scan and select a random location near the middle of the left or right half (see Fig. 16 in the appendix). With 888 CT scans, this strategy gave us a 99.1% successes rate. A more precise way is to execute an existing nodule detection algorithm to find a random micro nodule. To improve speed, the algorithm can be given only a few slices and implemented with early stopping. In our evaluation, we used the algorithm in [55], though many other options are available.

   Removal: A removal location can be found by selecting the largest nodule with [55], or by using a pre-trained deep learning cancer detection model.[13]

3. **Scale.** x'_r is scaled to the original 1:1:1 ratio using 3D spline interpolation.[14] The ratio information is available in the DICOM meta data with the tags (`0x0028`,`0x0030`) and (`0x0018`,`0x0050`). Scaling is necessary because each scan is stored with a different aspect ratio, and a GAN needs consistent units to produce accurate results. To minimize the computations, the cuboid cut in step 2 is cut with the exact dimensions so that the result of the rescaling process produces a 32^3 cube.

4-5. **Equalize & Normalize.** Histogram equalization is applied to the cube to increase contrast. This is a critical step since it enables the GAN to learn subtle features in the anatomy Normalization is then applied using the formula $X_n = 2\frac{X-\min(X)}{\max(X)-\min(X)} - 1$. This normalization ensures that all values fall on the range of $[-1,1]$ which helps the GAN learn the features better. The output of this process is the cube x_r.

6. **Mask.** In the center of x_r, a 16^3 cube is masked with zeros to form x^*_r. The masked area will be in-painted (completed) by the *generator*, and the unmasked area is the context).

7. **Inject/Remove.** x^*_r is passed through the chosen *discriminator* (G_{inj} or G_{rem}) creating a new sample (x_g) with new 3D generated content.

8-10. **Reverse Preprocessing.** x_g is unnormalized, unequalized, and then rescaled with spline interpolation back to its original proportions, forming x'_g.

11. **Touch-up.** The result of the interpolation usually blurs the imagery. In order to hide this artifact from the radiologists, we added Gaussian noise to the sample: we set $\mu = 0$ and σ to the sampled standard deviation of x'_r. To get a clean sample of the noise, we only measured voxels with values less than -600 HU. Moreover, to copy the relevant content into the scan, we merged the original cuboid (x'_r) with the generated one (x'_g) using a sigmoid weighted average.

    Let W be the weight function defined as

$$W_{\alpha,\beta}(x) = \frac{1}{1+e^{-(x+\alpha)/\beta}} * G(x) \qquad (1)$$

    where parameter α is the HU threshold between wanted and unwanted tissue densities, and parameter β controls the smoothness of the cut edges. The function $G(x)$ returns a 0-1 normalized Gaussian kernel with the dimensions of

[13] Pre-trained models are available here: https://concepttoclinic.drivendata.org/algorithms

[14] In Python: `scipy.ndimage.interpolation.zoom`

x. $G(x)$ is used to decay the contribution of each voxel the further it is the cuboid's center.

With W, we define the merging function as

$$merge_{\alpha,\beta}(x,y) = W_{\alpha,\beta}(x) * x + \left(1 - W_{\alpha,\beta}(x)\right) * y \quad (2)$$

where x is source (x'_g) and y is the destination (x'_r). We found that setting $\alpha = 500$ and $\beta = 70$ worked best. By applying these touch-ups, the final cuboid x^*_g is produced.

12. **Paste.** The cuboid x^*_g is pasted back into the CT scan at the selected location. See Fig. 16 in the appendix for one slice of a complete scan.

13. **Repeat.** If the attacker is removing cancer, then return to step 2 until no more nodules with a diameter $>$ 3mm are found. If the attacker is injecting cancer, then (optionally) return to step 2 until four injections have been performed. The reason for this is because the risk of a patient being diagnosed with cancer is statistically greater in the presence of exactly four solitary pulmonary nodules having a diameter $>$ 8mm [53].

14. **Return Data.** The scan is converted back into the original format (e.g. DICOM) and returned back to the source.

The quality of the injection/removal process can be viewed in figures 9 and 10. Fig. 9 presents a variety of examples before and after tampering, and Fig. 10 provides a 3D visualization of a cancer being injected and removed. More visual samples can be found in the appendix (figures 19 and 20).

We note that although some steps involve image touch-ups, the entire process is automatic (unlike Photoshop) and thus can be deployed in an autonomous malware or inside a viewing application (real-time tampering). We note that the same neural architecture and tampering process works on other modalities and medical conditions. For example, Fig. 18 in the appendix shows CT-GAN successfully injecting brain tumors into MRI head scans.

6 Evaluation

In this section we present our evaluation on how well the CT-GAN attack can fool expert radiologists and state-of-the-art AI.

6.1 Experiment Setup

To evaluate the attack, we recruited three radiologists (denoted **R1**, **R2**, and **R3**) with 2, 5, and 7 years of experience respectively. We also used a trained lung cancer screening model (denoted **AI**), the same deep learning model which won the 2017 Kaggle Data Science Bowl (a $1 million competition for diagnosing lung cancer).[15]

The experiment was performed in two trials: blind and open. In the blind trial, the radiologists were asked to diagnose 80 *complete* CT scans of lungs, but they were not told the purpose of the experiment or that some of the scans were manipulated.

[15] Source code and model available here: https://github.com/lfz/DSB2017

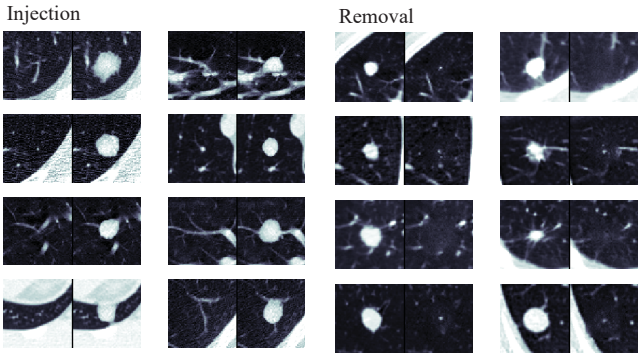


Figure 9: Sample injections (left) and removals (right). For each image, the left side is before tampering and the right side is after. Note that only the middle 2D slice is shown.

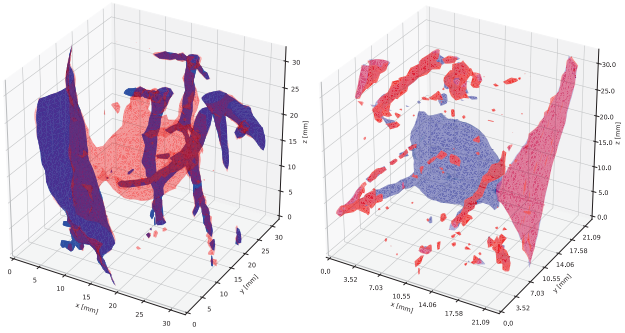


Figure 10: A 3D sample of injection (left) and removal (right) before (blue) and after (red) tampering with the CT scan.

Table 2: Summary of the scans and the relevant notations

	# Scans				# Slices	# Nodules		
	TB	TM	FB	FM		TM	FB	FM
Blind Trial	10	10	30	30	17,941	23	61	34
Open Trial	5	5	5	5	5,296	12	11	7

True-Benign (TB): *an original instance with no cancer*
True-Malign (TM): *an original instance with cancer*
False-Benign (FB): *a tampered instance with cancer removed*
False-Malign (FM): *a tampered instance with cancer injected*

In the open trial, the radiologists were told about the attack, and were asked to identify fake, real, and removed nodules in 20 CT scans. In addition, the radiologists were asked to rate the confidence of their decisions. After each trial, we gave the radiologists a questionnaire to assess how susceptible they were to the attacks. In all cases, the radiologists were asked to only detect and diagnose pulmonary nodules which have a diameter greater than 3mm.

The CT scans were taken from the LIDC-IDRI dataset [54]. The set of CT scans used in each trial and the notations used in this section are available in Table 2.

False benign (FB) and true malign (TM) scans truthfully contained at least one nodule with a diameter between 10mm and 16mm. FB scans were made by removing every nodule in the scan. FM scans were made by randomly injecting 1-4

Table 3: Cancer Detection Performance - Blind Trial

Evaluated by...		*Detector*																													
		Radiologist #1 (R1)								**Radiologist #2 (R2)**								**Radiologist #3 (R3)**								**Artificial Intelligence (AI)**							
		FP	TP	FN	TN	FPR	TPR	FNR	TNR	FP	TP	FN	TN	FPR	TPR	FNR	TNR	FP	TP	FN	TN	FPR	TPR	FNR	TNR	FP	TP	FN	TN	FPR	TPR	FNR	TNR
Instance	FB	0	2	59	0	-	0.03	0.97	-	0	3	58	0	-	0.05	0.95	-	0	4	57	0	-	0.07	0.93	-	0	4	57	0	-	0.07	0.93	-
	FM	32	0	0	2	0.94	-	-	0.06	33	0	0	1	0.97	-	-	0.03	33	0	0	1	0.97	-	-	0.03	34	0	0	0	1.00	-	-	0.00
	TB	0	0	0	15	0.00	-	-	1.00	0	0	0	15	0.00	-	-	1.00	0	0	0	15	0.00	-	-	1.00	0	0	0	15	0.00	-	-	1.00
	TM	0	21	2	0	-	0.91	0.09	-	0	19	4	0	-	0.83	0.17	-	0	20	3	0	-	0.87	0.13	-	0	21	2	0	-	0.91	0.09	-
Patient	FB	0	2	28	0	-	0.07	0.93	-	0	3	27	0	-	0.10	0.90	-	0	4	26	0	-	0.13	0.87	-	0	4	26	0	-	0.13	0.87	-
	FM	29	0	0	1	0.97	-	-	0.03	30	0	0	0	1.00	-	-	0.00	30	0	0	0	1.00	-	-	0.00	30	0	0	0	1.00	-	-	0.00
	TB	0	0	0	10	0.00	-	-	1.00	0	0	0	10	0.00	-	-	1.00	0	0	0	10	0.00	-	-	1.00	0	0	0	10	0.00	-	-	1.00
	TM	0	9	1	0	-	0.90	0.10	-	0	10	0	0	-	1.00	0.0	-	0	10	0	0	-	1.00	0.00	-	0	10	0	0	-	1.00	0.00	-

Table 4: Attack Detection Confusion Matrix - Open Trial
Evalauted by Instance

Seemingly Benign	**R1**		**R2**		**R3**	
	FB	TB	FB	TB	FB	TB
Cancer Removed: FB	0	11	2	7	0	10
No Cancer: TB	1	6	0	6	4	6

Seemingly Malign	**R1**		**R2**		**R3**	
	FM	TM	FM	TM	FM	TM
Fake Cancer: FM	0	3	5	1	3	4
Real Cancer: TM	2	6	3	6	2	4

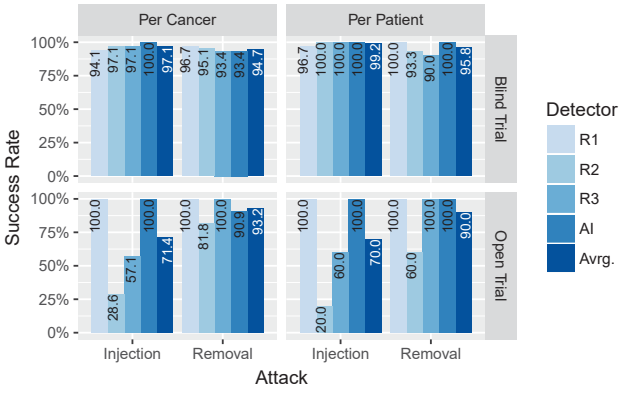


Figure 11: Attack success rates - Both Trials.

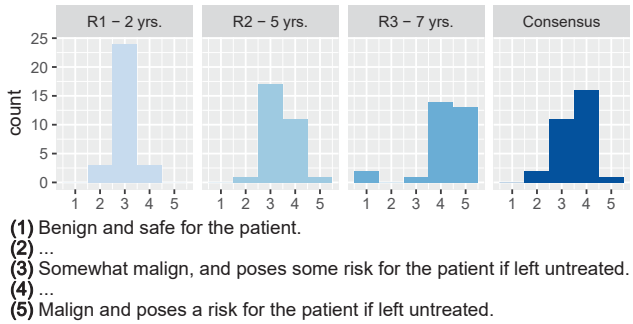


Figure 12: Malignancy of injected cancers (FM) - Blind Trial.

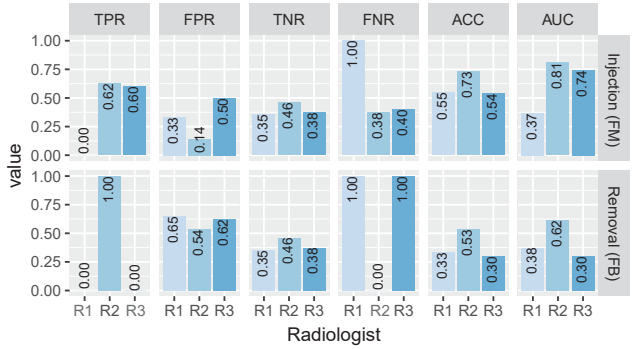


Figure 13: Attack detection performance - Open Trial.

nodules into a benign scan, where the injected nodules had a diameter of 14.4mm on average. In total, there were 100 CT scans analyzed by each of the radiologists, and the radiologists spent approximately 10 minutes analyzing each of these scans.

We note that the use of three radiologists is common practice in medical research (e.g., [56]). Moreover, we found that radiologists (and AI) significantly agreed with each other's diagnosis per patient and per nodule. We verified this agreement by computing Fliess' kappa [57] (a statistical interrater reliability measure) which produced an excellent kappa of 0.84 (p-value < 0.0001). Therefore, adding more radiologists will likely not affect the results.

6.2 Results: Blind Trial

In Table 3 we present the cancer detection performance of the radiologists and AI. The table lists the number of false-positives (FP - *detected a non-existent cancer*), true-positives (TP - *detected a real cancer*), false-negatives (FN - *missed a real cancer*), and their respective rates. The TCIA annotations (nodule locations) were used as our ground truth for measuring the performance on FB and TM scans. We evaluated these metrics per instance of cancer, and per patient as a whole. All four detectors performed well on the baseline (TB and TM) having an average TPR of 0.975 and a TNR of 1.0 in diagnosing the patients, meaning that we can rely on their diagnosis.

The top of Fig. 11 summarizes the attack success rates for the blind trial. In general, the attack had an average success rate of 99.2% for cancer injection and 95.8% for cancer removal. The AI was fooled completely which is an important aspect since some radiologists use AI tools to support their analysis (e.g. the Philips IntelliSite Pathology Solution). The radiologists were fooled less so, primarily due to human error (e.g., missing a nodule). When asked, none of the radiologists reported anything abnormal with the scans with the exception

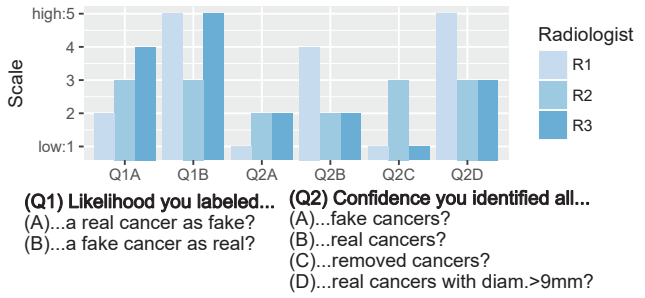


Figure 14: Confidence in detecting attacks - Open Trial.

of R2 who noted some noise in the area of one removal (FB). This may be attributed to "inattentional blindness," where one may miss an obvious event (artifacts) while engaged in a different task (searching for large nodules). In [58], the authors showed that this phenomenon also affects radiologists.

With regards to the injected cancers (FM), the consensus among the radiologists was that one-third of the injections require an immediate surgery/biopsy, and that all of the injections require follow-up treatments/referrals. When asked to rate the overall malignancy of the FM patients, the radiologists said that nearly all cases were significantly malign and pose a risk to the patient if left untreated. Fig. 12 summarizes radiologists' ratings of the FM patients. One interesting observation is that the malignancy rating increased with the experience of the radiologist. Finally, we note that an attacker could increase the overall malignancy of the injections if CT-GAN were trained only on samples with high malignancy and/or a larger diameter.

6.3 Results: Open Trial

In Table 4 we present the radiologists' attack detection performance with knowledge of the attack. Fig. 13 summarizes these results and provides the radiologists' accuracy (ACC) and area under the curve (AUC). An AUC of 1.0 indicates a perfect binary classifier, whereas an AUC of 0.5 indicates random guessing. The results show that the radiologists could not consistently tell the difference between real and fake cancers or identify the locations of removed cancers.

With regards to the attack success rates (bottom of Fig. 11), knowledge of the attack did not significantly affect cancer removal (90% from 95.8%). However, the success of the cancer injection was affected (70% from 99.2%). Moreover, R2 also picked up on a particular pattern which gave away several instances. This is a promising result, since it indicates that a portion of CT-GAN's attacks can be mitigated by educating radiologists. However, aside from low accuracy (61% for detecting an injection and 39% for detecting a removal), there was a significant number of false positives. With a high likelihood of making a mistake, a radiologist may choose not to report abnormalities. This is also apparent from the low confidence scores which the radiologists gave decisions (see Fig. 14).

In summary, both radiologists and AI are highly susceptible to CT-GAN's image tampering attacks. Although knowledge of the attack can help mitigate some cases of cancer injection, the error rates and confidence scores suggest that the attack may not be reported in most cases.

6.4 Baseline Analysis

As mentioned earlier, the attacker cannot use Photoshop on the images. This may be due to a time restriction: the radiologist may perform the diagnosis immediately after the scan has been taken, or the malware may be in the viewing application and operating in real-time. This may also be due to a connectivity restriction: many PACS do not have a direct connection to the Internet. Therefore, the attacker would need to infect the PACS with a malware that can perform the tampering autonomously.

However, for cancer injection, one may argue that there are simpler *automated* methods than using a GAN. For example, the malware could simply paste a pre-saved cropped image of cancer into a scan at a candidate location (found using steps 1 and 2 in Fig. 8). Therefore, to validate the use of a GAN, we evaluated this splice attack as a baseline comparison.

The experiment was setup as follows. To create our set of 'pre-saved cancers' we carefully cut out ten 32x32x32 cubes of real cancerous nodules from [54]. We then removed the irrelevant background tissue by producing transparency (alpha) channels using equation (1).

The evaluation dataset contained 20 scans in total: 10 FM and 10 TM. To create a FM scan, we took a random healthy scan and used steps 1 and 2 of Fig. 8 to select a candidate injection location. We then pasted one of the pre-saved cancers into the scan at that location using the transparency channel. To create the FM scans, we randomly selected 10 sick patients with exactly one nodule each.

To evaluate the baseline attack, we asked radiologist **R2** to analyze the 20 scans and determine which of the scans were authentic (TM) and which were tampered (FM) –similar to the open trial. This task took the radiologist approximately two hours to perform.

In summary, **R2** correctly labeled all of the tampered scans as fake. The only error was that **R2** mislabeled three of the authentic scans as fake. These results indicate that the baseline cancer injection attack cannot trick a radiologist, in comparison to CT-GAN which succeeds nearly every time.

The reason the baseline attack failed to trick **R2** is because the process of automatically pasting cancer creates obvious artifacts. This is because the pasted samples ignore the surrounding anatomy and may contain inconsistent noise patterns (textures). Fig. 15 illustrates some examples of these abnormalities such as cut bronchi, inconsistent noise patterns, and unaligned borders. CT-GAN does not produce these artifacts because it uses in-painting which considers the original content and surrounding anatomy.

7 Countermeasures

The tampering of DICOM medical files is a well-known concern. In the section we provide a brief overview of solutions for preventing and detecting this attack.

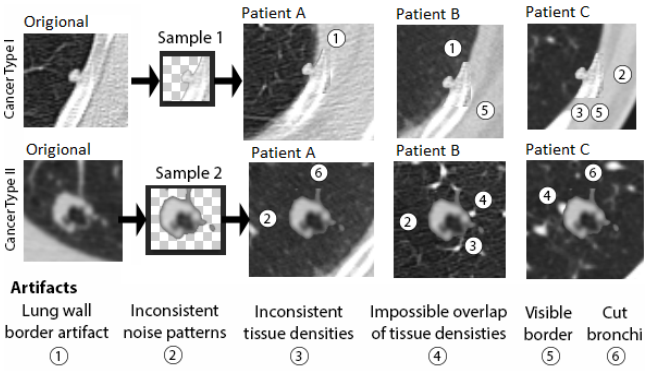


Figure 15: An illustration showing artifacts which can occur when using an *unsupervised* splice attack instead of CT-GAN. Only the middle slice is shown.

7.1 Prevention

To mitigate this threat, administrators should secure both the data-in-motion (DiM) and the data-at-rest (DaR). To secure data-in-motion, admins should enable encryption between the hosts in their PACS network using proper SSL certificates. This may seem trivial, but after discovering this flaw in the hospital we pen-tested, we turned to the PACS software provider for comment. The company, with over 2000 installations worldwide, confirmed to us that their hospitals do not enable encryption in their PACS because "it is not common practice". We were also told that some of the PACS don't support encryption at all.[16] To secure the DaR, servers and anti-virus software on modality and radiologist workstations should be kept up to date, and admins should also limit the exposure which their PACS server has to the Internet.

7.2 Detection

The best way to detect this attack is to have the scanner sign each scan with a digital signature. The DICOM image file standard already allows users to store signatures within the file's data structure [59, 60]. However, although some PACS software providers offer this feature, we have not seen it in use within a PACS. If enabled, admins should check that valid certificates are being used and that the radiologists' viewing applications are indeed verifying the signatures.

Another method for detecting this attack is digital watermarking (DW). A DW is a hidden signal embedded into an image such that tampering corrupts the signal and thus indicates a loss of integrity. For medical images, this subject has been researched in depth [20] and can provide a means for localizing changes in a tampered image. However, we did not find any medical devices or products which implement DW techniques. This may be due to the fact that they add noise to images which may harm the medical analysis.

Tampered images can also be detected with machine learning. In the supervised setting (where models are trained on examples of tampered images) the authors in [61] propose detection by (1) extracting a scan's noise pattern using a Wiener filter, then (2) applying a multi-resolution regression filter on the noise, and then (3) executing an SVM and ELM together via a Bayesian Sum Rule model. Many domain specific methods exist for detecting images tampered by GANs (e.g., images/videos of faces [62–64]). However, the supervised approach in [65] is more suitable for detecting our attack since it is domain generic.

Several approaches have been proposed for unsupervised setting as well. These approaches attempt to detect anomalies (inconsistencies) within the tampered images. To detect these inconsistencies, researchers have considered JPEG blocks, signal processing, and compression/resampling artifacts [66]. For example, in a recent work the authors trained a Siamese network to predict the probability that a pair of patches from two images have the same EXIF metadata (e.g., focal length and shutter speed) [67]. In [67], the model is trained using a dataset of real images only. In [68], the authors proposed 'noiseprint' which uses a Siamese network to extract the camera's unique noise pattern from an image (PRNU) to find inconsistent areas. In their evaluation, the authors show that they can detect GAN-based inpainting. In [69], the authors proposed three strategies for using PRNU-based tampering localization techniques with multi-scale analysis. Using this method, the authors were able to detect forgeries of all shapes and sizes.

While these countermeasures may apply to CT-GAN in some cases, they do admit some caveats; namely, that (1) medical scans are usually not compressed so compression methods are irrelevant, (2) these methods were tested on 2D images and not 3D volumetric imagery, and (3) CT/MR imaging systems produce very different noise patterns than standard cameras. For example, we found that the PRNU method in [69] does not work out-of-the-box on our tampered CT scans. This is because the noise patterns of CT images are altered by a radon transform used to construct the image. As future work, we plan to research how these techniques can be applied to detecting attacks such as CT-GAN.

8 Conclusion

In this paper we introduced the possibility of an attacker modifying 3D medical imagery using deep learning. We explained the motivations for this attack, discussed the attack vectors (demonstrating one of them), and presented a manipulation framework (CT-GAN) which can be executed by a malware autonomously. As a case study, we demonstrated how an attacker can use this approach to inject or remove lung cancer from full resolution 3D CT scans using free medical imagery from the Internet. We also evaluated the attack and found that CT-GAN can fool both humans and machines: radiologists and state-of-the-art AI. This paper also demonstrates how we should be wary of closed world assumptions: both human experts and advanced AI can be fooled if they fully trust their observations.

[16] See [52] for further comments.

References

[1] P. I, W. LR, et al. Health care spending in the united states and other high-income countries. *JAMA*, 319(10):1024–1039, 2018.

[2] J. R. Haaga. *CT and MRI of the Whole Body*. No. v. 1 in CT and MRI of the Whole Body. Mosby/Elsevier, 2008. ISBN 9780323053754.

[3] H. I. News. The biggest healthcare data breaches of 2018 (so far). https://www.healthcareitnews.com/projects/biggest-healthcare-data-breaches-2018-so-far, 2019.

[4] T. George. Feeling the pulse of cyber security in healthcare, securityweek.com. https://www.securityweek.com/feeling-pulse-cyber-security-healthcare, 2018.

[5] I. Institute. Cybersecurity in the healthcare industry. https://resources.infosecinstitute.com/cybersecurity-in-the-healthcare-industry, 2016.

[6] L. Coventry and D. Branley. Cybersecurity in healthcare: A narrative review of trends, threats and ways forward. *Maturitas*, 113:48 – 52, 2018. ISSN 0378-5122.

[7] M. S. Jalali and J. P. Kaiser. Cybersecurity in hospitals: A systematic, organizational perspective. *Journal of medical Internet research*, 20(5), 2018.

[8] C. Beek. Mcafee researchers find poor security exposes medical data to cybercriminals, mcafee blogs. https://securingtomorrow.mcafee.com/other-blogs/mcafee-labs/mcafee-researchers-find-poor-security-exposes-medical-data-to-cybercriminals/, 2018.

[9] H. Huang. *PACS-Based Multimedia Imaging Informatics: Basic Principles and Applications*. Wiley, 2019. ISBN 9781118795736.

[10] Verizon. Protected health information data breach report. *white paper*, 2018.

[11] F. Bray, J. Ferlay, et al. Global cancer statistics 2018: Globocan estimates of incidence and mortality worldwide for 36 cancers in 185 countries. *CA: a cancer journal for clinicians*, 68(6):394–424, 2018.

[12] X. Wu, K. Xu, et al. A survey of image synthesis and editing with generative adversarial networks. *Tsinghua Science and Technology*, 22(6):660–674, 2017.

[13] I. Goodfellow, J. Pouget-Abadie, et al. Generative adversarial nets. In *Advances in neural information processing systems*, pp. 2672–2680. 2014.

[14] W. Hu and Y. Tan. Generating adversarial malware examples for black-box attacks based on gan. *arXiv preprint arXiv:1702.05983*, 2017.

[15] M. Rigaki and S. Garcia. Bringing a gan to a knife-fight: Adapting malware communication to avoid detection. In *2018 IEEE Security and Privacy Workshops (SPW)*, pp. 70–75. IEEE, 2018.

[16] R. Chesney and D. K. Citron. Deep fakes: A looming challenge for privacy, democracy, and national security. *U of Texas Law, Public Law Research Paper No. 692; U of Maryland Legal Studies Research Paper No. 2018-21*, 2018.

[17] P. Isola, J.-Y. Zhu, et al. Image-to-image translation with conditional adversarial networks. *arXiv preprint*, 2017.

[18] T. Seals. Rsa conference 2019: Ultrasound hacked in two clicks, threatpost. https://threatpost.com/ultrasound-hacked/142601/, 2019.

[19] J.-Y. Zhu, T. Park, et al. Unpaired image-to-image translation using cycle-consistent adversarial networks. *arXiv preprint*, 2017.

[20] A. K. Singh, B. Kumar, et al. *Medical Image Watermarking Techniques: A Technical Survey and Potential Challenges*, pp. 13–41. Springer International Publishing, Cham, 2017. ISBN 978-3-319-57699-2.

[21] S. Sadeghi, S. Dadkhah, et al. State of the art in passive digital image forgery detection: copy-move image forgery. *Pattern Analysis and Applications*, 21(2):291–306, May 2018. ISSN 1433-755X.

[22] A. Kharboutly, W. Puech, et al. Ct-scanner identification based on sensor noise analysis. In *2014 5th European Workshop on Visual Information Processing (EUVIP)*, pp. 1–5. Dec 2014.

[23] Y. Duan, D. Bouslimi, et al. Computed tomography image origin identification based on original sensor pattern noise and 3d image reconstruction algorithm footprints. *IEEE journal of biomedical and health informatics*, 21(4):1039–1048, 2017.

[24] X. Yi, E. Walia, et al. Generative adversarial network in medical imaging: A review. *arXiv preprint arXiv:1809.07294*, 2018.

[25] L. Bi, J. Kim, et al. Synthesis of Positron Emission Tomography (PET) Images via Multi-channel Generative Adversarial Networks (GANs). pp. 43–51. Springer, Cham, 2017.

[26] A. Ben-Cohen, E. Klang, et al. Virtual PET Images from CT Data Using Deep Convolutional Networks: Initial Results. pp. 49–57. Springer, Cham, 2017.

[27] —. Cross-Modality Synthesis from CT to PET using FCN and GAN Networks for Improved Automated Lesion Detection. 2 2018.

[28] Q. Dou, C. Ouyang, et al. Unsupervised Cross-Modality Domain Adaptation of ConvNets for Biomedical Image Segmentations with Adversarial Loss. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, pp. 691–697. International Joint Conferences on Artificial Intelligence Organization, California, 7 2018. ISBN 9780999241127.

[29] C.-B. Jin, H. Kim, et al. Deep CT to MR Synthesis using Paired and Unpaired Data. 5 2018.

[30] C. Bermudez, A. J. Plassard, et al. Learning implicit brain mri manifolds with deep learning. In *Medical Imaging 2018: Image Processing*, vol. 10574, p. 105741L. International Society for Optics and Photonics, 2018.

[31] M. Frid-Adar, I. Diamant, et al. GAN-based Synthetic Medical Image Augmentation for increased CNN Performance in Liver Lesion Classification. 3 2018.

[32] J. M. Wolterink, T. Leiner, et al. Blood Vessel Geometry Synthesis using Generative Adversarial Networks. In *1st Conference on Medical Imaging with Deep Learning (MIDL 2018)*. Amsterdam, The Netherlands, The Netherlands, 2018.

[33] C. Baur, S. Albarqouni, et al. Melanogans: High resolution skin lesion synthesis with gans. *arXiv preprint arXiv:1804.04338*, 2018.

[34] A. Madani, M. Moradi, et al. Chest x-ray generation and data augmentation for cardiovascular abnormality classification. In *Medical Imaging 2018: Image Processing*, vol. 10574, p. 105741M. International Society for Optics and Photonics, 2018.

[35] M. J. Chuquicusma, S. Hussein, et al. How to fool radiologists with generative adversarial networks? a visual turing test for lung cancer diagnosis. In *Biomedical Imaging (ISBI 2018), 2018 IEEE 15th International Symposium on*, pp. 240–244. IEEE, IEEE, 4 2018. ISBN 978-1-5386-3636-7.

[36] W. Hruby. *Digital (R)Evolution in Radiology*. Springer Vienna, 2013. ISBN 9783709137079.

[37] A. Peck. *Clark's Essential PACS, RIS and Imaging Informatics*. Clark's Companion Essential Guides. CRC Press, 2017. ISBN 9781498763462.

[38] C. Carter and B. Veale. *Digital Radiography and PACS*. Elsevier Health Sciences, 2018. ISBN 9780323547598.

[39] B. Siwicki. Cloud-based pacs system cuts imaging costs by half for rural hospital | healthcare it news. https://www.healthcareitnews.com/news/cloud-based-pacs-system-cuts-imaging-costs-half-rural-hospital.

[40] J. Bresnick. Picture archive communication system use widespread in hospitals. https://healthitanalytics.com/news/picture-archive-communication-system-use-widespread-in-hospitals, 2016.

[41] S. Jodogne, C. Bernard, et al. Orthanc-a lightweight, restful dicom server for healthcare and medical research. In *Biomedical Imaging (ISBI), 2013 IEEE 10th International Symposium on*, pp. 190–193. IEEE, 2013.

[42] C. Costa, C. Ferreira, et al. Dicoogle-an open source peer-to-peer pacs. *Journal of digital imaging*, 24(5):848–856, 2011.

[43] L. Adefala. Healthcare experiences twice the number of cyber attacks as other industries. https://www.fortinet.com/blog/business-and-technology/healthcare-experiences-twice-the-number-of-cyber-attacks-as-othe.html, 2018.

[44] J. B. Rebecca Weintraub. 11 things the health care sector must do to improve cybersecurity. https://hbr.org/2017/06/11-things-the-health-care-sector-must-do-to-improve-cybersecurity, 2017.

[45] C. Osborne. Us hospital pays $55,000 to hackers after ransomware attack | zdnet. https://www.zdnet.com/article/us-hospital-pays-55000-to-ransomware-operators/, 2018.

[46] J. Muniz and A. Lakhani. *Penetration testing with raspberry pi*. Packt Publishing Ltd, 2015.

[47] M. Vanhoef and F. Piessens. Key reinstallation attacks: Forcing nonce reuse in wpa2. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1313–1328. ACM, 2017.

[48] A. NG. Security researchers find flaws in chips used in hospitals, factories and stores - cnet. https://www.cnet.com/news/security-researchers-find-flaws-in-chips-used-in-hospitals-factories-and-stores/, 2018.

[49] R. M. Robin Henry and J. Corke. Hospitals to struggle for days | news | the sunday times. https://www.thetimes.co.uk/article/nhs-cyberattack-bitcoin-wannacry-hospitals-to-struggle-for-days-k0nhk7p2b, 2017.

[50] DHS. Philips isite/intellispace pacs vulnerabilities (update a), ics-cert. https://ics-cert.us-cert.gov/advisories/ICSMA-18-088-01, 2018.

[51] J. E. Dunn. Imagine you're having a ct scan and malware alters the radiation levels – it's doable • the register. https://www.theregister.co.uk/2018/04/11/hacking_medical_devices/, 2018.

[52] K. Zetter. Hospital viruses: Fake cancerous nodes in ct scans, created by malware, trick radiologists. https://www.washingtonpost.com/technology/2019/04/03/hospital-viruses-fake-cancerous-nodes-ct-scans-created-by-malware-trick-radiologists/, April 2019.

[53] H. MacMahon, D. P. Naidich, et al. Guidelines for management of incidental pulmonary nodules detected on ct images: from the fleischner society 2017. *Radiology*, 284(1):228–243, 2017.

[54] S. G. Armato III, G. McLennan, et al. The lung image database consortium (lidc) and image database resource initiative (idri): a completed reference database of lung nodules on ct scans. *Medical physics*, 38(2):915–931, 2011.

[55] K. Murphy, B. van Ginneken, et al. A large-scale evalua-

tion of automatic pulmonary nodule detection in chest ct using local image features and k-nearest-neighbour classification. *Medical image analysis*, 13(5):757–770, 2009.

[56] A. Esteva, B. Kuprel, et al. Dermatologist-level classification of skin cancer with deep neural networks. *Nature*, 542(7639):115, 2017.

[57] A. J. Conger. Integration and generalization of kappas for multiple raters. *Psychological Bulletin*, 88(2):322, 1980.

[58] T. Drew, M. L.-H. Võ, et al. The invisible gorilla strikes again: Sustained inattentional blindness in expert observers. *Psychological science*, 24(9):1848–1853, 2013.

[59] F. Cao, H. Huang, et al. Medical image security in a hipaa mandated pacs environment. *Computerized medical imaging and graphics*, 27(2-3):185–196, 2003.

[60] NEMA. Digital imaging and communications in medicine (dicom) digital signatures. ftp://medical.nema.org/medical/dicom/final/sup41_ft.pdf, 2001.

[61] A. Ghoneim, G. Muhammad, et al. Medical image forgery detection for smart healthcare. *IEEE Communications Magazine*, 56(4):33–37, 2018.

[62] A. Rössler, D. Cozzolino, et al. Faceforensics++: Learning to detect manipulated facial images. *arXiv preprint arXiv:1901.08971*, 2019.

[63] F. Matern, C. Riess, et al. Exploiting visual artifacts to expose deepfakes and face manipulations. In *2019 IEEE Winter Applications of Computer Vision Workshops (WACVW)*, pp. 83–92. IEEE, 2019.

[64] S. Tariq, S. Lee, et al. Detecting both machine and human created fake face images in the wild. In *Proceedings of the 2nd International Workshop on Multimedia Privacy and Security*, pp. 81–87. ACM, 2018.

[65] D. Cozzolino, J. Thies, et al. Forensictransfer: Weakly-supervised domain adaptation for forgery detection. *arXiv preprint arXiv:1812.02510*, 2018.

[66] L. Zheng, Y. Zhang, et al. A survey on image tampering and its detection in real-world photos. *Journal of Visual Communication and Image Representation*, 58:380–399, 2019.

[67] M. Huh, A. Liu, et al. Fighting fake news: Image splice detection via learned self-consistency. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 101–117. 2018.

[68] D. Cozzolino and L. Verdoliva. Noiseprint: a cnn-based camera model fingerprint. *arXiv preprint arXiv:1808.08396*, 2018.

[69] P. Korus and J. Huang. Multi-scale analysis strategies in prnu-based tampering localization. *IEEE Trans. on Information Forensics & Security*, 2017.

Appendix

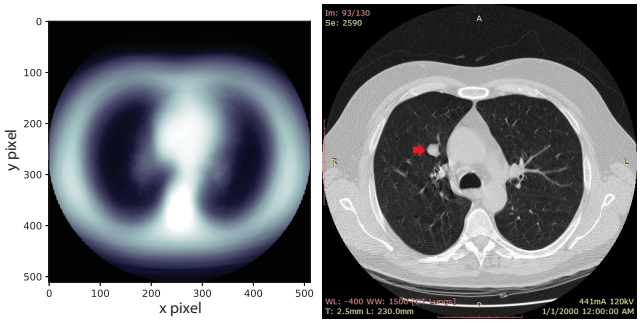


Figure 16: Left: the average of 888 CT scans' middle slices *before* scaling to 1:1:1 ratio (black ares are candidate injection locations). Right: a full slice with an injected nodule.

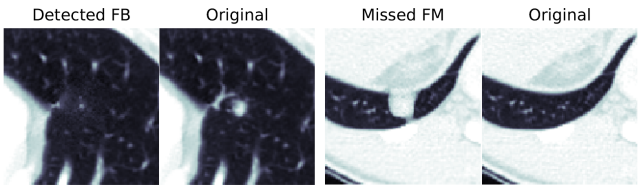


Figure 17: Examples of where the attack failed in the blind trial. Left: a removal (FB) detected as 'ground-glass' cancer due to too much additive noise. Right: an injection missed due to human error.

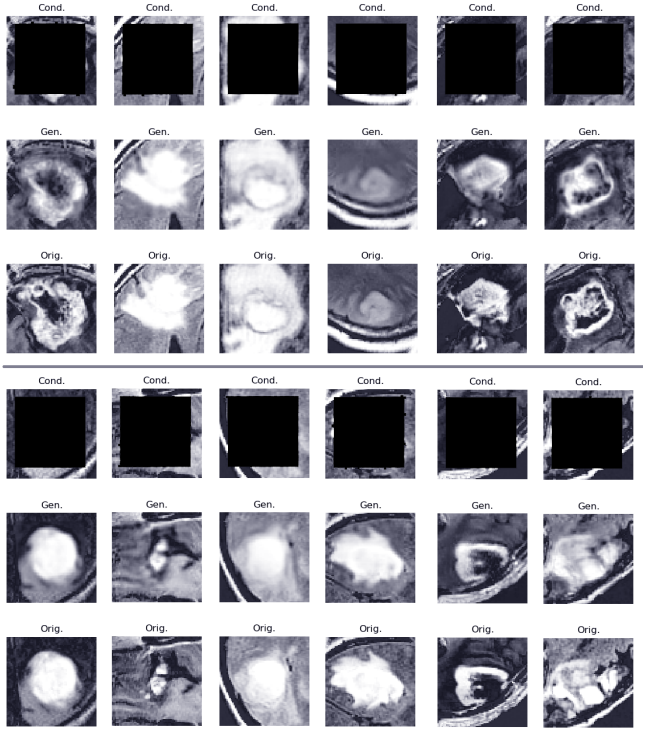


Figure 18: CT-GAN used to inject brain tumors into MRIs of healthy brains. Similar to Fig. 7, Top: context, middle: in-painted result, and bottom: ground-truth. Showing one slice in a 64x64x16 cuboid.

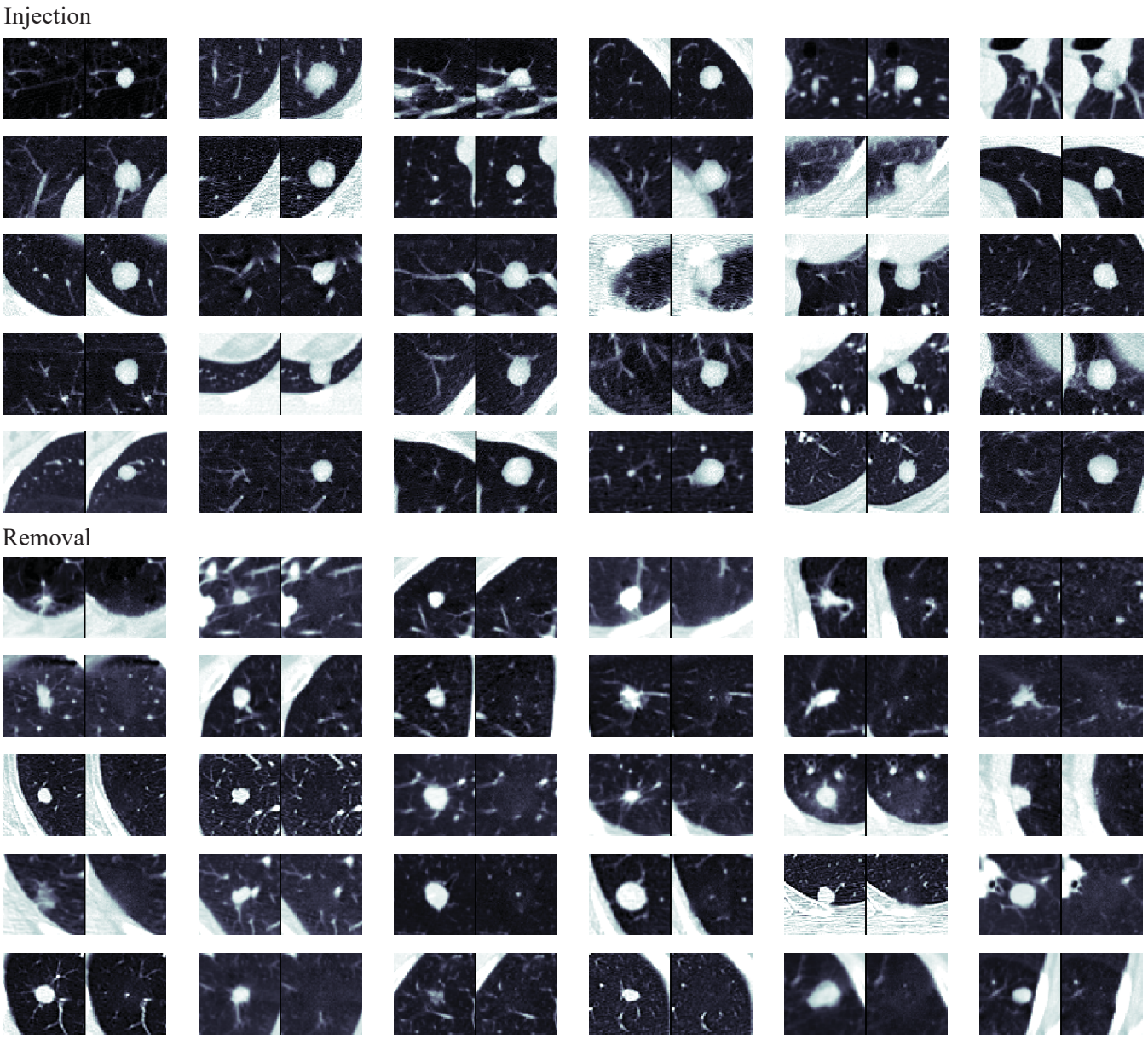


Figure 19: Samples of injected (top) and removed (bottom) pulmonary nodules. For each image, the left side is before tampering and the right side is after. Note, only the middle 2D slice is shown and the images are scaled to different ratios (the source scan).

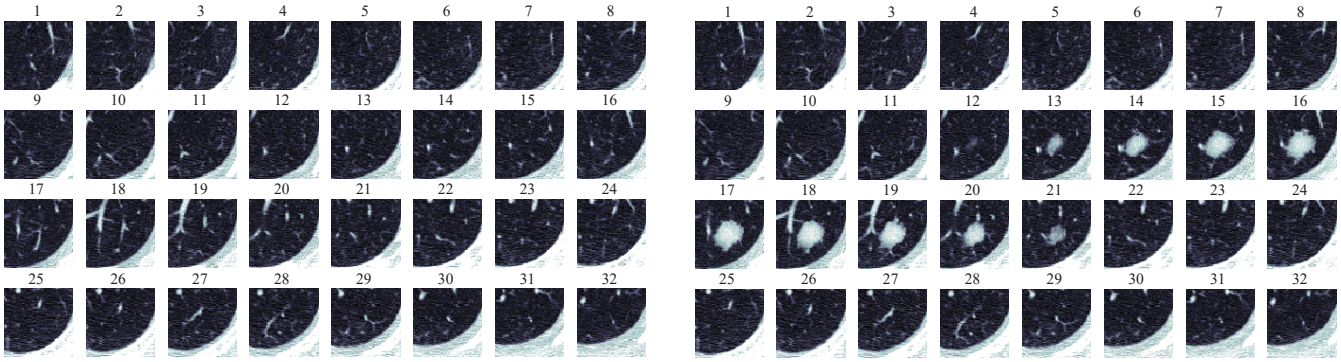


Figure 20: All 32 slices from a sample injection before (left) and after (right) tampering with the CT scan.

Misleading Authorship Attribution of Source Code using Adversarial Learning

Erwin Quiring, Alwin Maier and Konrad Rieck

Technische Universität Braunschweig, Germany

Abstract

In this paper, we present a novel attack against authorship attribution of source code. We exploit that recent attribution methods rest on machine learning and thus can be deceived by adversarial examples of source code. Our attack performs a series of semantics-preserving code transformations that mislead learning-based attribution but appear plausible to a developer. The attack is guided by Monte-Carlo tree search that enables us to operate in the discrete domain of source code. In an empirical evaluation with source code from 204 programmers, we demonstrate that our attack has a substantial effect on two recent attribution methods, whose accuracy drops from over 88% to 1% under attack. Furthermore, we show that our attack can imitate the coding style of developers with high accuracy and thereby induce false attributions. We conclude that current approaches for authorship attribution are inappropriate for practical application and there is a need for resilient analysis techniques.

1 Introduction

The source code of a program often contains peculiarities that reflect individual coding style and can be used for identifying the programmer. These peculiarities—or *stylistic patterns*—range from simple artifacts in comments and code layout to subtle habits in the use of syntax and control flow. A programmer might, for example, favor while-loops even though the use of for-loops would be more appropriate. The task of identifying a programmer based on these stylistic patterns is denoted as *authorship attribution*, and several methods have been proposed to recognize the authors of source code [1, 4, 9, 13] and compiled programs [3, 10, 17, 22].

While techniques for authorship attribution have made great progress in the last years, their robustness against attacks has received only little attention so far, and the majority of work has focused on achieving high accuracy. The recent study by Simko et al. [25], however, shows that developers can manually tamper with the attribution of source code and thus it becomes necessary to reason about attacks that can forge stylistic patterns and mislead attribution methods.

In this paper, we present the first black-box attack against authorship attribution of source code. Our attack exploits that recent attribution methods employ machine learning and thus can be vulnerable to adversarial examples [see 20]. We combine concepts from adversarial learning and compiler engineering, and create adversarial examples in the space of semantically-equivalent programs.

Our attack proceeds by iteratively transforming the source code of a program, such that stylistic patterns are changed while the underlying semantics are preserved. To determine these transformations, we interpret the attack as a game against the attribution method and develop a variant of Monte-Carlo tree search [24] for constructing a sequence of adversarial but plausible transformations. This black-box strategy enables us to construct *untargeted attacks* that thwart a correct attribution as well as *targeted attacks* that imitate the stylistic patterns of a developer.

As an example, Figure 1 shows two transformations performed by our attack on a code snippet from the Google Code Jam competition. The first transformation changes the for-loop to a while-loop, while the second replaces the C++ operator `<<` with the C-style function `printf`. Note that the format string is automatically inferred from the variable type. Both transformations change the stylistic patterns of author A and, in combination, mislead the attribution to author B.

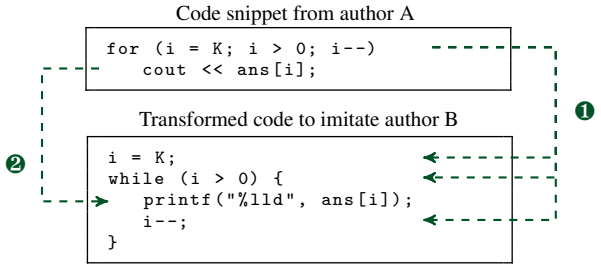


Figure 1: Two iterations of our attack: Transformation ❶ changes the control statement `for` → `while` and transformation ❷ manipulates the API usage `ostream` → `printf` to imitate the stylistic patterns of author B.

We conduct a series of experiments to evaluate the efficacy of our attack using the source code of 204 programmers from the Google Code Jam competition. As targets we consider the recent attribution methods by Caliskan et al. [9] and Abuhamad et al. [1], which provide superior performance compared to related approaches. In our first experiment, we demonstrate that our attack considerably affects both attribution methods [1, 9], whose accuracy drops from over 88% to 1% under attack, indicating that authorship attribution can be automatically thwarted at large scale. In our second experiment, we investigate the effect of targeted attacks. We show that in a group of programmers, each individual can be impersonated by 77% to 81% of the other developers on average. Finally, we demonstrate in a study with 15 participants that code transformed by our attack is plausible and hard to discriminate from unmodified source code.

Our work has implications on the applicability of authorship attribution in practice: We find that both, untargeted and targeted attacks, are effective, rendering the reliable identification of programmers questionable. Although our approach builds on a fixed set of code transformations, we conclude that features regularly manipulated by compilers, such as specific syntax and control flow, are not reliable for constructing attribution methods. As a consequence, we suggest to move away from these features and seek for more reliable means for identifying authors in source code.

Contributions. In summary, we make the following major contributions in this paper:

- *Adversarial learning on source code.* We present the first automatic attack against authorship attribution of source code. We consider targeted as well as untargeted attacks of the attribution method.
- *Monte-Carlo tree search.* We introduce Monte-Carlo tree search as a novel approach to guide the creation of adversarial examples, such that feasibility constraints in the domain of source code are satisfied.
- *Black-box attack strategy.* The devised attack does not require internal knowledge of the attribution method, so that it is applicable to any learning algorithm and suitable for evading a wide range of attribution methods.
- *Large-scale evaluation.* We empirically evaluate our attack on a dataset of 204 programmers and demonstrate that manipulating the attribution of source code is possible in the majority of the considered cases.

The remainder of this paper is organized as follows: We review the basics of program authorship attribution in Section 2. The design of our attack is lay out in Section 3, while Section 4 and 5 discuss technical details on code transformation and adversarial learning, respectively. An empirical evaluation of our attack is presented in Section 6 along with a discussion of limitations in Section 7. Section 8 discusses related work and Section 9 concludes the paper.

2 Authorship Attribution of Source Code

Before introducing our attack, we briefly review the design of methods for authorship attribution. To this end, we denote the source code of a program as x and refer to the set of all possible source codes by $\mathcal{X}$. Moreover, we define a finite set of authors $\mathcal{Y}$. Authorship attribution is then the task of identifying the author $y \in \mathcal{Y}$ of a given source code $x \in \mathcal{X}$ using a classification function f such that $f(x) = y$. In line with the majority of previous work, we assume that the programs in $\mathcal{X}$ can be attributed to a single author, as the identification of multiple authors is an ongoing research effort [see 12, 17].

Equipped with this basic notation, we proceed to discuss the two main building blocks of current methods for authorship attribution: (a) the extraction of features from source code and (b) the application of machine learning for constructing the classification function.

2.1 Feature Extraction

The coding habits of a programmer can manifest in a variety of stylistic patterns. Consequently, methods for authorship attribution need to extract an expressive set of features from source code that serve as basis for inferring these patterns. In the following, we discuss the major types of these features and use the code sample in Figure 2 as a running example throughout the paper.

```
1 int foo(int a){
2   int b;
3   if (a < 2)       // base case
4     return 1;
5   b = foo(a - 1); // recursion
6   return a * b;
7 }
```

Figure 2: Exemplary code sample (see Figure 3, 5, and 6)

Layout features. Individual preferences of a programmer often manifest in the layout of the code and thus corresponding features are a simple tool for characterizing coding style. Examples for such features are the indentation, the form of comments and the use of brackets. In Figure 2, for instance, the indentation width is 2, comments are provided in C++ style, and curly braces are opened on the same line.

Layout features are trivial to forge, as they can be easily modified using tools for code formatting, such as GNU indent. Moreover, many integrated development editors automatically normalize source code, such that stylistic patterns in the layout are unified.

Lexical features. A more advanced type of features can be derived from the lexical analysis of source code. In this analysis stage, the source code is partitioned into so-called *lexems*, tokens that are matched against the terminal symbols of the language grammar. These lexems give rise to a strong

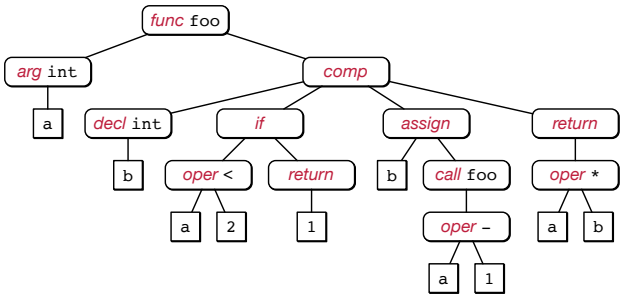


Figure 3: Abstract syntax tree (AST) for code sample in Figure 2.

set of string-based features jointly covering keywords and symbols. For example, in Figure 2, the frequency of the lexem `int` is 3, while it is 2 for the lexem `foo`.

In contrast to code layout, lexical features cannot be easily manipulated, as they implicitly describe the syntax and semantics of the source code. While the lexem `foo` in the running example could be easily replaced by another string, adapting the lexem `int` requires a more involved code transformation that introduces a semantically equivalent data type. We introduce such a transformation in Section 4.

Syntactic features. The use of syntax and control flow also reveals individual stylistic patterns of programmers. These patterns are typically accessed using the *abstract syntax tree* (AST), a basic data structure of compiler design [2]. As an example, Figure 3 shows a simplified AST of the code snippet from Figure 2. The AST provides the basis for constructing an extensive set of syntactic features. These features can range from the specific use of syntactic constructs, such as unary and ternary operators, to generic features characterizing the tree structure, such as the frequency of adjacent nodes. In Figure 3, there exist 21 pairs of adjacent nodes including, for example, `(func foo)`→`(arg int)` and `(return)`→`(1)`.

Manipulating features derived from an AST is challenging, as even minor tweaks in the tree structure can fundamentally change the program semantics. As a consequence, transformations to the AST need to be carefully designed to preserve the original semantics and to avoid unintentional side effects. For example, removing the node pair `(decl int)`→`(b)` from the AST in Figure 3 requires either replacing the type or the name of the variable without interfering with the remaining code. In practice, such transformations are often non-trivial and we discuss the details of manipulating the AST in Section 4.

2.2 Machine Learning

The three feature types (layout, lexical, syntactic) provide a broad view on the characteristics of source code and are used by many attribution methods as the basis for applying machine-learning techniques [e.g., 1, 4, 9, 21]

From code to vectors. Most learning algorithms are designed to operate on vectorial data and hence the first step for application of machine learning is the mapping of code to a vector space using the extracted features. Formally, this mapping can be expressed as $\phi : \mathcal{X} \longrightarrow \mathcal{F} = \mathbb{R}^d$ where $\mathcal{F}$ is a d dimensional vector space describing properties of the extracted features. Different techniques can be applied for constructing this map, which may include the computation of specific metrics as well as generic embeddings of features and their relations, such as a TF-IDF weighting [1, 9].

Surprisingly, the feature map ϕ introduces a non-trivial hurdle for the construction of attacks. The map ϕ is usually not bijective, that is, we can map a given source code x to a feature space but are unable to automatically construct the source code x' for a given point $\phi(x')$. Similarly, it is difficult to predict how a code transformation $x \mapsto x'$ changes the position in feature space $\phi(x) \mapsto \phi(x')$. We refer to this problem as the *problem-feature space dilemma* and discuss its implications in Section 3.

Multiclass classification. Using a feature map ϕ, we can apply machine learning for identifying the author of a source code. Typically, this is done by training a *multiclass classifier* $g : \mathcal{X} \longrightarrow \mathbb{R}^{|\mathcal{Y}|}$ that returns scores for all authors $\mathcal{Y}$. An attribution is obtained by simply computing

$$f(x) = \arg\max_{y \in \mathcal{Y}} g_y(x).$$

This setting has different advantages: First, one can investigate all top-ranked authors. Second, one can interpret the returned scores for determining the confidence of an attribution. We make use of the latter property for guiding our attack strategy and generating adversarial examples of source code (see Section 5)

Different learning algorithms have been used for constructing the multiclass classifier g, as for example, support vector machines [21], random forests [9], and recurrent neural networks [1, 4]. Attacking each of these learning algorithms individually is a tedious task and thus we resort to a *black-box attack* for misleading authorship attribution. This attack does not require any knowledge of the employed learning algorithm and operates with the output $g(x)$ only. Consequently, our approach is agnostic to the learning algorithm as we demonstrate in the evaluation in Section 6.

3 Misleading Authorship Attribution

With a basic understanding of authorship attribution, we are ready to investigate the robustness of attribution methods and to develop a corresponding black-box attack. To this end, we first define our threat model and attack scenario before discussing technical details in the following sections.

3.1 Threat Model

For our attack, we assume an adversary who has black-box access to an attribution method. That is, she can send an arbitrary source code x to the method and retrieve the corresponding prediction $f(x)$ along with prediction scores $g(x)$. The training data, the extracted features, and the employed learning algorithm, however, are unknown to the adversary, and hence the attack can only be guided by iteratively probing the attribution method and analyzing the returned prediction scores. This setting resembles a classic *black-box attack* as studied by Tramèr et al. [26] and Papernot et al. [19]. As part of our threat model, we consider two types of attacks—*untargeted* and *targeted attacks*—that require different capabilities of the adversary and have distinct implications for the involved programmers.

Untargeted attacks. In this setting, the adversary tries to mislead the attribution of source code by changing the classification into *any* other programmer. This attack is also denoted as *dodging* [23] and impacts the correctness of the attribution. As an example, a benign programmer might use this attack strategy for concealing her identity before publishing the source code of a program.

Targeted attacks. The adversary tries to change the classification into a chosen *target* programmer. This attack resembles an *impersonation* and is technically more advanced, as we need to transfer the stylistic patterns from one developer to another. A targeted attack has more severe implications: A malware developer, for instance, could systematically change her source code to blame a benign developer.

Furthermore, we consider two scenarios for targeted attacks: In the first scenario, the adversary has no access to source code from the target programmer and thus certain features, such as variable names and custom types, can only be guessed. In the second scenario, we assume that the adversary has access to two files of source code from the target developer. Both files are not part of the training- or test set and act as external source for extracting template information, such as recurring custom variable names.

In addition, we test a scenario where the targeted attack solely rests on a separate training set, without access to the output of the original classifier. This might be the case, for instance, if the attribution method is secretly deployed, but code samples are available from public code repositories. In this scenario, the adversary can learn a substitute model with the aim that her adversarial example—calculated on the substitute—also transfers to the original classifier.

3.2 Attack Constraints

Misleading the attribution of an author can be achieved with different levels of sophistication. For example, an adversary may simply copy code snippets from one developer for impersonation or heavily obfuscate source code for dodging. These trivial attacks, however, generate implausible code and are easy to detect. As a consequence, we define a set of constraints for our attack that should make it hard to identify manipulated source code.

Preserved semantics. We require that source code generated by our attack is semantically equivalent to the original code. That is, the two codes produce identical outputs given the same input. As it is undecidable whether two programs are semantically equivalent, we take care of this constraint during the design of our code transformations and ensure that each transformation is as semantics-preserving as possible.

Plausible code. We require that all transformations change the source code, such that the result is syntactically correct, readable and plausible. The latter constraint corresponds to the aspect of imperceptibility when adversarial examples are generated in the image domain [11]. In our context, plausibility is important whenever the adversary wants to hide the modification of a source file, for instance, when blaming another developer. For this reason, we do not include junk code or unusual syntax that normal developers would not use.

No layout changes. Layout features such as the tendency to start lines with spaces or tabs are trivial to change with tools for code formatting (see Section 6.4). Therefore, we restrict our attack to the forgery of lexical and syntactic features of source code. In this way, we examine our approach under a more difficult scenario for the attacker where no layout features are exploitable to mislead the attribution.

3.3 Problem-Feature Space Dilemma

The described threat model and attack constraints pose unique challenges to the design of our attack. Our attack jointly operates in two domains: On the one hand, we aim at attacking a classifier in the feature space $\mathcal{F}$. On the other hand, we require the source code to be semantically equivalent and plausible in the problem space $\mathcal{X}$. For most feature maps ϕ, a one-to-one correspondence, however, does not exist between the two spaces and thus we encounter a dilemma.

Problem space $\rightsquigarrow$ feature space. Each change in the source code x may impact a set of features in $\phi(x)$. The exact amount of change is generally not controllable. The correlation of features and post-processing steps in ϕ, such as a TF-IDF weighting, may alter several features, even if only a single statement is changed in the source code. This renders target-oriented modification of the source code difficult.

For example, if the declaration of the variable `b` in line 2 of Figure 2 is moved to line 5, a series of lexical and syntactic features change, such as the frequency of the lexem `b` or the subtree under the node `assign` in Figure 3.

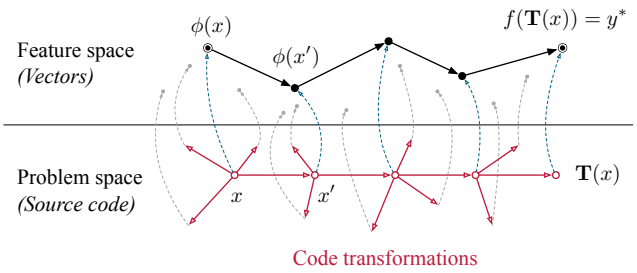


Figure 4: Schematic depiction of our approach. The attack is realized by moving in the problem space using code transformations while being guided by Monte-Carlo tree search in the feature space.

Feature space $\rightsquigarrow$ problem space. Any change to a feature vector $\phi(x)$ must ensure that there exists a plausible source code x in the problem space. Unfortunately, determining x from $\phi(x)$ is not tractable for non-bijective feature maps, and it is impossible to directly apply techniques from adversarial learning that operate in the feature space.

For example, if we calculate the difference of two vectors $\phi(z) = \phi(x) - \phi(x')$, we have no means for determining the resulting source code z. Even worse, it might be impossible to construct z, as the features in $\phi(z)$ can violate the underlying programming language specification, for example, due to feature combinations inducing impossible AST edges.

This dilemma has received little attention in the literature on adversarial learning so far, and it is often assumed that an adversary can change features almost arbitrarily [e.g. 6, 11, 18]. Consequently, our attack does not only pinpoint weaknesses in authorship attribution but also illustrates how adversarial learning can be conducted when the problem and feature space are disconnected.

3.4 Our Attack Strategy

To tackle this challenge, we adopt a mixed attack strategy that combines concepts from compiler engineering and adversarial learning. For the problem space, we develop code transformations (source-to-source compilations) that enable us to maneuver in the problem space and alter stylistic patterns without changing the semantics. For the feature space, we devise a variant of Monte-Carlo tree search that guides the transformations towards a target. This variant considers the attack as a game against the attribution method and aims at reaching a desired output with few transformations.

An overview of our attack strategy is illustrated in Figure 4. As the building blocks of our approach originate from different areas of computer science, we discuss their technical details in separate sections. First, we introduce the concept of semantics-preserving code transformations and present five families of source-to-source transformations (Section 4). Then, we introduce Monte-Carlo tree search as a generic black-box attack for chaining transformations together such that a target in the feature space is reached (Section 5).

4 Code Transformations

The automatic modification of code is a well-studied problem in compiler engineering and source-to-source compilation [2]. Consequently, we build our code transformations on top of the compiler frontend *Clang* [28], which provides all necessary primitives for parsing, transforming and synthesizing C/C++ source code. Note that we do *not* use code obfuscation methods, since their changes are (a) clearly visible, and (b) cannot mislead a classifier to a targeted author. Before presenting five families of transformations, we formally define the task of *code transformation* and introduce additional program representations.

Definition 1. A code transformation $T : \mathcal{X} \longrightarrow \mathcal{X},\ x \mapsto x'$ takes a source code x and generates a transformed version x', such that x and x' are semantically equivalent.

While code transformations can serve various purposes in general [2], we focus on *targeted* transformations that modify only minimal aspects of source code. If multiple source locations are applicable for a transformation, we use a pseudo-random seed to select one location. To chain together targeted transformations, we define *transformation sequences* as follows:

Definition 2. A transformation sequence $\mathbf{T} = T_1 \circ T_2 \circ \cdots \circ T_n$ is the subsequent application of multiple code transformations to a source code x.

To efficiently perform transformations, we make use of different program representations, where the AST is the most important one. To ease the realization of involved transformations, however, we employ two additional program representations that augment our view on the source code.

Control-flow graph with use-define chains. The control flow of a program is typically represented by a *control-flow graph* (CFG) where nodes represent statements and edges the flow of control. Using the CFG, it is convenient to analyze the execution order of statements. We further extend the CFG provided by Clang with *use-define chains* (UDCs). These chains unveil dependencies between usages and the definitions of a variable. With the aid of UDCs, we can trace the flow of data through the program and identify data dependencies between local variables and function arguments. Figure 5 shows a CFG with use-define chains.

Declaration-reference mapping. We additionally introduce a declaration-reference mapping (DRM) that extends the AST and links each declaration to all usages of the declared variable. As an example, Figure 6 shows a part of the AST together with the respective DRM for the code sample from Figure 2. This code representation enables navigation between declarations and variables, which allows us to efficiently rename variables or check for the sound transformation of data types. Note the difference between use-define

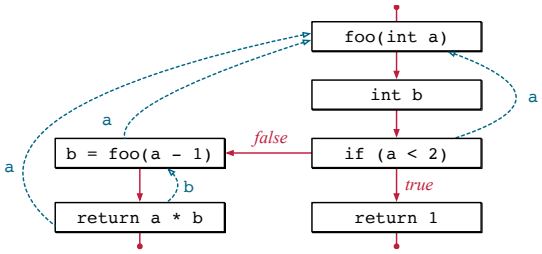


Figure 5: Control-flow graph with use-define chains for the code snippet from Figure 2. The control flow is shown in red (solid), use-define chains in blue (dashed).

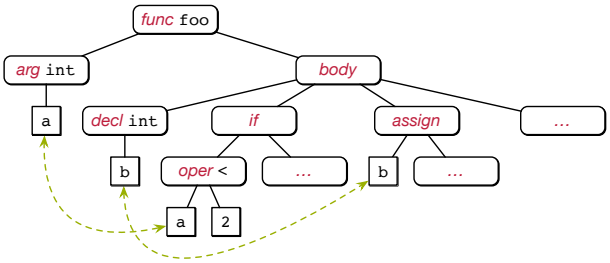


Figure 6: Abstract syntax tree with declaration-reference mapping for the code snippet from Figure 2. Declaration references are shown in green (dashed).

Table 1: Implemented families of transformations.

Transformation family	#	AST	CFG	UDC	DRM
Control transformations	5	•	•	•	
Declaration transformations	14	•			•
API transformations	9	•	•		•
Template transformations	4	•			•
Miscellaneous transformations	4	•			

chains and declaration-reference mappings. The former connects variable usages to variable definitions, while the latter links variable usages to variable declarations.

Based on these program representations, we develop a set of generic code transformations that are suitable for changing different stylistic patterns. In particular, we implement 36 transformers that are organized into five families. Table 1 provides an overview of each family together with the program representation used by the contained transformers.

In the following, we briefly introduce each of the five families. For a detailed listing of all 36 transformations, we refer the reader to Table 8 in Appendix C.

Control transformations. The first family of source-to-source transformations rewrites control-flow statements or modifies the control flow between functions. In total, the family contains 5 transformations. For example, the control-flow statements `while` and `for` can be mutually interchanged by two transformers. These transformations address a developer's preference to use a particular iteration type. As another example, Figure 7 shows the automatic creation of a function. The transformer moves the inner block of the for-statement to a newly created function. This transformation involves passing variables as function arguments, updating their values and changing the control flow of the caller and callee.

Declaration transformations. This family consists of 14 transformers that modify, add or remove declarations in source code. For example, in a widening conversion, the type of a variable is changed to a larger type, for example, `int` to `long`. This rewriting mimics a programmer's preference for particular data types. Declaration transformations make it necessary to update all usages of variables which

```
for (int j = i; j < i + k; j++) {
  if (s[j] == '-')
    s[j] = '+';
  else
    s[j] = '-';
}
```

```
inline void setarray(string& s, int& j) {
  if (s[j] == '-')
    s[j] = '+';
  else
    s[j] = '-';
}
[...]
for (int j = i; j < i + k; j++)
  setarray(s, j);
```

❶ ❷

Figure 7: Example of a control transformation. ❶ moves the compound statement into an own function and passes all variables defined outside the block as parameters. ❷ calls the new function at the previous location.

can be elegantly carried out using the DRM representation. Replacing an entire data type is a more challenging transformation, as we need to adapt all usages to the type, including variables, functions and return values. Figure 8 shows the replacement of the C++ `string` object with a conventional `char` array, where the declaration and also API functions, such as `size`, are modified. Note that in our current implementation of the transformer the `char` array has a fixed size and thus is not strictly equivalent to the C++ `string` object.

API transformations. The third family contains 9 transformations and exploits the fact that various APIs can be used

```
string s;
cin >> s;
for (int i = 0; i < s.size(); i++) {
    if (s[i] == '+')
```

```
char s[1000];
cin >> s;
for (int i = 0; i < strlen(s); i++) {
  if (s[i] == '+')
```

❶ ❷

Figure 8: Example of a declaration transformation. ❶ replaces the declaration of the C++ `string` object with a `char` array, ❷ adapts all uses of the object.

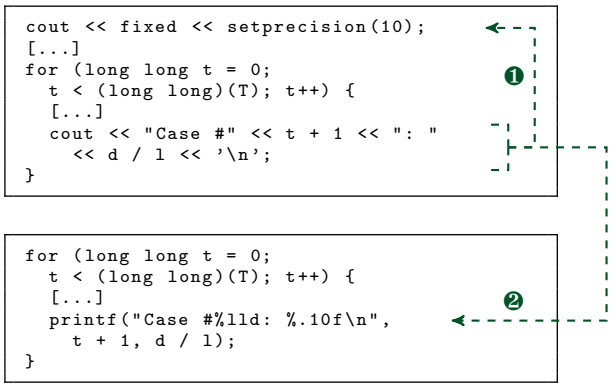


Figure 9: Example of an API transformation. ❶ determines the current precision for output; ❷ replaces the C++ API with a C-style printf. The format specifier respects the precision and the data type of the variable.

to solve the same problem. Programmers are known to favor different APIs and thus tampering with API usage is an effective strategy for changing stylistic patterns. For instance, we can choose between various ways to output information in C++, such as `printf`, `cout`, or `ofstream`.

As an example, Figure 9 depicts the replacement of the object `cout` by a call to `printf`. To this end, the transformer first checks for the decimal precision of floating-point values that `cout` employs, that is, we use the CFG to find the last executed `fixed` and `setprecision` statement. Next, the transformer uses the AST to resolve the final data type of each `cout` entry and creates a respective format string for `printf`.

Template transformations. The fourth family contains 4 transformations that insert or change code patterns based on a give template. For example, authors tend to reuse specific variable names, constants, and type definitions. If a template file is given for a target developer, these information are extracted and used for transformations. Otherwise, default values that represent general style patterns are employed. For instance, variable names can be iteratively renamed into default names like `i`, `j`, or `k` until a developer's tendency to declare control statement variables is lost (dodging attack) or gets matched (impersonation attack).

Miscellaneous transformations. The last family covers 4 transformations that conduct generic changes of code statements. For example, the use of curly braces around compound statements is a naive but effective stylistic pattern for identifying programmers. The compound statement transformer thus checks if the body of a control statement can be enclosed by curly braces or the other way round. In this way, we can add or remove a compound statement in the AST.

Another rather simple stylistic pattern is the use of return statements, where some programmers omit these statements in the `main` function and others differ in whether they return a constant, integer or variable. Consequently, we design a transformer that manipulates return statements.

5 Monte-Carlo Tree Search

Equipped with different code transformations for changing stylistic patterns, we are ready to determine a sequence of these transformations for untargeted and targeted attacks. We aim at a short sequence, which makes the attack less likely to be detected. Formally, our objective is to find a short transformation sequence $\mathbf{T}$ that manipulates a source file x, such that the classifier f predicts the target label y^*:

$$f\big(\mathbf{T}(x)\big) = y^* \,. \tag{1}$$

In the case of an untargeted attack, y^* represents any other developer than the original author y^s, that is, $y^* \neq y^s$. In the case of a targeted attack, y^* is defined as a particular target author y^t.

As we are unable to control how a transformation $T(x)$ moves the feature vector $\phi(x)$, several standard techniques for solving the problem in (1) are not applicable, such as gradient-based methods [e.g. 11]. Therefore, we require an algorithm that works over a search space of discrete objects such as the different transformations of the source code. As a single transformation does not necessarily change the score of the classifier, simple approximation techniques like Hill Climbing that only evaluate the neighborhood of a sample fail to provide appropriate solutions.

As a remedy, we construct our attack algorithm around the concept of *Monte-Carlo tree search* (MCTS)—a strong search algorithm that has proven effective in AI gaming with AlphaGo [24]. Similar to a game tree, our variant of MCTS creates a search tree for the attack, where each node represents a state of the source code and the edges correspond to transformations. By moving in this tree, we can evaluate the impact of different transformation sequences before deciding on the next move. Figure 10 depicts the four basic steps of our algorithm: selection, simulation, expansion and backpropagation.

Selection. As the number of possible paths in the search tree grows exponentially, we require a *selection policy* to identify the next node for expansion. This policy balances the tree's exploration and exploitation by alternately selecting nodes that have not been evaluated much (exploration) and nodes that seem promising to obtain a better result (exploitation). Following this policy, we start at the root node and recursively select a child node until we find a node u which was not evaluated before. Appendix A gives more information about the used selection policy.

Simulation & Expansion. We continue by generating a set of unique transformation sequences with varying length that start at u. We bound the length of each sequence by a predefined value. In our experiments, we create sequences with up to 5 transformations. For each sequence, we determine the classifier score by providing the modified source code to the attribution method. The right plot in Figure 10 exemplifies

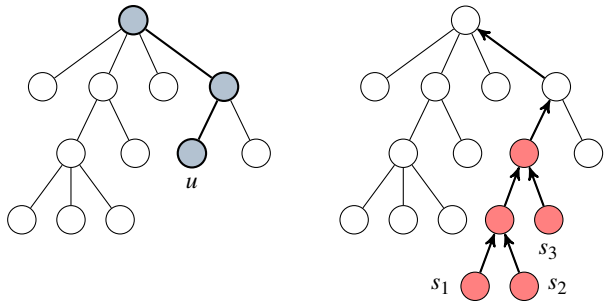


Figure 10: Basic steps of Monte-Carlo tree search. The left plot shows the selection step, the right plot the simulation, expansion and backpropagation.

the step: we create three sequences where two have the same first transformation. Next, we create the respective tree nodes. As two sequences start with the same transformation, they also share a node in the search tree.

Backpropagation. As the last step, we propagate the obtained classifier scores from the leaf node of each sequence back to the root. During this propagation, we update two statistics in each node on the path: First, we increment a counter that keeps track of how often a node has been part of a transformation sequence. In Figure 10, we increase the visit count of node u and the nodes above by 3. Second, we store the classifier scores in each node that have been observed in its subtree. For example, node u in Figure 10 stores the scores from s_1, s_2 and s_3. Both statistics are used by the selection policy and enable us to balance the exploration and exploitation of the tree in the next iterations.

Iteration. We repeat these four basic steps until a predefined iteration constraint is reached. After obtaining the resulting search tree, we identify the root's child node with the highest average classifier score and make it the novel root node of the tree. We then repeat the entire process again. The attack is stopped if we succeed, we reach a previously fixed number of iterations, or we do not obtain any improvement over multiple iterations.

Appendix A provides more implementation details on our variant of MCTS. We finally note that the algorithm resembles a black-box attack, as the inner working of the classifier f is not considered.

6 Evaluation

We proceed with an empirical evaluation of our attacks and investigate the robustness of source-code authorship attribution in a series of experiments. In particular, we investigate the impact of untargeted and targeted attacks on two recent attribution methods (Section 6.2 & 6.3). Finally, we verify in Section 6.4 that our initially imposed attack constraints are fulfilled.

6.1 Experimental Setup

Our empirical evaluation builds on the methods developed by Caliskan et al. [9] and Abuhamad et al. [1], two recent approaches that operate on a diverse set of features and provide superior performance in comparison to other attribution methods. For our evaluation, we follow the same experimental setup as the authors, re-implement their methods and make use of a similar dataset.

Dataset & Setup. We collect C++ files from the 2017 *Google Code Jam* (GCJ) programming competition [29]. This contest consists of various rounds where several participants solve the same programming challenges. This setting enables us to learn a classifier for attribution that separates stylistic patterns rather than artifacts of the different challenges. Moreover, for each challenge, a test input is available that we can use for checking the program semantics. Similar to previous work, we select eight challenges from the competition and collect the corresponding source codes from all authors who solved these challenges.

In contrast to prior work [1, 9], however, we set more stringent restrictions on the source code. We filter out files that contain incomplete or broken solutions. Furthermore, we format each source code using `clang-format` and expand macros, which removes artifacts that some authors introduce to write code more quickly during the contest. Our final dataset consists of 1,632 files of C++ code from 204 authors solving the same 8 programming challenges of the competition.

For the evaluation, we use a *stratified* and *grouped* k-fold cross-validation where we split the dataset into $k-1$ challenges for training and 1 challenge for testing. In this way, we ensure that training is conducted on different challenges than testing. For each of the k folds, we perform feature selection on the extracted features and then train the respective classifier as described in the original publications. We report results averaged over all 8 folds.

Implementation. We implement the attribution methods and our attack on top of Clang [28], an open-source C/C++ frontend for the LLVM compiler framework. For the method of Caliskan et al. [9], we re-implement the AST extraction and use the proposed random forest classifier for attributing programmers. The approach by Abuhamad et al. [1] uses lexical features that are passed to a long short-term memory (LSTM) neural network for attribution. Table 2 provides an overview of both methods. For further details on the fea-

Method	Lex	Syn	Classifier	Accuracy
Caliskan et al. [9]	•	•	RF	90.4% ± 1.7%
Abuhamad et al. [1]	•		LSTM	88.4% ± 3.7%

Table 2: Implemented attribution methods and their reproduced accuracy. (Lex = Lexical features, Syn = Syntactic features)

Method	Success rate of our attack		
	Untargeted	Targeted T+	Targeted T-
Caliskan et al. [9]	99.2%	77.3%	71.2%
Abuhamad et al. [1]	99.1%	81.3%	69.1%

Table 3: Performance of our attack as average success rate. The targeted attack is conducted with template (T+) and without template (T-).

ture extraction and learning process, we refer the reader to the respective publications [1, 9].

As a sanity check, we reproduce the experiments conducted by Caliskan et al. [9] and Abuhamad et al. [1] on our dataset. Table 2 shows the average attribution accuracy and standard deviation over the 8 folds. Our re-implementation enables us to differentiate the 204 developers with an accuracy of 90% and 88% on average, respectively. Both accuracies come close to the reported results with a difference of less than 6%, which we attribute to omitted layout features and the stricter dataset.

6.2 Untargeted Attack

In our first experiment, we investigate whether an adversary can manipulate source code such that the original author is not identified. To this end, we apply our untargeted attack to each correctly classified developer from the 204 authors. We repeat the attack for all 8 challenges and aggregate the results.

Attack performance. Table 3 presents the performance of the attack as the ratio of successful evasion attempts. Our attack has a strong impact on both methods and misleads the attribution in 99% of the cases, irrespective of the considered features and learning algorithm. As a result, the source code of almost all authors can be manipulated such that the attribution fails.

Attack analysis. To investigate the effect of our attack in more detail, we compute the ratio of changed features per adversarial sample. Figure 11 depicts the distribution over all samples. The method by Caliskan et al. [9] exhibits a bimodal distribution. The left peak shows that a few changes, such as the addition of include statements, are often sufficient to mislead attribution. For the majority of samples, however, the attack alters 50% of the features, which indicates the tight correlation between different features (see Section 3.3). A key factor to this correlation is the TF-IDF weighting that distributes minor changes over a large set of features.

In comparison, less features are necessary to evade the approach by Abuhamad et al. [1], possibly due to the higher sparsity of the feature vectors. Each author has 12.11% non-zero features on average, while 53.12% are set for the method by Caliskan et al. [9]. Thus, less features need to be changed and in consequence each changed feature impacts fewer other features that remain zero.

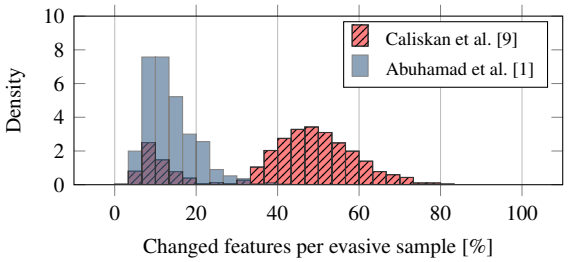


Figure 11: Untargeted attack: Histogram over the number of changed features per successful evasive sample for both attribution methods.

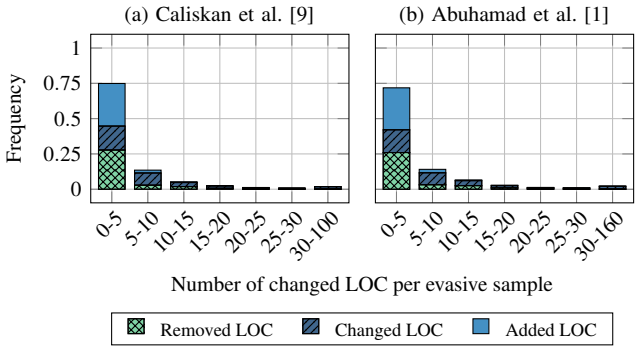


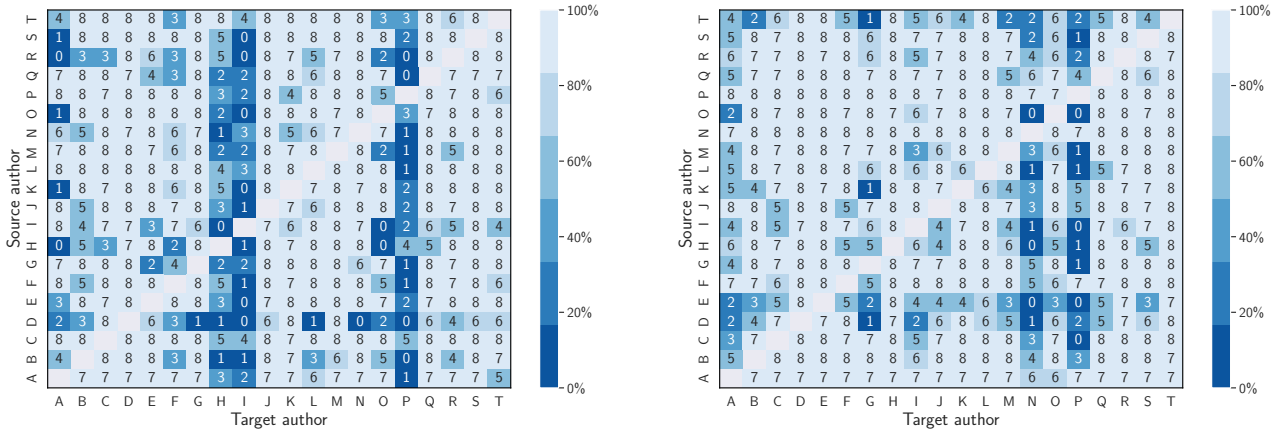
Figure 12: Untargeted attack: Stacked histogram over the number of changed lines of code (LOC) per successful evasive sample for both attribution methods. The original source files have 74 lines on average (std: 38.44).

Although we observe a high number of changed features, the corresponding changes to the source code are minimal. Figure 12 shows the number of added, changed and removed lines of code (LOC) determined by a context-diff with difflib for each source file before and after the attack. For the majority of cases in both attribution methods, less than 5 lines of code are added, removed or changed. This low number exemplifies the targeted design of our code transformations that selectively alter characteristics of stylistic patterns.

Summary. Based on the results from this experiment, we summarize that our untargeted attack severely impacts the performance of the methods by Caliskan et al. [9] and Abuhamad et al. [1]. We conclude that other attribution methods employing similar features and learning algorithms also suffer from this problem and hence cannot provide a reliable attribution in presence of an adversary.

6.3 Targeted Attack

We proceed to study the targeted variant of our attack. We consider pairs of programmers, where the code of the source author is transformed until it is attributed to the target author. Due to the quadratic number of pairs, we perform this experiment on a random sample of 20 programmers. This results in 380 source-target pairs each covering the source code of 8 challenges. Table 7 in Appendix B provides a list of the



(a) Caliskan et al. [9]

(b) Abuhamad et al. [1]

Figure 13: Impersonation matrix for both attribution methods. Each cell indicates the number of successful attack attempts for the 8 challenges.

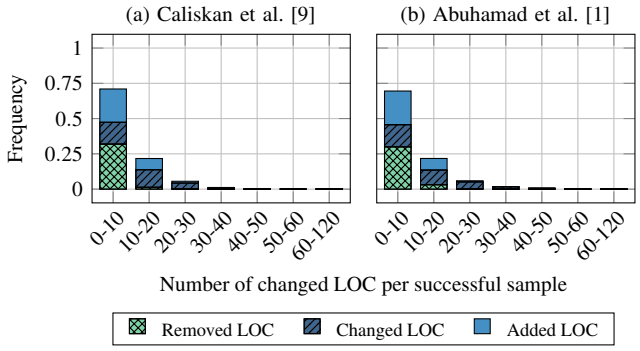


Figure 14: Targeted attack: Stacked histogram over the number of changed lines of code (LOC) per successful impersonation for both attribution methods. The original source files have 74 lines on average (std: 38.44).

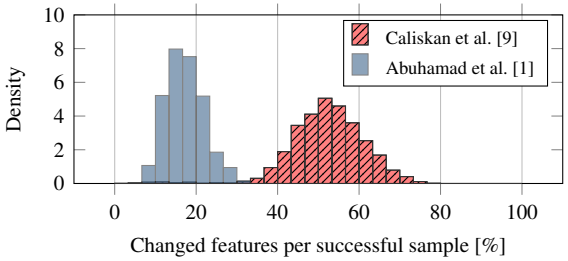


Figure 15: Targeted attack: Histogram over the number of changed features per successful impersonation for both attribution methods.

selected authors. We start with the scenario where we retrieve two samples of source code for each of the 20 programmers from various GCJ challenges—not part of the fixed 8 train-test challenges—to support the template transformations.

Attack performance. Table 3 depicts the success rate of our attack for both attribution methods. We can transfer the prediction from one to another developer in 77% and 81% of all cases, respectively, indicating that more than three out of four programmers can be successfully impersonated.

In addition, Figure 13 presents the results as a matrix, where the number of successful impersonations is visually depicted. Note that the value in each cell indicates the absolute number of successful impersonations for the 8 challenges associated with each author pair. We find that a large set of developers can be imitated by almost every other developer. Their stylistic patterns are well reflected by our transformers and thus can be easily forged. By contrast, only the developers I and P have a small impersonation rate for Caliskan et al. [9], yet 68% and 79% of the developers can still imitate the style of I and P in at least one challenge.

Attack analysis. The number of altered lines of code also remains small for the targeted attacks. Figure 14 shows that in most cases only 0 to 10 lines of code are affected. At the same time, the feature space is substantially changed. Figure 15 depicts that both attribution methods exhibit a similar distribution as before in the untargeted attack—except that the left peak vanishes for the method of Caliskan et al. [9]. This means that each source file requires more than a few targeted changes to achieve an impersonation.

Table 4: Usage of transformation families for impersonation

Transformation Family	Cal. [9]	Abu. [1]
Control Transformers	8.43%	9.72%
Declaration Transformers	14.11%	17.88%
API Transformers	29.90%	19.60%
Miscellaneous Transformers	9.15%	4.76%
Template Transformers	38.42%	48.04%

Table 4 shows the contribution of each transformation family to the impersonation success. All transformations are necessary to achieve the reported attack rates. A closer look reveals that the method by Abuhamad et al. [1] strongly rests on the usage of template transformers, while the families are more balanced for the approach by Caliskan et al. [9]. This

Source Author I

```
1  cout << std::fixed;
2
3
4  for (long long ccr = 1; ccr <= t; ++ccr) {
5    double d, n, ans = INT_MIN;
6    cin >> d >> n;
7    for (double i = 0; i < n; ++i) {
8      double k, s;
9      cin >> k >> s;
10     [...]
11   }
12
13   ans = d / ans;
14   cout << "Case #" << ccr << ": " <<
         setprecision(7) << ans << "\n";
15
16 }
```

Source ⟶ Target

```
1  typedef double td_d;                          ❶
2  [...]
3  long long ccr = 1;
4  while (ccr <= t) {                            ❷
5    td_d d, n, ans = INT_MIN;
6    cin >> d >> n;
7    td_d i;                                     ❸
8    for (i = 0; i < n; ++i) {
9      td_d k, s;
10     cin >> k >> s;
11     [...]
12   }
13   ans = d / ans;
14   printf("Case #%lld: %.7f\n",                ❹
15     ccr, ans);
16   ++ccr;
17 }
```

Target Author P

```
1  int T, cas = 0;
2  cin >> T;
3  while (T--) {
4    int d, n;
5    cin >> d >> n;
6    double t = 0;
7    while (n--) {
8      int k, s;
9      cin >> k >> s;
10     t = max((1.0 * d - k) / s, t);
11   }
12   double ans = d / t;
13   printf("Case #%d: %.10f\n", ++cas, ans);
14 }
```

Iteration	Transformer	Description
❶	Typedef	adds typedef and replaces all locations with previous type by novel typedef.
❷	For statement	converts `for`-statement into an equivalent `while`-statement, as target tends to solve problems via `while`-loops.
❸	Init-Decl	moves a declaration out of the control statement which mimics the declaration behavior of while-statements.
❹	Output API	substitutes C++ API for writing output by C API `printf`. To this end, it determines the precision of output statements by finding `fixed` (line 1) and `setprecision` (line 14) commands.

Figure 16: Impersonation example from our evaluation for the GCJ problem *Steed 2: Cruise Control*. The upper left listing shows the original source file, the upper right its modified version such that it is classified as the target author. For comparison, the lower left listing shows the original source file from the target author (which was not available for the attacker). The table lists the necessary transformations.

difference can be attributed to the feature sets, where the former method relies on simple lexical features only and the latter extracts more involved features from the AST.

Case Study. To provide further intuition for a successful impersonation, Figure 16 shows a case study from our evaluation. The upper two panels present the code from the source author in original and transformed form. The lower left panel depicts the original source text from the target author for the same challenge. Note that the attack has no access to this file. The table lists four conducted transformations. For instance, the target author has the stylistic pattern to use `while` statements, C functions for the output, and particular typedefs. By changing these patterns, our attack succeeds in misleading the attribution method.

Attack without template. We additionally examine the scenario when the adversary has no access to a template file of the target developer. In this case, our template transformers can only try common patterns, such as the iteration variables `i`, `j`, ..., `k` or typedef `ll` for the type `long long`. Table 3 shows the results of this experiment as well. Still, we achieve an impersonation rate of 71% and 69%—solely by relying on the feedback from the classifier. The number of altered lines of code and features correspond to Figures 14 and 15.

Contrary to expectation, without a template, the approach by Abuhamad et al. [1] is harder to fool than the method by Caliskan et al. [9]. As the lexical features rest more on simple declaration names and included libraries, they are harder to guess without a template file. However, if a template file is available, this approach is considerably easier to evade.

Attack with substitute model. We finally demonstrate that an impersonation is even possible without access to the prediction of the original classifier, only relying on a substitute model trained from separate data. We split our training set into disjoint sets with three files per author to train the original and substitute model, respectively. We test the attack on the method by Caliskan et al. [9], which is the more robust attribution under attack. By the nature of this scenario, the adversary can use two files to support the template transformations.

Adversarial examples—generated with the substitute model—transfer in 79% of the cases to the original model, that is, attacks successful against the substitute model are also effective against the original in the majority of the cases. This indicates that our attack successfully changes indicative features for a target developer across models. The success rate of our attack on the original model is 52%. Due to the reduced number of training files in this experiment, the attack

is harder, as the coding habits are less precisely covered by the original and substitute models. Still, we are able to impersonate every second developer with no access to the original classifier.

Summary. Our findings show that an adversary can automatically impersonate a large set of developers without and with access to a template file. We conclude that both considered attribution methods can be abused to trigger false allegations—rendering a real-world application dangerous.

6.4 Preserved Semantics and Plausibility

In the last experiment, we verify that our adversarial code samples comply with the attack constraints specified in Section 3.2. That is, we empirically check that (a) the semantics of the transformed source code are preserved, (b) the generated code is plausible to a human analyst, and (c) layout features can be trivially evaded.

Preserved semantics. We begin by verifying the semantics of the transformed source code. In particular, we use the test file from each challenge of the GCJ competition to check that the transformed source code provides the same solution as the original code. In all our experiments, we can verify that the output remains unchanged for each manipulated source code sample before and after our attack.

Plausible code. Next, we check that our transformations lead to plausible code and conduct a discrimination test with 15 human subjects. The group consists of 4 undergraduate students, 6 graduate students and 5 professional computer scientists. The structure of the test follows an *AXY-test*: Every participant obtains 9 files of source code—each from a different author but for the same GCJ challenge. These 9 files consists of 3 unmodified source codes as reference (A) and 6 sources codes (XY) that need to be classified as either *original* or *modified*. The participants are informed that 3 of the samples are modified. We then ask each participant to identify the unknown samples and to provide a short justification.

The results of this empirical study are provided in Table 5. On average, the participants are able to correctly classify 60% of the provided files which is only marginally higher than random guessing. This result highlights that it is hard to decide whether source code has been modified by our attack or not. In several cases, the participants falsely assume that unused `typedef` statements or an inconsistent usage of operators are modifications.

Evasion of layout features. Finally, we demonstrate that layout features can be trivially manipulated, so that it is valid to restrict our approach to the forgery of lexical and syntactic features. To this end, we train a random forest classifier *only* on layout features as extracted by Caliskan et al. [9]. We then compare the attribution accuracy of the classifier on the test set with and without the application of the formatting tool `clang-format`, which normalizes the layout of the code.

Table 5: Study on plausibility of transformed source code.

Participant Group	Accuracy	Std
Undergraduate students	66.7%	23.6%
Graduate students	55.6%	15.7%
Professionals	60.0%	24.9%
Total	60.0%	21.8%
Random guessing	50.0%	—

While the attribution method can identify 27.5% of the programmers based on layout features if the code is not formatted, the performance decreases to 4.5% if we apply the formatting tool to the source code. We thus conclude that it is trivial to mislead an attribution based on layout features.

7 Limitations

Our previous experiments demonstrate the impact of our attack on program authorship attribution. Nonetheless, our approach has limitations which we discuss in the following.

Adversarial examples $\neq$ anonymization. Our attack enables a programmer to hide their identity in source code by misleading an attribution. While such an attack protects the privacy of the programmer, it is not sufficient for achieving anonymity. Note that k-anonymity would require a set of k developers that are equally likely to be attributed to the source code. In our setting, the code of the programmer is transformed to match a different author and an anonymity set of sufficient size is not guaranteed to exist. Still, we consider anonymization as promising direction for further research, which can build on the concepts of code transformations developed in this paper.

Verification of semantics. Finally, we consider two programs to be semantically equivalent if they return the same output for a given input. In particular, we verify that the transformed source code is semantically equivalent by applying the test cases provided by the GCJ competition. Although this approach is reasonable in our setting, it cannot guarantee strict semantic equivalence in all possible cases. Some of the exchanged API functions, for example, provide the same functionality but differ in corner cases, such as when the memory is exhausted. We acknowledge this limitation, yet it does not impact the general validity of our results.

8 Related Work

The automatic attack of source-code authorship attribution touches different areas of security research. In this section, we review related methods and concepts.

Authorship attribution of source code. Identifying the author of a program is a challenging task of computer security that has attracted a large body of work in the last years. Starting from early approaches experimenting with hand-crafted features [14, 16], the techniques for examining source code have constantly advanced, for example, by incorporating expressive features, such as n-grams [e.g., 1, 8, 13] and abstract syntax trees [e.g., 4, 9, 21]. Similarly, techniques for analyzing native code and identifying authors of compiled programs have advanced in the last years [e.g., 3, 10, 17, 22].

Two notable examples for source code are the approach by Caliskan et al. [9] and by Abuhamad et al. [1]. The former inspects features derived from code layout, lexical analysis and syntactic analysis. Regarding comprehensiveness, this work can be considered as the current state of the art. The work by Abuhamad et al. [1] focuses on lexical features as input for recurrent neural networks. Their work covers the largest set of authors so far and makes use of recent advances in deep learning. Table 6 shows the related approaches.

Method	Lay	Lex	Syn	Authors	Results
***Abuhamad et al. [1]**		•		**8903**	**92%**
***Caliskan et al. [9]**	•	•	•	**250**	**95%**
Alsulami et al. [4]			•	70	89%
Frantzeskou et al. [13]	•	•		30	97%
Krsul and Spafford [14]	•	•	•	29	73%
Burrows et al. [8]	•	•		10	77%

Table 6: Comparison of approaches for source code authorship attribution. Lay = Layout features, Lex = Lexical features, Syn = Syntactic features. *Attacked in this paper.

Previous work, however, has mostly ignored the problem of untargeted and targeted attacks. Only the empirical study by Simko et al. [25] examines how programmers can mislead the attribution by Caliskan et al. [9] by mimicking the style of other developers. While this study provides valuable insights into the risk of forgeries, it does not consider automatic attacks and thus is limited to manipulations by humans. In this paper, we demonstrate that such attacks can be fully automated. Our generated forgeries even provide a higher success rate than the handcrafted samples in the study. Moreover, we evaluate the impact of different feature sets and learning algorithms by evaluating two attribution methods.

Adversarial machine learning. The security of machine learning techniques has also attracted a lot of research recently. A significant fraction of work on attacks has focused on scenarios where the problem and feature space are mainly identical [see 6, 11, 18]. In these scenarios, changes in the problem space, such as the modification of an image pixel, have a one-to-one effect on the feature space, such that sophisticated attack strategies can be applied. By contrast, a one-to-one mapping between source code and the extracted features cannot be constructed and thus we are required to introduce a mixed attack strategy (see Section 3).

Creating evasive PDF malware samples [27, 31] and adversarial examples for text classifiers [e.g., 5, 15] represent two similar scenarios, where the practical feasibility needs to be ensured. These works typically operate in the problem space, where search algorithms such as hill climbing or genetic programming are guided by information from the feature space. MCTS represents a novel concept in the portfolio of creating adversarial examples under feasibility constraints, previously examined by Wicker et al. [30] in the image context only.

Also related is the approach by Sharif et al. [23] for misleading face recognition systems using painted eyeglasses. The proposed attack operates in the feature space but ensures practical feasibility by refining the optimization problem. In particular, the calculated adversarial perturbations are required to match the form of eyeglasses, to be printable, and to be invariant to slight head movements. In our attack scenario, such refinements of the optimization problem are not sufficient for obtaining valid source code, and thus we resort to applying code transformations in the problem space.

9 Conclusion

Authorship attribution of source code can be a powerful tool if an accurate and robust identification of programmers is possible. In this paper, however, we show that the current state of the art is insufficient for achieving a robust attribution. We present a black-box attack that seeks adversarial examples in the domain of source code by combining Monte-Carlo tree search with concepts from source-to-source compilation. Our empirical evaluation shows that automatic untargeted and targeted attacks are technically feasible and successfully mislead recent attribution methods.

Our findings indicate a need for alternative techniques for constructing attribution methods. These techniques should be designed with robustness in mind, such that it becomes harder to transfer stylistic patterns from one source code to another. A promising direction are generative approaches of machine learning, such as generative adversarial networks, that learn a decision function while actively searching for its weak spots. Similarly, it would help to systematically seek for stylistic patterns that are inherently hard to manipulate, either due to their complexity or due to their tight coupling with program semantics.

Public dataset and implementation. To encourage further research on program authorship attribution and, in particular, the development of robust methods, we make our dataset and implementation publicly available.[1] The attribution methods, the code transformers as well as our attack algorithm are all implemented as individual modules, such that they can be easily combined and extended.

[1] www.tu-braunschweig.de/sec/research/code/imitator

Acknowledgment

The authors would like to thank Johannes Heidtmann for his assistance during the project, and the anonymous reviewers for their suggestions and comments. Furthermore, the authors acknowledge funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972 and the research grant RI 2469/3-1.

References

[1] M. Abuhamad, T. AbuHmed, A. Mohaisen, and D. Nyang. Large-scale and language-oblivious code authorship identification. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, 2018.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley, 2006.

[3] S. Alrabaee, P. Shirani, L. Wang, M. Debbabi, and A. Hanna. On leveraging coding habits for effective binary authorship attribution. In *Proc. of European Symposium on Research in Computer Security (ESORICS)*, 2018.

[4] B. Alsulami, E. Dauber, R. E. Harang, S. Mancoridis, and R. Greenstadt. Source code authorship attribution using long short-term memory based networks. In *Proc. of European Symposium on Research in Computer Security (ESORICS)*, 2017.

[5] M. Alzantot, Y. Sharma, A. Elgohary, B.-J. Ho, M. Srivastava, and K.-W. Chang. Generating natural language adversarial examples. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2018.

[6] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli. Evasion attacks against machine learning at test time. In *Machine Learning and Knowledge Discovery in Databases*. Springer, 2013.

[7] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1), 2012.

[8] S. Burrows, A. L. Uitdenbogerd, and A. Turpin. Application of information retrieval techniques for source code authorship attribution. In *Proc. of Conference on Database Systems for Advanced Applications (DASFAA)*, 2009.

[9] A. Caliskan, R. Harang, A. Liu, A. Narayanan, C. R. Voss, F. Yamaguchi, and R. Greenstadt. De-anonymizing programmers via code stylometry. In *Proc. of USENIX Security Symposium*, 2015.

[10] A. Caliskan, F. Yamaguchi, E. Tauber, R. Harang, K. Rieck, R. Greenstadt, and A. Narayanan. When coding style survives compilation: De-anonymizing programmers from executable binaries. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2018.

[11] N. Carlini and D. A. Wagner. Towards evaluating the robustness of neural networks. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2017.

[12] E. Dauber, A. Caliskan, R. E. Harang, and R. Greenstadt. Git blame who?: Stylistic authorship attribution of small, incomplete source code fragments. Technical Report abs/1701.05681, arXiv, Computing Research Repository, 2017.

[13] G. Frantzeskou, E. Stamatatos, S. Gritzalis, and S. Katsikas. Effective identification of source code authors using byte-level information. In *Proc. of International Conference on Software Engineering (ICSE)*, 2006.

[14] I. Krsul and E. H. Spafford. Authorship analysis: identifying the author of a program. *Computers & Security*, 16(3), 1997.

[15] J. Li, S. Ji, T. Du, B. Li, and T. Wang. Textbugger: Generating adversarial text against real-world applications. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2019.

[16] S. MacDonell, A. Gray, G. MacLennan, and P. Sallis. Software forensics for discriminating between program authors using case-based reasoning, feed-forward neural networks and multiple discriminant analysis. In *Proc. of International Conference on Neural Information Processing (ICONIP)*, 1999.

[17] X. Meng, B. P. Miller, and K.-S. Jun. Identifying multiple authors in a binary program. In *Proc. of European Symposium on Research in Computer Security (ESORICS)*, 2017.

[18] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The limitations of deep learning in adversarial settings. In *Proc. of IEEE European Symposium on Security and Privacy (EuroS&P)*, 2016.

[19] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. Berkay Celik, and A. Swami. Practical black-box attacks against machine learning. In *Proc. of ACM Asia Conference on Computer Computer and Communications Security (ASIA CCS)*, 2017.

[20] N. Papernot, P. D. McDaniel, A. Sinha, and M. P. Wellman. Sok: Security and privacy in machine learning. In *Proc. of IEEE European Symposium on Security and Privacy (EuroS&P)*, 2018.

[21] B. N. Pellin. Using classification techniques to determine source code authorship. Technical report, Department of Computer Science, University of Wisconsin, 2000.

[22] N. E. Rosenblum, X. Zhu, and B. P. Miller. Who wrote this code? identifying the authors of program binaries. In *Proc. of European Symposium on Research in Computer Security (ESORICS)*, 2011.

[23] M. Sharif, S. Bhagavatula, L. Bauer, and M. K. Reiter. Accessorize to a Crime: real and stealthy attacks on state-of-the-art face recognition. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, 2016.

[24] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529, 2016.

[25] L. Simko, L. Zettlemoyer, and T. Kohno. Recognizing and imitating programmer style: Adversaries in program authorship attribution. *Proceedings on Privacy Enhancing Technologies*, 1, 2018.

[26] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Stealing machine learning models via prediction apis. In *Proc. of USENIX Security Symposium*, 2016.

[27] N. Šrndić and P. Laskov. Practical evasion of a learning-based classifier: A case study. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2014.

[28] Website. Clang: C language family frontend for LLVM. LLVM Project, `https://clang.llvm.org`, 2019. last visited May 2019.

[29] Website. Google Code Jam. `https://code.google.com/codejam/`, 2019. last visited May 2019.

[30] M. Wicker, X. Huang, and M. Kwiatkowska. Feature-guided black-box safety testing of deep neural networks. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2018.

[31] W. Xu, Y. Qi, and D. Evans. Automatically evading classifiers: A case study on pdf malware classifiers. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2016.

A Monte-Carlo Tree Search

In this section, we provide further details about our variant of *Monte-Carlo tree search*. Algorithm 1 gives an overview of the attack. The procedure ATTACK starts with the root node r_0 that represents the original source code x. The algorithm then works in two nested loops:

- The outer loop in lines 3–5 repetitively builds a search tree for the current state of source code r, and takes a single move (i.e. a single transformation). To do so, in each iteration, we choose the child node with the highest average classifier score. This process is repeated until the attack succeeds or a stop criterion is fulfilled (we reach a fixed number of outer iterations or we do not observe any improvement over multiple iterations) (line 3).
- The procedure MCTS represents the inner loop. It iteratively builds and extends the search tree under the current root node r. As this procedure is the main building block of our attack, we discuss the individual steps in more detail in the following.

Algorithm 1 Monte-Carlo Tree Search

```
1: procedure ATTACK(r_0)
2:     r ← r_0
3:     while not SUCCESS(r) and not STOPCRITERION(r) do
4:         MCTS(r)                          ▷ Extend the search tree under r
5:         r ← CHILDWITHBESTSCORE(r)        ▷ Perform next move
6: procedure MCTS(r)
7:     for i ← 1, N do
8:         u ← SELECTION(r, i)
9:         T ← SIMULATIONS(u)
10:        EXPANSION(u, T)
11:        BACKPROPAGATION(T)
```

Selection. Algorithm 2 shows the pseudocode to find the next node which is evaluated. The procedure recursively selects a child node according to a selection policy. We stop if the current node has no child nodes or if we have not marked it before in the current procedure SELECTION. The procedure finally returns the node that will be evaluated next.

As the number of possible paths grows exponentially (we have up to 36 transformations as choice at each node), we cannot evaluate all possible paths. The tree creation thus crucially depends on a selection policy. We use a simple heuristic to approximate the *Upper Confidence Bound for Trees* algorithm that is often used as selection policy (see [7]). Depending on the current iteration index i of SELECTION, the procedure SELECTIONPOLICY alternately returns the decision rule to choose the child with the highest average score, the lowest visit count or the highest score standard deviation. This step balances the *exploration* of less-visited nodes and the *exploitation* of promising nodes with a high average score.

Simulations. Equipped with the node u that needs to be evaluated, the next step generates a set of transformation

Algorithm 2 Selection Procedure of MCTS

```
1: procedure SELECTION(r, i)
2:     D ← SELECTIONPOLICY(i)
3:     u ← r
4:     while u has child nodes do
5:         v ← SELECTCHILD(u, D)                ▷ Child of u w.r.t. to D
6:         if v not marked as visited then
7:             Mark v as visited
8:             return v
9:         else
10:            u ← v
```

sequences $\mathcal{T}$ that start at u:

$$\mathcal{T} = \{\mathbf{T}_j \,|j = 1,\ldots,k \text{ and } |\mathbf{T}_j| \leq M\}\,, \tag{2}$$

where $|\mathbf{T}_j|$ is the number of transformations in $\mathbf{T}_j$. The sequences are created randomly and have a varying length which is, however, limited by M. In our experiments, we set $M = 5$ to reduce the number of possible branches.

In contrast to the classic game use-case, we can use the returned scores $g(x)$ as early feedback and thus we do not need to play out a full game. In other words, it is not necessary to evaluate the complete path to obtain feedback. For each sequence, we determine the classifier score by passing the modified source code at the end of each sequence to the attribution method. We further pinpoint a difference to the general MCTS algorithm. Instead of evaluating only one path, we create a batch of sequences that can be efficiently executed in parallel. In this way, we reduce the computation time and obtain the scores for various paths.

Expansion. We continue by inserting the respective transformations from the sequences as novel tree nodes under u (see Algorithm 3). For each sequence, we start with u and the first transformation. We check if a child node with the same transformation already exists under u. If not, a new node v is created and added as child under u. Otherwise, we use the already existing node v. We repeat this step with v and the next transformation. Figure 10 from Section 5 exemplifies this expansion step.

Algorithm 3 Expansion Procedure of MCTS

```
1: procedure EXPANSION(u, T)
2:     for T in T do                            ▷ For each sequence
3:         z ← u
4:         for T in T do                        ▷ For each transformer
5:             if z has no child with T then
6:                 v ← CREATENEWNODE(T)
7:                 z.add_child(v)
8:             else
9:                 v ← z.GETCHILDWITH(T)
10:            z ← v
```

Backpropagation. Algorithm 4 shows the last step that backpropagates the classifier scores to the root. For each sequence, the procedure first determines the last node n of the current sequence and the observed classifier score s at node n. Next, all nodes on the path from n to the root node of the search tree are updated. First, the visit count of each path node is incremented. Second, the final classifier score s is added to the score list of each path node. Both statistics are used by SELECTCHILD to choose the next promising node for evaluation. Furthermore, CHILDWITHBESTSCORE uses the score list to obtain the child node with the highest average score.

Algorithm 4 Backpropagation Procedure of MCTS

```
1: procedure BACKPROPAGATION(T)
2:     for T in T do
3:         s ← GETSCORE(T)
4:         get n as tree leaf of current sequence
5:         while n is not None do                ▷ Backpropagate to root
6:             n.visitCount ← n.visitCount + 1   ▷ Increase visit count
7:             n.scores = n.scores ∪ s           ▷ Append score
8:             n ← n.parent                      ▷ Will be None for root node
```

We finally note a slight variation for the scenario with a substitute model (see Section 3.1). To improve the transferability rate from the substitute to the original model, we do not terminate at the first successful adversarial example. Instead, we collect all successful samples and stop the outer loop after a predefined number of iterations. We choose the sample with the highest score on the substitute to be tested on the original classifier.

B List of Developers For Impersonation

Table 7 maps the letters to the 20 randomly selected programmers from the 2017 GCJ contest.

Table 7: List of developers for impersonation

Letter	Author	Letter	Author
A	4yn	K	chocimir
B	ACMonster	L	csegura
C	ALOHA.Brcps	M	eugenus
D	Alireza.bh	N	fragusbot
E	DAle	O	iPeter
F	ShayanH	P	jiian
G	SummerDAway	Q	liymouse
H	TungNP	R	sdya
I	aman.chandna	S	thatprogrammer
J	ccsnoopy	T	vudduu

C List of Code Transformations

A list of all 36 developed code transformations is presented in Table 8. The transformers are grouped accordingly to the family of their implemented transformations, i.e, transformations altering the control flow, transformations of declarations, transformations replacing the used API, template transformations, and miscellaneous transformations.

Table 8: List of Code Transformations

Control Transformations

Transformer	Description of Transformations
For statement transformer	Replaces a `for`-statement by an equivalent `while`-statement.
While statement transformer	Replaces a `while`-statement by an equivalent `for`-statement.
Function creator	Moves a whole block of code to a standalone function and creates a call to the new function at the respective position. The transformer identifies and passes all parameters required by the new function. It also adapts statements that change the control flow (e.g. the block contains a `return` statement that also needs to be back propagated over the caller).
Deepest block transformer	Moves the deepest block in the AST to a standalone function.
If statement transformer	Split the condition of a single `if`-statement at logical operands (e.g., `&&` or `||`) to create a cascade or a sequence of two `if`-statements depending on the logical operand.

Declaration Transformations

Transformer	Description of Transformation
Array transformer	Converts a static or dynamically allocated array into a C++ vector object.
String transformer	*Array option:* Converts a char array (C-style string) into a C++ string object. The transformer adapts all usages in the respective scope, for instance, it replaces all calls to `strlen` by calling the instance methods `size`. *String option:* Converts a C++ string object into a char array (C-style string). The transformer adapts all usages in the respective scope, for instance, it deletes all calls to `c_str()`.
Integral type transformer	Promotes integral types (`char`, `short`, `int`, `long`, `long long`) to the next higher type, e.g., `int` is replaced by `long`.
Floating-point type transformer	Converts `float` to `double` as next higher type.
Boolean transformer	*Bool option:* Converts true or false by an integer representation to exploit the implicit casting. *Int option:* Converts an integer type into a boolean type if the integer is used as boolean value only.
Typedef transformer	*Convert option:* Convert a type from source file to a new type via `typedef`, and adapt all locations where the new type can be used. *Delete option:* Deletes a type definition (`typedef`) and replace all usages by the original data type.
Include-Remove transformer	Removes includes from source file that are not needed.
Unused code transformer	*Function option:* Removes functions that are never called. *Variable option:* Removes global variables that are never used.
Init-Decl transformer	*Move into option:* Moves a declaration for a control statement if defined outside into the control statement. For instance, `int i; ...; for(i = 0; i < N; i++)` becomes `for(int i = 0; i < N; i++)`. *Move out option:* Moves the declaration of a control statement's initialization variable out of the control statement.

API Transformations

Transformer	Description of Transformations
Input interface transformer	*Stdin option:* Instead of reading the input from a file (e.g. by using the API `ifstream` or `freopen`), the input to the program is read from stdin directly (e.g. `cin` or `scanf`). *File option:* Instead of reading the input from stdin, the input is retrieved from a file.
Output interface transformer	*Stdout option:* Instead of printing the output to a file (e.g. by `ofstream` or `freopen`), the output is written directly to stdout (e.g. `cout` or `printf`). *File option:* Instead of writing the output directly to stdout, the output is written to a file.
Input API transformer	*C++-Style option:* Substitutes C APIs used for reading input (e.g., `scanf`) by C++ APIs (e.g., usage of `cin`). *C-Style option:* Substitutes C++ APIs used for reading input (e.g., usage of `cin`) by C APIs (e.g., `scanf`).
Output API transformer	*C++-Style option:* Substitutes C APIs used for writing output (e.g., `printf`) by C++ APIs (e.g., usage of `cout`). *C-Style option:* Substitutes C++ APIs used for writing output (e.g., usage `cout`) by C APIs (e.g., `printf`).
Sync-with-stdio transformer	Enable or remove the synchronization of C++ streams and C streams if possible.

Table 8: List of Code Transformations (continued)

Template Transformers

Transformer	Description of Transformations
Identifier transformer	Renames an identifier, i.e., the name of a variable or function. If no template is given, default values are extracted from the 2016 Code Jam Competition set that was used by Caliskan et al. [9] and that is not part of the training- and test set. We test default values such as `T`, `t`, ..., `i`.
Include transformer	Adds includes at the beginning of the source file. If no template is given, the most common includes from the 2016 Code Jam Competition are used as defaults.
Global declaration transformer	Adds global declarations to the source file. Defaults are extracted from the 2016 Code Jam Competition.
Include-typedef transformer	Inserts a type using `typedef`, and updates all locations where the new type can be used. Defaults are extracted from the 2016 Code Jam Competition.

Miscellaneous Transformers

Transformer	Description of Transformations
Compound statement transformer	*Insert option:* Adds a compound statement ({...}). The transformer adds a new compound statement to a control statement (`if`, `while`, etc.) given their body is not already wrapped in a compound statement. *Delete option:* Deletes a compound statement ({...}). The transformer deletes compound statements that have no effect, i.e., compound statements containing only a single statement.
Return statement transformer	Adds a return statement. The transformer adds a return statement to the main function to explicitly return 0 (meaning success). Note that main is a non-void function and is required to return an exit code. If the execution reaches the end of main without encountering a return statement, zero is returned implicitly.
Literal transformer	Substitutes a return statement returning an integer literal, by a statement that returns a variable. The new variable is declared by the transformer and initialized accordingly.

Terminal Brain Damage: Exposing the Graceless Degradation in Deep Neural Networks Under Hardware Fault Attacks

Sanghyun Hong, Pietro Frigo†, Yiğitcan Kaya, Cristiano Giuffrida†, Tudor Dumitraș

University of Maryland, College Park
†*Vrije Universiteit Amsterdam*

Abstract

Deep neural networks (DNNs) have been shown to tolerate "brain damage": cumulative changes to the network's parameters (e.g., pruning, numerical perturbations) typically result in a graceful degradation of classification accuracy. However, the limits of this natural resilience are not well understood in the presence of small adversarial changes to the DNN parameters' underlying memory representation, such as bit-flips that may be induced by hardware fault attacks. We study the effects of bitwise corruptions on 19 DNN models—six architectures on three image classification tasks—and we show that most models have at least one parameter that, after *a specific bit-flip in their bitwise representation*, causes an accuracy loss of over 90%. For large models, we employ simple heuristics to identify the parameters likely to be vulnerable and estimate that 40–50% of the parameters in a model might lead to an accuracy drop greater than 10% when individually subjected to such single-bit perturbations. To demonstrate how an adversary could take advantage of this vulnerability, we study the impact of an exemplary hardware fault attack, *Rowhammer*, on DNNs. Specifically, we show that a Rowhammer-enabled attacker co-located in the same physical machine can inflict significant accuracy drops (up to 99%) even with single bit-flip corruptions and no knowledge of the model. Our results expose the limits of DNNs' resilience against parameter perturbations induced by real-world fault attacks. We conclude by discussing possible mitigations and future research directions towards fault attack-resilient DNNs.

1 Introduction

Deep neural networks (DNNs) are known to be resilient to "brain damage" [32]: typically, cumulative changes to the network's parameters result in a graceful degradation of classification accuracy. This property has been harnessed in a broad range of techniques, such as network pruning [35], which significantly reduces the number of parameters in the network and leads to improved inference times. Besides structural resilience, DNN models can tolerate slight noise in their parameters with minimal accuracy degradation [2]. Researchers have proposed utilizing this property in defensive techniques, such as adding Gaussian noise to model parameters to strengthen DNN models against adversarial examples [69]. As a result, this natural resilience is believed to make it difficult for attackers to significantly degrade the overall accuracy by corrupting network parameters.

Recent work has explored the impact of *hardware faults* on DNN models [34,42,45]. Such faults can corrupt the memory storing the victim model's parameters, stress-testing DNNs' resilience to bitwise corruptions. For example, Qin et al. [42], confirming speculation from previous studies [34,35], showed that a DNN model for CIFAR10 image classification does not lose more than 5% accuracy when as many as 2,600 parameters out of 2.5 million are corrupted by *random errors*. However, this analysis is limited to a specific scenario and only considers accidental errors rather than attacker-induced corruptions by means of fault attacks. The widespread usage of DNNs in many mission-critical systems, such as self-driving cars or aviation [12, 51], requires a comprehensive understanding of the security implications of such adversarial bitwise errors.

In this paper, we explore the security properties of DNNs under bitwise errors that can be induced by practical hardware fault attacks. Specifically, we ask the question: *How vulnerable are DNNs to the atomic corruption that a hardware fault attacker can induce?* This paper focuses on single bit-flip attacks that are realistic as they well-approximate the constrained memory corruption primitive of practical hardware fault attacks such as Rowhammer [48]. To answer this question, we conduct a comprehensive study that characterizes the DNN model's responses to single-bit corruptions in each of its parameters.

First, we implement a systematic vulnerability analysis framework that flips each bit in a given model's parameters and measures the misclassification rates on a validation set. Using our framework, we analyze 19 DNN models composed of six different architectures and their variants on three pop-

ular image classification tasks: MNIST, CIFAR10, and ImageNet. Our experiments show that, on average, ∼50% of model parameters are vulnerable to single bit-flip corruptions, causing relative accuracy drops above 10%, and that all 19 DNN models include parameters that can cause an accuracy drop of over 90%[1]. The results expose the limits of the DNN's resilience to numerical changes, as adversarial bitwise errors can lead to a *graceless degradation* of classification accuracy.

Our framework also allows us to characterize the vulnerability by examining the impact of various factors: the bit position, bit-flip direction, parameter sign, layer width, activation function, normalization, and model architecture. Our key findings include: 1) the vulnerability is caused by drastic spikes in a parameter value; 2) the spikes in positive parameters are more threatening, however, an activation function that allows negative outputs renders the negative parameters vulnerable as well; 3) the number of vulnerable parameters increases proportionally as the DNN's layers get wider; 4) two common training techniques, e.g., dropout [52] and batch normalization [24], are ineffective in preventing the massive spikes bit-flips cause; and 5) the ratio of vulnerable parameters is almost constant across different architectures (e.g., AlexNet, VGG16, and so on). Further, building on these findings, we propose heuristics for speeding up the analysis of vulnerable parameters in large models.

Second, to understand the practical impact of this vulnerability, we use Rowhammer [26] as an exemplary hardware fault attack. While a variety of hardware fault attacks are documented in literature [11, 26, 38, 57], Rowhammer is particularly amenable to practical, real-world exploitation. Rowhammer takes advantage of a widespread vulnerability in modern DRAM modules and provides an attacker with the ability to trigger controlled memory corruptions directly from unprivileged software execution. As a result, even a constrained Rowhammer-enabled attacker, who only needs to perform a specific memory access pattern, can mount practical attacks in a variety of real-world environments, including cloud [44, 67], browsers [9, 15, 19, 48], mobile [15, 62], and servers [36, 60].

We analyze the feasibility of Rowhammer attacks on DNNs by simulating a Machine-Learning-as-a-Service (MLaaS) scenario, where the victim and attacker VMs are co-located on the same host machine in the cloud. The co-location leads the victim and the attacker to share the same physical memory, enabling the attacker to trigger Rowhammer bit-flips in the victim's data [44, 67]. We focus our analysis to models with an applicable memory footprint, which can realistically be targeted by hardware fault attacks such as Rowhammer.

Our Rowhammer results show that in a *surgical* attack scenario, with the capability of flipping specific bits, the attacker can reliably cause severe accuracy drops in practical settings. Further, even in a *blind* attack scenario, the attacker can still mount successful attacks without any control over the locations of bit-flips landed in memory. Moreover, we also reveal a potential vulnerability in the *transfer learning* scenario; in which a surgical attack targets the parameters in the layers victim model contains in common with a public one.

Lastly, we discuss directions for viable protection mechanisms, such as reducing the number of vulnerable parameters by preventing significant changes in a parameter value. In particular, this can be done by 1) restricting activation magnitudes and 2) using low-precision numbers for model parameters via quantization or binarization. We show that, when we restrict the activations using the ReLU-6 activation function, the ratio of vulnerable parameters decreases from 47% to 3% in AlexNet, and also, the accuracy drops are largely contained within 10%. Moreover, quantization and binarization reduce the vulnerable parameter ratio from 50% to 1-2% in MNIST. While promising, such solutions cannot deter practical hardware fault attacks in the general case, and often require training the victim model from scratch; hinting that more research is required towards fault attack-resilient DNNs.

Contributions. We make three contributions:

- We show DNN models are more vulnerable to bit-flip corruptions than previously assumed. In particular, we show adversarial bitwise corruptions induced by hardware fault attacks can easily inflict severe *indiscriminate damages* by drastically increasing or decreasing the value of a model parameter.
- We conduct the first comprehensive analysis of DNN models' behavior against single bit-flips and characterize the vulnerability that a hardware fault attack can trigger.
- Based on our analysis, we study the impact of practical hardware fault attacks in a representative DL scenario. Our analysis shows that a Rowhammer-enabled attacker can inflict significant accuracy drops (up to 99%) on a victim model even with constrained bit-flip corruptions and no knowledge of the model.

2 Preliminaries

Here, we provide an overview of the required background knowledge.

Deep neural networks. A DNN can be conceptualized as a function that takes an input and returns a prediction, i.e., the inferred *label* of the input instance. The network is composed of a sequence of layers that is individually parameterized by a set of matrices, or *weights*. Our work focuses on feed-forward DNNs—specifically on convolutional neural networks (CNNs)—in the supervised learning setting, i.e., the weights that minimize the inference error are learned from

[1] The vulnerability of a parameter requires a *specific* bit in its bitwise representation to be flipped. There also might be multiple such bits in the representation that, when flipped separately, trigger the vulnerability.

a labeled *training set*. In a feed-forward network, each layer applies a linear transformation, defined by its weight matrix, to its input—the output of the previous layer—and a bias parameter is added optionally. After the linear transformation, a non-linear *activation function* is applied; as well as other optional layer structures, such as dropout, pooling or batch normalization. During training, the DNN's *parameters*, i.e., the weights in each layer and in other optional structures, are updated iteratively by *backpropagating* the error on the training data. Once the network converges to an acceptable error rate or when it goes through sufficient iterations, training stops and the network, along with all its parameters, is stored as a trained network. During testing (or inference), we load the full model into the memory and produce the prediction for a given input instance, usually not in the training data.

Single precision floating point numbers. The parameters of a DNN model are usually represented as IEEE754 32-bit single-precision floating-point numbers. This format leverages the exponential notation and trades off the large range of possible values for reduced precision. For instance, the number 0.15625 in exponential notation is represented as 1.25×2^{-3}. Here, 1.25 expresses the *mantissa*; whereas -3 is the *exponent*. The IEEE754 single-precision floating-point format defines 23 bits to store the mantissa, 8 bits for the exponent, and one bit for the sign of the value. The fact that different bits have different influence on the represented value makes this format interesting from an adversarial perspective. For instance, continuing or example, flipping the 16th bit in the mantissa increases the value from 0.15625 to 0.15625828; hence, a usually negligible perturbation. On the other hand, a flipping the highest exponent bit would turn the value into 1.25×2^{125}. Although both of these rely on the same bit corruption primitive, they yield vastly different results. In Sec 4, we analyze how this might lead to a vulnerability when a DNN's parameters are corrupted via single bit-flips.

Rowhammer attacks. Rowhammer is the most common instance of software-induced fault attacks [9, 15, 19, 44, 48, 60, 62, 67]. This vulnerability provides an aggressor with a single-bit corruption primitive at DRAM level; thus, it is an ideal attack for the purpose of our analysis. Rowhammer is a remarkably versatile fault attack since it only requires an attacker to be able to access content in DRAM; an ubiquitous feature of every modern system. By simply carrying out specific memory access patterns—which we explain in Sec 5—the attacker is able to cause extreme stress on other memory locations triggering faults on other stored data.

3 Threat Model

Prior research has extensively validated a DNN's resilience to parameter changes [2, 32, 34, 35, 42, 69], by considering random or deliberate perturbations. However, from a security perspective, these results provide only limited insights as they study a network's expected performance under cumulative changes. In contrast, towards a successful and feasible attack, an adversary is usually interested in inflicting the worst-case damage under minimal changes.

We consider a class of modifications that an adversary, using hardware fault attacks, can induce in practice. We assume a cloud environment where the victim's deep learning system is deployed inside a VM—or a container—to serve the requests of external users. For making test-time inferences, the trained DNN model and its parameters are loaded into the system's (shared) memory and remain constant in normal operation. Recent studies describe this as a typical scenario in MLaaS [61].

To understand the DNNs' vulnerability in this setting, we consider the atomic change that an adversary may induce—the single bit-flip—and we, in Sec 4, systematically characterize the damage such change may cause. We then, in Sec 5, investigate the feasibility of inducing this damage in practice, by considering adversaries with different capabilities and levels of knowledge.

Capabilities. We consider an attacker *co-located* in the same physical host machine as the victim's deep learning system. The attacker, due to co-location, can take advantage of a well-known software-induced fault attack, Rowhammer [44,67], for corrupting the victim model stored in DRAM. We take into account two possible scenarios: 1) a *surgical* attack scenario where the attacker can cause a bit-flip at an intended location in the victim's process memory by leveraging advanced memory massaging primitives [44, 62] to obtain more precise results; and 2) a *blind* attack where the attacker lacks fine-grained control over the bit-flips; thus, is completely unaware of where a bit-flip lands in the layout of the model.

Knowledge. Using the existing terminology, we consider two levels for the attacker's knowledge of the victim model, e.g., the model's architecture and its parameters as well as their placement in memory: 1) a *black-box* setting where the attacker has no knowledge of the victim model. Here, both the surgical and blind attackers only hope to trigger an accuracy drop as they cannot anticipate what the impact of their bit-flips would be; and 2) a *white-box* setting where the attacker knows the victim model, at least partially. Here, the surgical attacker can deliberately tune the attack's inflicted accuracy drop—from minor to catastrophic damage. Optionally, the attacker can force the victim model to misclassify a specific input sample without significantly damaging the overall accuracy. However, the blind attacker gains no significant advantage over the black-box scenario as the lack of capability prevents the attacker from acting on the knowledge.

4 Single-Bit Corruptions on DNNs

In this section, we expose DNNs' vulnerability to single bit-flips. We start with an overview of our experimental setup and methodology. We then present our findings of DNNs' vulnerability to single bit corruptions. For characterizing the vulnerability, we analyze the impact of 1) the bitwise representation of the corrupted parameter, and 2) various DNN properties; on the resulting *indiscriminate damage*[2]. We also discuss the broader implications of the vulnerability for both the blind and surgical attackers. Finally, we turn our attention to two distinct attack scenarios single bit-flips lead to.

4.1 Experimental Setup and Methodology

Our vulnerability analysis framework systematically flips the bits in a model, individually, and quantifies the impact using the metrics we define. We implement the framework using Python 3.7[3] and PyTorch 1.0[4] that supports CUDA 9.0 for accelerating computations by using GPUs. Our experiments run on the high performance computing cluster that has 488 nodes, where each is equipped with Intel E5-2680v2 2.8GHz 20-core processors, 180 GB of RAM, and 40 of which have 2 Nvidia Tesla K20m GPUs. We achieve a significant amount of speed-up by leveraging a parameter-level parallelism.

Datasets. We use three popular image classification datasets: MNIST [31], CIFAR10 [29], and ImageNet [47]. MNIST is a grayscale image dataset used for handwritten digits (zero to nine) recognition, containing 60,000 training and 10,000 validation images of 28x28 pixels. CIFAR10 and ImageNet are colored image datasets used for object recognition. CIFAR10 includes 32x32 pixels, colored natural images of 10 classes, containing 50,000 training and 10,000 validation images. For ImageNet, we use the ILSVRC-2012 subset [46], resized at 224x224 pixels, composed of 1,281,167 training and 50,000 validation images from 1,000 classes.

Models. We conduct our analysis on 19 different DNN models. For MNIST, we define a baseline architecture, Base (B), and generate four variants with different layer configurations: B-Wide, B-PReLU, B-Dropout, and B-DP-Norm. We also examine well-known LeNet5 (L5) [31] and test two variants of it: L5-Dropout and L5-D-Norm. For CIFAR10, we employ the architecture from [55] as a baseline and experiment on its three variants: B-Slim, B-Dropout and B-D-Norm. In the following sections, we discuss why we generate these variants. In Appendix A, we describe the details of these custom architectures; in Appendix C, we present the hyper-parameters. For CIFAR10, we also employ two off-the-shelf network architectures: AlexNet [30] and VGG16 [50]. For ImageNet, we use five well-known DNNs to understand the vulnerability of large models: AlexNet, VGG16, ResNet50 [22], DenseNet161 [23] and InceptionV3 [56][5].

Metrics. To quantify the indiscriminate damage of single bit-flips, we define the Relative Accuracy Drop as $\text{RAD} = {}^{(\text{Acc}_{pristine} - \text{Acc}_{corrupted})}/_{\text{Acc}_{pristine}}$; where $\text{Acc}_{pristine}$ and $\text{Acc}_{corrupted}$ denote the classification accuracies of the pristine and the corrupted models, respectively. In our experiments, we use [RAD > 0.1] as the criterion for indiscriminate damage on the model. We also measure the accuracy of each class in the validation set to analyze whether a single bit-flip causes a subset of classes to dominate the rest. In MNIST and CIFAR10, we simply compute the Top-1 accuracy on the test data (as a percentage) and use the accuracy for analysis. For ImageNet, we consider both the Top-1 and Top-5 accuracy; however, for the sake of comparability, we report only Top-1 accuracy in Table 1. We consider a parameter as *vulnerable* if it, in its bitwise representation, contains at least one bit that triggers severe indiscriminate damage when flipped. For quantifying the vulnerability of a model, we simply count the number of these vulnerable parameters.

Methodology. On our 8 MNIST models, we carry out a complete analysis: we flip each bit in all parameters of a model, in both directions—(0→1) and (1→0)—and compute the RAD over the entire validation. However, a complete analysis of the larger models requires infeasible computational time—the VGG16 model for ImageNet with 138M parameters would take ≈ 942 days on our setup. Therefore, based on our initial results, we devise three speed-up heuristics that aid the analysis of CIFAR10 and ImageNet models.

Speed-up techniques. The following three heuristics allow us to feasibly and accurately estimate the vulnerability in larger models:

• *Sampled validation set (SV).* After a bit-flip, deciding whether the bit-flip leads to a vulnerability [RAD > 0.1] requires testing the corrupted model on the validation set; which might be cost prohibitive. This heuristic says that we can still estimate the model accuracy—and the RAD—on a sizable subset of the validation set. Thus, we randomly sample 10% the instances from each class in the respective validation sets, in both CIFAR10 and ImageNet experiments.

• *Inspect only specific bits (SB).* In Sec 2, we showed how flipping different bits of a IEEE754 floating-point number results in vastly different outcomes. Our the initial MNIST analysis in Sec 4.3 shows that mainly the exponent bits lead to perturbations strong enough to cause indiscriminate damage.

[2] We use this term to indicate the severe overall accuracy drop in the model.
[3] https://www.python.org
[4] https://pytorch.org
[5] The pre-trained ImageNet models we use are available at: https://pytorch.org/docs/stable/torchvision/models.html.

Dataset	Network	Base acc.	# Params	Speed-up heuristics			Vulnerablility	
				SV	SB	SP	# Params	Ratio
MNIST	B(ase)	95.71	21,840	✗	✗	✗	10,972	50.24%
	B-Wide	98.46	85,670	✗	✗	✗	42,812	49.97%
	B-PReLU	98.13	21,843	✗	✗	✗	21,663	99.18%
	B-Dropout	96.86	21,840	✗	✗	✗	10,770	49.35%
	B-DP-Norm	97.97	21,962	✗	✗	✗	11,195	50.97%
	L5	98.81	61,706	✗	✗	✗	28,879	46.80%
	L5-Dropout	98.72	61,706	✗	✗	✗	27,806	45.06%
	L5-D-Norm	99.05	62,598	✗	✗	✗	30,686	49.02%
CIFAR10	B(ase)	83.74	776,394	✓(83.74)	✓(exp.)	✗	363,630	46.84%
	B-Slim	82.19	197,726	✓(82.60)	✓(exp.)	✗	92,058	46.68%
	B-Dropout	81.18	776,394	✓(80.70)	✓(exp.)	✗	314,745	40.54%
	B-D-Norm	80.17	777,806	✓(80.17)	✓(exp.)	✗	357,448	45.96%
	AlexNet	83.96	2,506,570	✓(85.00)	✓(exp.)	✗	1,185,957	47.31%
	VGG16	91.34	14,736,727	✓(91.34)	✓(exp.)	✗	6,812,359	46.23%
ImageNet	AlexNet	56.52 / 79.07	61,100,840	✓(51.12 / 75.66)	✓(31st bit)	✓(20,000)	9,467 SP	47.34%
	VGG16	79.52 / 90.38	138,357,544	✓(64.28 / 86.56)	✓(31st bit)	✓(20,000)	8,414 SP	42.07%
	ResNet50	76.13 / 92.86	25,610,152	✓(69.76 / 89.86)	✓(31st bit)	✓(20,000)	9,565 SP	47.82%
	DenseNet161	77.13 / 93.56	28,900,936	✓(72.48 / 90.94)	✓(31st bit)	✓(20,000)	9,790 SP	48.95%
	InceptionV3	69.54 / 88.65	27,197,488	✓(65.74 / 86.24)	✓(31st bit)	✓(20,000)	8,161 SP	40.84%

SV = Sampled Validation set SB = Specific Bits SP = Sampled Parameters set

Table 1: **Indiscriminate damages to 19 DNN models caused by single bit-flips.**

This observation is the basis of our *SB* heuristic that tells us to examine the effects of flipping only the exponent bits for CIFAR10 models. For ImageNet models, we use a stronger SB heuristic and only inspect the most significant exponent bit of a parameter to achieve a greater speed-up. This heuristic causes us to miss the vulnerability the remaining bits might lead to, therefore, its results can be interpreted as a conservative estimate of the actual number of vulnerable parameters.

• *Sampled parameters (SP) set.* Our MNIST analysis also reveals that almost 50% of all parameters are vulnerable to bit-flips. This leads to our third heuristic: uniformly sampling from the parameters of a model would still yield an accurate estimation of the vulnerability. We utilize the SP heuristic for ImageNet models and uniformly sample a fixed number of parameters—20,000—from all parameters in a model. In our experiments, we perform this sampling five times and report the average vulnerability across all runs. Uniform sampling also reflects the fact that a black-box attacker has a uniform probability of corrupting any parameter.

4.2 Quantifying the Vulnerability That Leads to Indiscriminate Damage

Table 1 presents the results of our experiments on single-bit corruptions, for 19 different DNN models. We reveal that an attacker, armed with a single bit-flip attack primitive, can successfully cause indiscriminate damage [RAD > 0.1] and that the ratio of vulnerable parameters in a model varies between 40% to 99%; depending on the model. The consistency between MNIST experiments, in which we examine every possible bit-flip, and the rest, in which we heuristically examine only a subset, shows that, in a DNN model, approximately half of the parameters are vulnerable to single bit-flips. Our experiments also show small variability in the chances of a successful attack—indicated by the ratio of vulnerable parameters. With 40% vulnerable parameters, the InceptionV3 model is the most apparent outlier among the other ImageNet models; compared to 42-49% for the rest. We define the vulnerability based on [RAD > 0.1] and, in Appendix B, we also give how vulnerability changes within the range [0.1 $\leq$ RAD $\leq$ 1]. In the following subsections, we characterize the vulnerability in relation to various factors and discuss our results in more detail.

4.3 Characterizing the Vulnerability: Bitwise Representation

Here, we characterize the interaction how the features of a parameter's bitwise representation govern its vulnerability.

Impact of the bit-flip position. To examine how much change in a parameter's value leads to indiscriminate damage, we focus on the position of the corrupted bits. In Figure 1, for each bit position, we present the number of bits—in the log-scale—that cause indiscriminate damage when flipped, on MNIST-L5 and CIFAR10-AlexNet models. In our MNIST

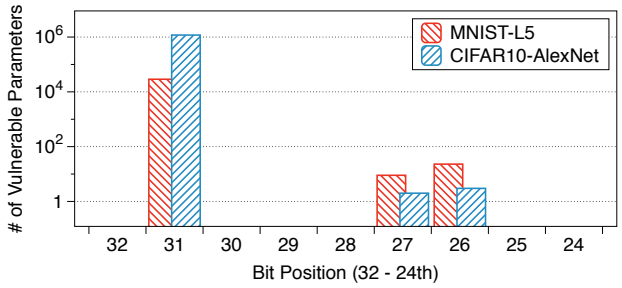


Figure 1: **The impact of the bit position.** The number of vulnerable parameters in bit positions 32nd to 24th.

experiments, we examine all bit positions and we observe that bit positions other than the exponents mostly do not lead to significant damage; therefore, we only consider the exponent bits. We find that *the exponent bits, especially the 31st-bit, lead to indiscriminate damage*. The reason is that a bit-flip in the exponents causes to a drastic change of a parameter value, whereas a flip in the mantissa only increases or decreases the value by a small amount—$[0,1]$. We also observe that flipping the 30th to 28th bits is mostly inconsequential as these bits, in the IEEE754 representation, are already set to one for most values a DNN parameter usually takes—$[3.0517 \times 10^{-5}, 2]$.

Impact of the flip direction. We answer which direction of the bit-flip, (0→1) or (1→0), leads to greater indiscriminate damage. In Table 2, we report the number of effective bit-flips, i.e., those that inflict [RAD > 0.1] for each direction, on 3 MNIST and 2 CIFAR10 models. We observe that *only (0→1) flips cause indiscriminate damage* and *no (1→0) flip leads to vulnerability*. The reason is that a (1→0) flip can only decrease a parameter's value, unlike a (0→1) flip. The values of model parameters are usually normally distributed—$N(0,1)$—that places most of the values within [-1,1] range. Therefore, a (1→0) flip, in the exponents, can decrease the magnitude of a typical parameter at most by one; which is not a strong enough change to inflict critical damage. Similarly, in the sign bit, both (0→1) and (1→0) flips cannot cause severe damage because they change the magnitude of a parameter at most by two. On the other hand, a (0→1) flip, in the exponents, can increase the parameter value significantly; thus, during the forward-pass, the extreme neuron activation caused by the corrupted parameter overrides the rest of the activations.

Direction	**Models (M: MNIST, C: CIFAR10)**				
(32-24th bits)	**M-B**	**M-PReLU**	**M-L5**	**C-B**	**C-AlexNet**
0→1	11,019	21,711	28,902	314,768	1,185,964
1→0	0	0	0	0	0
Total	11,019	21,711	28,902	314,768	1,185,964

Table 2: **The impact of the flip direction.** The number of effective bit-flips in 3 MNIST and 2 CIFAR10 models.

Impact of the parameter sign. As our third feature, we investigate whether the sign—positive or negative—of the corrupted parameter impacts the vulnerability. In Figure 2, we examine the MNIST-L5 model and present the number of vulnerable positive and negative parameters in each layer—in the log-scale. Our results suggest that *positive parameters are more vulnerable to single bit-flips than negative parameters*. We identify the common ReLU activation function as the reason: ReLU immediately zeroes out the negative activation values, which are usually caused by the negative parameters. As a result, the detrimental effects of corrupting a negative parameter fail to propagate further in the model. Moreover, we observe that *in the first and last layers, the negative parameters, as well as the positive ones, are vulnerable*. We hypothesize that, in the first convolutional layer, changes in the parameters yield a similar effect to corrupting the model inputs directly. On the other hand, in their last layers, DNNs usually have the Softmax function that does not have the same zeroing-out effect as ReLU.

4.4 Characterizing the Vulnerability: DNN Properties

We continue our analysis by investigating how various properties of a DNN model affect the model's vulnerability to single bit-flips.

Impact of the layer width. We start our analysis by asking whether increasing the width of a DNN affects the number of vulnerable parameters. In Table 1, in terms of the number of vulnerable parameters, we compare the MNIST-B model with the MNIST-B-Wide model. In the wide model, all the convolutional and fully-connected layers are twice as wide as the corresponding layer in the base model. We see that the ratio of vulnerable parameters is almost the same for both models: 50.2% vs 50.0%. Further, experiments on the CIFAR10-B-Slim and CIFAR10-B—twice as wide as the slim model—produce consistent results: 46.7% and 46.8%. We conclude that *the number of vulnerable parameters grows proportionally with the DNN's width* and, as a result, *the ratio of vulnerable parameters remains constant at around 50%*.

Impact of the activation function. Next, we explore whether the choice of activation function affects the vulnerability. Previously, we showed that ReLU can neutralize the effects of large negative parameters caused by a bit-flip; thus, we experiment on different activation functions that allow negative outputs, e.g., PReLU [21], LeakyReLU, or RReLU [68]. These ReLU variants have been shown to improve the training performance and the accuracy of a DNN. In this experiment, we train the MNIST-B-PReLU model; which is exactly the same as the MNIST-B model, except that it replaces ReLU with PReLU. Figure 3 presents the layer-wise number of

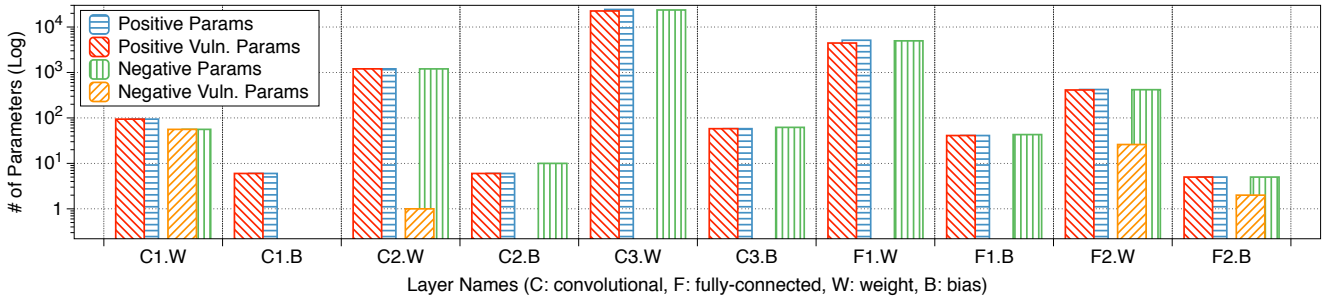


Figure 2: **The impact of the parameter sign.** The number of vulnerable positive and negative parameters, in each layer of MNIST-L5.

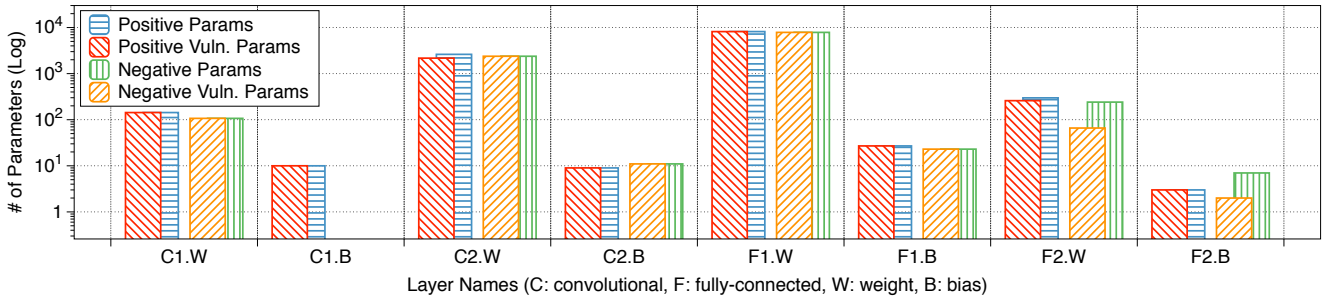


Figure 3: **The impact of the activation function.** The number of vulnerable positive and negative parameters, in each layer of MNIST-PReLU.

vulnerable positive and negative parameters in MNIST-B-PReLU. We observe that *using PReLU causes the negative parameters to become vulnerable and, as a result, leads to a DNN approximately twice as vulnerable as the one that uses ReLU*—50.2% vs. 99.2% vulnerable parameters.

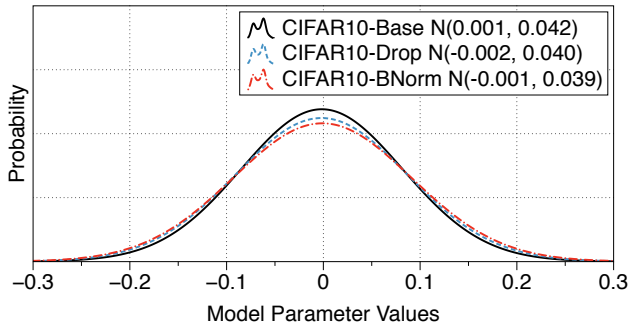


Figure 4: **The impact of the dropout and batch normalization.** The distributions of the parameter values of three CIFAR10 models variants.

Impact of dropout and batch normalization. We confirmed that successful bit-flip attacks increase a parameter's value drastically to cause indiscriminate damage. In consequence, we hypothesize that common techniques that tend to constrain the model parameter values to improve the performance, e.g., dropout [52] or batch normalization [24], would result in a model more resilient to single bit-flips. Besides the base CIFAR10 and MNIST models, we train the B-Dropout and B-DNorm models for comparison. In B-Dropout models, we apply dropout before and after the first fully-connected layers; in B-DNorm models, in addition to dropout, we also apply batch normalization after each convolutional layer. In Figure 4, we compare our three CIFAR10 models and show how dropout and batch normalization have the effect of reducing the parameter values. However, when we look into the vulnerability of these models, we surprisingly find that *the vulnerability is mostly persistent regardless of dropout or batch normalization*—with at most 6.3% reduction in vulnerable parameter ratio over the base network.

Impact of the model architecture. Table 1 shows that the vulnerable parameter ratio is mostly consistent across different DNN architectures. However, we see that the InceptionV3 model for ImageNet has a relatively lower ratio—40.8%—compared to the other models—between 42.1% and 48.9%. We hypothesize that the reason is the auxiliary classifiers in the InceptionV3 architecture that have no function at test-time. To confirm our hypothesis, we simply remove the parameters in the auxiliary classifiers; which bring the vulnerability ratio closer to the other models—46.5%. Interestingly, we also observe that the parameters in batch normalization layers are resilient to a bit-flip: corrupting `running_mean` and `running_var` cause negligible damage. In consequence, ex-

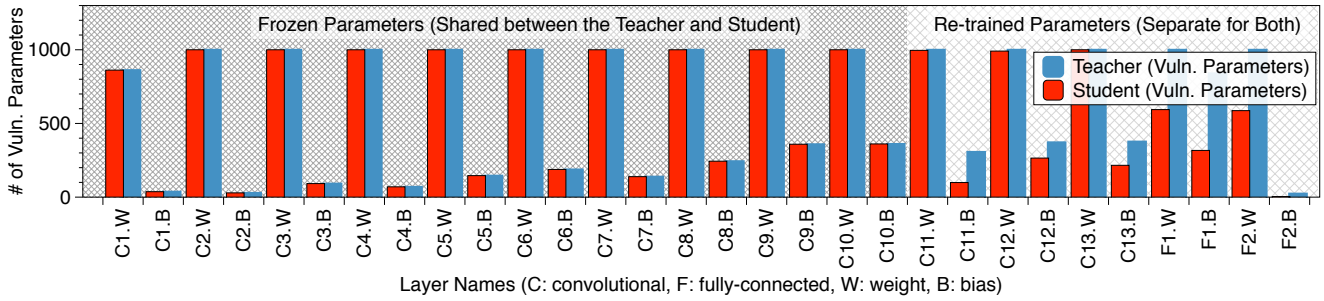


Figure 5: **The security threat in a transfer learning scenario.** The victim model—student—that is trained by transfer learning is vulnerable to the surgical attacker, who can see the parameters the victim has in common with the teacher model.

cluding the parameters in InceptionV3's multiple batch normalization layers leads to a slight increase in vulnerability—by 0.02%.

4.5 Implications for the Adversaries

In Sec 3, we defined four attack scenarios: the blind and surgical attackers, in the black-box and white-box settings. First, we consider the strongest attacker: the surgical, who can flip a bit at a specific memory location, white-box, with the model knowledge for anticipating the impact of flipping the said bit. To carry out the attack, this attacker identifies: 1) how much indiscriminate damage, the RAD goal, she intends to inflict, 2) a vulnerable parameter that can lead to the RAD goal, 3) in this parameter, the bit location, e.g., 31st-bit, and the flip direction, e.g., (0→1), for inflicting the damage. Based on our [RAD > 0.1] criterion, approximately 50% of the parameters are vulnerable in all models; thus, for this goal, the attacker can easily achieve 100% success rate. For more severe goals [$0.1 \leq \text{RAD} \leq 0.9$], our results in Appendix B suggest that the attacker can still find vulnerable parameters. In Sec 5.1, we discuss the necessary primitives, in a practical setting, for this attacker.

For a black-box surgical attacker, on the other hand, the best course of action is to target the 31st-bit of a parameter. This strategy maximizes the attacker's chance of causing indiscriminate damage, even without knowing what, or where, the corrupted parameter is. Considering, the VGG16 model for ImageNet, the attack's success rate is 42.1% as we report in Table 1; which is an upper-bound for the black-box attackers. For the weakest—black-box blind—attacker that cannot specifically target the 31st-bit, we conservatively estimate the lower-bound as 42.1% / 32-bits = 1.32%; assuming only the 31st-bits lead to indiscriminate damage. Note that the success rate for the white-box blind attacker is still 1.32% as acting upon the knowledge of the vulnerable parameters requires an attacker to target specific parameters. In Sec 5.2, we evaluate the practical success rate of a blind attacker.

4.6 Distinct Attack Scenarios

In this section, other than causing indiscriminate damage, we discuss two distinct attack scenarios single bit-flips might enable: transfer learning and targeted misclassification.

Transfer learning scenario. Transfer learning is a common technique for *transferring* the knowledge in a pre-trained *teacher* model to a *student* model; which, in many cases, outperforms training a model from scratch. In a practical scenario, a service provider might rely on publicly available teacher as a starting point to train commercial student models. The teacher's knowledge is transferred by *freezing* some of its layers and embedding them into the student model; which, then, trains the remaining layers for its own task. The security risk is that, for an attacker who knows the teacher but not the student, a black-box attack on the student might escalate into a white-box attack on the teacher's frozen layers. The attacker first downloads the pre-trained teacher from the Internet. She then loads the teacher into the memory and waits for the *deduplication* [66] to happen. During deduplication, the memory pages with the same contents—the frozen layers—are merged into the shared pages between the victim and attacker. In consequence, a bit-flip in the attacker's pages can also affect the student model in the victim's memory. We hypothesize that a surgical attacker, who can identify the teacher's vulnerable parameters and trigger bit-flips in these parameters, can cause indiscriminate damage to the student model. In our experiments, we examine two transfer learning tasks in [63]: the traffic sign (GTSRB) [53] and flower recognition (Flower102) [41]. We initialize the student model by transferring first ten frozen layers of the teacher—VGG16 or ResNet50 on ImageNet. We then append a new classification layer and train the resulting student network for its respective task by only updating the new unfrozen layer. We corrupt the 1,000 parameters sampled from each layer in the teacher and monitor the damage to the student model. Figure 5 reports our results: we find that *all vulnerable parameters in the frozen layers and more than a half in the re-trained layers are shared by the teacher and the student*.

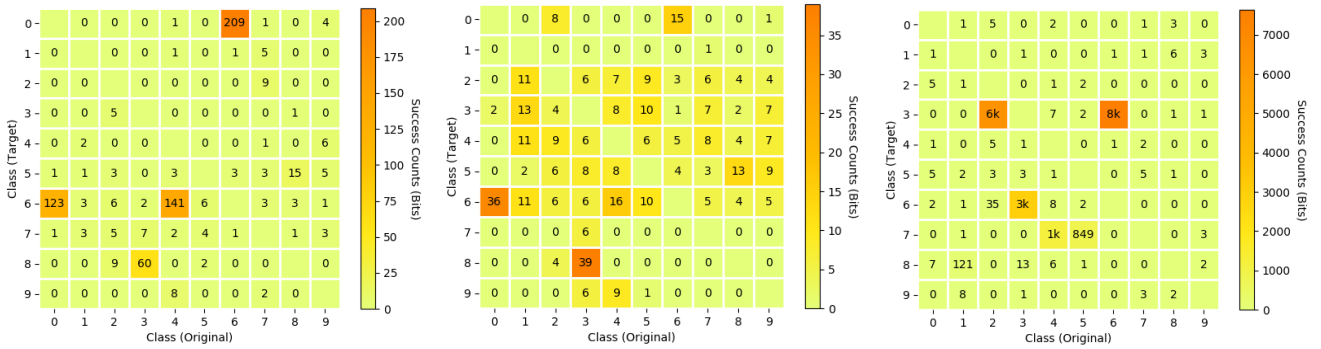


Figure 6: **The vulnerable parameters for a targeted attack in 3 DNN models.** Each cell reports the number of bits that lead to the misclassification of a target sample, whose original class is given by the x-axis, as the target class, which is given by the y-axis. From left to right, the models are MNIST-B, MNIST-L5 and CIFAR10-AlexNet.

Targeted misclassification. Although our main focus is showing DNNs' graceless degradation, we conduct an additional experiment to see whether a single bit-flip primitive could be used in the context of *targeted misclassification* attacks. A targeted attack aims to preserve the victim model's overall accuracy while causing the network to misclassify a specific target sample into the target class. We experiment with a target sample from each class in MNIST or CIFAR10—we use MNIST-B, MNIST-L5, and CIFAR10-AlexNet models. Our white-box surgical attacker also preserves the accuracy by limiting the [RAD < 0.05] as in [55]. We find that *the number of vulnerable parameters for targeted misclassifications is lower than that of for causing indiscriminate damage*. In Figure 6, we also see that for some *(original–target class) pairs*, the vulnerability is more evident. For example, in MNIST-B, there are 141 vulnerable parameters for (class 4–class 6) and 209 parameters for (class 6–class 0). Simlarly, in CIFAR10-AlexNet, there are 6,000 parameters for (class 2–class 3); 3,000 parameters for (class 3–class 6); and 8,000 parameters for (class 6–class 3).

5 Exploiting Using Rowhammer

In order to corroborate the analysis made in Sec 4 and prove the viability of hardware fault attacks against DNN, we test the resiliency of these models against Rowhammer. At a high level, Rowhammer is a software-induced fault attack that provides the attacker with a single-bit write primitive to specific physical memory locations. That is, an attacker capable of performing specific memory access patterns (at DRAM-level) can induce persistent and repeatable bit corruptions from software. Given that we focus on single-bit perturbations on DNN's parameters in practical settings, Rowhammer represents the perfect candidate for the task.

DRAM internals. In Figure 7, we show the internals of a DRAM *bank*. A bank is a bi-dimensional array of memory *cells* connected to a *row buffer*. Every DRAM chip contains multiple banks. The cells are the actual storage of one's data. They contain a capacitor whose charge determines the value of a specific bit in memory. When a read is issued to a specific row, this row gets *activated*, which means that its content gets transferred to the row buffer before being sent to the CPU. Activation is often requested to recharge a row's capacitors (i.e., *refresh* operation) since they leak charge over time.

Rowhammer mechanism. Rowhammer is a DRAM disturbance error that causes spurious bit-flips in DRAM cells generated by frequent activations of a neighboring row. Here, we focus on double-sided Rowhammer, the most common and effective Rowhammer variant used in practical attacks [15, 44, 62]. Figure 8 exemplifies a typical double-sided Rowhammer attack. The victim's data is stored in a row enclosed between two aggressor rows that are repeatedly accessed by the attacker. Due to the continuous activations of the neighboring rows, the victim's data is under intense duress. Thus, there is a large probability of bit-flips on its content.

To implement such attack variant, the attacker usually needs some knowledge or control over the physical memory layout. Depending on the attack scenario, a Rowhammer-enabled attacker can rely on a different set of primitives for this purpose. In our analysis, we consider two possible scenarios: 1) we

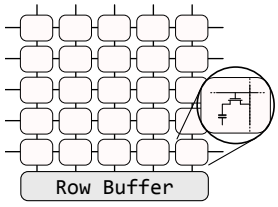


Figure 7: **DRAM bank structure.** Zoom-in on a cell containing the capacitor storing data.

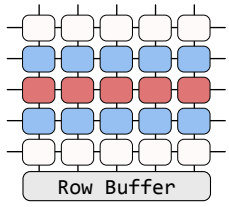


Figure 8: **Double-sided Rowhammer.** Aggressor rows , and a victim row .

DRAM	# (0→1) flips	DRAM	# (0→1) flips
A_2	21,538	A_4	5,577
E_2	16,320	I_1	4,781
H_1	10,608	J_1	4,725
G_1	7,851	E_1	4,175
A_1	4,367	A_3	1,541
F_1	5,927	C_1	1,365

Table 3: **Hammertime database [58].** We report the number of (0→1) bit-flips in 12 different DRAM setups. (The rows in gray are used for the experiments in Figure 9.)

initially consider the *surgical* attacker; that is, an attacker with the capability of causing bit-flips at the specific locations, and we demonstrate how, under these assumptions, she can induce indiscriminate damage to a co-located DNN application. 2) We then deprive the attacker of this ability to analyze the outcome of a *blind* attacker and we demonstrate that, even in a more restricted environment, the attacker can still cause indiscriminate damage by causing bitwise corruptions.

Experimental setup. For our analysis, we constructed a simulated environment[6] relying on a database of the Rowhammer vulnerability in 12 DRAM chips, provided by Tatar et al [58]. Different memory chips have a different degree of susceptibility to the Rowhammer vulnerability, enabling us to study the impact of Rowhammer attacks on DNNs in different real-world scenarios. Table 3 reports the susceptibility of the different memory chips to Rowhammer. Here, we only include the numbers for (0→1) bit-flips since these are the more interesting ones for the attacker targeting a DNN model according to our earlier analysis in Sec 4.3 and Sec 4.4.

We perform our analysis on an exemplary deep learning application implemented in PyTorch, constantly querying an ImageNet model. We use ImageNet models since we focus on a scenario where the victim has a relevant memory footprint that can be realistically be targeted by hardware fault attacks such as Rowhammer in practical settings. While small models are also potential targets, the number of interesting locations to corrupt is typically limited to draw general conclusions on the practical effectiveness of the attack.

5.1 Surgical Attack Using Rowhammer

We start our analysis by discussing a surgical attacker, who has the capability of causing a bit-flip at the specific location in memory. The two surgical attackers are available: the attacker with the knowledge of the victim model (white-box) and without (black-box). However, in this subsection, we assume that the strongest attacker knows the parameters to compromise and is capable of triggering bit-flips on its corresponding memory location. Then, this attacker can take advantage of accurate memory massing primitives (e.g., memory deduplication) to achieve 100% attack success rate.

[6]We first implemented all the steps described in our paper on a physical system, considering using end-to-end attacks for our analysis. After preliminary testing of this strategy on our own DRAMs, we concluded it would be hard to generalize the findings of such an analysis and decided against it—in line with observations from prior work [59].

Network	Vuln. Objects *(Vuln./Total)*	Vuln. Params *(in 20k params)*	#Hammer Attempts *(min/med/max)*
AlexNet	7/16	9,522	4/64/4,679
VGG16	12/32	8,140	4/64/4,679
ResNet50	9/102	3,466	4/64/4,679
DenseNet161	63/806	5,117	4/64/4,679
InceptionV3	53/483	6,711	4/64/4,679

Table 4: **Effectiveness of surgical attacks.** We examine five different ImageNet models analyzed in Sec 4.

Memory templating. Since a surgical attacker knows the location of vulnerable parameters, she can *template* the memory up front [44]. That is, the attacker scans the memory by inducing Rowhammer bit-flips in her own allocated chunks and looking for exploitable bit-flips. A surgical attacker aims at specific bit-flips. Hence, while templating the memory, the attacker simplifies the scan by looking for bit-flips located at specific offsets from the start address of a memory *page* (i.e., 4 KB)—the smallest possible chunk allocated from the OS. This allows the attacker to find memory pages vulnerable to Rowhammer bit-flips at a given page offset (i.e., *vulnerable templates*), which they can later use to predictably attack the victim data stored at that location.

Vulnerable templates. To locate the parameters of the attacker's interest (i.e., vulnerable parameters) within the memory page, she needs to find *page-aligned* data in the victim model. Modern memory allocators improve performances by storing large objects (usually multiples of the page size) page-aligned whereas smaller objects are not. Thus, we first analyzed the allocations performed by the PyTorch framework running on Python to understand if it performs such optimized page-aligned allocations for large objects similar to other programs [16, 39]. We discovered this to be the case for all the objects larger than 1 MB—i.e., our attacker needs to target the parameters such as weight, bias, and so on, stored as tensor objects in layers, larger than 1 MB.

Then, again focusing on the ImageNet models, we analyzed them to identify the objects that satisfy this condition. Even if the ratio between the total number of objects and target objects may seem often unbalanced in favor of the small ones[7], we found that the number of vulnerable parameters in the target objects is still significant (see Table 4). Furthermore, it is

[7]The `bias` in convolutional or dense layers, and the `running_mean` and `running_var` in batch-norms are usually the small objects (< 1 MB).

important to note that when considering a surgical attacker, she only needs one single vulnerable template to compromise the victim model, and there is only 1,024 possible offsets where we can store a 4-byte parameter within a 4 KB page.

Memory massaging. After finding a vulnerable template, the attacker needs to *massage* the memory to land the victim's data on the vulnerable template. This can be achieved, for instance, by exploiting memory deduplication [9, 44, 67]. Memory deduplication is a system-level memory optimization that merges read-only pages for different processes or VMs when they contain the same data. These pages re-split when a write is issued to them. However, Rowhammer behaves as invisible bit-wise writes that do not trigger the spit, breaking the process boundaries. If the attacker knows (even if partially) the content of the victim model can take advantage of this merging primitive to compromise the victim service.

Experimental results. Based on the results of the experiments in Sec 4.3 and Sec 4.4, we analyzed the requirements for a surgical (white-box) attacker to carry out a successful attack. Here, we used one set of the five sampled parameters for each model. In Table 4, we report *min, median, and max* values of the number of rows that an attacker needs to hammer to find the first vulnerable template on the 12 different DRAM setups for each model. This provides a meaningful metric to understand the success rate of a surgical attack. As you can see in Table 4, the results remain unchanged among all the different models. That is, for every model we tested in the best case, it required us to hammer only 4 rows (*A_2* DRAM setup) to find a vulnerable template all the way up to 4,679 in the worst case scenario (*C_1*). The reason why the results are equal among the different models is *due to the number of vulnerable parameters which largely exceeds the number of possible offsets within a page that can store such parameters* (i.e., 1024). Since every vulnerable parameter yields indiscriminate damage [RAD > 0.1], we simply need to identify a template that could match any given vulnerable parameter. This means that an attacker can find a vulnerable template at best in a matter of few seconds[8] and at worst still within minutes. Once the vulnerable template is found, the attacker can leverage memory deduplication to mount an effective attack against the DNN model—with no interference with the rest of the system.

5.2 Blind Attack Using Rowhammer

While in Sec 5.1 we analyzed the outcome of a surgical attack, here we abstract some of the assumptions made above and study the effectiveness of a blind attacker oblivious of the bit-flip location in memory. To bound the time of the lengthy blind Rowhammer attack analysis, we specifically focus our experiments on the ImageNet-VGG16 model.

We run our PyTorch application under the pressure of Rowhammer bit-flips indiscriminately targeting both code and data regions of the process's memory. Our goal is twofold: 1) to understand the effectiveness of such attack vector in a less controlled environment and 2) to examine the robustness of a running DNN application to Rowhammer bit-flips by measuring the number of failures (i.e., crashes) that our blind attacker may inadvertently induce.

Attacker's capabilities. We consider a blind attacker who cannot control the bit-flips caused by Rowhammer. As a result, the attacker may corrupt bits in the DNN's parameters as well as the code blocks in the victim process's memory. In principle, since Rowhammer bit-flips propagate at the DRAM level, a fully blind Rowhammer attacker may also inadvertently flip bits in other system memory locations. In practice, even an attacker with limited knowledge of the system memory allocator, can heavily influence the physical memory layout by means of specially crafted memory allocations [17, 18]. Since this strategy allows attackers to achieve co-location with the victim memory and avoid unnecessary fault propagation in practical settings, we restrict our analysis to a scenario where bit-flips can only (blindly) corrupt memory of the victim deep learning process. This also generalizes our analysis to arbitrary deployment scenarios, since the effectiveness of blind attacks targeting arbitrary system memory is inherently environment-specific.

Methods. For every one of the 12 vulnerable DRAM setups available in the database, we carried out 25 experiments where we performed at most 300 "hammering" attempts—value chosen after the surgical attack analysis where a median of 64 attempts was required. The experiment has three possible outcomes: 1) we trigger one(or more) effective bit-flip(*s*) that compromise the model, and we record the relative accuracy drop when performing our testing queries; 2) we trigger one(or more) effective bit-flip(*s*) in other victim memory locations that result in a crash of the deep learning process; 3) we reach the "timeout" value of 300 hammering attempts. We set such "timeout" value to bound our experimental analysis which would otherwise result too lengthy.

Experimental results. In Figure 9, we present the results for three sampled DRAM setups. We picked *A_2*, *I_1*, and *C_1* as representative samples since they are the most, least, and moderately vulnerable DRAM chips (see Table 3). Depending on the DRAM setup, we obtain fairly different results. We found *A_2* obtains successful indiscriminate damages to the model in 24 out of 25 experiments while, in less vulnerable environments such as *C_1*, the number of successes decreases to only one while the other 24 times out. However, it is im-

[8] We assume 200ms to hammer a row.

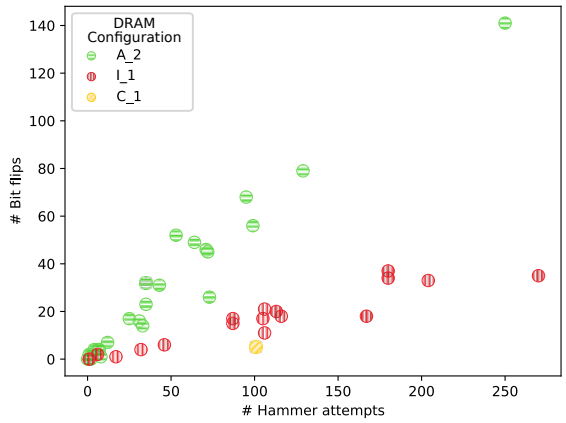


Figure 9: **The successful runs of a blind attack execution over three different DRAM setups (*A_2*-most, *I_1*-least, and *C_1*-moderately vulnerable).** We report the success in terms of #*flips* and #*hammer attempts* required to obtain an indiscriminate damage to the victim model. We observe the successes within few hammering attempts.

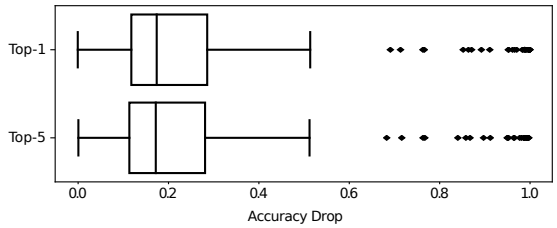


Figure 10: **The distribution of relative accuracy drop for Top-1 and Top-5.** We compute them over the effective #*flips* in our experiments on the ImageNet-VGG16 model.

portant to note that a timeout does not represent a negative result—a crash. Contrarily, while *C_1* only had a single successful attack, it also represents a peculiar case corroborating the analysis presented in Sec 4. The corruption generated in this single successful experiment was induced by a single bit-flip, which caused one of the most significant RADs detected in the entire experiment, i.e., 0.9992 and 0.9959 in Top-1 and Top-5. Regardless of this edge case, we report a mean of 15.6 out of 25 effective attacks for this Rowhammer variant over the different DRAM setups. Moreover, we report the distribution of accuracy drops for Top-1 and Top-5 in Figure 10. In particular, the median drop for Top-1 and Top-5 confirms the claims made in the previous sections, i.e., the blind attacker can expect [RAD > 0.1] on average.

Interestingly, when studying the robustness of the victim process to Rowhammer, we discovered it to be quite resilient to spurious bit-flips. We registered only 6 crashes over all the different DRAM configurations and experiments—300 in total. This shows that the model effectively dominates the memory footprint of the victim process and confirms findings from our earlier analysis that bit-flips in non-vulnerable model elements have essentially no noticeable impact.

5.3 Synopsis

Throughout the section, we analyzed the outcome of surgical and blind attacks against large DNN models and demonstrated how Rowhammer can be deployed as a feasible attack vector against these models. These results corroborate our findings in Sec 4 where we estimated at least 40% of a model's parameters to be vulnerable to single-bit corruptions. Due to this large attack surface, in Sec 5.1, we showed that a Rowhammer-enabled attacker armed with knowledge of the network's parameters and powerful memory massaging primitives [44, 62, 67] can carry out precise and effective indiscriminate attacks in a matter of, at most, few minutes in our simulated environment. Furthermore, this property, combined with the resiliency to spurious bit-flips of the (perhaps idle) code regions, allowed us to build successful blind attacks against the ImageNet-VGG16 model and inflict "terminal brain damage" even when hiding the model from the attacker.

6 Discussion

In this section, we discuss and evaluate some potential mitigations to protect against single-bit attacks on DNN models. We discuss two research directions towards making DNN models resilient to bit-flips: *to restrict activation magnitudes* and *to use low-precision numbers*. Prior work on defenses against Rowhammer attacks suggest system-level defenses [10, 27] that often even require specific hardware support [6, 26]. Yet they have not been widely deployed since they require infrastructure-wide changes from cloud host providers. Moreover, even though the infrastructure is robust to Rowhammer attacks, an adversary can leverage other vectors to exploit bit-flips attacks to corrupt a model. Thus, we focus on the solutions that our victim can apply to his models.

6.1 Restricting Activation Magnitudes

In Sec 4.3, we found that the vulnerable parameter ratio changes based on inherent properties of a DNN; for instance, using PReLU activation function allows a model to propagate negative extreme activations. Hence, if we opt for an activation function that always bounds the output within a specific range, a bit-flip is hard to cause indiscriminate damage. There are several functions, such as Tanh or HardTanh [25], that suppresses the activations; however, using ReLU-6 [28] function provides two key advantages over the others: 1) the victim only needs to substitute the existing activation functions from ReLU to ReLU-6 without re-training, and 2) ReLU-6 allows

Network	Train	Base acc.	# Params	Vulnerability
Base (ReLU)	Scr	98.13	21,840	10,972 (50.2%)
Base (ReLU6)	Scr	98.16		313 (1.4%)
Base (Tanh)	Scr	97.25		507 (2.3%)
Base (ReLU6)	Sub	95.71		542 (2.4%)
AlexNet (ReLU)	-	56.52 / 79.07	20,000 (61M)	9.467 (47.34%)
AlexNet (ReLU6)	Sub	39.80 / 65.82		560 (2.8%)
AlexNet (ReLUA)	Sub	56.52 / 79.07		1,063 (5.32%)
VGG16 (ReLU)	-	64.28 / 86.56	20,000 (138M)	8,227 (41.13%)
VGG16 (ReLU6)	Sub	38.58 / 64.84		2,339 (11.67%)
VGG16 (ReLUA)	Sub	64.28 / 86.56		2,427 (12.14%)

Table 5: **Effectiveness of restricting activation.**

the victim to control the level of permitted activation by modifying the bounds, e.g., using other limits instead of 6, which minimizes the performance loss by bounding the activation. What the victim can do is to monitor the activation values over the validation set and to decide the limits that only suppresses the abnormal activation by bit-flips. For example, in our experiments with ImageNet-AlexNet, we set the limits to $[0, max]$, where max is defined adaptively by looking at the maximum activation from each layer (ReLU-A).

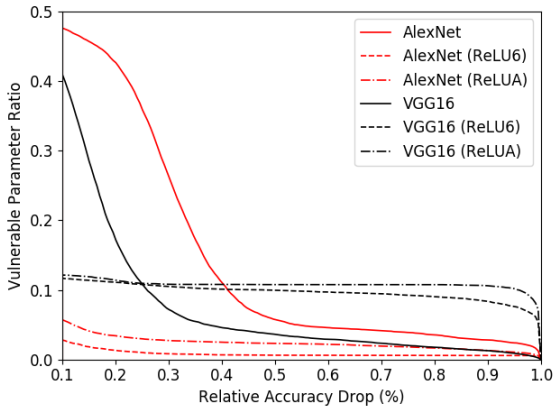


Figure 11: **The vulnerability of DNN models using different criterions.** We illustrate the ImageNet-AlexNet (red lines) and -VGG16 (black lines) cases with ReLU-6 and ReLU-A.

Experiments. We use three DNN models in Sec 4: the MNIST-B, ImageNet-AlexNet, and ImageNet-VGG16 models. We evaluate four activation functions: ReLU (default), Tanh, ReLU-6, and ReLU-A (only for AlexNet and VGG16), and two training methods: training a model from scratch (Scr) or substituting the existing activation into another (Sub). We use the notation as the network names with the activation in parenthesis, e.g., AlexNet (ReLU-6). For larger models, we use the same speed-up heuristics in Sec 4.2; SV, SB, and SP.

Table 5 shows the effectiveness of our proposal. For each network (Column 1), we list the training method, the base accuracy, the number of examined parameters, and the vulnerability (Column 2-5). We found that restricting activation magnitudes with Tanh and ReLU-6 in some instances can reduce the vulnerability; For instance, in the MNIST models, we observed that the number of vulnerable parameters is reduced from 50% to 1.4-2.4% without incurring in significant performance loss. Further, we discovered that ReLU-6 achieves a similar effect without re-training of a model like Tanh. However, there are the vulnerable parameters after the restrictions since we cannot apply the ReLU-6 function to the last layer. In AlexNet and VGG16, the decrease in the number of vulnerable parameters is also generally significant, namely from 47.34% to 2.8% and 41.13% to 11.67%. However, we observe the models suffer from large accuracy drops caused by restricting the activation. To minimize the loss, we control the bounds of activation in AlexNet (ReLU-A) and VGG16 (ReLU-A) by choosing the maximum activation from each layer. With the ReLU-A, we can trade accuracy for the number of vulnerable parameters as we show in Table 5. Nevertheless, it is interesting to see that by employing ReLU-A, while the number of vulnerable parameters remains significant, the RAD also suffers from the new activation function limiting the possible effects of the corruption. In Figure 11, the dashed lines are for ReLU-6, the dashed-dot lines are for ReLU-A, and the straight lines are for ReLU. We found the ReLU-A lines are between the ReLU and ReLU-6 in AlexNet.

Network	Method	Base acc.	# Params	Vulnerability
L5	-	99.24	62,598	30,686 (49.0%)
L5	8-bit Quantized	99.03	62,600	0 (0.0%)
L5	XNOR Binarized	98.39	62,286	623 (1.0%)

Table 6: **Effectiveness of using low-precision.**

Takeaways. Our experimental results with restricting activation magnitudes suggest that: this mechanism 1) allows a defender to control the trade-off between the relative accuracy drop and reducing the vulnerable parameters and 2) enables ad-hoc defenses to DNN models, which does not require training the network from scratch. However, the remaining number of vulnerable parameters shows that the Rowhammer attacker still could inflict damage, with a reduced success rate.

6.2 Using Low-precision Numbers

Another direction is to represent the model parameters as low-precision numbers by using quantization and binarization. In Sec 4.3, we found that the vulnerability exploits the bit-wise representation of the corrupted parameter to induce the dramatic chances in the parameter value. Our intuition is to use low-precision numbers hard to be increased dramatically by a bit-flip; for example, an integer expressed as the 8-bit quantized format can be increased at most 128 by a flip in the

MSB (8th bit). Thus, the attacker only can increase a model parameter with such a restricted bound. Training models using low-precision numbers are supported by the popular deep learning frameworks such as TensorFlow[9]. The victim can train and deploy the model with the quantized or binarized parameters by utilizing the frameworks.

Experiments. To validate our intuition, we use 3 DNN models: the MNIST-L5 (baseline) and its quantized and binarized models. When we quantize the MNIST-L5 model, we use the 8-bit quantization in [7, 64] which converts the model parameters in all layers into integers between 0 and 255. For the binarization, we employ the method in XNOR-Net [43] which converts the model parameters to -1 and 1, except the first convolutional layer. Using the trained models, we evaluate the vulnerability to single bit-flips, and report the accuracy, total parameters, and vulnerability, without the speed-up heuristics.

Table 6 shows the effectiveness of using low-precision parameters. For each network (Column 1), we note the quantization method, the accuracy, the number of vulnerable parameters, and their percentage (Column 2-5). We found that *using low-precision parameters reduces the vulnerability*; in all cases, the percentage of vulnerable parameters are reduced from 49% (Baseline) to 0-2% (surprisingly 0% with the quantization). We focus on analyzing which layer has vulnerable parameters in the binarization model. We found that mostly the parameters in the first convolutional (150 parameters) and classification (last) layers (420 parameters) are vulnerable to a bit-flip, which corroborates what observed in Sec 4.3.

Takeaways. Even though we showed the elimination of the vulnerability through 8-bit quantization, in a real-world, training the large model such as [65] from scratch can take a week on a supercomputing cluster.

7 Related Work

DNN's resilience to perturbations. Prior work has utilized the *graceful degredation* of DNN models under parameter perturbations in a wide range of applications. For example, network quantization [3, 5], by quantizing a DNN model's high-precision parameter into low-precision, reduces the size and inference time of a model with negligible performance penalty. This property has also been used as a primitive for improving the security of DNNs. For example, modifying the parameter slightly to inject a watermark to allow model owners to prove ownership [1]; adding Gaussian noise to model parameter for reducing the reliability of test-time adversarial attacks on DNNs [69]; and fine-tuning the parameters for mitigating the malicious backdoors in a model [37]. Further, the resilience to structural changes has lead to pruning techniques [4, 20, 35] which improve the efficiency of a DNN model by removing unimportant neurons along with their parameters. In our work, we study the *graceless degredation* of DNNs under hardware fault attacks that induce single bit-flips to individual parameters.

Indiscriminate poisoning attacks on DNNs. Recent work on adversarial machine learning has demonstrated many attack scenarios to inflict indiscriminate damage on a model. One of the well-studied vectors is *indiscriminate poisoning attacks* [8] in which the adversary, by injecting malicious data in the victim's training set, aims to hurt the model. Previous studies suggest that such attack might require significant amount of poisonous instances [40]. For example, Steinhardt et al. [54] shows that, with IMDB dataset, an attacker needs to craft 3% of the total training instances to achieve 11% of accuracy drop compared to the pristine model. Further, the defenses based on robust outlier removal techniques could render poison injection ineffective by filtering it out [14, 54]. Moreover, to achieve targeted damages without harming the model's overall accuracy, *targeted poisoning attacks* [49, 55] have been studied. In this paper, we analyze a test-time vulnerability that does not require the adversary's contact to the victim model during its training. This vulnerability inflicts indiscriminate damage, similar to indiscriminate poisoning attacks, through a different attack medium.

Hardware fault injection attacks. Hardware fault injection is a class of attacks that rely on hardware glitches on the system to corrupt victim's data. These glitches generally provide a single-bit write primitive at the physical memory; which could potentially lead to privilege escalation [67]. While in the past these attacks required physical access to the victim's system [11, 38], recently they have gained more momentum since the software-based version of these attacks were demonstrated [26, 57]. Instances of these attacks are 1) the CLKSCREW attack [57] that leverages dynamic voltage and frequency scaling on mobile processors generate faults on instructions; or 2) the well-known Rowhammer vulnerability that triggers bitwise corruptions in DRAM. Rowhammer has been used in the context of cloud VMs [44, 67], on desktops [48] and mobile [62] and even to compromise browsers from JavaScript [9, 15, 19]. In the context of DNNs, fault attacks have been proposed as an alternative for inflicting indiscriminate damages. Instead of injecting poisonous instances, fault attacks directly induce perturbations to the models running on hardware [11, 13, 34, 38, 45]. These studies have considered the adversaries with direct access to the victim hardware [11, 13] and adversaries who randomly corrupt parameters [34, 38, 45]. We utilize Rowhammer as an established fault attack to demonstrate practical implications of the graceless degradation of DNNs. Our threat model follows the realistic single bit-flip capability of a fault attack and modern

[9] https://www.tensorflow.org/lite/performance/post_training_quantization

application of DNNs in a cloud environment, where physical access to the hardware is impractical.

8 Conclusions

This work exposes the limits of DNN's resilience against the parameter perturbations. We study the vulnerability of DNN models to single bit-flips. We evaluated 19 DNN models with six architectures on three image classification tasks and showed that: we can easily find 40-50% vulnerable parameters where an attacker can cause indiscriminate damage [RAD > 0.1] by a bit-flip. We further characterize this vulnerability based on the impact of various factors: the bit position, bit-flip direction, parameter sign, layer width, activation function, training techniques, and model architecture. Understanding this emerging threat, we leverage the software-induced fault injection, Rowhammer, to demonstrate the feasibility of the bit-flip attacks in practice. In experiments with RowHammer, we found that, without knowing the victim's deep learning system, the attacker can inflict indiscriminate damage without system crashes. Lastly, motivated by the attacks, we discuss two potential directions of mitigation: restricting activation magnitudes and using low-precision numbers.

Acknowledgments

We thank Tom Goldstein, Dana Dachman-Soled, our shepherd, David Evans, and the anonymous reviewers for their feedback. We also acknowledge the University of Maryland super-computing resources[10] (DeepThought2) made available for conducting the experiments reported in our paper. This research was partially supported by the Department of Defense, by the United States Office of Naval Research (ONR) under contract N00014-17-1-2782 (BinRec), by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 786669 (ReAct) and No. 825377 (UNICORE), and by the Netherlands Organisation for Scientific Research through grant NWO 639.021.753 VENI (PantaRhei). This paper reflects only the authors' view. The funding agencies are not responsible for any use that may be made of the information it contains.

References

[1] Yossi Adi, Carsten Baum, Moustapha Cisse, Benny Pinkas, and Joseph Keshet. Turning your weakness into a strength: Watermarking deep neural networks by backdooring. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1615–1631, Baltimore, MD, 2018. USENIX Association.

[2] G. An. The effects of adding noise during backpropagation training on a generalization performance. *Neural Computation*, 8(3):643–674, April 1996.

[3] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Fixed point optimization of deep convolutional neural networks for object recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 1131–1135. IEEE, 2015.

[4] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 13(3):32, 2017.

[5] Yiwen Guo Lin Xu Yurong Chen Aojun Zhou, Anbang Yao. Incremental network quantization: Towards lossless cnns with low-precision weights. In *International Conference on Learning Representations (ICLR)*, 2017.

[6] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. Anvil: Software-based protection against next-generation rowhammer attacks. *ACM SIGPLAN Notices*, 51(4):743–755, 2016.

[7] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. Scalable methods for 8-bit training of neural networks. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 5145–5153. Curran Associates, Inc., 2018.

[8] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning attacks against support vector machines. In *Proceedings of the 29th International Coference on International Conference on Machine Learning*, ICML'12, pages 1467–1474, USA, 2012. Omnipress.

[9] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *2016 IEEE symposium on security and privacy (SP)*, pages 987–1004. IEEE, 2016.

[10] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. Can't touch this: Software-only mitigation against rowhammer attacks targeting kernel memory. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 117–130, Vancouver, BC, 2017. USENIX Association.

[11] Jakub Breier, Xiaolu Hou, Dirmanto Jap, Lei Ma, Shivam Bhasin, and Yang Liu. Practical fault attack on deep neural networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 2204–2206, New York, NY, USA, 2018. ACM.

[12] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730, 2015.

[13] Joseph Clements and Yingjie Lao. Hardware trojan attacks on neural networks, 2018.

[14] Ilias Diakonikolas, Gautam Kamath, Daniel M. Kane, Jerry Li, Jacob Steinhardt, and Alistair Stewart. Sever: A robust meta-algorithm for stochastic optimization, 2018.

[15] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand pwning unit: accelerating microarchitectural attacks with the gpu. In *Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU*, page 0. IEEE, 2018.

[16] Sanjay Ghemawat. Tcmalloc : Thread-caching malloc, 2018.

[17] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. Page cache attacks. *arXiv preprint arXiv:1901.01161*, 2019.

[18] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another flip in the wall of rowhammer defenses. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 245–261. IEEE, 2018.

[19] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in javascript. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 300–321. Springer, 2016.

[10] http://hpcc.umd.edu

[20] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *International Conference on Learning Representations (ICLR)*, 2016.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[23] Forrest Iandola, Matt Moskewicz, Sergey Karayev, Ross Girshick, Trevor Darrell, and Kurt Keutzer. Densenet: Implementing efficient convnet descriptor pyramids. *arXiv preprint arXiv:1404.1869*, 2014.

[24] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR.

[25] Barry L Kalman and Stan C Kwasny. Why tanh: Choosing a sigmoidal function. In *Neural Networks, 1992. IJCNN., International Joint Conference on*, volume 4, pages 578–581. IEEE, 1992.

[26] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 361–372. IEEE Press, 2014.

[27] Radhesh Krishnan Konoth, Marco Oliverio, Andrei Tatar, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. Zebram: Comprehensive and compatible software protection against rowhammer attacks. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 697–710, Carlsbad, CA, 2018. USENIX Association.

[28] Alex Krizhevsky and Geoff Hinton. Convolutional deep belief networks on cifar-10. *Unpublished manuscript*, 40(7), 2010.

[29] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[30] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[31] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[32] Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605, 1990.

[33] Yann LeCun et al. Lenet-5, convolutional neural networks. *URL: http://yann. lecun. com/exdb/lenet*, page 20, 2015.

[34] Guanpeng Li, Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, Karthik Pattabiraman, Joel Emer, and Stephen W Keckler. Understanding error propagation in deep learning neural network (dnn) accelerators and applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 8. ACM, 2017.

[35] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[36] Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster. Nethammer: Inducing rowhammer faults through network requests. *arXiv preprint arXiv:1805.04956*, 2018.

[37] Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. Fine-pruning: Defending against backdooring attacks on deep neural networks. In *Research in Attacks, Intrusions, and Defenses (RAID)*, pages 273–294, 2018.

[38] Yannan Liu, Lingxiao Wei, Bo Luo, and Qiang Xu. Fault injection attack on deep neural network. In *Proceedings of the 36th International Conference on Computer-Aided Design*, pages 131–138. IEEE Press, 2017.

[39] Jemalloc manual. Jemalloc: general purpose memory allocation functions. http://jemalloc.net/jemalloc.3.html, 2019.

[40] Blaine Nelson, Marco Barreno, Fuching Jack Chi, Anthony D Joseph, Benjamin IP Rubinstein, Udam Saini, Charles A Sutton, J Doug Tygar, and Kai Xia. Exploiting machine learning to subvert your spam filter. *LEET*, 8:1–9, 2008.

[41] Maria-Elena Nilsback and Andrew Zisserman. Automated flower classification over a large number of classes. In *Computer Vision, Graphics & Image Processing, 2008. ICVGIP'08. Sixth Indian Conference on*, pages 722–729. IEEE, 2008.

[42] Minghai Qin, Chao Sun, and Dejan Vucinic. Robustness of neural networks against storage media errors, 2017.

[43] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *ECCV*, 2016.

[44] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1–18, Austin, TX, 2016. USENIX Association.

[45] Brandon Reagen, Udit Gupta, Lillian Pentecost, Paul Whatmough, Sae Kyu Lee, Niamh Mulholland, David Brooks, and Gu-Yeon Wei. Ares: A framework for quantifying the resilience of deep neural networks. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[46] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[47] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[48] Mark Seaborn and Thomas Dullien. Exploiting the dram rowhammer bug to gain kernel privileges.

[49] Ali Shafahi, W. Ronny Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and Tom Goldstein. Poison frogs! targeted clean-label poisoning attacks on neural networks. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 6106–6116. Curran Associates, Inc., 2018.

[50] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *International Conference on Learning Representations (ICLR)*, 2015.

[51] Nikolai Smolyanskiy, Alexey Kamenev, Jeffrey Smith, and Stan Birchfield. Toward low-flying autonomous mav trail navigation using deep neural networks for environmental awareness. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4241–4247. IEEE, 2017.

[52] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[53] Johannes Stallkamp, Marc Schlipsing, Jan Salmen, and Christian Igel. Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition. *Neural networks*, 32:323–332, 2012.

[54] Jacob Steinhardt, Pang Wei W Koh, and Percy S Liang. Certified defenses for data poisoning attacks. In *Advances in Neural Information Processing Systems*, pages 3517–3529, 2017.

[55] Octavian Suciu, Radu Marginean, Yigitcan Kaya, Hal Daume III, and Tudor Dumitras. When does machine learning FAIL? generalized transferability for evasion and poisoning attacks. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1299–1316, Baltimore, MD, 2018. USENIX Association.

[56] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[57] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: Exposing the perils of security-oblivious energy management. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1057–1074, Vancouver, BC, 2017. USENIX Association.

[58] Andrei Tatar, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Defeating software mitigations against rowhammer: a surgical precision hammer. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 47–66. Springer, 2018.

[59] Andrei Tatar, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Defeating software mitigations against rowhammer: a surgical precision hammer. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 47–66. Springer, 2018.

[60] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer attacks over the network and defenses. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 213–226, Boston, MA, 2018. USENIX Association.

[61] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 601–618, Austin, TX, 2016. USENIX Association.

[62] Victor Van Der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1675–1689. ACM, 2016.

[63] Bolun Wang, Yuanshun Yao, Bimal Viswanath, Haitao Zheng, and Ben Y. Zhao. With great training comes great vulnerability: Practical attacks against transfer learning. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1281–1297, Baltimore, MD, 2018. USENIX Association.

[64] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 7675–7684. Curran Associates, Inc., 2018.

[65] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[66] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. Security implications of memory deduplication in a virtualized environment. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2013.

[67] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 19–35, Austin, TX, 2016. USENIX Association.

[68] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network, 2015.

[69] Yan Zhou, Murat Kantarcioglu, and Bowei Xi. Breaking transferability of adversarial samples with randomness, 2018.

Appendix

A Network Architectures

We use 19 DNN models in our experiments: six architecture and their variants. Table 7 describes two architectures and their six variations for MNIST. For CIFAR10, we employ the base architecture from [55] that has four convolutional layers and a fully-connected layer, and we make three variations of it. CIFAR10-AlexNet[11] and CIFAR10-VGG16[12] are from the community. For ImageNet, we use the DNN architectures available from the Internet[13]. In Sec 6.2, we employ two networks (8-bit quantized[14] and binarized versions of MNIST-L5) from the community[15] with adjustments.

B The Vulnerability Using Different Criterion

We examine the vulnerable parameter ratio (vulnerability) using the different RAD criterion with 15 DNN models. Our results are in Figure 12. Each figure describe the vulnerable parameter ratio on a specific RAD criterion; for instance, in MNIST-L5, the model has 40% of vulnerable parameters that cause [RAD > 0.5], which estimates the upper bound of the blind attacker. In MNIST, CIFAR10, and two ImageNet models, the vulnerability decreases as the attacker aims to inflict the severe damage; however, in ImageNet, ResNet50, DenseNet161, and InceptionV3 have almost the same vulnerability (∼50%) with the high criterion [RAD > 0.8].

C Hyper-parameters for Training

In our experiments, we use these hyper-parameters:

- **MNISTs.** For MNIST models, we use: SGD, 40 epochs, 0.01 learning rate (lr), 64 batch, 0.1 momentum, and adjust learning rate by 0.1, in every 10 epochs.
- **CIFAR10s.** For Base models we use: SGD, 50 epochs, 0.02 lr, 32 batch, 0.1 momentum, and adjust lr by 0.5, in every 10 epochs. For AlexNet, we use: 300 epochs,

[11] https://github.com/bearpaw/pytorch-classification/blob/master/models/cifar/alexnet.py
[12] https://github.com/kuangliu/pytorch-cifar/blob/master/models/vgg.py
[13] https://github.com/pytorch/vision/tree/master/torchvision/models
[14] https://github.com/eladhoffer/quantized.pytorch
[15] https://github.com/jiecaoyu/XNOR-Net-PyTorch

Table 7: **8 Network Architectures for MNIST.** We take the two baselines (Base and LeNet5) and make four and two variants from them, respectively. Note that we highlight the variations from the baselines in red color.

Base		**Base (Wide)**		**Base (Dropout)**		**Base (PReLU)**	
Layer Type	**Layer Size**	**Layer Type**	**Layer Size**	**Layer Type**	**Layer Size**	**Layer Type**	**Layer Size**
Conv (R)	5x5x10 (2)	Conv (R)	5x5x20 (2)	Conv (R)	5x5x10 (2)	Conv (P)	5x5x10 (2)
Conv (R)	5x5x20 (2)	Conv (R)	5x5x40 (2)	Conv (-)	5x5x20 (2)	Conv (P)	5x5x20 (2)
-	-	-	-	Dropout (R)	0.5	-	-
FC (R)	50	FC (R)	100	FC (R)	50	FC (P)	50
-	-	-	-	Dropout (R)	0.5	-	-
FC (S)	10	FC (S)	10	FC (S)	10	FC (S)	10
Base (D-BNorm)		**LeNet5** [33]		**LeNet5 (Dropout)**		**LeNet5 (D-BNorm)**	
Layer Type	**Layer Size**	**Layer Type**	**Layer Size**	**Layer Type**	**Layer Size**	**Layer Type**	**Layer Size**
Conv (-)	5x5x10 (2)	Conv (R)	5x5x6 (2)	Conv (R)	5x5x6 (2)	Conv (-)	5x5x6 (2)
BatchNorm (R)	10	-	-	-	-	BatchNorm (R)	6
-	-	MaxPool (-)	2x2	MaxPool (-)	2x2	MaxPool (-)	2x2
Conv (-)	5x5x20 (2)	Conv (R)	5x5x16 (2)	Conv (R)	5x5x16 (2)	Conv (-)	5x5x16 (2)
BatchNorm (R)	20	-	-	-	-	BatchNorm (R)	16
-	-	MaxPool (-)	2x2	MaxPool (-)	2x2	MaxPool (-)	2x2
-	-	Conv (R)	5x5x120 (2)	Conv (R)	5x5x120 (2)	Conv (R)	5x5x120 (2)
-	-	-	-	-	-	BatchNorm (R)	120
Dropout (R)	0.5	-	-	Dropout (R)	0.5	Dropout (R)	0.5
-	-	MaxPool (-)	2x2	MaxPool (-)	2x2	MaxPool (-)	2x2
-	-	Conv (R)	5x5x120 (1)	Conv (R)	5x5x120 (1)	Conv (R)	5x5x120 (1)
FC (R)	50	FC (R)	84	FC (R)	84	FC (R)	84
Dropout (R)	0.5	-	-	Dropout (R)	0.5	Dropout (R)	0.5
FC (S)	10	FC (S)	10	FC (S)	10	FC (S)	10

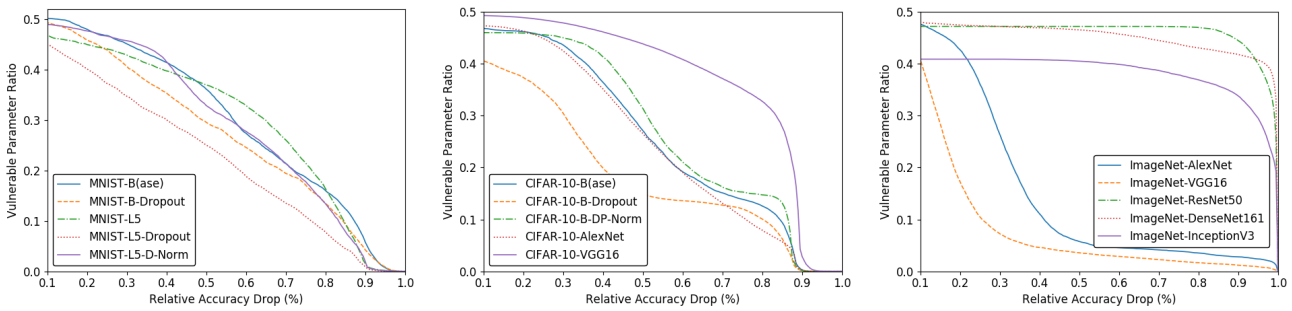


Figure 12: **The vulnerability of 15 DNN models using different criterions.** We plot the vulnerable parameter ratio based on the different RADs that an attacker aims; 5 from MNIST (left), 5 from CIFAR10 (middle), and 5 from ImageNets (right).

0.01 lr, 64 batch, 0.1 momentum, and adjust lr by 0.95, in every 10 epochs. For VGG16, we use: 300 epochs, 0.01 lr, 128 batch, 0.1 momentum, and adjust lr by 0.15, in every 100 epochs.

- **GTSRB.** We fine-tune VGG16 pre-trained on ImageNet, using: SGD, 40 epochs, 0.01 lr, 32 batch, 0.1 momentum, and adjust lr by 0.1 and 0.05, in 15 and 25 epochs. We freeze the parameters of the first 10 layers.

- **Flower102.** We fine-tune ResNet50 pre-trained on ImageNet, using: SGD, 40 epochs, 0.01 lr, 50 batch, 0.1 momentum, and adjust lr by 0.1, in 15 and 25 epochs. We freeze the parameters of the first 10 layers.

CSI NN: Reverse Engineering of Neural Network Architectures Through Electromagnetic Side Channel

Lejla Batina
Radboud University, The Netherlands

Shivam Bhasin
Nanyang Technological University, Singapore

Dirmanto Jap
Nanyang Technological University, Singapore

Stjepan Picek
Delft University of Technology, The Netherlands

Abstract

Machine learning has become mainstream across industries. Numerous examples prove the validity of it for security applications. In this work, we investigate how to reverse engineer a neural network by using side-channel information such as timing and electromagnetic (EM) emanations. To this end, we consider multilayer perceptron and convolutional neural networks as the machine learning architectures of choice and assume a non-invasive and passive attacker capable of measuring those kinds of leakages.

We conduct all experiments on real data and commonly used neural network architectures in order to properly assess the applicability and extendability of those attacks. Practical results are shown on an ARM Cortex-M3 microcontroller, which is a platform often used in pervasive applications using neural networks such as wearables, surveillance cameras, etc. Our experiments show that a side-channel attacker is capable of obtaining the following information: the activation functions used in the architecture, the number of layers and neurons in the layers, the number of output classes, and weights in the neural network. Thus, the attacker can effectively reverse engineer the network using merely side-channel information such as timing or EM.

1 Introduction

Machine learning, and more recently deep learning, have become hard to ignore for research in distinct areas, such as image recognition [25], robotics [21], natural language processing [47], and also security [53, 26] mainly due to its unquestionable practicality and effectiveness. Ever increasing computational capabilities of the computers of today and huge amounts of data available are resulting in much more complex machine learning architectures than it was envisioned before. As an example, AlexNet architecture consisting of 8 layers was the best performing algorithm in image classification task ILSVRC2012 (`http://www.image-net.org/challenges/LSVRC/2012/`). In 2015, the best performing architecture for the same task was ResNet consisting of 152 layers [15]. This trend is not expected to stagnate any time soon, so it is prime time to consider machine/deep learning from a novel perspective and in new use cases. Also, deep learning algorithms are gaining popularity in IoT edge devices such as sensors or actuators, as they are indispensable in many tasks, like image classification or speech recognition. As a consequence, there is an increasing interest in deploying neural networks on low-power processors found in always-on systems, e.g., ARM Cortex-M microcontrollers.

In this work, we focus on two neural network algorithms: multilayer perceptron (MLP) and convolutional neural networks (CNNs). We consider feed-forward neural networks and consequently, our analysis is conducted on such networks only.

With the increasing number of design strategies and elements to use, fine-tuning of hyper-parameters of those algorithms is emerging as one of the main challenges. When considering distinct industries, we are witnessing an increase in intellectual property (IP) models strategies. Basically, in cases when optimized networks are of commercial interest, their details are kept undisclosed. For example, EMVCo (formed by MasterCard and Visa to manage specifications for payment systems and to facilitate worldwide interoperability) nowadays requires deep learning techniques for security evaluations [43]. This has an obvious consequence in: 1) security labs generating (and using) neural networks for evaluation of security products and 2) they treat them as IP, exclusively for their customers.

There are also other reasons for keeping the neural network architectures secret. Often, these pre-trained models might provide additional information regarding the training data, which can be very sensitive. For example, if the model is trained based on a medical record of a patient [9], confidential information could be encoded into the network during the training phase. Also, machine learning models that are used for guiding medical treatments are often based on a patient's genotype making this extremely sensitive from the privacy perspective [10]. Even if we disregard privacy issues,

obtaining useful information from neural network architectures can help acquiring trade secrets from the competition, which could lead to competitive products without violating intellectual property rights [3]. Hence, determining the layout of the network with trained weights is a desirable target for the attacker. One could ask the following question: Why would an attacker want to reverse engineer the neural network architecture instead of just training the same network on its own? There are several reasons that are complicating this approach. First, the attacker might not have access to the same training set in order to train his own neural network. Although this is admittedly a valid point, recent work shows how to solve those limitations [49]. Second, as the architectures have become more complex, there are more and more parameters to tune and it could be extremely difficult for the attacker to pinpoint the same values for the parameters as in the architecture of interest.

After motivating our use case, the main question that remains is on the feasibility of reverse engineering such architectures. Physical access to a device could allow readily reverse engineering based on the binary analysis. However, in a confidential IP setting, standard protections like blocking binary readback, blocking JTAG access [20], code obfuscation, etc. are expected to be in place and preventing such attacks. Nevertheless, even when this is the case, a viable alternative is to exploit side-channel leakages.

Side-channel analysis attacks have been widely studied in the community of information security and cryptography, due to its potentially devastating impact on otherwise (theoretically) secure algorithms. Practically, the observation that various physical leakages such as timing delay, power consumption, and electromagnetic emanation (EM) become available during the computation with the (secret) data has led to a whole new research area. By statistically combining this physical observation of a specific internal state and hypothesis on the data being manipulated, it is possible to recover the intermediate state processed by the device.

In this study, our aim is to highlight the potential vulnerabilities of standard (perhaps still naive from the security perspective) implementations of neural networks. At the same time, we are unaware of any neural network implementation in the public domain that includes side-channel protection. For this reason, we do not just pinpoint to the problem but also suggest some protection measures for neural networks against side-channel attacks. Here, we start by considering some of the basic building blocks of neural networks: the number of hidden layers, the basic multiplication operation, and the activation functions.

For instance, the complex structure of the activation function often leads to conditional branching due to the necessary exponentiation and division operations. Conditional branching typically introduces input-dependent timing differences resulting in different timing behavior for different activation function, thus allowing the function identification. Also, we notice that by observing side-channel leakage, it is possible to deduce the number of nodes and the number of layers in the networks.

In this work, we show it is possible to recover the layout of unknown networks by exploiting the side-channel information. Our approach does not need access to training data and allows for network recovery by feeding known random inputs to the network. By using the known divide-and-conquer approach for side-channel analysis, (i.e., the attacker's ability to work with a feasible number of hypotheses due to, e.g., the architectural specifics), the information at each layer could be recovered. Consequently, the recovered information can be used as input for recovering the subsequent layers.

We note that there exists somewhat parallel research to ours also on reverse engineering by "simply" observing the outputs of the network and training a substitute model. Yet, this task is not so simple since one needs to know what kind of architecture is used (e.g., convolutional neural network or multilayer perceptron, the number of layers, the activation functions, access to training data, etc.) while limiting the number of queries to ensure the approach is realistic [39]. Some more recent works have tried to overcome a few of the highlighted limitations [49, 18].

To our best knowledge, this kind of observation has never been used before in this context, at least not for leveraging on (power/EM) side-channel leakages with reverse engineering the neural networks architecture as the main goal. We position our results in the following sections in more detail. To summarize, our main motivation comes from the ever more pervasive use of neural networks in security-critical applications and the fact that the architectures are becoming proprietary knowledge for the security evaluation industry. Hence, reverse engineering a neural network has become a new target for the adversaries and we need a better understanding of the vulnerabilities to side-channel leakages in those cases to be able to protect the users' rights and data.

1.1 Related Work

There are many papers considering machine learning and more recently, deep learning for improving the effectiveness of side-channel attacks. For instance, a number of works have compared the effectiveness of classical profiled side-channel attacks, so-called template attacks, against various machine learning techniques [30, 19]. Lately, several works explored the power of deep learning in the context of side-channel analysis [32]. However, this line of work is using machine learning to derive a new side-channel distinguisher, i.e., the selection function leading to the key recovery.

On the other hand, using side-channel analysis to attack machine learning architectures has been much less investigated. Shokri et al. investigate the leakage of sensitive information from machine learning models about individual

data records on which they were trained [44]. They show that such models are vulnerable to membership inference attacks and they also evaluate some mitigation strategies. Song et al. show how a machine learning model from a malicious machine learning provider can be used to obtain information about the training set of a model [45]. Hua et al. were first to reverse engineer two convolutional neural networks, namely AlexNet and SqueezeNet through memory and timing side-channel leaks [17]. The authors measure side-channel through an artificially introduced hardware trojan. They also need access to the original training data set for the attack, which might not always be available. Lastly, in order to obtain the weights of neural networks, they attack a very specific operation, i.e., zero pruning [40]. Wei et al. have also performed an attack on an FPGA-based convolutional neural network accelerator [52]. They recovered the input image from the collected power consumption traces. The proposed attack exploits a specific design choice, i.e., the line buffer in a convolution layer of a CNN.

In a nutshell, both previous reverse engineering efforts using side-channel information were performed on very special designs of neural networks and the attacks had very specific and different goals. Our work is more generic than those two as it assumes just a passive adversary able to measure physical leakages and our strategy remains valid for a range of architectures and devices. Although we show the results on the chips that were depackaged prior to experiments in order to demonstrate the leakage available to powerful adversaries, our findings remain valid even without depackaging. Basically, having EM as an available source of side-channel leakage, it comes down to using properly designed antennas and more advanced setups, which is beyond the scope of this work.

Several other works doing somewhat related research are given as follows. Ohrimenko et al. used a secure implementation of MapReduce jobs and analyzed intermediate traffic between reducers and mappers [37]. They showed how an adversary observing the runs of typical jobs can infer precise information about the inputs. In a follow-up work they discuss how machine learning algorithms can be exploited by various side-channels [38]. Consequently, they propose data-oblivious machine learning algorithms that prevent exploitation of side channels induced by memory, disk, and network accesses. They note that side-channel attacks based on power and timing leakages are out of the scope of their work. Xu et al. introduced controlled-channel attacks, which is a type of side-channel attack allowing an untrusted operating system to extract large amounts of sensitive information from protected applications [54]. Wang and Gong investigated both theoretically and experimentally how to steal hyper-parameters of machine learning algorithms [51]. In order to mount the attack in practice, they estimate the error between the true hyper-parameter and the estimated one.

In this work, we further explore the problem of reverse engineering of neural networks from a more generic perspective. The closest previous works to ours have reverse engineered neural networks by using cache attacks that work on distinct CPUs and are basically micro-architectural attacks (albeit using timing side-channel). Our approach utilizes EM side-channel on small embedded devices and it is supported by practical results obtained on a real-world architecture. Finally, our attack is able to recover both the hyper-parameters (parameter external to the model, e.g., the number of layers) and parameters (parameter internal to the model, like weights) of neural networks.

1.2 Contribution and Organization

The main contributions of this paper are:

1. We describe full reverse engineering of neural network parameters based on side-channel analysis. We are able to recover the key parameters such as activation function, pre-trained weights, number of hidden layers and neurons in each layer. The proposed technique does not need any information on the (sensitive) training data as that information is often not even available to the attacker. We emphasize that, for our attack to work, we require the knowledge of some inputs/outputs and side-channel measurements, which is a standard assumption for side-channel attacks.
2. All the proposed attacks are practically implemented and demonstrated on two distinct microcontrollers (i.e., 8-bit AVR and 32-bit ARM).
3. We highlight some interesting aspects of side-channel attacks when dealing with real numbers, unlike in everyday cryptography. For example, we show that even a side-channel attack that failed can provide sensitive information about the target due to the precision error.
4. Finally, we propose a number of mitigation techniques rendering the attacks more difficult.

We emphasize that the simplicity of our attack is its strongest point, as it minimizes the assumption on the adversary (no pre-processing, chosen-plaintext messages, etc.)

2 Background

In this section, we give details about artificial neural networks we consider in this paper and their building blocks. Next, we discuss the concepts of side-channel analysis and several types of attacks we use in this paper.

2.1 Artificial Neural Networks

Artificial neural networks (ANNs) is an umbrella notion for all computer systems loosely inspired by biological neural networks. Such systems are able to "learn" from examples, which makes them a strong (and very popular) paradigm in

the machine learning domain. Any ANN is built from a number of nodes called artificial neurons. The nodes are connected in order to transmit a signal. Usually, in an ANN, the signal at the connection between artificial neurons is a real number and the output of each neuron is calculated as a nonlinear function of the sum of its inputs. Neurons and connections have weights that are adjusted as the learning progresses. Those weights are used to increase or decrease the strength of a signal at a connection. In the rest of this paper, we use the notions of an artificial neural network, neural network, and network interchangeably.

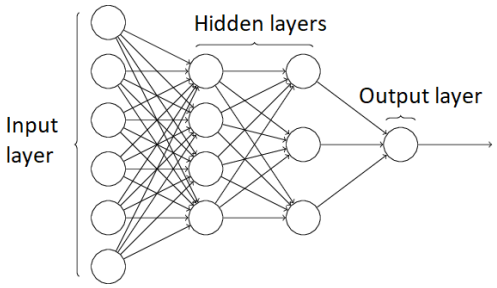


Figure 1: Multilayer perceptron.

2.1.1 Multilayer Perceptron

A very simple type of a neural network is called perceptron. A perceptron is a linear binary classifier applied to the feature vector as a function that decides whether or not an input belongs to some specific class. Each vector component has an associated weight w_i and each perceptron has a threshold value θ. The output of a perceptron equals "1" if the direct sum between the feature vector and the weight vector is larger than zero and "-1" otherwise. A perceptron classifier works only for data that are linearly separable, i.e., if there is some hyperplane that separates all the positive points from all the negative points [34].

By adding more layers to a perceptron, we obtain a multilayer perceptron algorithm. Multilayer perceptron (MLP) is a feed-forward neural network that maps sets of inputs onto sets of appropriate outputs. It consists of multiple layers of nodes in a directed graph, where each layer is fully connected to the next one. Consequently, each node in one layer connects with a certain weight w to every node in the following layer. Multilayer perceptron algorithm consists of at least three layers: one input layer, one output layer, and one hidden layer. Those layers must consist of nonlinearly activating nodes [7]. We depict a model of a multilayer perceptron in Figure 1. Note, if there is more than one hidden layer, then it can be considered a deep learning architecture. Differing from linear perceptron, MLP can distinguish data that are not linearly separable. To train the network, the backpropagation algorithm is used, which is a generalization of the least mean squares algorithm in the linear perceptron. Backpropagation is used by the gradient descent optimization algorithm to adjust the weight of neurons by calculating the gradient of the loss function [34].

2.1.2 Convolutional Neural Network

CNNs represent a type of neural networks which were first designed for 2-dimensional convolutions as it was inspired by the biological processes of animals' visual cortex [28]. From the operational perspective, CNNs are similar to ordinary neural networks (e.g., multilayer perceptron): they consist of a number of layers where each layer is made up of neurons. CNNs use three main types of layers: convolutional layers, pooling layers, and fully-connected layers. Convolutional layers are linear layers that share weights across space. Pooling layers are non-linear layers that reduce the spatial size in order to limit the number of neurons. Fully-connected layers are layers where every neuron is connected with all the neurons in the neighborhood layer. For additional information about CNNs, we refer interested readers to [12].

2.1.3 Activation Functions

An activation function of a node is a function f defining the output of a node given an input or set of inputs, see Eq. (1). To enable calculations of nontrivial functions for ANN using a small number of nodes, one needs nonlinear activation functions as follows.

$$y = Activation(\sum(weight \cdot input) + bias). \tag{1}$$

In this paper, we consider the logistic (sigmoid) function, tanh function, softmax function, and Rectified Linear Unit (ReLU) function. The logistic function is a nonlinear function giving smooth and continuously differentiable results [14]. The range of a logistic function is $[0,1]$, which means that all the values going to the next neuron will have the same sign.

$$f(x) = \frac{1}{1+e^{-x}}. \tag{2}$$

The tanh function is a scaled version of the logistic function where the main difference is that it is symmetric over the origin. The tanh function ranges in $[-1,1]$.

$$f(x) = tanh(x) = \frac{2}{1+e^{-2x}} - 1. \tag{3}$$

The softmax function is a type of sigmoid function able to map values into multiple outputs (e.g., classes). The softmax function is ideally used in the output layer of the classifier in order to obtain the probabilities defining a class for each input [5]. To denote a vector, we represent it in bold style.

$$f(\mathbf{x})_j = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}},\ for\ j = 1,\ldots,K. \tag{4}$$

The Rectified Linear Unit (ReLU) is a nonlinear function that is differing from the previous two activation functions as it does not activate all the neurons at the same time [35]. By activating only a subset of neurons at any time, we make the network sparse and easier to compute [2]. Consequently, such properties make ReLU probably the most widely used activation function in ANNs today.

$$f(x) = max(0, x). \tag{5}$$

2.2 Side-channel Analysis

Side-channel Analysis (SCA) exploits weaknesses on the implementation level [33]. More specifically, all computations running on a certain platform result in unintentional physical leakages as a sort of physical signatures from the reaction time, power consumption, and Electromagnetic (EM) emanations released while the device is manipulating data. SCA exploits those physical signatures aiming at the key (secret data) recovery. In its basic form, SCA was proposed to perform key recovery attacks on the implementation of cryptography [23, 22]. One advantage of SCA over traditional cryptanalysis is that SCA can apply a divide-and-conquer approach. This means that SCA is typically recovering small parts of the key (sub-keys) one by one, which is reducing the attack complexity.

Based on the analysis technique used, different variants of SCA are known. In the following, we recall a few techniques used later in the paper. Although the original terms suggest power consumption as the source of leakage, the techniques apply to other side channels as well. In particular, in this work, we are using the EM side channel and the corresponding terms are adapted to reflect this.

Simple Power (or Electromagnetic) Analysis (SPA or SEMA). Simple power (or EM) analysis, as the name suggests, is the most basic form of SCA [22]. It targets information from the sensitive computation that can be recovered from a single or a few traces. As a common example, SPA can be used against a straightforward implementation of the RSA algorithm to distinguish square from multiply operation, leading to the key recovery. In this work, we apply SPA, or actually SEMA to reverse engineer the architecture of the neural network.

Differential Power (or Electromagnetic) Analysis (DPA or DEMA). DPA or DEMA is an advanced form of SCA, which applies statistical techniques to recover secret information from physical signatures. The attack normally tests for dependencies between actual physical signature (or measurements) and hypothetical physical signature, i.e., predictions on intermediate data. The hypothetical signature is based on a leakage model and key hypothesis. Small parts of the secret key (e.g., one byte) can be tested independently. The knowledge of the leakage model comes from the adversary's intuition and expertise. Some commonly used leakage models for representative devices are the Hamming weight for microcontrollers and the Hamming distance in FPGA, ASIC, and GPU [4, 31] platforms. As the measurements can be noisy, the adversary often needs many measurements, sometimes millions. Next, statistical tests like correlation [6] are applied to distinguish the correct key hypothesis from other wrong guesses. In the following, DPA (DEMA) is used to recover secret weights from a pre-trained network.

3 Side-channel Based Reverse Engineering of Neural Networks

In this section, we discuss the threat model we use, the experimental setup and reverse engineering of various elements of neural networks.

3.1 Threat Model

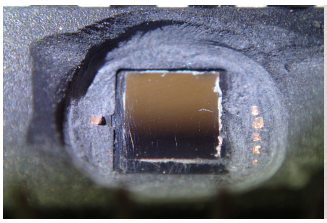
The main goal of this work is to recover the neural network architecture using only side-channel information.

Scenario. We select to work with MLP and CNNs since: 1) they are commonly used machine learning algorithms in modern applications, see e.g., [16, 11, 36, 48, 25, 21]; 2) they consist of different types of layers that are also occurring in other architectures like recurrent neural networks; and 3) in the case of MLP, the layers are all identical, which makes it more difficult for SCA and could be consequently considered as the worst-case scenario.

We choose our attack to be as generic as possible. For instance, we have no assumption on the type of inputs or its source, as we work with real numbers. If the inputs are in the form of integers (like the MNIST database), the attack becomes easier, since we would not need to recover mantissa bytes and deal with precision. We also assume that the implementation of the machine learning algorithm does not include any side-channel countermeasures.

Attacker's capability. The attacker in consideration is a passive one. This implies him/her acquiring measurements of the device while operating "normally" and not interfering with its internal operations by evoking faulty computations and behavior by e.g., glitching the device, etc. More in details, we consider the following setting:

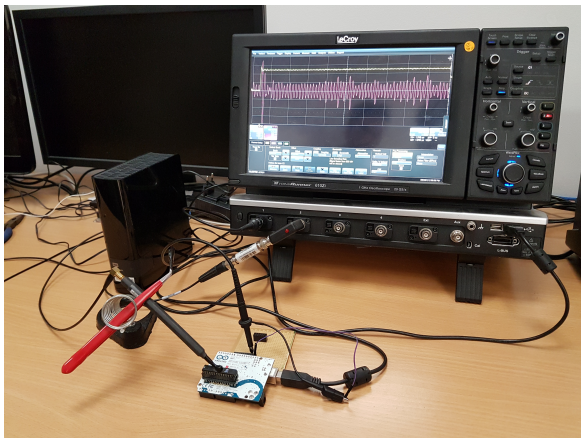
1. **Attacker does not know the architecture of the used network but can feed random (and hence known) inputs to the architecture. We note that the attacks and analysis presented in our work do not rely on any assumptions on the distributions of the inputs, although a common assumption in SCA is that they are chosen uniformly at random.** Basically, we assume that the attacker has physical access to the device (can be remote, via EM signals) and he/she knows that the device runs some neural net. The attacker only controls the execution of it through selecting the inputs, but



(a) Target 8-bit microcontroller mechanically decapsulated



(b) Langer RF-U 5-2 Near Field Electromagnetic passive Probe



(c) The complete measurement setup

Figure 2: Experimental Setup

he/she can observe the outputs and side-channel information (but not individual intermediate values). The attack scenario is often referred to as *known-plaintext attack*. An adequate use case would be when the attacker legally acquires a copy of the network with API access to it and aims at recovering its internal details e.g. for IP theft.

2. **Attacker is capable of measuring side-channel information leaked from the implementation of the targeted architecture.** The attacker can collect multiple side-channel measurements while processing the data and use different side-channel techniques for her analysis. In this work, we focus on timing and EM side channels.

3.2 Experimental Setup

Here we describe the attack methodology, which is first validated on Atmel ATmega328P. Later, we also demonstrate the proposed methodology on ARM Cortex-M3.

The side-channel activity is captured using the Lecroy WaveRunner 610zi oscilloscope. For each known input, the attacker gets one measurement (or trace) from the oscilloscope. In the following, nr. of inputs or nr. of traces are used interchangeably. Each measurement is composed of many samples (or points). The number of samples (or length of the trace) depends on sampling frequency and execution time. As shown later, depending on the target, nr. of samples can vary from thousands (for multiplication) to millions (for a whole CNN network). The measurements are synchronized with the operations by common handshaking signals like start and stop of computation. To further improve the quality of measurements, we opened the chip package mechanically (see Figure 2a). An RF-U 5-2 near-field electromagnetic (EM) probe from Langer is used to collect the EM measurements (see Figure 2b). The setup is depicted in Figure 2c. We use the probe as an antenna for spying on the EM side-channel leakage from the underlying processor running ML. Note that EM measurements also allow to observe the timing of all the operations and thus the setup allows for timing side-channels analysis as well. Our choice of the target platform is motivated by the following considerations:

- Atmel ATmega328P: This processor typically allows for high quality measurements. We are able to achieve a high signal-to-noise ratio (SNR) measurements, making this a perfect tuning phase to develop the methodology of our attacks.
- ARM Cortex-M3: This is a modern 32-bit microcontroller architecture featuring multiple stages of the pipeline, on-chip co-processors, low SNR measurements, and wide application. We show that the developed methodology is indeed versatile across targets with a relevant update of measurement capability.

In addition, real-world use cases also justify our platforms of choice. Similar micro-controllers are often used in wearables like Fitbit (ARM Cortex-M4), several hardware crypto wallets, smart home devices, etc. Additionally, SCA on a GPU or an FPGA platform is practically demonstrated in several instances, thus our methodology can be directly adapted for those cases as well. For different platforms, the leakage model could change, but this would not limit our approach and methodology. In fact, adequate leakage models are known for platforms like FPGA [4] and GPU [31]. Moreover, as for ARM Cortex-M3, low SNR of the measurement might force the adversary to increase the number of measurements and apply signal pre-processing techniques, but the main principles behind the analysis remain valid.

As already stated above, the exploited leakage model of the target device is the Hamming weight (HW) model. A microcontroller loads sensitive data to a data bus to perform indicated instructions. This data bus is pre-charged to all '0's' before every instruction. Note that data bus being pre-charged is a natural behavior of microcontrollers and not a vulnerability introduced by the attacker. Thus, the power consumption (or EM radiation) assigned to the value of the data being loaded is modeled as the number of bits equal to '1'. In other words, the power consumption of loading data

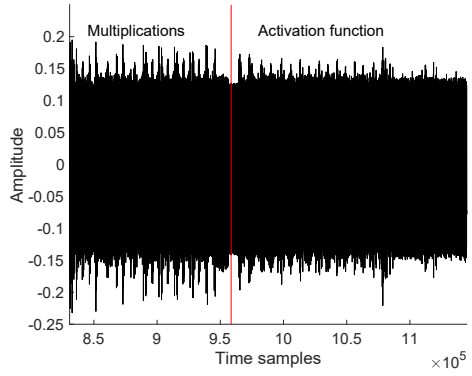


Figure 3: Observing pattern and timing of multiplication and activation function

x is:

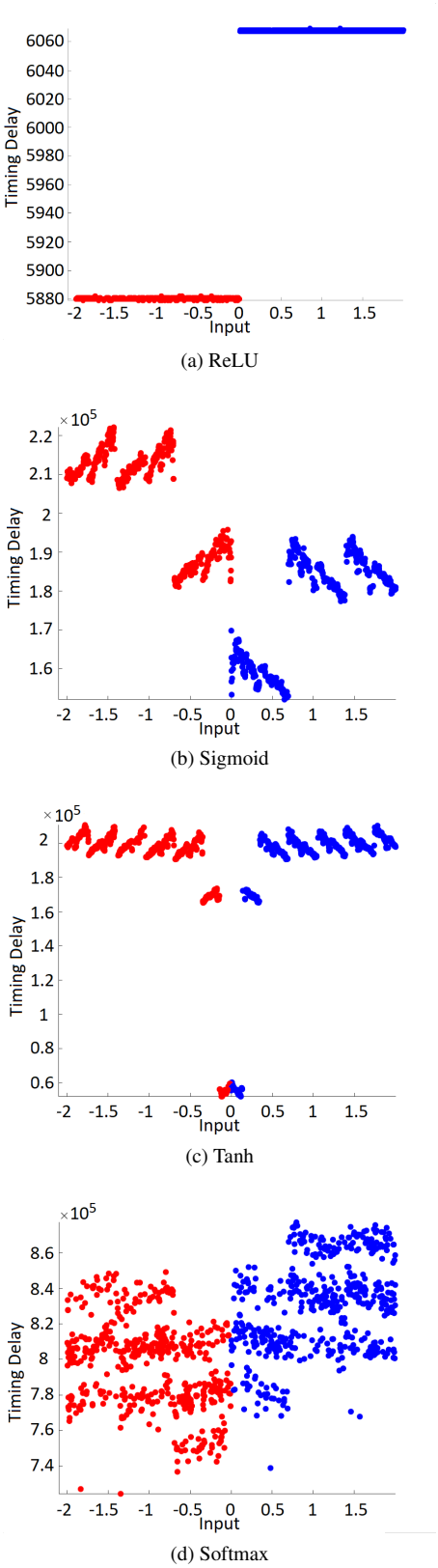
$$HW(x) = \sum_{i=1}^{n} x_i \ , \tag{6}$$

where x_i represents the ith bit of x. In our case, it is the secret pre-trained weight which is regularly loaded from memory for processing and results in the HW leakage. To conduct the side-channel analysis, we perform the divide-and-conquer approach, where we target each operation separately. The full recovery process is described in Section 3.6.

Several pre-trained networks are implemented on the board. The training phase is conducted offline, and the trained network is then implemented in C language and compiled on the microcontroller. In these experiments, we consider multilayer perceptron architectures consisting of a different number of layers and nodes in those layers. Note that, with our approach, there is no limit in the number of layers or nodes we can attack, as the attack scales linearly with the size of the network. The methodology is developed to demonstrate that the key parameters of the network, namely the weights and activation functions can be reverse engineered. Further experiments are conducted on deep neural networks with three hidden layers but the method remains valid for larger networks as well.

3.3 Reverse Engineering the Activation Function

We remind the reader that nonlinear activation functions are necessary in order to represent nonlinear functions with a small number of nodes in a network. As such, they are elements used in virtually any neural network architecture today [25, 15]. If the attacker is able to deduce the information on the type of used activation functions, he/she can use that knowledge together with information about input values to deduce the behavior of the whole network.



(a) ReLU

(b) Sigmoid

(c) Tanh

(d) Softmax

Figure 4: Timing behavior for different activation functions

Table 1: Minimum, Maximum, and Mean computation time (in *ns*) for different activation functions

Activation Function	Minimum	Maximum	Mean
ReLU	5 879	6 069	5 975
Sigmoid	152 155	222 102	189 144
Tanh	51 909	210 663	184 864
Softmax	724 366	877 194	813 712

We analyze the side-channel leakage from different activation functions. We consider the most commonly used activation functions, namely ReLU, sigmoid, tanh, and softmax [14, 35]. The timing behavior can be observed directly on the EM trace. For instance, as shown later in Figure 8a, a multiplication is followed by activation with individual signatures. For a similar architecture, we test different variants with each activation function. We collect EM traces and measure the timing of the activation function computation from the measurements. The measurements are taken when the network is processing random inputs in the range, i.e., $x \in \{-2,2\}$. A total of 2000 EM measurements are captured for each activation function. As shown in Figure 3, the timing behavior of the four tested activation functions have distinct signatures allowing easy characterization.

Different inputs result in different processing times. Moreover, the timing behavior for the same inputs largely varies depending on the activation function. For example, we can observe that ReLU will require the shortest amount of time, due to its simplicity (see Figure 4a). On the other hand, tanh and sigmoid might have similar timing delays, but with different pattern considering the input (see Figure 4b and Figure 4b), where tanh is more symmetric in pattern compared to sigmoid, for both positive and negative inputs. We can observe that softmax function will require most of the processing time, since it requires the exponentiation operation which also depends on the number of neurons in the output layer. As neural network algorithms are often optimized for performance, the presence of such timing side-channels is often ignored. A function such as tanh or sigmoid requires computation of e^x and division and it is known that such functions are difficult to implement in constant time. In addition, constant time implementations might lead to substantial performance degradation. Other activation functions can be characterized similarly. Table 1 presents the minimum, maximum, and mean computation time for each activation function over 2000 captured measurements. While ReLU is the fastest one, the timing difference for other functions stands out sufficiently, to allow for a straightforward recovery. To distinguish them, one can also do some pattern matching to determine which type of function is used, if necessary. Note, although Sigmoid and Tanh have similar Maximum and mean values, the Minimum value differs significantly. Moreover, the attacker can sometimes pre-characterize (or profile) the timing behavior of the target activation function independently for better precision, especially when common libraries are used for standard functions like multiplication, activation function, etc.

3.4 Reverse Engineering the Multiplication Operation

A well-trained network can be of significant value. Main distinguishing factors for a well trained network against a poorly trained one, for a given architecture, are the weights. With fine-tuned weights, we can improve the accuracy of the network. In the following, we demonstrate a way to recover those weights by using SCA.

For the recovery of the weights, we use the Correlation Power Analysis (CPA) i.e., a variant of DPA using the Pearson's correlation as a statistical test.[1] CPA targets the multiplication $m = x \cdot w$ of a known input x with a secret weight w. Using the HW model, the adversary correlates the activity of the predicted output m for all hypothesis of the weight. Thus, the attack computes $\rho(t,w)$, for all hypothesis of the weight w, where ρ is the Pearson correlation coefficient and t is the side-channel measurement. The correct value of the weight w will result in a higher correlation standing out from all other wrong hypotheses w^*, given enough measurements. Although the attack concept is the same as when attacking cryptographic algorithms, the actual attack used here is quite different. Namely, while cryptographic operations are always performed on fixed length integers, in ANN we are dealing with real numbers.

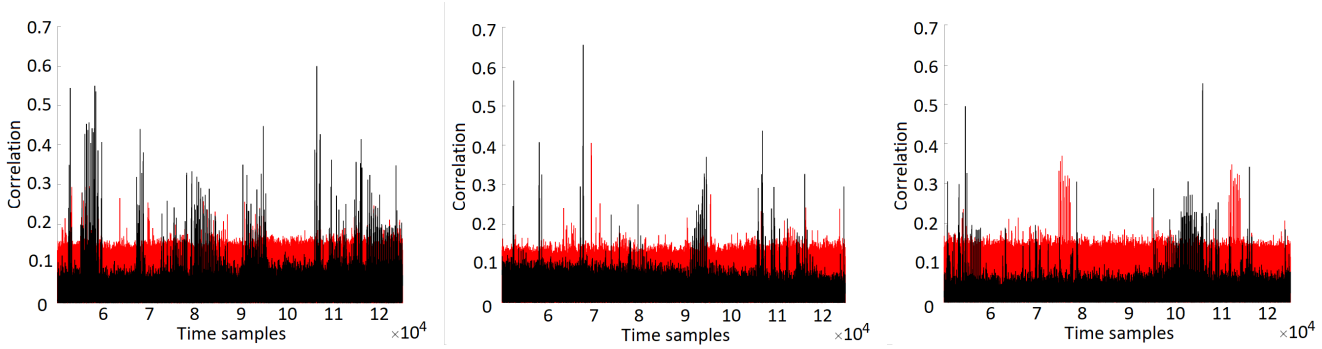
We start by analyzing the way the compiler is handling floating-point operations for our target. The generated assembly is shown in Table 2, which confirms the usage of IEEE 754 compatible representation as stated above. The knowledge of the representation allows one to better estimate the leakage behavior. Since the target device is an 8-bit microcontroller, the representation follows a 32-bit pattern $(b_{31}...b_0)$, being stored in 4 registers. The 32-bit consist of: 1 sign bit (b_{31}), 8 biased exponent bits $(b_{30}...b_{23})$ and 23 mantissa (fractional) bits $(b_{22}...b_0)$. It can be formulated as:

$$(-1)^{b_{31}} \times 2^{(b_{30}...b_{23})_2 - 127} \times (1.b_{22}...b_0)_2.$$

For example, the value 2.43 can be expressed as $(-1)^0 \times 2^{(1000000)_2 - 127} \times (1.00110111000010100011111)_2$. The measurement t is considered when the computed result m is stored back to the memory, leaking in the HW model i.e., $HW(m)$. Since 32-bit m is split into individual 8-bits, each byte of m is recovered individually. Hence, by recovering this representation, it is enough to recover the estimation of the real number value.

To implement the attack two different approaches can be considered. The first approach is to build the hypothesis on

[1] It is called CEMA in case of EM side channel.



(a) First byte mantissa for weight = 2.43 (b) Second byte mantissa for weight = 2.43 (c) Third byte mantissa for weight = 2.43

Figure 5: Correlation of different weights candidate on multiplication operation

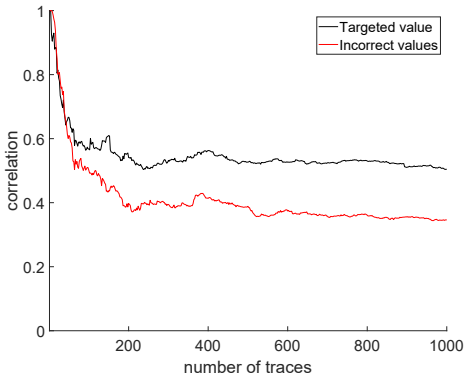
Table 2: Code snippet of the returned assembly for multiplication: $x = x \cdot w (= 2.36$ or 0x3D0A1740 in IEEE 754 representation). The multiplication itself is not shown here, but from the registers assignment, our leakage model assumption holds.

#	Instruction	Comment
11a	ldd r22, Y+1	0x01
11c	ldd r23, Y+2	0x02
11e	ldd r24, Y+3	0x03
120	ldd r25, Y+4	0x04
122	ldi r18, 0x3D	61
124	ldi r19, 0x0A	10
126	ldi r20, 0x17	23
128	ldi r21, 0x40	64
12a	call 0xa0a	multiplication
12e	std Y+1, r22	0x01
130	std Y+2, r23	0x02
132	std Y+3, r24	0x03
134	std Y+4, r25	0x04

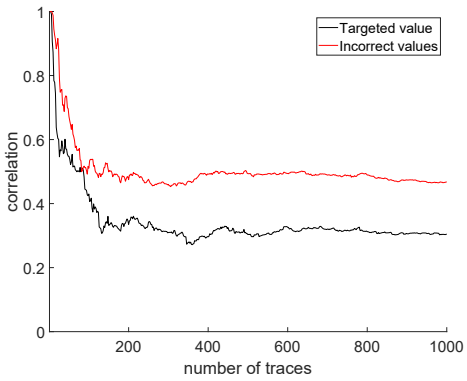
the weight directly. For this experiment, we target the result of the multiplication m of known input values x and unknown weight w. For every input, we assume different possibilities for weight values. We then perform the multiplication and estimate the IEEE 754 binary representation of the output. To deal with the growing number of possible candidates for the unknown weight w, we assume that the weight will be bounded in a range $[-N,N]$, where N is a parameter chosen by the adversary, and the size of possible candidates is denoted as $s = 2N/p$, where p is the precision when dealing with floating-point numbers.

Then, we perform the recovery of the 23-bit mantissa of the weight. The sign and exponent could be recovered separately. Thus, we are observing the leakage of 3 registers, and based on the best CPA results for each register, we can reconstruct the mantissa. Note that the recovered mantissa does not directly relate to the weight, but with the recovery of the sign and exponent, we could obtain the unique weight value. The traces are measured when the microcontroller performs secret weight multiplication with uniformly random values between -1 and 1 ($x \in \{-1,1\}$) to emulate normalized input values. We set $N = 5$ and to reduce the number of possible candidates, we assume that each floating-point value will have a precision of 2 decimal points, $p = 0.01$. Since we are dealing with mantissa only, we can then only check the weight candidates in the range $[0,N]$, thus reducing the number of possible candidates. We note here that this range $[-5,5]$ is based on the previous experiments with MLP. Although, in the later phase of the experiment, we targeted the floating point and fixed-point representation (2^{32} in the worst case scenario on a 32-bit microcontroller, but could be less if the value is for example normalized), instead of the real value, which could in principle cover all possible floating values.

In Figure 5, we show the result of the correlation for each byte with the measured traces. The horizontal axis shows the time of execution and vertical axis correlation. The experiments were conducted on 1 000 traces for each case. In the figure, the black plot denotes the correlation of the "correct" mantissa weight ($|m(\hat{w}) - m(w)| < 0.01$), whereas the red plots are from all other weight candidates in the range described earlier. Since we are only attacking mantissa in this phase, several weight candidates might have similar correlation peaks. After the recovery of the mantissa, the sign bit and exponent can be recovered similarly, which narrows down the list candidate to a unique weight. Another observation is that the correlation value is not very high and scattered across different clock cycles. This is due to the reason that the measurements are noisy and since the operation is not constant-time, the interesting time samples are distributed across multiple clock cycles. Nevertheless, it is shown that the side-channel leakage can be exploited to recover the weight up to certain precision. Multivariate side channel analysis [42] can be considered if distributed samples hinder recovery.

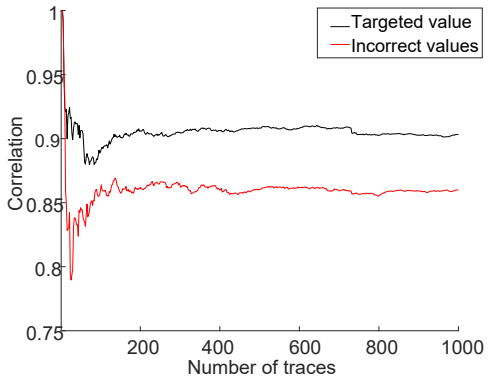


(a) weight = 1.635

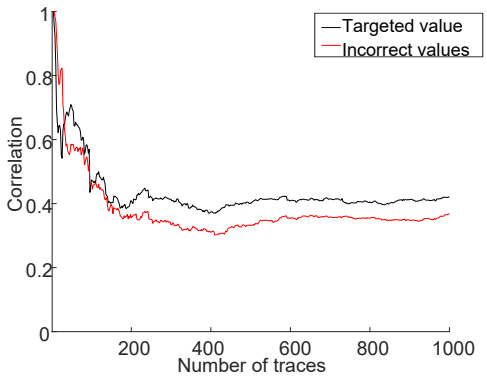


(b) weight = 0.890

Figure 6: Correlation comparison between the correct and incorrect mantissa of the weights. (a) Correct mantissa can be recovered (correct values/black line has a higher value compared to max incorrect values/red line). (b) A special case where the incorrect value of mantissa has a higher correlation, recovering 0.896025 (1100100000..00) instead of 0.89 (1100011110...10), still within precision error limits resulting in attack success



(a) First byte recovery (sign and 7-bit exponent)



(b) Second byte recovery (lsb exponent and mantissa)

Figure 7: Recovery of the weight

We emphasize that attacking real numbers as in the case of weights of ANN can be easier than attacking cryptographic implementations. This is because cryptography typically works on fixed-length integers and exact values must be recovered. When attacking real numbers, small precision errors due to rounding off the intermediate values still result in useful information.

To deal with more precise values, we can target the mantissa multiplication operation directly. In this case, the search space can either be $[0, 2^{23} - 1]$ to cover all possible values for the mantissa (hence, more computational resources will be required) or we can focus only on the most significant bits of the mantissa (lesser candidates but also with lesser precision). Since the 7 most significant bits of the mantissa are processed in the same register, we can aim to target only those bits, assigning the rest to 0. Thus, our search space is now $[0, 2^7 - 1]$. The mantissa multiplication can be performed as $1.mantissa_x \times 1.mantissa_w$, then taking the 23 most significant bits after the leading 1, and normalization (updating the exponent if the result overflows) if necessary.

In Figure 6, we show the result of the correlation between the HW of the first 7-bit mantissa of the weight with the traces. Except for Figure 6b, the other results show that the correct mantissa can be recovered. Although the correlation is not increasing, it is important that the difference becomes stable after a sufficient amount of traces is used and eventually distinguishing correct weight from wrong weight hypotheses. The most interesting result is shown in Figure 6b, which at first glance looks like a failure of the attack. Here, the target value of the mantissa is **1100011**110...10, while the attack recovers **1100100**000..00. Considering the sign and exponents, the attack recovers *0.890625* instead of *0.89*, i.e., a precision error at 4th place after decimal point. Thus, in both cases, we have shown that we can recover the weights from the SCA leakage.

In Figure 7, we show the composite recovery of 2 bytes of the weight representation i.e., a low precision setting where

we recover sign, exponent, and most significant part of mantissa. Again, the targeted (correct) weight can be easily distinguished from the other candidates. Hence, once all the necessary information has been recovered, the weight can be reconstructed accordingly.

3.5 Reverse Engineering the Number of Neurons and Layers

After the recovery of the weights and the activation functions, now we use SCA to determine the structure of the network. Mainly, we are interested to see if we can recover the number of hidden layers and the number of neurons for each layer. To perform the reverse engineering of the network structure, we first use SPA (SEMA). SPA is the simplest form of SCA which allows information recovery in a single (or a few) traces with methods as simple as a visual inspection. The analysis is performed on three networks with different layouts.

The first analyzed network is an MLP with one hidden layer with 6 neurons. The EM trace corresponding to the processing of a randomly chosen input is shown in Figure 8a. By looking at the EM trace, the number of neurons can be easily counted. The observability arises from the fact that multiplication operation and the activation function (in this case, it is the Sigmoid function) have completely different leakage signatures. Similarly, the structures of deeper networks are also shown in Figure 8b and Figure 8c. The recovery of output layer then provides information on the number of output classes. However, distinguishing different layers might be difficult, since the leakage pattern is only dependent on multiplication and activation function, which are usually present in most of the layers. We observe minor features allowing identification of layer boundaries but only with low confidence. Hence, we develop a different approach based on CPA to identify layer boundaries.

The experiments follow a similar methodology as in the previous experiments. To determine if the targeted neuron is in the same layer as previously attacked neurons, or in the next layer, we perform a weight recovery using two sets of data.

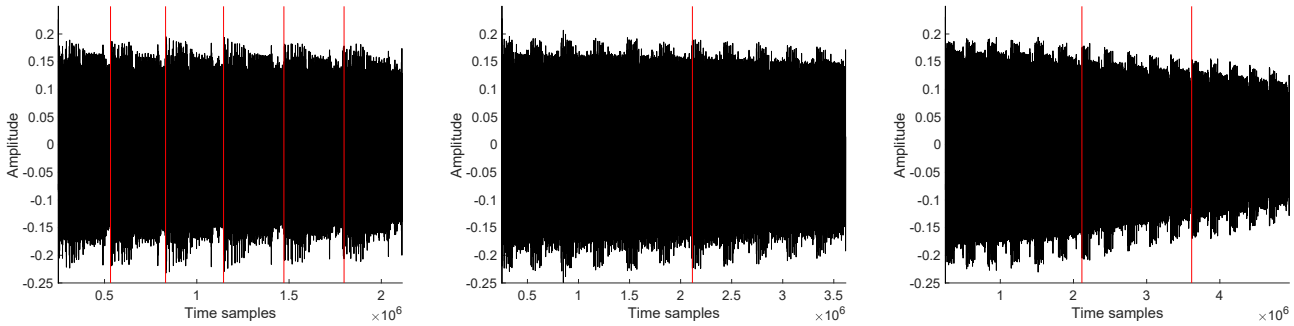
Let us assume that we are targeting the first hidden layer (the same approach can be done on different layers as well). Assume that the input is a vector of length N_0, so the input x can be represented $x = \{x_1, ..., x_{N_0}\}$. For the targeted neuron y_n in the hidden layer, perform the weight recovery on 2 different hypotheses. For the first hypothesis, assume that the y_n is in the first hidden layer. Perform weight recovery individually using x_i, for $1 \leq i \leq N_0$. For the second hypothesis, assume that y_n is in the next hidden layer (the second hidden layer). Perform weight recovery individually using y_i, for $1 \leq i \leq (n-i)$. For each hypothesis, record the maximum (absolute) correlation value, and compare both. Since the correlation depends on both inputs to the multiplication operation, the incorrect hypothesis will result in a lower correlation value. Thus, this can be used to identify layer boundaries.

3.6 Recovery of the Full Network Layout

The combination of previously developed individual techniques can thereafter result in full reverse engineering of the network. The full network recovery is performed layer by layer, and for each layer, the weights for each neuron have to be recovered one at a time. Let us consider a network consisting of N layers, $L_0, L_1, ..., L_{N-1}$, with L_0 being the input layer and L_{N-1} being the output layer. Reverse engineering is performed with the following steps:

1. The first step is to recover the weight w_{L_0} of each connection from the input layer (L_0) and the first hidden layer (L_1). Since the dimension of the input layer is known, the CPA/CEMA can be performed n_{L_0} times (the size of L_0). The correlation is computed for 2^d hypotheses (d is the number of bits in IEEE 754 representation, normally it is 32 bits, but to simplify, 16 bits can be used with lesser precision for the mantissa). After the weights have been recovered, the output of the sum of multiplication can be calculated. This information provides us with input to the activation function.
2. In order to determine the output of the sum of the multiplications, the number of neurons in the layer must be known. This can be recovered by the combination of SPA/SEMA and DPA/DEMA technique described in the previous subsection (2 times CPA for each weight candidate w, so in total $2n_{L_0}2^d$ CPA required), in parallel with the weight recovery. When all the weights of the first hidden layer are recovered, the following steps are executed.
3. Using the same set of traces, timing patterns for different inputs to the activation function can be built, similar to Figure 4. Timing patterns or average timing can then be compared with the profile of each function to determine the activation function (a comparison can be based on simple statistical tools like correlation, distance metric, etc). Afterward, the output of the activation function can be computed, which provides the input to the next layer.
4. The same steps are repeated in the subsequent layers ($L_1, ..., L_{N-1}$, so in total at most $2Nn_L2^d$, where n_L is $max(n_{L_0}, ..., n_{L_{N-1}})$) until the structure of the full network is recovered.

The whole procedure is depicted in Figure 9. In general, it can be seen that the attack scales linearly with the size of the network. Moreover, the same set of traces can be reused for various steps of the attack and attacking different layers, thus reducing measurement effort.



(a) One hidden layer with 6 neurons (b) 2 hidden layers (6 and 5 neurons each) (c) 3 hidden layers (6,5,5 neurons each)

Figure 8: SEMA on hidden layers

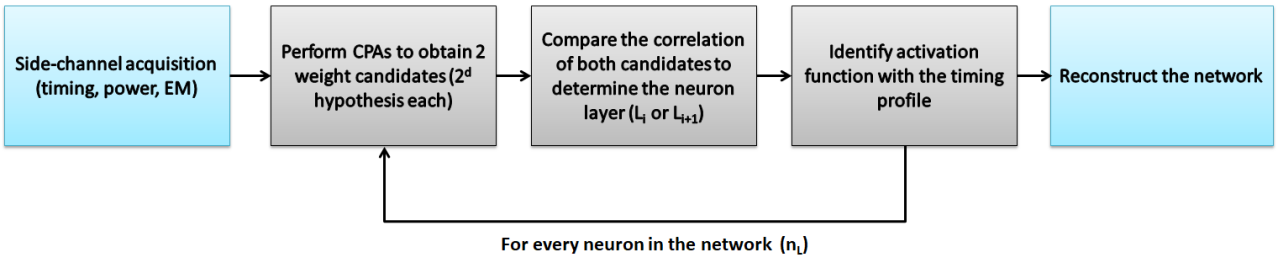


Figure 9: Methodology to reverse engineer the target neural network

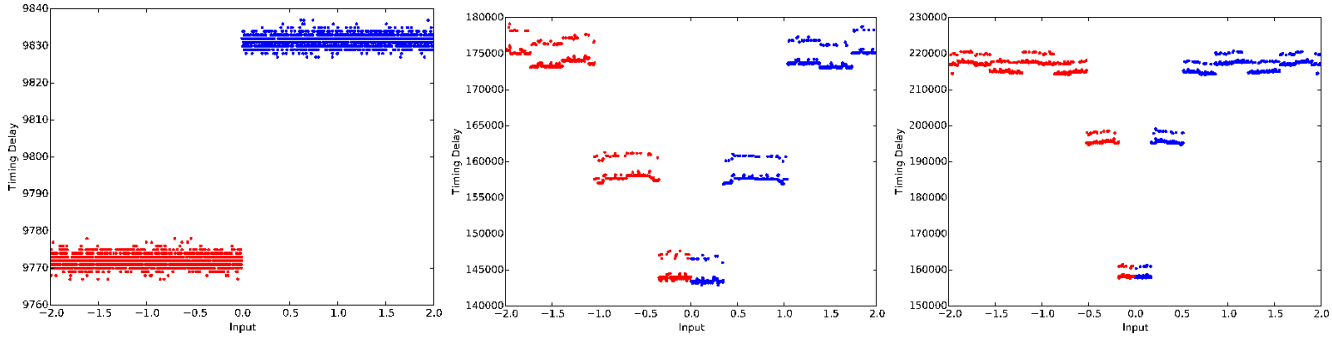
4 Experiments with ARM Cortex-M3

In the previous section, we propose a methodology to reverse engineer sensitive parameters of a neural network, which we practically validated on an 8-bit AVR (Atmel ATmega328P). In this section, we extend the presented attack on a 32-bit ARM microcontroller. ARM microcontrollers form a fair share of the current market with huge dominance in mobile applications, but also seeing rapid adoption in markets like IoT, automotive, virtual and augmented reality, etc. Our target platform is the widely available *Arduino due* development board which contains an *Atmel SAM3X8E ARM Cortex-M3* CPU with a 3-stage pipeline, operating at 84 MHz. The measurement setup is similar to previous experiments (Lecroy WaveRunner 610zi oscilloscope and RF-U 5-2 near-field EM probe from Langer). The point of measurements was determined by a benchmarking code running AES encryption. After capturing the measurements for the target neural network, one can perform reverse engineering. Note that ARM Cortex-M3 (as well as M4 and M7) have support for deep learning in the form of CMSIS-NN implementation [27].

The timing behavior of various activation functions is shown in Figure 10. The results, though different from previous experiments on AVR, have unique timing signatures, allowing identification of each activation function. Here, sigmoid and tanh activation functions have similar minimal computation time but the average and maximum values are higher for tanh function. To distinguish, one can obtain multiple inputs to the function, build patterns and do pattern matching to determine which type of function is used. The activity of a single neuron is shown in Figure 11a, which uses sigmoid as an activation function (the multiplication operation is shown separated by a vertical red line).

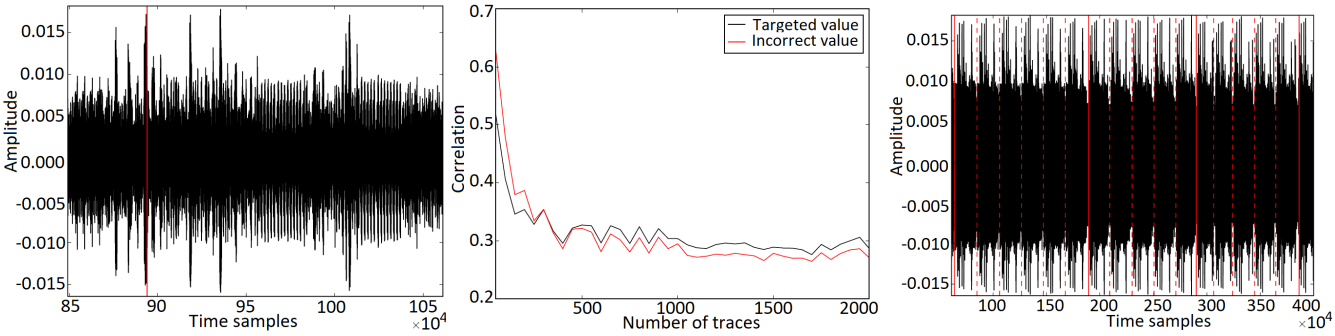
A known input attack is mounted on the multiplication to recover the secret weight. One practical consideration in attacking multiplication is that different compilers will compile it differently for different targets. Modern microcontrollers also have dedicated floating point units for handling operations like multiplication of real numbers. To avoid the discrepancy of the difference of multiplication operation, we target the output of multiplication. In other words, we target the point when multiplication operation with secret weight is completed and the resultant product is updated in general purpose registers or memory. Figure 11b shows the success of attack recovering secret weight of 2.453, with a known input. As stated before, side-channel measurements on modern 32-bit ARM Cortex-M3 may have lower SNR thus making attack slightly harder. Still, the attack is shown to be practical even on ARM with 2× more measurements. In our setup, getting 200 extra measurements takes less than a minute. Similarly, the setup and number of measurements can be updated for other targets like FPGA, GPU, etc.

Finally, the full network layout is recovered. The activity



(a) ReLU (b) Sigmoid (c) Tanh

Figure 10: Timing behavior for different activation functions



(a) Observing pattern and timing of multiplication and activation function (b) Correlation comparison between correct and incorrect mantissa for weight=2.453 (c) SEMA on hidden layers with 3 hidden layers (6,5,5 neurons each)

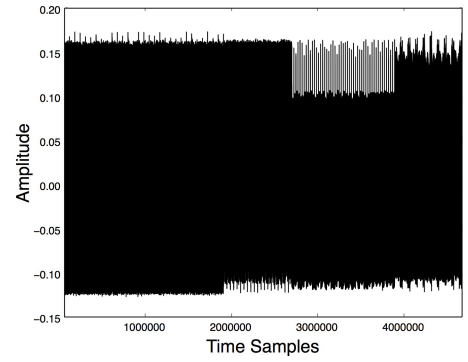
Figure 11: Analysis of an (6,5,5,) neural network

of a full network with 3 hidden layers composed of 6, 5, and 5 neurons each is shown in Figure 11c. All the neurons are observable by visual inspection. The determination of layer boundaries (shown by a solid red line) can be determined by attacking the multiplication operation and following the approach discussed in Section 3.6.

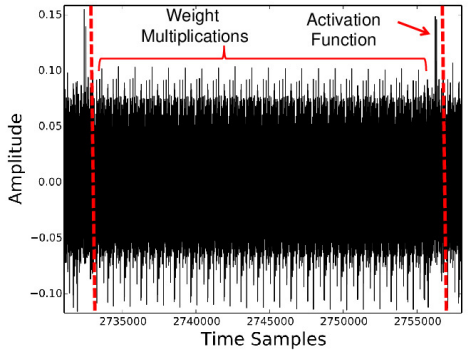
4.1 Reverse Engineering MLP

The migration of our testbed to ARM Cortex-M3 allowed us to test bigger networks, which are used in some relevant case-studies. First, we consider an MLP that is used in profiling side-channel analysis [41]. Our network of choice comes from the domain of side-channel analysis which has seen the adoption of deep learning methods in the past. With this network, a state-of-the-art profiled SCA was conducted when considering several datasets where some even contain implemented countermeasures. Since the certification labs use machine learning to evaluate the resilience of cryptographic implementations to profiled attacks, an attacker being able to reverse engineer that machine learning would be able to use it to attack implementations on his own. The MLP we investigate has 4 hidden layers with dimensions $(50, 30, 20, 50)$, it uses ReLU activation function and has Softmax at the output. The whole measurement trace is shown in Figure 12(a) with a zoom on 1 neurons in the third layer in Figure 12(b). When measuring at 500 $MSamples/s$, each trace had ~ 4.6 million samples. The dataset is DPAcontest v4 with 50 samples and 75 000 measurements [46]. The first 50 000 measurements are used for training and the rest for testing. We experiment with the Hamming weight model (meaning there are 9 output classes). The original accuracy equals 60.9% and the accuracy of the reverse engineered network is 60.87%. While the previously developed techniques are directly available, there are a few practical issues.

- As the average run time is $9.8ms$, each measurement would take long considering the measurement and data saving time. To boost up the SNR, averaging is recommended. We could use the oscilloscope in-built feature for averaging. Overall, the measurement time per trace was slightly over one second after averaging 10 times.
- The measurement period was too big to measure the whole period easily at a reasonable resolution. This was resolved by measuring two consecutive layers at a time



(a)



(b)

Figure 12: (a) Full EM trace of the MLP network from [41], (b) zoom on one neuron in the third hidden layer showing 20 multiplications, followed by a ReLU activation function. 50 such patterns can be seen in (a) identifying third layer in (50,30,20,50) MLP

in independent measurements. It is important to always measure two consecutive layers and not individual layer to determine layer boundaries. This issue otherwise can be solved with a high-end oscilloscope.

- We had to resynchronize traces each time according to the target neuron which is a standard pre-processing in side-channel attacks.

Next, we experiment with an MLP consisting of 4 hidden layers, where each layer has 200 nodes. We use the MNIST database as input to the MLP [29]. The MNIST database contains 60 000 training images and 10 000 testing images where each image has 28×28 pixel size. The number of classes equals 10. The accuracy of the original network is equal to 98.16% while the accuracy of the reverse engineered network equals 98.15%, with an average weight error converging to 0.0025.

We emphasize that both attacks (on DPAcontest v4 and MNIST) were performed following exactly the same procedure as in previous sections leading to a successful recovery of the network parameters. Finally, in accordance with the conclusions that our attack scales linearly with the size of the network, we did not experience additional difficulties when compared to attacking smaller networks.

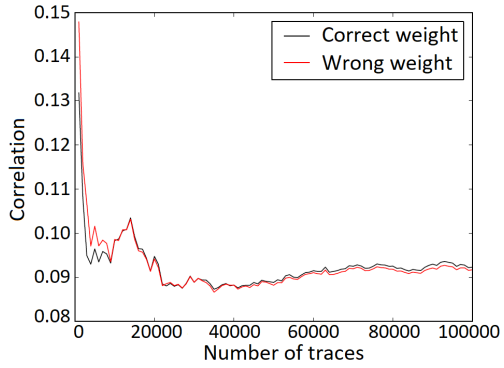
4.2 Reverse Engineering CNN

When considering CNN, the target is the CMSIS-NN implementation [27] on ARM Cortex-M3 with measurement setup same as in previous experiments. Here, as input, we target the CIFAR-10 dataset [24]. This dataset consists of 60 000 32×32 color images in 10 classes. Each class has 6 000 images and there are in total 50 000 training images and 10 000 test images. The CNN we investigate is the same as in [27] and it consists of 3 convolutional layers, 3 max pooling layers, and one fully-connected layer (in total 7 layers).

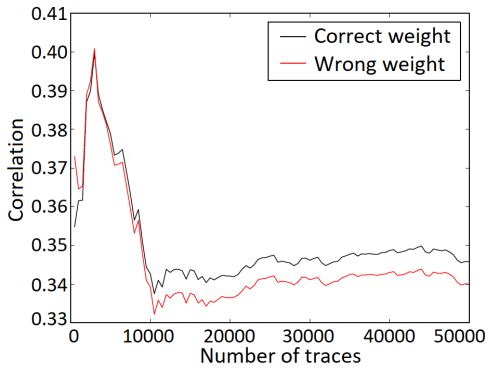
We choose as target the multiplication operation from the input with the weight, similar as in previous experiments. Differing from previous experiments, the operations on real values are here performed using fixed-point arithmetic. Nevertheless, the idea of the attack remains the same. In this example, numbers are stored using 8-bit data type – `int8` (q7). The resulting multiplication is stored in temporary `int` variable. This can also be easily extended to `int16` or `int32` for more precision. Since we are working with integer values, we use the Hamming weight model of the hypothetical outputs (since the Hamming weight model is more straightforward in this case).

If the storing of temporary variable is targeted, as can be seen from Figure 13(a), around 50 000 traces will be required before the correct weight can be distinguished from the wrong weights. This is based on 0.01 precision (the absolute difference from the actual weight in floating number). However, in this case, it can be observed that the correlation value is quite low (~ 0.1). In the case that the conversion to `int8` is performed after the multiplication, this can be also targeted. In Figure 13(b), it can be seen that after 10 000 traces, the correct weight candidate can be distinguished, and the correlation is slightly higher (~ 0.34).

Next, for pooling layer, once the weights in the convolution part are recovered, the output can be calculated. Most CNNs use max pooling layers, which makes it also possible to simply guess the pooling layer type. Still, because the max pooling layer is based on the following conditional instruction, $conditional(if(a > max)max = a)$, it is straightforward to differentiate it from the average pooling that has summation and division operations. This technique is then repeated to reverse engineer any number of convolutional and pooling layers. Finally, the CNN considered here uses ReLU activation function and has one fully-connected layer, which are reverse engineered as discussed in previous sections. In our experiment, the original accuracy of the CNN equals 78.47% and the accuracy of the recovered CNN is 78.11%. As it can be seen, by using sufficient measurements (e.g., $\sim$ 50 000), we are able to reverse engineer CNN architecture as well.



(a) `int` scenario



(b) `int8` scenario

Figure 13: The correlation of correct and wrong weight hypotheses on different number of traces targeting the result of multiplication operation stored as different variable type: (a) `int`, (b) `int8`

5 Mitigation

As demonstrated, various side-channel attacks can be applied to reverse engineer certain components of a pre-trained network. To mitigate such a recovery, several countermeasures can be deployed:

1. Hidden layers of an MLP must be executed in sequence but the multiplication operation in individual neurons within a layer can be executed independently. An example is shuffling [50] as a well-studied side-channel countermeasure. It involves shuffling/permuting the order of execution of independent sub-operations. For example, given N sub-operations $(1,\ldots,N)$ and a random permutation σ, the order of execution becomes $(\sigma(1),\ldots,\sigma(N))$ instead. We propose to shuffle the order of multiplications of individual neurons within a hidden layer during every classification step. Shuffling modifies the time window of operations from one execution to another, mitigating a classical DPA/DEMA attack.
2. Weight recovery can benefit from the application of masking countermeasures [8, 42]. Masking is another widely studied side-channel countermeasure that is even accompanied by a formal proof of security. It involves assuring that sensitive computations are with random values to remove the dependencies between actual data and side-channel signatures, thus preventing the attack. Every computation of $f(x,w)$ is transformed into $f_m(x\oplus m_1, w\oplus m_2) = f(x,w)\oplus m$, where m_1, m_2 are uniformly drawn random masks, and f_m is the masked function which applies mask m at the output of f, given masked inputs $x\oplus m_1$ and $w\oplus m_2$. If each neuron is individually masked with an independently drawn uniformly random mask for every iteration and every neuron, the proposed attacks can be prevented. However, this might result in a substantial performance penalty.
3. The proposed attack on activation functions is possible due to the non-constant timing behavior. Mostly considered activation functions perform exponentiation operation. Implementation of constant time exponentiation has been widely studied in the domain of public key cryptography [13]. Such ideas can be adjusted to implement constant time activation function processing.

Note, the techniques we discuss here represent well-explored methods of protecting against side-channel attacks. As such, they are generic and can be applied to any implementation. Unfortunately, all those countermeasures also come with an area and performance cost. Shuffling and masking require a true random number generator that is typically very expensive in terms of area and performance. Constant time implementations of exponentiation [1] also come at performance efficiency degradation. Thus, the optimal choice of protection mechanism should be done after a systematic resource and performance evaluation study.

6 Further Discussions and Conclusions

Neural networks are widely used machine learning family of algorithms due to its versatility across domains. Their effectiveness depends on the chosen architecture and fine-tuned parameters along with the trained weights, which can be proprietary information. In this work, we practically demonstrate reverse engineering of neural networks using side-channel analysis techniques. Concrete attacks are performed on measured data corresponding to implementations of chosen networks. To make our setting even more general, we do not assume any specific form of the input data (except that inputs are real values).

We conclude that using an appropriate combination of SEMA and DEMA techniques, all sensitive parameters of the network can be recovered. The proposed methodology is demonstrated on two different modern controllers, a classic 8-bit AVR and a 32-bit ARM Cortex-M3 microcontroller. As also shown in this work, the attacks on modern devices are

typically somewhat harder to mount, due to lower SNR for side-channel attacks, but remain practical. In the presented experiments, the attack took twice as many measurements, requiring roughly 20 seconds extra time. Overall, the attack methodology scales linearly with the size of the network. The attack might be easier in some setting where a new network is derived from well known network like VGG-16, Alexnet, etc. by tuning hyper-parameters or transfer learning. In such cases, the side-channel based approach can reveal the remaining secrets. However, analysis of such partial cases is currently out of scope.

The proposed attacks are both generic in nature and more powerful than the previous works in this direction. Finally, suggestions on countermeasures are provided to help the designer mitigate such threats. The proposed countermeasures are borrowed mainly from side-channel literature and can incur huge overheads. Still, we believe that they could motivate further research on optimized and effective countermeasures for neural networks. Besides continuing working on countermeasures, as the main future research goal, we plan to look into more complex CNNs. Naturally, this will require stepping aside from low power ARM devices and using for instance, FPGAs. Additionally, in this work, we considered only feed-forward networks. It would be interesting to extend our work to other types of networks like recurrent neural networks. Since such architectures have many same elements like MLP and CNNs, we believe our attack should be (relatively) easily extendable to such neural networks.

References

[1] AL HASIB, A., AND HAQUE, A. A. M. M. A comparative study of the performance and security issues of AES and RSA cryptography. In *Convergence and Hybrid Information Technology, 2008. ICCIT'08. Third International Conference on* (2008), vol. 2, IEEE, pp. 505–510.

[2] ALBERICIO, J., JUDD, P., HETHERINGTON, T., AAMODT, T., JERGER, N. E., AND MOSHOVOS, A. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)* (June 2016), pp. 1–13.

[3] ATENIESE, G., MANCINI, L. V., SPOGNARDI, A., VILLANI, A., VITALI, D., AND FELICI, G. Hacking Smart Machines with Smarter Ones: How to Extract Meaningful Data from Machine Learning Classifiers. *Int. J. Secur. Netw. 10*, 3 (Sept. 2015), 137–150.

[4] BHASIN, S., GUILLEY, S., HEUSER, A., AND DANGER, J.-L. From cryptography to hardware: analyzing and protecting embedded Xilinx BRAM for cryptographic applications. *Journal of Cryptographic Engineering 3*, 4 (2013), 213–225.

[5] BISHOP, C. M. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[6] BRIER, E., CLAVIER, C., AND OLIVIER, F. Correlation power analysis with a leakage model. In *International Workshop on Cryptographic Hardware and Embedded Systems* (2004), Springer, pp. 16–29.

[7] COLLOBERT, R., AND BENGIO, S. Links Between Perceptrons, MLPs and SVMs. In *Proceedings of the Twenty-first International Conference on Machine Learning* (New York, NY, USA, 2004), ICML '04, ACM, pp. 23–.

[8] CORON, J.-S., AND GOUBIN, L. On boolean and arithmetic masking against differential power analysis. In *International Workshop on Cryptographic Hardware and Embedded Systems* (2000), Springer, pp. 231–237.

[9] DOWLIN, N., GILAD-BACHRACH, R., LAINE, K., LAUTER, K., NAEHRIG, M., AND WERNSING, J. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48* (2016), ICML'16, JMLR.org, pp. 201–210.

[10] FREDRIKSON, M., LANTZ, E., JHA, S., LIN, S., PAGE, D., AND RISTENPART., T. Privacy in Pharmacogenetics: An End-to-End Case Study of Personalized Warfarin Dosing. In *USENIX Security* (2014), pp. 17–32.

[11] GILMORE, R., HANLEY, N., AND O'NEILL, M. Neural network based attack on a masked implementation of AES. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)* (May 2015), pp. 106–111.

[12] GOODFELLOW, I., BENGIO, Y., AND COURVILLE, A. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[13] HACHEZ, G., AND QUISQUATER, J.-J. Montgomery exponentiation with no final subtractions: Improved results. In *International Workshop on Cryptographic Hardware and Embedded Systems* (2000), Springer, pp. 293–301.

[14] HAYKIN, S. *Neural Networks: A Comprehensive Foundation*, 2nd ed. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1998.

[15] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep Residual Learning for Image Recognition. *CoRR abs/1512.03385* (2015).

[16] HEUSER, A., PICEK, S., GUILLEY, S., AND MENTENS, N. Lightweight Ciphers and their Side-channel Resilience. *IEEE Transactions on Computers* (2017), 1–1.

[17] HUA, W., ZHANG, Z., AND SUH, G. E. Reverse Engineering Convolutional Neural Networks Through Side-channel Information Leaks. In *Proceedings of the 55th Annual Design Automation Conference* (New York, NY, USA, 2018), DAC '18, ACM, pp. 4:1–4:6.

[18] ILYAS, A., ENGSTROM, L., ATHALYE, A., AND LIN, J. Black-box Adversarial Attacks with Limited Queries and Information. *CoRR abs/1804.08598* (2018).

[19] JAP, D., STÖTTINGER, M., AND BHASIN, S. Support vector regression: exploiting machine learning techniques for leakage modeling. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy* (2015), ACM, p. 2.

[20] KHAN, A., GOODHUE, G., SHRIVASTAVA, P., VAN DER VEER, B., VARNEY, R., AND NAGARAJ, P. Embedded memory protection, Nov. 22 2011. US Patent 8,065,512.

[21] KOBER, J., AND PETERS, J. *Reinforcement Learning in Robotics: A Survey*, vol. 12. Springer, Berlin, Germany, 2012, pp. 579–610.

[22] KOCHER, P., JAFFE, J., AND JUN, B. Differential power analysis. In *Annual International Cryptology Conference* (1999), Springer, pp. 388–397.

[23] KOCHER, P. C. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference* (1996), Springer, pp. 104–113.

[24] KRIZHEVSKY, A., NAIR, V., AND HINTON, G. CIFAR-10 (Canadian Institute for Advanced Research).

[25] KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1* (USA, 2012), NIPS'12, Curran Associates Inc., pp. 1097–1105.

[26] KUČERA, M., TSANKOV, P., GEHR, T., GUARNIERI, M., AND VECHEV, M. Synthesis of Probabilistic Privacy Enforcement. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2017), CCS '17, ACM, pp. 391–408.

[27] LAI, L., SUDA, N., AND CHANDRA, V. CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs. *CoRR abs/1801.06601* (2018).

[28] LECUN, Y., BENGIO, Y., ET AL. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks 3361*, 10 (1995).

[29] LECUN, Y., AND CORTES, C. MNIST handwritten digit database.

[30] LERMAN, L., POUSSIER, R., BONTEMPI, G., MARKOWITCH, O., AND STANDAERT, F.-X. Template attacks vs. machine learning revisited (and the curse of dimensionality in side-channel analysis). In *International Workshop on Constructive Side-Channel Analysis and Secure Design* (2015), Springer, pp. 20–33.

[31] LUO, C., FEI, Y., LUO, P., MUKHERJEE, S., AND KAELI, D. Side-channel power analysis of a GPU AES implementation. In *Computer Design (ICCD), 2015 33rd IEEE International Conference on* (2015), IEEE, pp. 281–288.

[32] MAGHREBI, H., PORTIGLIATTI, T., AND PROUFF, E. Breaking cryptographic implementations using deep learning techniques. In *International Conference on Security, Privacy, and Applied Cryptography Engineering* (2016), Springer, pp. 3–26.

[33] MANGARD, S., OSWALD, E., AND POPP, T. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, December 2006. ISBN 0-387-30857-1, `http://www.dpabook.org/`.

[34] MITCHELL, T. M. *Machine Learning*, 1 ed. McGraw-Hill, Inc., New York, NY, USA, 1997.

[35] NAIR, V., AND HINTON, G. E. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning* (USA, 2010), ICML'10, Omnipress, pp. 807–814.

[36] NARAEI, P., ABHARI, A., AND SADEGHIAN, A. Application of multilayer perceptron neural networks and support vector machines in classification of healthcare data. In *2016 Future Technologies Conference (FTC)* (Dec 2016), pp. 848–852.

[37] OHRIMENKO, O., COSTA, M., FOURNET, C., GKANTSIDIS, C., KOHLWEISS, M., AND SHARMA, D. Observing and Preventing Leakage in MapReduce. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM, pp. 1570–1581.

[38] OHRIMENKO, O., SCHUSTER, F., FOURNET, C., MEHTA, A., NOWOZIN, S., VASWANI, K., AND COSTA, M. Oblivious Multi-party Machine Learning on Trusted Processors. In *Proceedings of the 25th USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2016), SEC'16, USENIX Association, pp. 619–636.

[39] PAPERNOT, N., MCDANIEL, P., GOODFELLOW, I., JHA, S., CELIK, Z. B., AND SWAMI, A. Practical Black-Box Attacks Against Machine Learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (New York, NY, USA, 2017), ASIA CCS '17, ACM, pp. 506–519.

[40] PARASHAR, A., RHU, M., MUKKARA, A., PUGLIELLI, A., VENKATESAN, R., KHAILANY, B., EMER, J., KECKLER, S. W., AND DALLY, W. J. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)* (June 2017), pp. 27–40.

[41] PICEK, S., HEUSER, A., JOVIC, A., BHASIN, S., AND REGAZZONI, F. The Curse of Class Imbalance and Conflicting Metrics with Machine Learning for Side-channel Evaluations. *IACR Transactions on Cryptographic Hardware and Embedded Systems 2019*, 1 (Nov. 2018), 209–237.

[42] PROUFF, E., AND RIVAIN, M. Masking against side-channel attacks: A formal security proof. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (2013), Springer, pp. 142–159.

[43] RISCURE. https://www.riscure.com/blog/automated-neural-network-construction-genetic-algorithm/, 2018.

[44] SHOKRI, R., STRONATI, M., SONG, C., AND SHMATIKOV, V. Membership Inference Attacks Against Machine Learning Models. In *2017 IEEE Symposium on Security and Privacy (SP)* (May 2017), pp. 3–18.

[45] SONG, C., RISTENPART, T., AND SHMATIKOV, V. Machine Learning Models That Remember Too Much. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2017), CCS '17, ACM, pp. 587–601.

[46] TELECOM PARISTECH SEN RESEARCH GROUP. DPA Contest (4th edition), 2013–2014. `http://www.DPAcontest.org/v4/`.

[47] TEUFL, P., PAYER, U., AND LACKNER, G. From NLP (Natural Language Processing) to MLP (Machine Language Processing). In *Computer Network Security* (Berlin, Heidelberg, 2010), I. Kotenko and V. Skormin, Eds., Springer Berlin Heidelberg, pp. 256–269.

[48] THOMAS, P., AND SUHNER, M.-C. A New Multilayer Perceptron Pruning Algorithm for Classification and Regression Applications. *Neural Processing Letters 42*, 2 (Oct 2015), 437–458.

[49] TRAMÈR, F., ZHANG, F., JUELS, A., REITER, M. K., AND RISTENPART, T. Stealing Machine Learning Models via Prediction APIs. *CoRR abs/1609.02943* (2016).

[50] VEYRAT-CHARVILLON, N., MEDWED, M., KERCKHOF, S., AND STANDAERT, F.-X. Shuffling against side-channel attacks: A comprehensive study with cautionary note. In *International Conference on the Theory and Application of Cryptology and Information Security* (2012), Springer, pp. 740–757.

[51] WANG, B., AND GONG, N. Z. Stealing Hyperparameters in Machine Learning. *CoRR abs/1802.05351* (2018).

[52] WEI, L., LIU, Y., LUO, B., LI, Y., AND XU, Q. I Know What You See: Power Side-Channel Attack on Convolutional Neural Network Accelerators. *CoRR abs/1803.05847* (2018).

[53] XU, X., LIU, C., FENG, Q., YIN, H., SONG, L., AND SONG, D. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2017), CCS '17, ACM, pp. 363–376.

[54] XU, Y., CUI, W., AND PEINADO, M. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2015), SP '15, IEEE Computer Society, pp. 640–656.

simTPM: User-centric TPM for Mobile Devices

Dhiman Chakraborty
CISPA Helmholtz Center for Information Security, Saarland University

Lucjan Hanzlik
CISPA Helmholtz Center for Information Security, Stanford University

Sven Bugiel
CISPA Helmholtz Center for Information Security

Abstract

Trusted Platform Modules are valuable building blocks for security solutions and have also been recognized as beneficial for security on mobile platforms, like smartphones and tablets. However, strict space, cost, and power constraints of mobile devices prohibit an implementation as dedicated on-board chip and the incumbent implementations are software TPMs protected by Trusted Execution Environments.

In this paper, we present simTPM, an alternative implementation of a mobile TPM based on the SIM card available in mobile platforms. We solve the technical challenge of implementing a TPM2.0 in the resource-constrained SIM card environment and integrate our simTPM into the secure boot chain of the ARM Trusted Firmware on a HiKey960 reference board. Most notably, we address the challenge of how a removable TPM can be bound to the host device's root of trust for measurement. As such, our solution not only provides a mobile TPM that avoids additional hardware while using a dedicated, strongly protected environment, but also offers promising synergies with co-existing TEE-based TPMs. In particular, simTPM offers a user-centric trusted module. Using performance benchmarks, we show that our simTPM has competitive speed with a reported TEE-based TPM and a hardware-based TPM.

1 Introduction

Trusted computing technology has become a valuable building block for security solutions. The most widely deployed form of trusted computing on end-consumer devices is the Trusted Platform Module (TPM), a dedicated hardware chip that offers facilities for crypto co-processing, protected credentials, secure storage, or even the attestation of its host platform's state. By today, software and system vendors have built various security solutions on top of TPM. For instance, Microsoft's BitLocker uses it to release disk-encryption credentials only to a trustworthy bootloader [49]; or Google's Chromium uses the TPM for a range of objectives [60], such as preventing software version rollback, protecting RSA keys, or attesting protected keys.

TPM is also of interest for the different stakeholders on mobile devices. However, the particular benefits that the TPM offers have historically hung on the TPM's implementation as a dedicated security chip that can act as a "local trusted third party" on devices. Mobile devices are, however, constrained in space, cost, and power consumption, which prohibits a classical deployment of TPM. To address the particular problems of the mobile domain, the Trusted Computing Group (TCG) introduced the Mobile Trusted Module (MTM) specifications [61]. Although the MTM concept has never left the prototype status, its ideas influenced the latest TPM2.0 specification [64]. The TPM2.0 mobile reference architecture [63] proposed different alternatives for implementing a TPM on a mobile device, including virtualization, dedicated cores, or hardware-based isolation. The de-facto implementation of mobile TPMs today are protected environments through hardware-based trusted execution environment (TEE) [23, 24, 32, 45, 46, 54], like ARM TrustZone that is available on virtually all mobile platforms today, where the TPM is implemented as protected software application inside the TEE.

Given the different proposals for realizing TPMs on mobile platforms, we conduct a systematic comparison of the different solutions in terms of security of the TPM itself, their applicability in current systems, and deploy-ability in the specific setting of mobile devices. While the solutions naturally differ in their security guarantees for the TPM (i.e., TPM state or execution) due to differences in the underlying technology (e.g., dedicated hardware chip vs. virtual machine), we see particularly shortcomings of the current solutions in terms of applicability and deploy-ability. In particular, the currently incumbent fTPM (firmware TPM) is strictly bound to the platform vendors and serves their purposes (e.g., securing vendor credentials), but is not or only very limited available to other stakeholders in the system, such as the user. Moreover, an fTPM [54] that is based on a TEE falls short on providing a fully measured boot by itself. The availability of an fTPM

depends on the availability of the TEE during boot, which is one of the last steps in the long boot-chain. In light of recent attacks against mobile bootloaders [55] and trusted software in TEE [3, 8, 18, 39, 41, 51, 56–58], this lacking support to attest the entire, early boot-chain, including the software in the TEE, is unsatisfactory.

To put a new perspective on solving those issues of fTPM, we add in this paper an alternative implementation of a hardware TPM called *simTPM* to the landscape of mobile TPM implementations by using the subscriber identity module (SIM) card. We have implemented a prototype of our solution on a Hikey960 reference board [1] and using a Gemalto Multos card as SIM card. Our simTPM solves the technical challenge of implementing TPM2.0 compliant functionality on the SIM card, which does not require any additional hardware for the TPM. This approach keeps the costs down and leverages dormant hardware capabilities of mobile devices. Through performance tests, we show that simTPM is competitively fast to reported fTPM implementations. A particular challenge of this design is the lack of the usual physical binding between the TPM and its host platform's root of trust for measurement (RTM), that is, a SIM card can be moved to another platform. We discuss two strategies in the particular setting of mobile devices on how to bind the simTPM to a device's RTM, either through an extended secure boot and TEE proxy or through a distance bounding protocol. Once bound to the device's RTM, we also integrated simTPM with the ARM Trusted Firmware (ATF) boot chain to augment the ATF secure boot with an authenticated boot. Our solution not only fills the gap of TEE-based TPMs for measured boots, but the co-existence of a fTPM and simTPM on a mobile device creates also promising synergies between the two TPMs (e.g., to support multiple stakeholders). Our contribution can be summarized as follows:

1. A systematic comparison of existing solutions for mobile TPMs and their enabling technologies. We discover that incumbent solutions fall short on applicability and deploy-ability aspects.

2. We implemented the first SIM card based TPM2.0 for mobile devices by developing a simTPM, which can be executed in this constrained environment. Our solution enables a user-centric trusted module offering a portable sealed storage.

3. We propose an integration with the on-board TEE to solve the problem of binding the simTPM to the RTM and discuss an alternative solution based on distance bounding. As a result of this binding, a fully measured boot on the ARM Trusted Firmware (ATF) secure boot chain is possible.

4. The performance of our simTPM is competitively fast to a reported fTPM implementation and is comparable with existing hardware TPMs.

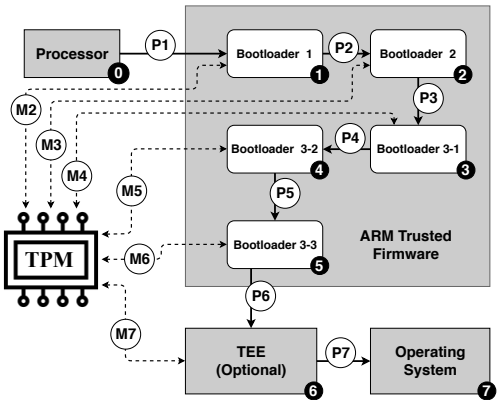


Figure 1: Trusted Boot Process with TPM; P(#) = boot chain path; M(#) = measurement of component #

2 Background

We briefly introduce necessary background information about ARM Trusted Firmware, TPM, and SIM cards.

2.1 ARM Trusted Firmware (ATF)

ATF implements a subset of the trusted board boot requirements for ARM reference platform [5]. Figure 1 illustrates the bootloader settings and boot chain. ATF is triggered when the platform is powered on. After the primary CPU and all other CPU cores are initialized successfully, the primary core triggers the ATF (P1). ATF is divided in five steps depending on modularity: ❶ BootLoader stage 1 (BL1) for AP trusted boot ROM, ❷ BootLoader stage 2 (BL2) for Trusted Boot Firmware, ❸ BootLoader stage 3-1 (BL3-1) for EL3 Runtime Firmware, ❹ BootLoader stage 3-2 (BL3-2) for Secure-EL1 Payload (optional), ❺ BootLoader stage 3-3 (BL3-3) for Non-trusted Firmware.

Secure boot: ATF implements a secure boot in which every component along the boot chain (P#) verifies the authenticity and integrity of the next component. Since BL1 does not have a preceding component, it has to be axiomatically trusted. Thus, BL1 verifies BL2, BL2 verifies BL3.x, and so forth. Verification is usually based on certificates, where a hash of a trusted (vendor) public key is fused into the hardware and is available to BL1 to ensure a trustworthy signature of BL2. At the end of a successful secure boot, every component in the boot chain has been checked for integrity and authenticity before handing control to it. If any verification fails, the boot aborts.

2.2 Trusted Platform Module (TPM)

TPM by the Trusted Computing Group is the most widespread trusted computing technology on end-user devices. By today, the TPM specification is in its version 2.0, addressing

many of the security issues and practical concerns of previous versions 1.0–1.2. According to this specification, a TPM provides a number of desirable hardware and security features. It is equipped with secure non-volatile memory, a set of platform configuration register (PCR) banks, a processor to run TPM code in isolation, co-processors for common cryptographic primitives (e.g., RSA, ECC, SHA-1, SHA-256), a clock, and a random number generator. By default, a TPM is deployed as a hardware chip soldered onto a platform's motherboard. Besides acting as a cryptographic co-processor, a TPM provides the facilities to securely store measurement about the host platform's configuration (e.g., software state) in its PCRs and to reliably report those measurements to a remote verifier (remote attestation based on a pre-installed endorsement key), as well as creating secure storage through TPM protected credentials and data sealing with extended authorization policies. Further, the TPM non-volatile memory, including secure monotonic counters, can be attractive for building security solutions, e.g., version rollback prevention for software updates.

By now, a number of real world applications make use of TPM. For instance, IBM's password manager uses it for storing keys, Microsoft windows management instrumentation uses TPM for cryptographic co-processing, Intel's Trusted eXecution Technology or AMD's Secure Technology rely on a hardware TPM, several VPN apps can make use of it, TPM is used in full disk encryption (e.g., Microsoft Bitlocker, dm-crypt), and even browsers like Chrome make use of TPM for different purposes.

Measured boot: Of particular relevance for this paper is measured (or authenticated) boot based on TPM (see Figure 1). During a measured boot, every component in the boot chain (P#) measures the next component—a cryptographic hash of the component—and then stores this measurement in the PCR of the TPM ((M#)) before passing on control. Since BL1 does not have a preceding component, it is not measured and acts as the *Root of Trust for Measurement*, which starts the measurement chain. In contrast to a secure boot, the components are not verified and the boot is not aborted, however, after a measured boot the software configuration of the boot components can be attested by the TPM or used to seal storage to this configuration (i.e., values in the PCR).

2.3 Subscriber Identification Module (SIM)

SIM card is the module that authenticates the mobile device in the network. The primary job of the SIM card is to prove the identity of the owner of subscription to the cellular carrier to enable services like calling, Internet, and various others.

Through physically separated pins, a SIM module can achieve the same degree of independence from power supply, reset capability, clock signal, and separated I/O communication with the host platform like a TPM.

Since SIM cards are smart cards, they use command-response communication and the application protocol data unit (APDU) to communicate with their reader. The Android radio interface layer can be extended to send specialized APDU commands to the SIM card, which we use in simTPM. It is worth noting, that this APDU command sent by the Android radio interface has to go through the baseband processor. The structure of the APDU commands are defined in the ISO/IEC 7816-4 standard and are recalled later on in Section 4.

3 Requirement Analysis & Systematization of Existing Solutions

There exists many approaches to realize TPM in a way different than using a dedicated hardware TPM. In this section, we systematically compare different solutions of trusted computing procedures using both hardware and software that are representative for the different implementation options. For comparison, we first re-enumerate the objectives a secure and practical TPM implementation needs to fulfill (Section 3.1) and then discuss the existing solutions (Sections 3.2 through 3.4). In particular, this systematization should help to understand the trade-off of the proposed solutions in comparison to the default hardware TPM and where our simTPM solution fits into. Table 1 summarizes the discussion in the remainder of this section.

3.1 Objectives

We start by briefly formulating the objectives a trusted module, in particular for mobile devices, should fulfill. We group them into security of the TPM itself, the applicability of the implementation, and desirable deploy-ability objectives.

3.1.1 Security of TPM

These are objectives that should be fulfilled to ensure the security of the TPM state, its execution and trustworthiness, and secure operations.

S1 Confidentiality and integrity of TPM state: The TPM state should be confidential and protected against untrusted code (e.g., host platform, non-TEE apps) and only be available to authorized entities. We assign ✔ if the confidentiality and integrity of the state is protected through strong security means (e.g., physical isolation), ✪ if they depend on software integrity (e.g., of the OS), and ✘ in other cases.

S2 Rollback Protection: Reverting the TPM state back to a former version must be prevented or at least be detectable. We assign ✔ if rollback protection is guaranteed through hardware means (e.g., hardware counters), ✪ if there is a dependency on untrusted OS but rollbacks can be detected, ✘ if no rollback protection or detection is provided.

S3 Trustworthy Endorsement: A TPM should be carrying an asymmetric encryption key called Endorsement key (EK) that can live as long as the TPM and for which credentials exist that verify the authenticity of the TPM and allow a verifier to recognize a genuine TPM. We assign ✔ if endorsement credentials are available to the TPM (e.g., pre-installed at manufacturing time or derived from other verifiable credentials), ✪ if the TPM has to create an EK and prove it is genuine through a remote verification, ✘ otherwise.

S4 Secure Counter: TPM has to provide secure, persistent monotonic counters, e.g., for its clients or extended authorization policies. We assign ✔ if the TPM provides such counters backed by hardware support or NV-storage of the TPM software state that is protected (i.e., **S1, S2** both ✔). We assign ✪ if the security of the counter depends on software integrity (e.g., of the OS or hypervisor). Otherwise ✘.

S5 Secure Clock: A clock is needed for attestation, for generation of timed attestation keys, and for authorization policies with lock-out time. If a secure clock is available to the TPM (e.g., its own hardware clock), we assign ✔; if the clock depends on shared resources but manipulation can be detected we assign ✪, otherwise ✘.

S6 Security of TPM Execution: The execution of the TPM code or firmware has to be protected against compromise. We assign ✔ if a strong security boundary exists between untrusted code and the TPM execution environment (e.g., dedicated physical chip). If the execution environment shares hardware resources (e.g., CPU or RAM) with untrusted code and the shared resources provide isolation (e.g., modes of operation of CPU and separate memory regions), we assign ✪, since the shared resources open an attack surface. If the security of the TPM execution environment is based purely on software means (e.g., hypervisor or OS), we assign ✘ for this weakest form of isolation.

3.1.2 Applicability

These are objectives related to the application of TPM, such as authenticated boot or providing secure storage to clients.

A1 Secure Persistent Storage: TPM provides a persistent storage to securely store limited amounts of data (e.g., certificates). We assign ✔ if the TPM provides such storage (e.g., NV-RAM in a dedicated chip) and ✪ if the persistent storage is part of an outsourced TPM state that is protected (i.e., **S1, S2** both ✔). We assign ✘ in other cases.

A2 Early Availability: A main use-case for TPM is storing the measurement of loaded software components, i.e., measured boot. To be able to attest the entire software stack, the TPM has to be early available during the boot sequence. If the trusted module is available as soon as the platform has power, we assign ✔. Otherwise, if the TPM becomes available at late stage during boot (e.g., after initializing a separate execution environment), we assign ✘.

A3 Multiple Stakeholders: Computer systems, in particular mobile platforms and enterprise devices, usually have multiple stakeholders co-existing with an interest in protecting credentials and software on the platform (e.g., end-user, administrator, network operator, software vendor). If the TPM was designed to support both platform software and users (e.g., distinct hierarchies), we assign ✔. If the TPM primarily supports the platform but offers limited functionality to the end-user, we give ✪. If the TPM was designed solely as support for the platform vendor, we give ✘.

3.1.3 Deploy-ability

Objectives related to the deployment of TPM, in particular if deployment complies with the requirements of mobile devices or if it is bound to a specific platform.

D1 Mobile Availability: We want to have the TPM available for mobile devices. This imposes strict constraints, such as not changing the current architecture by adding a new on-board chip. If the TPM implementation adheres to this constraints, we assign ✔, otherwise ✘.

D2 Movability: The TCG specification has introduced the TPM as being bound to its host platform (e.g., fixed part of the motherboard). However, depending on the context, the movability of the TPM to another platform is desirable, e.g., if an associated virtual machine migrates to another platform. If the TPM is generally easily moved to another platform, we assign ✔, if it is bound to a specific platform, we assign ✘.

D3 Bound RTM: The measurements during a measured boot are given to the TPM by the host platform, starting with the Root of Trust for Measurement (RTM). To ensure that the provided measurements indeed describe the TPM's host platform's configuration, TPM and RTM must be bound together on the same platform. If this binding is achieved via physical means (e.g., TPM and RTM are fixed parts of the same motherboard), we assign ✔. If the TPM receives those measurements from another trusted entity (e.g., another, bound TPM, or a secure boot anchored at the RTM), we assign ✪. If the TPM cannot establish trust into the RTM, we assign ✘.

In Section 2.2, while introducing the hardware TPM, we explained all its properties, which allow the TPM to achieve the objectives we defined in Section 3.1 and summarized in Table 1. Objectives **S1** to **S6** and **A1** to **A3** are our interpretation of properties derived from TCG's mobile TPM [61, 63] and standard TPM specification [62, 64]. We define *Deploy-ability* as added objectives that simTPM should achieve. The current TCG specifications do not stipulate a removable TPM. We will use the standard hardware TPM as the baseline that simTPM should achieve.

Table 1: Comparison of existing TPM implementations

Category		Objective	fTPM [54]	vTPM† [9]	Intel SGX [19]	simTPM	Hardware TPM
Security of TPM	Security of TPM state	**S1.** Confidentiality and integrity	✔	✔/ ✪	✲	✔	✔
		S2. Rollback protection	✔	✔/ ✪	✔	✔	✔
		S3. Trustworthy Endorsement	✔	✪/ ✪	✔	✔	✔
		S4. Secure counter	✔	✔/ ✪	✔	✔	✔
		S5. Secure clock	✪	✔/ ✘	✘	✔	✔
		S6. Security of TPM execution	✪	✔/ ✘	✲	✔	✔
Applicability		**A1.** Secure persistent storage	✪	✔/ ✘	✪	✔	✔
		A2. Early availability	✘	✔/ ✔	✘	✔	✔
		A3. Multiple stake holder	✘	✔/ ✔	✔	✔	✔
Deploy-ability		**D1.** Mobile availability	✔	✘/ ✘	✘	✔	✘
		D2. Movability	✘	✔/ ✔	✘	✔	✘
		D3. Bound RTM	✔	✘/ ✪	✔	✪	✔

✔ = fulfilled by the implementation; ✪ = partially fulfilled by the implementation; ✘ = not fulfilled by the implementation; ✲ = not applicable for the implementation

† First column is for *Secure co-processor based vTPM* (SCoP) implementation and second column is for *Software only vTPM* (SW-only) implementation

3.2 fTPM

Specifically for the mobile domain, a number of past implementations [25, 54, 66] leveraged trusted execution environments (TEE) to realize a software-based TPM. We use Microsoft's fTPM [54] as a representative for those implementations, since it is one of the most recent solutions. The fTPM implementation is widely deployed in Microsoft mobile devices using a TEE on top of ARM TrustZone (**D1:** ✔). TrustZone creates a memory and process isolation between the protected environment ("secure world") running inside the TEE and the "normal world" (i.e., Android or similar), and allows the execution to switch contexts between those two worlds via a secure monitor.

fTPM provides confidentiality, integrity (**S1:** ✔), and rollback protection (**S2:** ✔) for fTPM states by creating a trusted storage through a combination of encryption with fused keys, device UUID, and Replay Protected Memory Block (RPMB) with authenticated writes and write counter. Any form of secure persistent storage the fTPM offers to clients is based on this securely outsourced state (**A1:** ✪), which is also used to provide secure counters to clients (**S4:** ✔).

Due to ARM TrustZone, the execution of the fTPM environment is isolated from the normal world, however, both worlds still share the CPU and RAM (**S6:** ✪), which has opened TrustZone TEEs to attacks (e.g., [41]).

fTPM does not have a separate secure clock. It uses the clock of the system in cooperation with the untrusted OS (**S5:** ✪). To handle the shared clock situation, fTPM implements *fate sharing*, where fTPM refuses to provide any functionality if the OS does not cooperate.

fTPM is primarily designed to provide TPM support to the platform vendor (**A3:** ✘). The fTPM is a software implementation and bound to one device (**D2:** ✘), since it derives many of its credentials from device-specific keys or UUIDs, including its endorsement credentials (**S3:** ✔).

Since the fTPM is implemented as software in the TEE on top of ARM TrustZone, the fTPM becomes only available once the TEE has been initialized during the boot sequence (see also Section 2). That means the fTPM (or any TEE-based TPM) is not early enough available to store measurements of the early boot stages (**A2:** ✘). But this can be alleviated by introducing shared memory between the bootloaders and TEE for measurement storage. We will discuss this solution in more details in Section 4.3.

Although the fTPM is only available after the bootchain has created the TEE, the secure boot transitively extends the trust put into the RTM (BL1) to the remainder of the secure bootchain on the same platform as the TEE. Thus, fTPM can assume that the measurements are done as if by the RTM on the same platform (**D3:** ✔) if the measurements comes from a component of the secure bootchain.

3.3 vTPM

Another way of implementing a software TPM is by creating virtual instances over a physical TPM [9]. This, in particular, targets cloud environments in which virtual machines need a TPM, but sharing a single physical TPM (or providing an array

of physical TPM) is not an option. The representative work for virtual TPM, or vTPM, is based on the Xen hypervisor and proposes two different implementation options: 1) a software only implementation with vTPM instances running inside a privileged VM, and 2) a secure co-processor (SCoP) to run all vTPM instances with better isolation at the cost of additional hardware. Both options are not feasible for mobile TPMs (**D1:** ✘), since virtualization is not sufficiently supported or effective, and adding a secure co-processor is too costly in terms of space and power. However, by design vTPMs must be movable to different platforms to support migration of associated VMs between platforms (**D2:** ✔).

In both deployment options, a vTPM has to create its endorsement key at creation time. To establish trust into the EK for a remote verifier, a genuine, primary TPM on the platform (hardware TPM) must attest the trustworthiness of the vTPM's EK (**S3:** ✪).

In case of SCoP-vTPM, the TPM logic and vTPM instances are executed inside the secure co-processor (**S6|SCoP-vTPM:** ✔). Further, the secure co-processor used in [9] (an IBM PCIXCC) provides CMOS RAM backed persistent storage. We assume it provides the confidentiality, integrity, and rollback protection of the vTPM states as well as sufficient secure persistent storage to the vTPM clients (**S1, S2, A1|SCoP-vTPM**: ✔). The same co-processor also offers facilities for secure counters (**S4|SCoP-vTPM:** ✔) and a secure clock (**S5|SCoP-vTPM:** ✔).

For SW-only-vTPM the vTPM instances reside in Xen's privileged *dom0*. Thus, their execution is protected from untrusted VMs by only the Xen hypervisor (**S6|SW-only-vTPM:** ✘), and their state, when stored in persistent storage in *dom0*, is also protected by only the access control and isolation of the hypervisor and *dom0* (**S1, S2|SW-only-vTPM:** ✪). Similar, the protection of any persistent storage offered to vTPM clients depends on the integrity and trustworthiness of *dom0* (**A1|SW-only-vTPM:** ✘) as does any counter stored in the vTPM state (**S4|SCoP-vTPM:** ✪). A vTPM relies on the platform's clock shared between all vTPMs including untrusted code and not specifically protected (**S5|SCoP-vTPM:** ✘). Although vTPM instances are created after the host platform has booted up, a vTPM receives the initial measurement from the underlying hardware TPM of its platform, which also attests the vTPM trustworthiness, and *dom0* protects the vTPM state from migrating to an untrusted platform (**D3|SW-only-vTPM:** ✪). Further, vTPM instances are created together with their associated VM, hence, allowing the VM to measure its entire bootchain and store the measurements in its vTPM (**A2:** ✔).

In case of SCoP-vTPM, the TPM resides entirely in the IBM PCIXCC, a removable peripheral. Thus, no physical binding to the RTM exists and no authenticity/trustworthiness of the RTM is being ensured (**D3|SCoP-vTPM:** ✘), hence, an attacker could move the TPM to an untrusted platform that feeds the TPM with arbitrary measurements. This situation is very similar to our simTPM, which is also removable, and we discuss solutions to this challenge in Section 4.3, which might also be applicable to SCoP-vTPM.

The vTPM does not make any assumptions about which stakeholder—user or platform—within the associated VM uses the vTPM and supports, like a regular hardware TPM, multiple hierarchies (**A3:** ✔).

3.4 Intel SGX

Although Intel SGX is not an implementation of a TPM but a solution to allow applications to establish a TEE, *enclave* in SGX jargon (**S1, S7:** ✲), we include it here for comparison because it offers in many dimensions similar protections as a hardware TPM and shares a lot of a TPM's objectives (we mark non-applicable objectives with ✲ in Table 1). For this work we have only considered stock SGX implementations in Intel processor to keep the comparison on par with other candidates. SGX is currently only supported by desktop and server class Intel processors (**D1:** ✘) and binds any credentials, like generated and derived keys, and transitively sealed data strictly to the CPU (**D2:** ✘).

In SGX, *attestation* means verifying that a certain enclave code was initialized correctly and not tampered with by the untrusted host OS. For *remote attestation* in SGX an Intel-provided *Quoting Enclave* provides the facilities to enclaves to do direct anonymous attestation (DAA) using attestation keys endorsed by Intel (**S3**: ✔). The SGX extensions to the CPU measure the enclaves, hence, the enclaves are physically bound to their RTM (**D3**: ✔).

SGX supports enclaves in sealing data for storing it on untrusted persistent storage, since enclaves themselves do not have any persistent storage like NV-RAM (**A1:** ✪). In addition, Intel has added support for monotonic counters [30, 43] that allow rollback protection of sealed data (**S4, S2:** ✔).

It is a processor based technology, so it can fully utilize the clock of the system. But the current SGX implementation does not accommodate a trusted and fine grained clock for the user-level enclaves. There is an API provided by Intel, e.g., `get_trusted_time`, but this call can be arbitrarily modified by the untrusted OS, since it requires to make an *OCALL* [4, 6, 17, 31, 37]. Moreover, any timing mechanism must account for the fact that the OS can interrupt the enclave at any point in its execution, wait for an arbitrary period of time, and then resume the enclave using *ERESUME* (**S5:** ✘).

Both regular applications and system software can use enclaves and SGX is not restricted to particular stakeholders (**A3:** ✔). However, an early firmware initialized enclave is not possible, since the OS is needed for memory management of enclaves (**A2:** ✘).

3.5 Java-card based MTM

Dietrich and Winter proposed a way of implementing a mobile trusted module (MTM) in a Java-based smart-card for mobile devices [21, 26]. The implementation is for applications running on mobiles and the TPM communicates through NFC.

The TPM is installed as a set of applets in the Java-card, where a *master-applet* provides services to other applets, like TPM command handling and controls the access to the endorsement key. The actual processing of TPM commands is handled by specific applets implementing those commands.

Although this implementation seems like closest related work to our simTPM, their work described a proof-of-concept prototype and is unfortunately silent about many aspects, such as secure persistent storage, and some functionality is not available, such as attestation of the system or authenticated boot. The Java-card communicates with the system over NFC, so binding the card with the system is not possible and early availability of the trusted module is also not possible before the NFC driver is loaded.

Their implementation provides important insights on the implementation of MTM on mobile devices through a programmable TPM and presented pioneering work, but given the lack of documentation and also differences in engineering (see Section 4), we cannot provide a full and fair comparison with simTPM and exclude it from our systematization.

4 System Design and Security Analysis

The main component of simTPM is a smart card based implementation of a SIM TPM. However, to properly work it also requires changes in the bootloader and the operating system (i.e., Android). In this section, we describe the design and implementation of simTPM in more details. We also discuss how our solution solves the shortcomings described in Section 3 and argue about our design's security. Along with the design descriptions, we indicate how the objectives shown in Table 1 are met by simTPM.

4.1 SIM TPM

Modern SIM cards are usually general purpose smart cards running an applet created by the mobile network provider. The two most prominent smart card technologies are Java Cards and Multos cards. Both introduce a custom OS (i.e., Java Card OS and Multos OS) and APIs that can be used by programmers for cryptographic (e.g., encryption, signing) and non-cryptographic (e.g., memory allocation and copy) operations that are implemented and executed directly on the microprocessor. Depending on the technology, applets can be programmed in C/C++ (e.g., Multos cards) or in Java (e.g., Java Card). Additional cryptographic algorithms, not provided by the API, can be implemented in software.

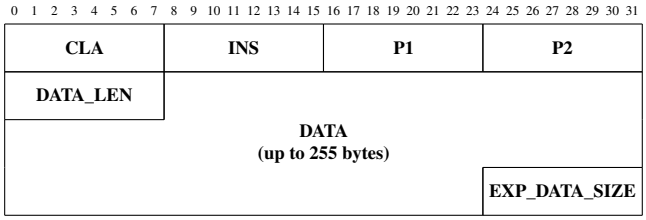


Figure 2: Generic APDU command structure

Both card technologies have support for multiple applets. To properly manage them, cards provide a specialized security manager that is responsible for installing and deleting of user defined applets. Once an applet is uploaded, the security manager creates its instance and allows the applet to create necessary objects and allocate memory.

4.1.1 API Limitations of Smart Cards

As mentioned above, each Smart Card OS provides a card specific API that allows applets to perform extended operations. This forces the programmer to use only a predefined set of functions. For example, in case of Java cards the API supports only a subset of the standard Java language and is limited to high level cryptographic operations (e.g., encryption, hashing, signing). There is no support for mathematical functions like modular multiplication or elliptic curve point addition, which are one of the main building blocks of public key cryptography. In other words, the developer cannot use hardware support for those low-level operations and is limited to software implementations that are inefficient due to the overhead of the virtualization layer.

Obviously, those limitation do not directly concern TPM commands that only use basic cryptographic operations. Unfortunately, the TPM standard defines a remote attestation scheme that is not supported by the cards API, because it uses, e.g., zero-knowledge proofs. This constitutes an interesting engineering problem that we solve. In particular, we were able to implement simTPM on a Gemalto MultiApp Multos smart card with an Infineon SLE78CLX family microprocessor. This card also helped us achieving process isolation from the general-purpose processor (**S6:** ✔). It is worth noting, that in this paper we focused mainly on the Multos API [42], because it supports a broader range of functions than the Java card API. In particular, we were able to efficiently implement a remote attestation scheme on-card.

4.1.2 Smart Cards and TPM Command Parsing

SIM cards are connected to the main processing unit over a separate bus and available for mobile telephony services (**D1:** ✔). Smart cards work in a command/response manner, i.e., given an input the card executes the code and returns a response. The input data is defined by an APDU command

(see Figure 2), which consists of a class byte (CLA), an instruction byte (INS), two bytes for parameters (P1, P2), one byte for the expected response length, one byte for the data length (DATA_LEN), and DATA_LEN bytes of data. The cards' response contains the response data and two bytes that constitute the status word (not shown in the figure). The data field is limited to 255 bytes. There exist an extended length APDU specification that allows for a larger data field but it is not widely implemented.

In a multi applet system, an APDU command will be forwarded to the currently selected applet. To select an applet, the `SELECT` APDU command with an unique applet identifier has to be sent to the card. This command is then recognized and executed by the OS. Once selected, the applet can parse incoming commands according to its work flow. In particular, this means that the developer can use the instruction and parameters bytes to program the behavior of the card.

The APDU data structure provides a convenient way to communicate with the card. We designed a custom APDU command that implements TPM commands. The data length size of up to 255 bytes is sufficient for the payload sizes of most TPM commands, and for TPM commands with larger payload sizes (e.g., sealed data blobs), we send the payload split across multiple APDU messages and use the parameter bytes to communicate the card if more data is to be expected.

Changes to Android's radio interface: The Android Radio interface layer (RIL) is responsible for communicating with the device's SIM card. To allow RIL to communicate with simTPM, we introduced a set of TPM commands. We implemented a custom RIL as a shared library, which sends APDU commands as bulk transfer to the simTPM and receives its responses.

4.1.3 TPM Commands

We now briefly discuss how we designed the card to handle basic TPM commands related to PCR banks and sealing. The former case is easy, the applet reserves enough non-volatile memory to store the PCRs. The number of banks is defined by the installation parameter of the TPM applet, which also defines the algorithm we use to extend the PCR (e.g., `SHA1` or `SHA256`). In a standard setup we use 24 PCRs. For the `TPM_EXTEND` and `TPM_READ` commands we used two separate instruction bytes (respectively, `0x10` and `0x20`) to form the APDU. In both cases the number of the PCRs is given using parameter P1.

To design (un-)sealing on a smart card was a bit harder. Due to the limited input data size, the card has to encrypt/decrypt the input in chunks, which are split across multiple APDU messages. The storage key for sealing is generated by the card after receiving the `TPM_INIT` command. The key is stored in the non-volatile memory that is allocated during installation of the applet (see next Section 4.1.4).

It is worth noting that smart cards can be programmed to execute all TPM commands that require basic cryptographic algorithms, on-card key generation, key agreement, or storing data in volatile/non-volatile memory. Unfortunately, the privacy-preserving variant of remote attestation (i.e., direct anonymous attestation, DAA) requires zero-knowledge proofs and other unsupported crypto operations. What is more, in versions below TPM 2.0 the specification defined only one algorithm for anonymous attestation [14], which is based on groups with hidden order (i.e., using a RSA modulus) and Camenisch-Lysyanskaya signatures. The TPM 2.0 specification, however, allows for algorithm agility. We leveraged this fact and used a custom scheme. We present the full scheme and security proofs in the technical report of our paper [16]. Here, we only draft the idea behind the scheme, which follows the generic approach used by other DAA schemes: The TPM receives a signature/certificate under its secret DAA key from an authority. It then uses this secret key to certify its attestation key using a proof. In this zero-knowledge proof the TPM shows that it knows a certificate under a DAA key and a signature created using this key under an attestation key. The scheme uses Boneh and Boyen [11] signatures and an efficient zero-knowledge proof for the above statement that is made non-interactive using the Fiat-Shamir transformation [28]. The main advantage of the scheme is that it can be executed solely by the TPM (i.e., on-card) and does not require any involvement of the host platform. To further improve efficiency of our scheme, we decided to optimize the workload between commands, i.e., if the `TPM_CREATE` command recognizes that the TPM is creating an attestation key, it already does some pre-computation for the DAA certification.

4.1.4 PCR and NV storage

All smart cards implement a small amount of non-volatile storage that can be used for various purposes. This memory of the smart card is by design tamper-resistant and therefore offers memory isolation from the rest of the system (**S6:** ✔). Modification of this memory is only possible by the applet that reserved it and we reserve some of the NV storage for the simTPM (**A1:** ✔). Smart cards are equipped with features preventing updates of its internal state by the outside world. To update stored content (e.g., applets), one has to issue an authorized command to the card manager to update storage or perform applet specific commands, e.g., PCR extension (**S1:** ✔). Our simTPM is equipped with PCR banks that are initialized when power cycling the device and, hence, the SIM card, and can only be changed between power cycles using `PCR_EXTEND`.

System software or user level software can keep a counter containing the current version of the software inside the NV-storage and updates to the counter are only allowed via authorized commands. This provides an easy setup for secure counter and rollback protection (**S4:** ✔).

4.1.5 Trustworthy endorsement & Clock

Trustworthy endorsement of a TPM is very important. The standard solution is to use an asymmetric encryption key called endorsement key. This key is unique per TPM and should stay alive as long as the TPM is alive. This key differentiates a genuine from a rogue TPM. simTPM can achieve secure endorsement by putting a (vendor) certified endorsement key inside its NV-storage and implementing TPM logic that ensures that the private portion of the key is never released to the outside world (**S3:** ✔).

SIM cards are equipped with a clock pin connected to the baseband processor. Thus, they cannot be clocked higher or lower by an untrusted application or OS. This separate clock helps simTPM to work on a different clock frequency not under direct influence of the main processor. What is more, the baseband processor can be used as a secure external clock. In particular, since the baseband processor is by default isolated with a strong security boundary from untrusted code on the platform, it can prepend any APDU command with an APDU command containing the current time (this can also be limited to time-sensitive TPM commands only). This way simTPM can be provided with a secure clock (**S5:** ✔).

4.1.6 Movability & Stakeholders

The other unique feature of the simTPM architecture is its movability (**D2:** ✔). simTPM implements the TPM inside the SIM card. So by design, simTPM can be transferred to a different device. This creates some interesting use-cases, which we discuss in more details in Section 6.2, but also challenges, which we discuss separately in Section 4.3. simTPM is not specifically bound to one particular stakeholder and supports the multiple stakeholder model proposed by TCG (**A3:** ✔), although we think the end-users and their apps are the primary beneficiaries of simTPM.

4.2 ATF boot-loader changes

In Section 2.1, we have briefly introduced ATF and its boot-loader chains. In this section we describe the changes we have implemented to enable communication between the bootloader components and the simTPM. Figure 1 can be helpful as a visual aid for understanding.

After turning on the secondary cores on the cold boot path, the processor kicks in the first stage BL1 of the bootloader (❶). Current bootloaders are not implemented such as to be able to communicate with a device like a SIM card and to run a command response protocol. Thus, we have extended all the boot-loaders with the capability to communicate with the SIM card via bus communication. This modification in ATF makes the simTPM already available to the early BL1 stage (**A2:** ✔). The bootloader software is capable of translating TPM commands to APDU commands, sending them to simTPM, receiving responses, and translating them to a meaningful response that can be used to make decisions (e.g., failed/successful PCR extension commands). One thing that needed to be addressed here is that except for BL3-3, all bootloaders are secure mode software (i.e., secure world in TrustZone). So during execution, simTPM has to be initialized as secure mode hardware to be available to the bootloader. We initialize the simTPM as a secure mode hardware, but after a successful boot chain verification, we switch simTPM to normal mode (of TrustZone). This allows us to maintain normal efficiency in the normal world, since the SIM card functionality (e.g., calls or text messages) is accessed by Android and switching context from normal world to secure world every time before accessing the SIM card in Android can interrupt the normal world execution and would be highly inefficient.

4.3 Bootstrapping trust for movable simTPM

Parno [53] was first to identify the problem of how to bootstrap trust into a hardware TPM and the possibility of *cuckoo attacks*. A fundamental problem of TPM is that the verifier (e.g., local user) does not know if they are talking to the *intended* (e.g., local) TPM, just that they are talking to *a genuine* TPM. In a cuckoo attack, an attacker that compromised the local platform can exploit this problem and fool the verifier into trusting the compromised platform: the attacker simply relays the verifier's communication to another (remote) TPM on an attacker-controlled platform, which then can attest an arbitrary, trustworthy state to the verifier. The preferred solutions to prevent cuckoo attacks are hardwired channels via a special purpose hardware interface to the on-board TPM or, alternatively, a cryptographically secured verifier-TPM communication where the verifier has knowledge of the public key of the TPM on the intended platform.

However, those solutions make an implicit assumption: Historically TPMs are soldered onto the motherboard, eliminating the issue of ensuring proper binding to the device's root of trust of measurement (RTM), usually in form of an immutable piece of trusted code in the BIOS. Due to this static design a TPM is ensured that the very first received measurement in a chain-of-trust is coming from a trusted, local RTM. Only a sophisticated hardware attack can break this binding. A TPM that is by-design movable, such as our simTPM or the PCI-attached secure co-processor for vTPM [9], raises an interesting question about how to re-establish this bond between TPM and RTM.

Lack of chain-of-trust: Without binding the TPM to a trusted, local RTM, the measurements of any authenticated boot cannot be trusted. An adversary could simply plug the simTPM into an attacker-controlled platform and replay[1] any desired measurements sequence, i.e., create arbitrary PCR values akin to a TPM reset attack [29, 36]. This allows the

[1] The TPM is a passive device to which the measurements have to be provided by its caller.

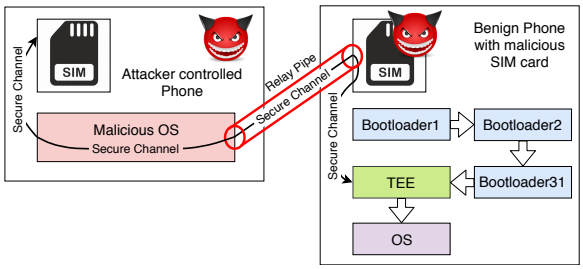


Figure 3: Using TEE as TPM proxy to bind simTPM with RTM and to mitigate the effects of relay attacks.

attacker to fool a remote verifier during remote attestation but also to gain access to sealed secrets, whose release is bound to the platform state (i.e., PCR values).

Binding simTPM and RTM: To create a binding between the simTPM and a trusted, local RTM, we need the simTPM to 1) authenticate the RTM to ensure its a trusted code (e.g., BL1 of ATF); and to 2) ensure policies (e.g., for data release) and commands (e.g., attestation) are only executed for exactly the platform for which the simTPM stores the measurements. To address those challenges, we identified two possible solutions, using the device's TEE as a proxy to the simTPM (see below) or using distance bounding protocols (discussion deferred to the technical report of our implementation [16]).

Using TEE as TPM proxy: One way to bind the simTPM with the device's RTM is by leveraging the platform security building blocks of mobile devices and using the TEE as a proxy to simTPM (see Figure 3). On a genuine device with secure boot in place, i.e., BL1 as a trusted RTM, the TEE has exclusive access to device-specific credentials that are certified by the device vendor. Using those credentials, the simTPM and TEE can establish a secure end-to-end channel. In this setup, simTPM will only respond to PCR extensions, attestation requests, or unsealing of encrypted data if the commands come via this secure channel. As a result, an attacker cannot forge arbitrary PCR values without compromising the device-specific key. Further, if the TPM enforces a particular device key, it can ensure that only the intended platform is using the simTPM; however, even without this strict set of device keys, this solution still ensures that any TPM commands, such as releasing data to the host platform, can only come from a genuine mobile platform with an intact secure boot from which it received the measurements. Considering previously mentioned software-based attacks against TEE (see Section 3.2), an attacker could compromise the TEE to steal the device-specific key and impersonate the TEE to the simTPM. This can be alleviated by using session keys instead of the long-term secret device-specific key for communication between TEE and simTPM, which could be setup during the bootstrapping and, hence, before untrusted code can attack the TEE. A drawback of this solution is that the simTPM requires the TEE to be bootstrapped to become itself operational, which prevents an early availability of the simTPM. Since the simTPM is not early available in this setup, ATF's secure boot has to be extended to store the measurements of verified software components and pass those measurements on to the TEE, which then can forward them to the simTPM via the secured channel (**D3:** ✪). It should be noted that while this extension to ATF would also provide a solution to the early availability of fTPM [54], simTPM gives a user-centric solution and additional interesting use-cases in comparison to fTPM (see Section 6). We discuss an alternative solution based on a distance bounding protocol in Appendix A.

4.4 Security analysis

Lastly, we analyze the security of simTPM in comparison to the closest solutions fTPM and hardware TPM, specifically considering the deployment of our TPM on a SIM card.

Off-chip protection: As mentioned in Section 3, fTPM depends on the integrity of the secure world, which has been under attack recently [3, 8, 18, 41, 51, 55–57]. Our simTPM implements an off-board TPM on the SIM card and, like a discrete TPM, is physically isolated from untrusted code. This provides a stronger protection of the simTPM's trusted computing base, however, we cannot fully exclude potential software attacks against the SIM card software. For instance, in the past smart cards have exhibited bugs [50] like hidden commands, buffer overflows, weaknesses of cryptographic protocols [52], or malicious applets [52]. Further, like a hardware TPM, simTPM is connected via a bus, which makes it prone to advanced bus attacks [12, 34, 36] that, however, are considered outside the attacker model for consumer grade hardware like the TPM.

SIM card cloning: Deployment on a SIM card also raises the concern of card cloning [65], which could easily enable impersonation attacks or theft of credentials. However, driven by the interests of telecommunication companies, modern SIM cards come with anti-cloning defenses that mitigate this attack vector [42].

SIM swapping attack: In SIM swap attack, an attacker obtains details about the victim and then tricks the telephony company to port the victim's phone number to a fraudulent SIM card owned by the attacker, usually with the goal to receive all SMS including highly sensitive information, like OTP for online banking. In our design, the TPM is not dependent on the SIM telephony functionalities. simTPM works as a local co-processor with desirable attributes. An attacker can port the telephony services to a fraudulent SIM card, but not the TPM state, as it is bound to the local SIM-card and would require explicit migration policies to other (SIM)TPM.

Side-channel attacks: To be compliant with the TPM 2.0 specification, the hardware has to implement cryptographic functions that are resilient to timing-based side-channel attacks. There exists a similar requirement for smart cards, which are designed to be resistant against various types of side-channel attacks. Thus, simTPM immediately benefits from the security features of the underlying smart card.

However, a motivated attacker can easily move simTPM to a controlled environment and mount different active side-channel attacks, such as clock frequency, heat measurement, probing [33], fault injection [35], or power analysis [40,47,48]. While similar attacks have been shown against ARM TrustZone (e.g., [39,58]) and discrete TPM chips [59], deploying the TPM on a removable card might ease mounting those attacks. Nevertheless, it should be noted that such sophisticated hardware attacks are not only strenuous, exorbitant, and inconsistent, but also beyond the protection that a consumer grade security chip can offer.

5 Performance Evaluation

We evaluate the performance of simTPM on a HiKey960 board in comparison with a hardware TPM. We focus on the most frequent commands executed by a TPM, i.e., key generation, sealing/unsealing of data, extending/reading a PCR, generating random bytes, and computing a hash value of an input. Beside simTPM we prepared two test setups equipped with an Infineon SLB 9670 TPM chip. One of these two test benches is a plug-able TPM on a Raspberry-Pi (*piTPM*) and the other one is an embedded TPM on a standard Lenovo laptop (*embTPM*). More details about the setups is given in [16]. We have used a TSS implementation by IBM [2] to communicate with the Infineon TPM. The results of our benchmarks are summarized in Figure 4. All results come from 50 measurements per command per device. We report the 95% confidence intervals.

5.1 Test cases and results

Key generation: We measured the time to generate a 256-bit ECC key and output the public part of the key. Our implementation of simTPM creates the key on average in $257 \pm 8.03ms$, comparable to the piTPM performance ($253 \pm 1.25ms$), but slower than the embTPM ($172 \pm 0.61ms$).

Create hash: We measured the time it takes for the TPM to hash 256 bits of input data with SHA-256 and output the digest. piTPM ($50 \pm 0.76ms$) and embTPM ($21 \pm 0.16ms$) outperform the simTPM ($72 \pm 10.13ms$) by a factor of 1.44 and 3.42, respectively.

Extending and reading a PCR: We evaluated the PCR extend and read commands. The former allows to extend the PCR with a new value, while the latter command is used to read the current value of a PCR. We use SHA-256 as hash algorithm and a 128 bit string as input value. For PCR extension, simTPM ($24 \pm 2.66ms$) is on par with embTPM ($21 \pm 0.11ms$), however, exhibits a higher instability of the performance. For reading PCRs, simTPM ($15 \pm 0.15ms$) is the fastest implementation, followed by embTPM ($21 \pm 0.13ms$). piTPM is the slowest implementation in both cases ($41 \pm 1.22ms$ and $57 \pm 2.58ms$) and exhibits an unstable performance, too.

Sealing and unsealing data: The TPM seal command takes a byte array, attaches a policy, encrypts it with a TPM storage key, and returns a blob to the caller. When unsealing, the TPM takes an encrypted blob, checks the policy, and decrypts the blob if the policy is satisfied by the TPM state. For our performance measurement we used 128 bits input data, a 256-bit ECC sealing key with ECIES, and an empty policy. The embTPM is the fatest solution for sealing and unsealing ($130 \pm 0.27ms$ and $89 \pm 0.46ms$) and outperforms our simTPM ($588 \pm 18.55ms$ and $376 \pm 22.30ms$) by a factor of 4.52 and 4.22, respectively.

Random number generation: We use the TPM to generate a 64 bit random number. Our simTPM is the fastest solution ($15 \pm 0.14ms$), followed by embTPM ($21 \pm 0.17ms$) and then piTPM ($63 \pm 1.63ms$).

5.2 Discussion of performance

Our test results show that there is no clear winner among our test systems. simTPM as well as embTPM excel for some commands and we would argue that our simTPM prototype shows a competitive performance. Unfortunately, the implementation of Infineon SLB 9670 TPM is not publicly available, thus commenting on the exact reasons for those differences would result in speculations. If we would venture to speculate, potential reasons for the differences could be the different communication buses. embTPM has a dedicated bus communication with the onboard processor and a faster processor, while piTPM is running on a Raspberry Pi and is connected over GPIO with lower bandwidth. On the other hand, simTPM is connected through the USB bus. Moreover, our simTPM implementation uses only the publicly available APIs of the smart card OS, which provide only an indirect access to hardware level commands. Hence, a vendor-supported implementation with direct access to the microprocessor would improve in efficiency.

The fTPM is unfortunately not available, precluding a direct comparison in our test suite; however, our observations for the embTPM speed are comparable to those reported by Raj et al. [54], although it is unclear which hardware TPM they evaluated. An fTPM, unsurprisingly, outperforms any other tested implementation here—e.g., slowest fTPM in [54]

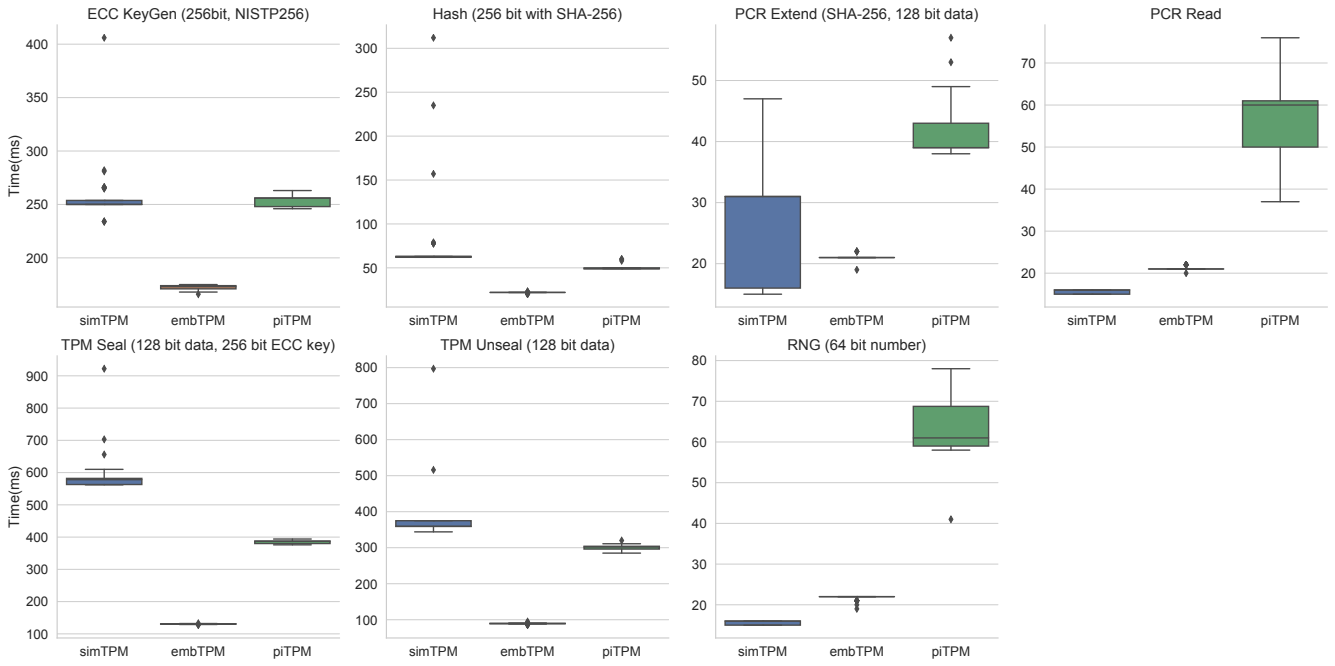


Figure 4: Performance comparison (in *ms*) of different TPM commands for simTPM and an Infineon SLB 9670 TPM2.0 on a Raspberry-Pi and Lenovo laptop.

was between 2.4–15.12 times faster than the fastest hardware TPM—since it is executed on the ARM Cortex main application processor, whereas discrete TPMs use slower microprocessors, as does our simTPM.

6 Use Cases

We discuss briefly how simTPM fits into the trusted computing landscape and explain scenarios that are of particular interest when simTPM and fTPM co-exist.

6.1 Multiple stakeholder model

The TPM specifications [64] as well as the obsolete Mobile Trusted Module (MTM) specifications [61] acknowledged the fact that a trusted platform might have multiple stakeholders. In particular, mobile platforms are not considered under the full management of the user, but critical mobile network management is the domain of the mobile carrier/network operator and the device vendor has high interest in keeping highest privileged operations (e.g., TEE and OS) under their control. The old and new TCG specifications define recommended capabilities and various implementation alternatives to allow multiple stakeholders to safely coexist. For instance, the MTM specification clearly differentiates between remote stakeholders and local stakeholders, each with their own TPM under their control. This concept is reflected in the recommended capabilities for a mobile TPM2.0 [63], which advise the isolation between stakeholders and their resources and policy-based authorization of stakeholder sensitive data. To realize this multiple stakeholder model, the reference architecture outlines different implementation alternatives. For instance, multiple TPMs within a protected environment like TEE, or virtual TPMs supported by a hypervisor [9], where stakeholders are isolated from each other based on the compartmentalization provided by the TEE's trusted OS or the hypervisor, respectively.

Our particular setting also fits well into the defined multiple stakeholder model: two distinct TPMs co-exist, each with a distinct affinity to a different stakeholder. The fTPM is by design designated to the platform stakeholder (i.e., device manufacturer) and it is bound to the device through the device-specific credentials within the TEE (e.g., eFuses) from which fTPM derives its endorsement key and to which it anchors its key/storage hierarchies. For instance, the fTPM described in [54] is designated entirely to the platform and its services. In contrast, the simTPM is designated to the end-user. This intuition is based on the observation that users use the SIM to authenticate themselves to the mobile network and rather stick to one SIM (i.e., phone number) while changing more frequently the device. Moreover, users have to explicitly authenticate themselves to the SIM card, i.e., their mobile carrier issued PIN. In this setting we are going beyond the initial proposals by the TCG reference architecture by actually as-

Table 2: Migrating user data when switching SIM card or device

	Data bound to device	Data **not** bound to device
New SIM card	Key duplication	Key duplication
New device	`TPM_Authorize` key policy	—

signing two distinct stakeholders to two *physically* separated TPM instances, SIM card versus TEE.

6.2 Switching SIM card or device

Since the SIM card is removable and exchangeable, two scenarios have to be considered: the user switches devices but keeps the SIM card, or the user keeps the device and switches to a new SIM card. How this affects migration of the user data protected with the simTPM is summarized in Table 2 and explained in the following.

Switching device: When switching the mobile device and migrating the user data to a new device, the complexity of the operation is dependent on whether the user bound any data to the device. For instance, during secure boot, BL1 has access to device-specific information like the board_id (or potentially values derived from the device-specific vendor key) that uniquely identifies the current platform. This board_id (like derived values) can be included in the measurements collected during secure boot (see Section 4.3) and allow the simTPM to bind data or keys to this particular platform.[2] If the user did not bind any data/keys to the platform, no further action is required beyond moving the SIM card to the new phone. The entire simTPM state including the key hierarchy is inherently migrated to the new device and can be used to decrypt the user data—i.e., a form of *portable sealed storage*. If the data is bound to the board_id, a new feature of TPM2.0 called `TPM_Authorize` has to be used to avoid the problem of *"brittle policies."* Without `TPM_Authorize`, the user data would be bound to one particular board_id and could never be decrypted on another device. With `TPM_Authorize` different possible board_id values can be signed off as valid for a successful verification of the platform state and, hence, decryption of data migrated with the SIM card. The valid board_id values can be signed off by the user to endorse a new phone to which data should be migrated, or by another entity, like the mobile carrier or the employer in BYOD settings.

Switching SIM card: If the user switches the SIM card and hence moves to another simTPM, all user data has to be migrated to the new SIM card, i.e., the necessary simTPM keys have to be moved to the new simTPM. Independent of whether the user data is bound to the device or not, switching the SIM card requires the simTPM keys used for securing the data to be duplicated to the new simTPM. This is an example scenario for TPM2.0 key duplication to migrate keys and associated data to another TPM and is supported by simTPM.

The bottom line of those two scenarios is that a user that wants to keep the option to migrate data secured with the simTPM to both new SIM cards and new devices should use duplicable keys with `TPM_Authorize`.

[2] Assuming a bond between the RTM and simTPM was established.

7 Discussion

The fTPM [54] is the incumbent deployment for a TPM on mobile devices and was part of the Windows Phone platform. However, it was designed primarily for vendor services and did not specifically target the end-user. In this work, we add to the landscape of mobile trusted computing and advocate using the dormant hardware capabilities of SIM cards to provide (additional) TPM support on mobile devices. Our systematization of related works shows that a simTPM can take a niche among the existing works and, in particular, inherently avoids problems of TEE-based deployments (e.g., protected state or secure clock) that currently require compromises and modifications to the TPM specification (e.g., "dark period" or cooperative checkpointing of fTPM) or that make additional hardware requirements (e.g., replay-protected memory blocks). On the other hand, a movable TPM raises the challenge of how to bind the TPM and the platform RTM. In this work, we proposed using the unique features of mobile devices—secure boot and TEE with device-specific, certified keys—to address this challenge. However, we find that this problem also affects prior solutions, like a vTPM based on a PCI-attached secure co-processor, and our solution might give insights into how to establish the TPM-RTM binding in those prior works.

Our simTPM implementation is based on a physical SIM card, thus it is currently not suitable for phones using eSIM (e.g., Apple iPhone). However, eSIM solutions are supported by separate hardware modules (such as JEDEC SON-8) and it might be worthwhile to investigate how those modules can be extended to implement a full TCG compliant TPM2.0.

Recently, Google introduced their Titan chip [67] as part of their Nexus 3 phones, which shows the need for hardware-backed security features in addition to TEE-based implementations on mobile end-user devices. Similar to the simTPM, Titan chip also provides hardware-backed security for system operations like verified booting as well as a hardware-implemented keystore for apps and users. But Titan is exclusive for Google devices, whereas our simTPM is portable between mobile devices and provides TPM2.0 compliant features. Since implementation details are yet unknown, we excluded the Titan chip from our systematization in Section 3.

8 Conclusion

In this paper we proposed simTPM, a hardware-based TPM implementation for mobile devices using the SIM card. Performance evaluation of our prototype shows that our implementation is comparable with an existing discrete TPM chip. Thus, we think simTPM is a practical solution to add user-centric trusted computing technology to mobile devices without the need to add hardware. A particular challenge of a movable TPM is the binding between TPM and the device RTM, which we addressed through a TEE-proxy or a distance bounding protocol. Future work includes a more detailed and formal write-up of the custom DAA scheme we used in our prototype, since it is particularly fitting for implementation on a smart card. Also future implementations of simTPM in industrial IoT or automotive settings for hardware based attestation could be worthwhile to pursue.

9 Acknowledgment

We are grateful to N. Asokan for his insightful suggestions. We are also thankful to the anonymous reviewers for their valuable comments.

This work is supported by the German Federal Ministry of Education and Research(BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA)(AutSec/FKZ: 16KIS0753) and the CISPA-Stanford Center for Cybersecurity (FKZ: 16KIS0762).

References

[1] Hikey960 android development board. https://www.96boards.org/product/hikey960/. Accessed: 02.08.2018.

[2] IBM's TPM 2.0 TSS. https://sourceforge.net/projects/ibmtpm20tss/. Accessed: 06.08.2018.

[3] Trustzone downgrade attack opens android devices to old vulnerabilities. http://bits-please.blogspot.com/2015/03/getting-arbitrary-code-execution-in.html, March 2015. Accessed: 02.08.2018.

[4] Fritz Alder, N. Asokan, Arseny Kurnikov, Andrew Paverd, and Michael Steiner. S-FaaS: Trustworthy and Accountable Function-as-a-Service using Intel SGX. *CoRR*, abs/1810.06080, 2018.

[5] Arm Limited. Trusted board boot design guide. https://github.com/ARM-software/arm-trusted-firmware/blob/master/docs/trusted-board-boot.rst, March 2018. Accessed: 04.08.2018.

[6] N. Asokan. On secure resource accounting for outsourced computation, 2018. Invited keynote at 3rd Workshop on System Software for Trusted Execution (SysTEX 2018).

[7] Samy Bengio, Gilles Brassard, Yvo G Desmedt, Claude Goutier, and Jean-Jacques Quisquater. Secure implementation of identification systems. *Journal of Cryptology*, 4(3):175–183, 1991.

[8] Gal Beniamini. Getting arbitrary code execution in trustzone's kernel from any context. https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html, July 2017. Accessed: 02.08.2018.

[9] Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vTPM: Virtualizing the Trusted Platform Module. In *Proc. 15th USENIX Security Symposium (SEC '06)*. USENIX Association, 2006.

[10] Thomas Beth and Yvo Desmedt. Identification tokens—or: Solving the chess grandmaster problem. In *Conference on the Theory and Application of Cryptography*, pages 169–176. Springer, 1990.

[11] Dan Boneh and Xavier Boyen. Short signatures without random oracles. In *Advances in Cryptology - EUROCRYPT 2004: International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2004.

[12] Jeremy Boone. Tpm genie: Attacking the hardware root of trust for less than $50, 2018. Accessed: 02/13/2019.

[13] Stefan Brands and David Chaum. Distance-bounding protocols. In *Workshop on the Theory and Application of Cryptographic Techniques on Advances in Cryptology (EUROCRYPT '93)*. Springer, 1994.

[14] Ernie Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In *Proc. 11th ACM Conference on Computer and Communication Security (CCS '04)*. ACM, 2004.

[15] Broadchip. BCT4303 Dual Sim card controller. www.chinesechip.com/files/2015-03/912ed043-de27-4e8a-95f7-c009ad22dd92.pdf. Last accessed: 22/01/19.

[16] Dhiman Chakraborty, Lucjan Hanzlik, and Sven Bugiel. simTPM: User-centric tpm for mobile devices (technical report). *CoRR*, abs/1905.08164, 2019.

[17] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. Detecting privileged side-channel attacks in shielded execution with déjà vu. In *Proc.*

12th ACM Symposium on Information, Computer and Communication Security (ASIACCS '17). ACM, 2017.

[18] Catalin Cimpanu. Trust Issues: Exploiting TrustZone TEEs. https://www.bleepingcomputer.com/news/security/trustzone-downgrade-attack-opens-android-devices-to-old-vulnerabilities/, September 2017. Accessed: 02.08.2018.

[19] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016(086):1–118, 2016.

[20] Cas Cremers, Kasper B Rasmussen, Benedikt Schmidt, and Srdjan Capkun. Distance hijacking attacks on distance bounding protocols. In *Proc. 33rd IEEE Symposium on Security and Privacy (SP '12)*. IEEE Computer Society, 2012.

[21] Kurt Dietrich and Johannes Winter. Towards customizable, application specific mobile trusted modules. In *Proc. 5th ACM workshop on Scalable trusted computing (STC '10)*. ACM, 2010.

[22] Saar Drimer, Steven J Murdoch, et al. Keep your enemies close: Distance bounding against smartcard relay attacks. In *Proc. 16th USENIX Security Symposium (SEC '07)*. USENIX Association, 2007.

[23] J. Ekberg, K. Kostiainen, and N. Asokan. The untapped potential of trusted execution environments on mobile devices. *IEEE Security Privacy*, 12(4):29–37, July 2014.

[24] Jan-Erik Ekberg. *Securing Software Architectures for Trusted Processor Environments*. PhD thesis, Aalto University, Helsinki, Finland, 2013.

[25] Jan-Erik Ekberg and Sven Bugiel. Trust in a small package: Minimized MRTM software implementation for mobile secure environments. In *Proc. 4th ACM workshop on Scalable trusted computing (STC '09)*. ACM, 2009.

[26] Paul England and Talha Tariq. Towards a programmable TPM. In *Proc. 2nd International Conference on Trust and Trustworthy Computing (TRUST '09)*. Springer, 2009.

[27] ETSI. TS 151 011 V4.15.0 (2005-06) Technical Specification Digital cellular telecommunications system (Phase 2+); Specification of the Subscriber Identity Module - Mobile Equipment (SIM-ME) interface (3GPP TS 51.011 version 4.15.0 Release 4). https://www.etsi.org/deliver/etsi_ts/151000_151099/151011/04.15.00_60/ts_151011v041500p.pdf. Last accessed: 22/01/19.

[28] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Proc. on Advances in cryptology (CRYPTO '86)*. Springer, 1987.

[29] Seunghun Han, Wook Shin, Jun-Hyeok Park, and HyoungChun Kim. A bad dream: Subverting trusted platform module while you are sleeping. In *Proc. 27th USENIX Security Symposium (SEC' 18)*. USENIX Association, 2018.

[30] Intel. Software guard extensions sdk: sgx_create_monotonic_counter. https://software.intel.com/en-us/sgx-sdk-dev-reference-sgx-create-monotonic-counter, May 2018.

[31] Intel Developer Zone. Platform Service Enclave and ME for Intel Xeon Server. https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/806502. Last accessed: 20/05/19.

[32] Jin Soo Jang, Sunjune Kong, Minsu Kim, Daegyeong Kim, and Brent Byunghoon Kang. SeCReT: Secure channel between rich execution environment and trusted execution environment. In *Proc. 22nd Annual Network and Distributed System Security Symposium (NDSS '15)*. The Internet Society, 2015.

[33] Timo Kasper, David Oswald, and Christof Paar. Information security applications. chapter EM Side-Channel Attacks on Commercial Contactless Smartcards Using Low-Cost Equipment, pages 79–93. Springer, 2009.

[34] Bernhard Kauer. Oslo: Improving the security of trusted computing. In *Proc. 16th USENIX Security Symposium (SEC '07)*. USENIX Association, 2007.

[35] Oliver Kömmerling and Markus G. Kuhn. Design principles for tamper-resistant smartcard processors. In *Proc. 1st Workshop on Smartcard Technology (Smartcard 1999)*, 1999.

[36] Nate Lawson. Tpm hardware attacks. https://rdist.root.org/2007/07/16/tpm-hardware-attacks/, July 2007. Accessed: 06.08.2018.

[37] Hongliang Liang and Mingyu Li. Bring the Missing Jigsaw Back: TrustedClock for SGX Enclaves. In *Proc. 11th European Workshop on Systems Security (EuroSec'18)*. ACM, 2018.

[38] Linear Technology. LTC4558 - Dual SIM/Smart Card Power Supply and Interface. https://www.analog.com/media/en/technical-documentation/data-sheets/4558fa.pdf. Last accessed: 22/01/19.

[39] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In *Proc. 25th USENIX Security Symposium (SEC' 16)*. USENIX Association, 2016.

[40] Junrong Liu, Yu Yu, François-Xavier Standaert, Zheng Guo, Dawu Gu, Wei Sun, Yijie Ge, and Xinjun Xie. Small tweaks do not help: Differential power analysis of milenage implementations in 3g/4g usim cards. In *Proc. 20th European Symposium on Research in Computer Security (ESORICS 2015)*. Springer, 2015.

[41] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. Boomerang: Exploiting the semantic gap in trusted execution environments. In *Proc. 24th Annual Network and Distributed System Security Symposium (NDSS '17)*. The Internet Society, 2017.

[42] MAOSCO Limited. Multos standard c-api. https://www.multos.com/uploads/CAPI.pdf, 2016. Accessed: 02.08.2018.

[43] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: Rollback protection for trusted execution. In *Proc. 26th USENIX Security Symposium (SEC' 17)*. USENIX Association, 2017.

[44] Sjouke Mauw, Zach Smith, Jorge Toro-Pozo, and Rolando Trujillo-Rasua. Distance-bounding protocols: Verification without time and location. In *Proc. 39th IEEE Symposium on Security and Privacy (SP '18)*. IEEE Computer Society, 2018.

[45] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for tcb minimization. In *Proc. 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems (Eurosys '08)*. ACM, 2008.

[46] Brian McGillion, Tanel Dettenborn, Thomas Nyman, and N. Asokan. Open-tee – an open virtual trusted execution environment. In *Proc. IEEE Trustcom/BigDataSE/ISPA - Volume 01 (TRUSTCOM '15)*. IEEE Computer Society, 2015.

[47] Thomas S. Messerges and Ezzy A. Dabbish. Investigations of power analysis attacks on smartcards. In *Proc. 1st Workshop on Smartcard Technology (Smartcard 1999)*, 1999.

[48] Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan. Power analysis attacks of modular exponentiation in smartcards. In *Proc. First International Workshop on Cryptographic Hardware and Embedded Systems (CHES'99)*, 1999.

[49] Microsoft. Secure the windows 10 boot process. https://docs.microsoft.com/en-us/windows/security/information-protection/secure-the-windows-10-boot-process, October 2017. Last accessed: 08/06/18.

[50] Wojciech Mostowski and Erik Poll. Malicious code on java card smartcards: Attacks and countermeasures. In *Proc. 8th IFIP WG 8.8/11.2 International Conference on Smart Card Research and Advanced Applications (CARDIS '08)*, 2008.

[51] Zhenyu Ning and Fengwei Zhang. Understanding the security of arm debugging features. In *Proc. 40th IEEE Symposium on Security and Privacy (SP '19)*. IEEE Computer Society, 2019.

[52] Karsten Nohl. Rooting sim cards. https://media.blackhat.com/us-13/us-13-Nohl-Rooting-SIM-cards-Slides.pdf, 2013. Blackhat USA 2013.

[53] Bryan Parno. Bootstrapping trust in a "trusted" platform. In *Proc. 3rd Conference on Hot Topics in Security (HOTSEC'08)*. USENIX Association, 2008.

[54] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Löser, Dennis Mattoon, Magnus Nyström, David Robinson, Rob Spiger, Stefan Thom, and David Wooten. fTPM: A Software-Only Implementation of a TPM Chip. In *Proc. 25th USENIX Security Symposium (SEC' 16)*. USENIX Association, 2016.

[55] Nilo Redini, Aravind Machiry, Dipanjan Das, Yanick Fratantonio, Antonio Bianchi, Eric Gustafson, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Bootstomp: on the security of bootloaders in mobile devices. In *Proc. 26th USENIX Security Symposium (SEC' 17)*. USENIX Association, 2017.

[56] Dan Rosenberg. Reflections on trusting trustzone. https://www.blackhat.com/docs/us-14/materials/us-14-Rosenberg-Reflections-on-Trusting-TrustZone.pdf, 2014. Accessed: 02.08.2018.

[57] Di Shen. Exploiting trustzone on android. https://www.blackhat.com/docs/us-15/materials/us-15-Shen-Attacking-Your-Trusted-Core-Exploiting-Trustzone-On-Android-wp.pdf, 2015. Accessed: 02.08.2018.

[58] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: Exposing the perils of security-oblivious energy management. In *Proc. 26th USENIX*

Security Symposium (SEC' 17). USENIX Association, 2017.

[59] Christopher Tarnovsky. Deconstructing a 'secure' processor, 2010. BlackHat DC.

[60] The Chromium Projects. TPM Usage. https://www.chromium.org/developers/design-documents/tpm-usage. Last accessed: 08/06/18.

[61] Trusted Computing Group. Mobile phone work group mobile trusted module specification. https://trustedcomputinggroup.org/resource/mobile-phone-work-group-mobile-trusted-module-specification/, 2010.

[62] Trusted Computing Group. Tpm main part 1 design principles. https://trustedcomputinggroup.org/wp-content/uploads/TPM-Main-Part-1-Design-Principles_v1.2_rev116_01032011.pdf, 2011.

[63] Trusted Computing Group. Tpm 2.0 mobile reference architecture specification. https://trustedcomputinggroup.org/resource/tpm-2-0-mobile-reference-architecture-specification/, 2014.

[64] Trusted Computing Group. Tpm 2.0 library specification. https://trustedcomputinggroup.org/resource/tpm-library-specification/, 2016.

[65] David Wagner. Gsm cloning. http://www.isaac.cs.berkeley.edu/isaac/gsm.html. Last accessed: 02/13/19.

[66] Johannes Winter. Trusted computing building blocks for embedded linux-based arm trustzone platforms. In *Proc. 3rd ACM workshop on Scalable trusted computing (STC '08)*. ACM, 2008.

[67] Xiaowen Xin. Titan M makes Pixel 3 our most secure phone yet. https://blog.google/products/pixel/titan-m-makes-pixel-3-our-most-secure-phone-yet/, October 2018. Accessed: 13.11.2018.

A Binding RTM with distance bounding

In Section 4.3 we discussed using the TEE as proxy in order to assert the authenticity of the RTM and mitigate the risks of a relay attack. Another way to bind the simTPM with its RTM is by using a distance bounding (DB) protocol [7, 10, 13]. Distance bounding is widely used for card-based payment systems. When a credit card is punched to the card reader, the reader runs a distance bounding protocol to check the proximity of the card to prevent a possible relay attack. We are facing the opposite scenario, in which the card is trying to assert the proximity of the device where the communication partner, here the RTM, resides.

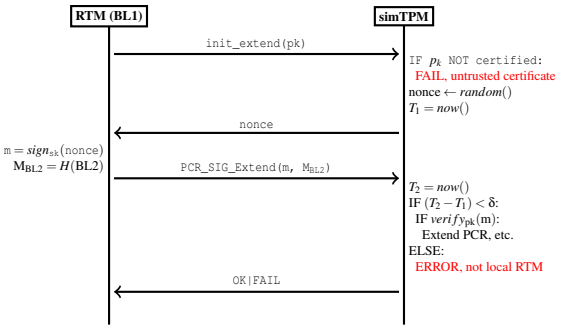


Figure 5: Prototypical distance bounding protocol for binding local RTM (BL1) and simTPM

Prototypical distance bounding: We assume the device vendors equipped the simTPM with certificates for their device-specific keys, which allows a verifier to distinguish trusted code with access to such secrets (e.g., early bootstages, like BL1 or the TEE) from untrusted code, like the host OS or apps. To assert the proximity of the RTM, only the very first measurement provided to the simTPM, i.e., the measurement by BL1 (RTM) of BL2, has to be checked for proximity. After that, the chain of trust of an authenticated boot will transitively extend this trust into the proximity of the RTM. Figure 5 illustrates a prototypical protocol for our scenario. We consider a two-step PCR extension by the RTM for verifying the proximity: (1) the RTM provides the public key *pk* of its device-specific key (or a key derived from it) to the TPM, which then can verify the authenticity of the RTM using the vendor-supplied certificate; (2) as in other distance bounding protocols, the simTPM (verifier) challenges the RTM (prover) with a nonce to which the RTM replies with the signed nonce value (using the authenticated private key) as well as the PCR extension arguments. If this reply of the signed nonce is received within a time threshold T and the signature verifies, simTPM assumes the RTM to be local and extends the PCR with the supplied measurement value M_{BL2}; if either condition fails, the simTPM aborts. For robustness of the protocol, the challenge-response can be repeated N times to decrease the chances of a legitimate, local RTM failing the threshold.

Prototypical setup: In general, calculating the threshold for distance bounding is difficult, because various factors can influence the response time. For instance, jitters of the network over which the verifier and prover communicate, interrupts of the prover's computation, cache and memory delays, etc. might introduce a high uncertainty of the expectable response time. At first glance, our particular scenario seems very favorable for a distance bounding protocol, since the prover (RTM) is the BL1 that has exclusively control of the CPU without interrupts or interference of an OS; and the RTM is connected

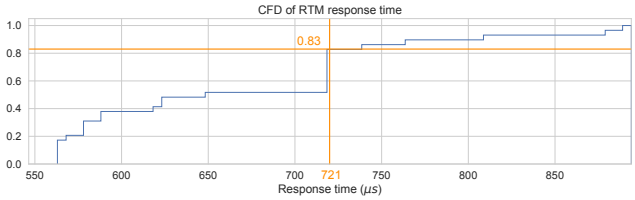


Figure 6: Cumulative frequency distribution of the RTM response time in our measurements ($N = 30$)

to the SIM card over a 480 mbps USB 2.0 bus, in modern devices even via a 5 gbps USB 3.0 bus, with no parallel transfers, providing favorable circumstances for a challenge-response protocol and small error-margin in which an attacker has to fall for a successful, undetected relay attack [20, 22, 44].
The SIM card is connected to the phone through a reader, which is directly connected to the baseband processor. The reader powers the smart card and provides it with the baseband's clock. The clock duty cycle shall be between 40% and 60% of the period during stable operation [27]. Modern smart cards support clock stop to allow preservation of power, which an attacker could use to tamper with the verifier's perception of time. However, this feature can be disabled by initializing the card as clock stop not allowed by setting the *VERIFY CHV* command to 0. Disabling this feature will increase the phone's battery consumption, but not in a significant amount, since the maximum current consumption of an idle SIM card should not exceed $200\mu A$.
The SIM card and the reader connection are in a contact connection and generally interfaces within $20ns$ [15, 38]. The reader connects to the baseband processor through Non-Level-Shifted bidirectional I/O. The connection in our test setup goes through an USB 2.0 bus with 480 mbps. Communication between SIM card and the CPU via this bus ranges between $35ns$ to $72ns$.

Measurements and threshold: We conducted measurements on our test device to evaluate the feasibility of distance bounding to bind the RTM and simTPM. We measured 30 times[3] the speed of the prover (RTM) for calculating the response to the challenge (64 bits nonce) using ECC with the NIST P-256 curve. In our test, the responses took 563–894μs, and the average response time was $669.759 \pm 49.804\mu s$ for a confidence level of 99%. Figure 6 shows the CFD of the RTM response time, where 83% of all responses were $\leq 721\mu s$ and 93% of all responses were $\leq 812\mu s$. The success chance of the distance bounding protocol P_{DB} for a single round is the cumulative probability sampled over the frequency distribution in Figure 6. If we were to set the threshold T for successful distance bounding to 721 μs:

[3] A single measurement requires ≈5min, since only a single measurement per power-cycle is possible on our test device.

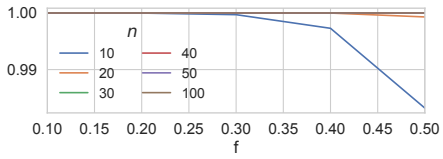


Figure 7: Success probability of local RTM for distance bounding depending on f and n for $T = 721\mu s$ ($p = 0.83$)

$$P_{DB} = \Pr[x \leq 721] = \sum_{i=563}^{721} \Pr[x = i] \approx 0.83$$

where 563 μs is the lowest latency in our dataset. Going below 721 μs reduces the probability of a successful bounding protocol for legitimate devices, i.e., to 0.52 for a threshold of $649\mu s$. To increase the chances for local RTM to pass the distance bounding check, a successful verification usually requires that the response is below T for a sufficient fraction f of the responses, means at-least $f \times n$ out of n responses should arrive within T. When modeling the challenge-response game as binomial distribution and requiring $f \times n$ responses within 721 μs out of n responses (i.e., success probability $P_{DB} = 0.83$), the cumulative probability distribution is:

$$\Pr[x \geq fn] = \sum_{i=fn}^{n} \binom{n}{i} (p)^i (1-p)^{n-i} \text{ where } p \in \{P_{DB}\}$$

Figure 7 shows the success probabilities for different choices of f and n. An optimal choice minimizes n (lower overall runtime overhead for the protocol) while maximizing $\Pr[x \geq fn]$ and minimizing the chance of the attacker to successfully relay. We have observed from our dataset that setting $f = 0.47$ for $n = 30$ (i.e., 14 out of 30 runs) offers a success rate $\Pr[x \geq fn]$=0.99999724049 for local RTM.

Attacker chances: The APDU package for the challenge is 112 bits and for the response 304 bits, which are transferred virtually instantly between verifier and prover ($\leq 1\mu s$). Thus, the response time measured in Figure 6 consists virtually only of the processing time of the RTM, which an attacker cannot speed up (see Figure 3). As a consequence, if an attacker requires more than $721 - 563 = 158\mu s$ to relay the challenge and the response, the relay attack has no chance of winning, since the RTM in our tests required at least $563\mu s$ to compute the response. Assuming a packet size of 55 bytes (minimal Ethernet frame size, IP header, and UDP package with 1 byte payload for the nonce/response), the attacker needs at least a relay bandwith of ≈ 5.87 mbps to have *any* chance of winning, which is a very reasonable assumption. Hence, attacks against this distance bounding are feasible. From our measurements it is hard to concretely model the attacker, however, the attack chance is already 0.1% when relaying via Ethernet and an IP network (55 bytes datasize) with a bandwith of ≈ 49 mbps, or when relaying only the APDU data of 14 bytes (e.g., via a custom build connection) with ≈ 10 mbps.

The Betrayal At Cloud City: An Empirical Analysis Of Cloud-Based Mobile Backends

Omar Alrawi*
Georgia Institute of Technology

Chaoshun Zuo*
The Ohio State University

Ruian Duan
Georgia Institute of Technology

Ranjita Pai Kasturi
Georgia Institute of Technology

Zhiqiang Lin
The Ohio State University

Brendan Saltaformaggio
Georgia Institute of Technology

Abstract

Cloud backends provide essential features to the mobile app ecosystem, such as content delivery, ad networks, analytics, and more. Unfortunately, app developers often disregard or have no control over prudent security practices when choosing or managing these services. Our preliminary study of the top 5,000 Google Play Store free apps identified 983 instances of N-day and 655 instances of 0-day vulnerabilities spanning across the software layers (OS, software services, communication, and web apps) of cloud backends. The mobile apps using these cloud backends represent between 1M and 500M installs each and can potentially affect hundreds of thousands of users. Further, due to the widespread use of third-party SDKs, app developers are often unaware of the backends affecting their apps and where to report vulnerabilities. This paper presents SkyWalker, a pipeline to automatically vet the backends that mobile apps contact and provide actionable remediation. For an input APK, SkyWalker extracts an enumeration of backend URLs, uses remote vetting techniques to identify software vulnerabilities and responsible parties, and reports mitigation strategies to the app developer. Our findings suggest that developers and cloud providers do not have a clear understanding of responsibilities and liabilities in regards to mobile app backends that leave many vulnerabilities exposed.

1 Introduction

Cloud-based mobile backends provide a wide array of features, such as ad networks, analytics, content delivery, and much more. These features are supported by multiple layers of software and multiple parties including content delivery networks (CDNs), hosting providers, and cloud providers who offer virtual/physical hardware, provisioned operating systems, and managed platforms. Due to the inherent complexity of cloud-based backends, deploying and maintaining them securely is challenging. Consequently, mobile app developers often disregard prudent security practices when choosing cloud infrastructure, building, or renting these backends.

Recent backend breaches of the British Airways [1] app and Air Canada [2] app demonstrate how wide-spread these incidents are. More recently, the hijacking of the Fortnite mobile game [3] showed how incrementally-downloaded content from mobile backends can allow an attacker to install additional mobile apps without the user's consent. Additional cases [4] involving the exposure of 43TB of enterprise customer names, email addresses, phone numbers, PIN reset tokens, device information, and password lengths was due to *insecure mobile backends* and not the developer's mobile app code.

Even for security-conscious developers, it is not clear what backends their mobile app will interact with because of third-party libraries. Third-party libraries do not expose their backends to developers, instead, they offer an application program interface (API) that developers use. Many of these vulnerabilities can be identified ahead of time if developers have the right tools and resources to evaluate the security of their backends. Further, identifying vulnerable software layers and the responsible party can expedite remediation and therefore lower the risk of exposure.

To deal with the complexities in cloud infrastructure, the research community surveyed [5] and proposed several taxonomies [6], ontologies [7], assessment models [8], and threat classifications [9]. Unfortunately, these approaches provide few practical recommendations for mobile app developers. Recent works on server-side vulnerability discovery of mobile apps [10]–[12] have shown that a lack of security awareness among app developers is a growing problem. Yet, these works only scratch the surface by examining only the software service layer of mobile backends.

A systematic study is needed to identify the most pressing issues facing mobile backends. Moreover, to conduct such a study, the analysis must be reproducible, transparent, and easy to interpret for developers. The study should be done on a representative mobile app ecosystem to provide real in-

*Authors contributed equally.

sight into the backend vulnerability landscape. Finally, the study should offer practical steps to guide and inform app developers on the security of their mobile backends.

To this end, this paper presents the design and implementation of SkyWalker, an analysis pipeline to study mobile backends. Using SkyWalker, we conducted an empirical analysis of the top 5,000 free mobile apps in the Google Play store from August 2018. Based on this study, we uncovered 655 0-day instances and 983 N-day instances affecting thousands of apps. We used Google Play Store metadata to measure the impact of our findings and estimate the number of affected users. We propose mitigation strategies for different types of vulnerabilities and guidelines for developers to follow. Lastly, we offer the SkyWalker analysis pipeline as a free public web-service to help developers identify what backends their mobile apps interact with, the security state of the backends, and recommendations to address any detected issues.

Our empirical study found 983 N-day instances of 52 vulnerabilities affecting hypervisors, operating systems, databases, mail servers, DNS servers, web servers, scripting language interpreters, and others. We found 655 0-day instances of SQL injection (SQLi), cross-site-scripting (XSS), and external XML entity (XXE). These affected thousands of mobile apps, with some apps having over 50M+ installs and more than 332,000 reviews. We present two case studies to demonstrate the vulnerabilities affecting a specific developer and vulnerabilities affecting a platform that is used by many developers.

We found these backends to be geographically distributed across the globe and hosted on 6,869 different networks. We notified all affected parties about the findings, and were careful to follow ethical and legal guidelines when conducting this study, additional details are in Section 8. We propose mitigation strategies for developers to follow based on the issues found and the types of backends. We conclude with recommendations for deploying and maintaining secure backends.

2 A Motivating Example

Mobile apps use cloud-based backend services to support extensive functions like ads, telemetry, content delivery, and analytics. Unfortunately, a mobile app developer who wants to audit the backends their app uses will quickly find that this is harder than it seems. The first thing the developer must do is simply enumerate those mobile backends. Consider the *Crime City Real Police Driver* (*com.vg.crazypoliceduty*) app, a mobile game with over 10M+ installs and 126,257 reviews. The mobile app uses several third-party SDK libraries including Amazon In-App Billing, SupersonicAds, Google AdMob, Unity3D, Nuance Speech Recognition Kit, and Xamarin Mono. The developer may not be aware of many of the backends that are invoked from imported native or Java libraries, i.e., the Unity3D backends. In most cases, the developer will first have to employ static binary analysis tools or dynamically instrument the app to track multiple levels of SDK inclusion. SkyWalker automatically identified 13 unique backends from this app's APK (shown in Table 1) and mapped them to the modules they were found in, i.e., library backends versus developer backends.

Party	Vendor	Backend	Purpose
Hybrid	Vasco Games	androidha.vascogames.com	Game Content
Third	Unity3D	api.uca.cloud.unity3d.com	Telemetry
		cdn-highwinds.unityads.unity3d.com	Ads
		config.uca.cloud.unity3d.com	Telemetry
		impact.applifier.com	Telemetry
	Sizmek	bs.serving-sys.com	Ads
		secure-ds.serving-sys.com	Ads
	Moat	px.moatads.com	Analytics
		z.moatads.com	Ads
	Google	googleads.g.doubleclick.net	Ads
		pagead2.googlesyndication.com	Ads
		tpc.googlesyndication.com	Ads
		www.google-analytics.com	Analytics

Table 1: Backends identified for *Crime City Real Police Drive* and their purpose. The red cells indicate vulnerable backends.

Backends have layers of software (components) that support the web application software (*AS*), including an operating system (*OS*), software services (*SS*), and communication services (*CS*). The developer now has to fingerprint the backends to inventory the software layers and identify the software type, version, and its purpose. Using this information, the developer can then check to see if any of their software is outdated or affected by a known vulnerability [13], a laborious and time-consuming task. SkyWalker identified that the game content backend runs *Debian 6* for the *OS*; *OpenSSH 6.5p1*, *Apache httpd 2.2.22*, *PHP/5.4.4-14*, and *Apache-Coyote/1.1* for the *SS*; and uses the *HTTP* protocol for *CS*. SkyWalker's search of the national vulnerability database (NVD) [13] and correlation with the fingerprint results showed multiple common vulnerability exposure (CVE) entries affecting PHP 5.4.4-14. Further, the Debian version running on the backend is no longer supported and does not receive any updates from the vendor.

In addition to these issues, the developer's *AS* can contain bugs that must be audited. The developer can check the *AS* by auditing the parameters passed to each API and testing for SQLi, XSS, XXE, or any other applicable vulnerabilities from OWASP's top 10 common issues [14]. This task requires secure programming experience and security domain expertise to identify bugs in the source code. SkyWalker found that the game content backend interface is vulnerable to SQLi for some parameters passed by the mobile app, which is due to the *AS* not properly sanitizing the input.

The developer must now remediate or mitigate these risks, but each backend layer may be operated by different entities that provide hardware and software as a service. Therefore,

before fixing any issues, they must figure out what party is responsible for each component. SkyWalker fingerprinted the *Crime City Real Police Driver* game content backend, *androidha.vascogames.com*, as being hosted on a Google Compute Engine Flexible Environment instance (which provides virtual hardware, operating system, and PHP). We refer to this type of backend model as *hybrid* since Google is partially responsible for the virtual environment and the developer is responsible for the *AS* and *CS*.

The developer must come up with a remediation strategy to address these problems. Google advertises that they patch any vulnerable software affecting the *OS* and *SS*, but this is only applicable to non-deprecated versions. In the case of *Crime City Real Police Driver* app, the developer is responsible for all the software layers since the *OS* and *SS* versions are deprecated. The developer must upgrade to a supported *OS*, apply patches to the PHP interpreter (*SS*), patch the *AS* source code against SQLi, and support *HTTPS* for secure *CS*.

The Unity3D, Sizmek, and Moat backends shown in Table 1 are called *third-party*, since the developer has no control over them. This evaluation must also be carried out on third-party backends to identify additional vulnerabilities (potentially affecting all apps which use those shared services). SkyWalker found that the *Crime City Real Police Driver* app uses the *config.uca.cloud.unity3d.com* backend, which contains an XXE vulnerability, and the *bs.servingsys.com* backend that contains an XSS vulnerability. Ideally, the developer could report those vulnerabilities to the platform through a bug bounty program or migrate their app to backends that are not vulnerable.

This manual assessment procedure is very involved and requires extensive security domain knowledge, which many app developers may not have. Instead, SkyWalker gives all mobile app developers the ability to identify the backends invoked by their app, assess their software layers, and suggest remediation strategies to improve the security for their mobile app backends.

3 Background

This section defines an abstraction to model mobile backends for our empirical study. We also define our labeling for backends and create a mapping between responsible stakeholders and resources. We outline how we count vulnerabilities and define them in the context of this work.

3.1 Mobile App Backend Model

We follow the standard definition for mobile backends used by industry leaders [15]–[18], which encompass many cloud features, such as storage, user management, notifications, and APIs for various services, regardless of who maintains/owns them. We breakdown mobile backends into a stack representation that consists of five layers:

- **Hardware** (*HW*) refers to the physical or virtual hardware that hosts the backend.
- **Operating System** (*OS*) refers to the OS running on the hardware, i.e., Linux or Windows.
- **Software Services** (*SS*) refers to software services running in the OS, i.e., database service, web service, etc.
- **Application Software** (*AS*) refers to the custom application interface used by mobile apps to interact with the running services.
- **Communication Services** (*CS*) refers to the communication channel supported between the mobile app and the mobile backend.

Our approach does not consider the hardware layer because 1) we would need root-level access on the backend to evaluate the hardware and 2) mobile app developers have no direct way of addressing hardware vulnerabilities, i.e., manufacturers must issue firmware updates or replace the hardware. It is important to note that this work does not consider the mobile app security, instead we leverage the mobile app to study the backend.

We differentiate between mobile backends by ownership, which provides a granular mapping between stakeholders and resources. We define four labels for the mobile backends with respect to the app developer:

- **First-Party** (B_{1st}) refers to backends that are fully managed by the mobile app developers (i.e., full control over the backend).
- **Third-Party** (B_{3rd}) refers to backends that are fully managed by third-parties (i.e., no control over the backend).
- **Hybrid** (B_{hyb}) refers to backends that are co-managed by third-parties and developers such as cloud infrastructure (i.e., some control over the mobile backend).
- **Unknown** (B_{ukn}) refers to backends that ownership could not be established with high confidence.

In our model, there are two primary stakeholders, the app developers (*D*) and the cloud service providers (*SP*). There are additional stakeholders, like app users and internet service providers (ISP), but they do not have direct remediation oversight. We define a mapping between backends layers, labels, and ownership, shown in Table 2.

The final piece of the model is the mitigation component that maps vulnerable backends to the proper mitigation strategies. There are five mitigation strategies for developers:

- **Upgrade** (*u*) the software to vendor supported versions.
- **Patch** (*p*) vulnerable software with a vendor patch.

Label	*HW*	*OS*	*SS*	*AS*	*CS*
First-Party (B_{1st})	○	○	○	○	○
Third-Party (B_{3rd})	●	●	●	●	●
Hybrid (B_{hyb})	●	◑	◑	○	○

Table 2: Backend labels (first-party - B_{1st}, third-party B_{3rd}, and hybrid - B_{hyb}) and cloud layers (hardware - *HW*, operating system - *OS*, software services - *SS*, application software - *AS*, and communication services - *CS*) mapping to stakeholders (developers - ○, service providers - ●, and shared - ◑)

- **Block** (*b*) incoming internet traffic to exposed services.
- **Report** (*r*) the vulnerability to the responsible party.
- **Migrate** (*m*) the backend to secure infrastructure.

In many cases, the developer may not have control or authority to fix the issues but still has the option to report it (*r*) or change service provider (*m*).

3.2 Counting Vulnerabilities

This work considers vulnerabilities which are software bugs that exist in the backend software stack, including the operating system (*OS*), services (*SS*), application (*AS*), and communication (*CS*). We consider *N-Day* vulnerabilities to be those vulnerabilities which have an associated common vulnerabilities and exposure (CVE) number assigned by the national institute of standards and technology (NIST) and indexed in the national vulnerability database (NVD) [13]. In our findings, we count *N-Day* vulnerabilities by class and instance, where class refers to the CVE number of a particular vulnerability and an instance refers to the vulnerability affecting a specific interface or software component on a mobile backend. For example, Apache Struts vulnerability *CVE-2017-5638* that affects Apache Struts 2.3.*x* before 2.3.32 and 2.5.*x* before 2.5.10.1 is counted as a single vulnerability (class), but it can affect multiple backends that run different versions of Apache Struts (instances).

Some software versions are affected by multiple CVEs, in this case, we **do not** count every CVE as an instance. We generally assume patching the latest CVE should address all previous unpatched CVEs. We only consider the latest CVE affecting the vulnerable software and count it once. Further, a vulnerability instance is a tuple of the backend's domain name, IP address, and the vulnerable software version. As for *0-Day* vulnerabilities, they are associated with the software application (*AS*) running on the backend. This work looks at three classes of *0-Day* vulnerabilities, SQLi, XSS, and XXE and counts each instance per API interface endpoint on the mobile backend. The defined model, labels, mitigations, mappings, and vulnerabilities are the basis for our methodology, which we describe next.

4 Methodology

In this section, we provide an overview of our assessment and details about implementing SkyWalker. Figure 1 is an overview of SkyWalker's internal components. We divide the implementation into four phases, namely binary analysis, labeling, fingerprinting, and vulnerability analysis. Each phase provides input to the next phase, starting from an input app APK to the final vulnerability/mitigation report.

4.1 Binary Analysis

SkyWalker leverages our prior work, Smartgen [19], to perform the binary analysis and extract query messages from an APK binary. SkyWalker dynamically executes the code paths to the network functions and extracts the native usage of the backend APIs. The native usage of an API includes the URI path and their parameter types/values.

4.2 Backend Labels

Backend labeling assigns one of the four labels defined in our model. The labels are used to map the responsible parties and the mitigation strategies needed (excluding unknown), shown in Table 2. Moreover, the labels are used to identify where the most common issues are found. To perform the labeling, we curate three unique lists using the ipcat [20] datacenter dataset. The first list is called *CP* and contains cloud providers, content delivery networks (CDNs), and mobile platform cloud services. The second list, *Colo*, contains a list of collocation centers. The third list is a list of *SDK* libraries that we extracted using LibScout [21] (Table 3), which help SkyWalker identify third-party backends. OSSPolice [22] provides a more comprehensive list, including native libraries used by the mobile app, but our binary analysis technique only instruments Java code, therefore, we limit the third-party SDK identification to LibScout.

To perform the labeling we generate a tuple for each extracted backend B that contains the effective-second level domain d, IP address ip, a boolean flag lib indicating if the backend belongs to an SDK library, and the developer or vendor name v. We define a function $owner()$ that parses WHOIS, MaxMind [23], and ASN records to extract ownership information. The $owner()$ function uses text tokenization, normalization, and aliasing to consolidate varying records.

SkyWalker uses Algorithm 1 to assign labels to each backend. Algorithm 1 takes as input a list of backends, β, containing tuples $B = \{d, ip, lib, v\}$ and returns a list of labeled backends β'. The algorithm uses the *CP* and *Colo* list to check membership for the domains and IPs to determine the appropriate label. The first check is to determine the origin of the backend (was it extracted from an SDK library?) then

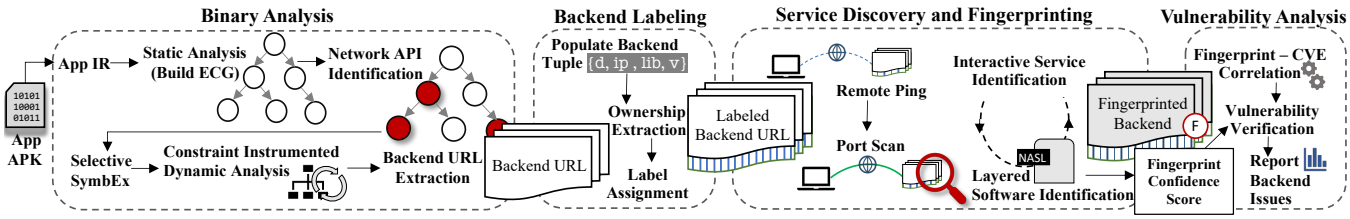


Figure 1: SkyWalker Overview. Phase 1 (Binary Analysis) extracts backend URLs through a dynamic binary instrumentation technique. Phase 2 labels backends into first-party, third-party, and hybrid. Phase 3 discovers and fingerprints the backend services to collect cloud layer information. Phase 4 (vulnerability analysis) uses the fingerprints and correlates them with public vulnerabilities to identify vulnerable backends.

Algorithm 1: Assigning Labels to Backends

Input: β = List of backend tuple $B = \{d, ip, lib, v\}$
Output: β' = Ownership labeled backend list
SDK: List of backend domains found in the SDK libraries;
CP: List of cloud and hosting providers (domains, net prefix, and ASNs);
Colo: List of collocation providers (domains, net prefix, and ASNs);
for $\forall B \in \beta$ **do**
  if $B.lib \vee B.d \in SDK$ **then**
    `// Backend from Java lib`
    $B.label \leftarrow$ "third-party";
    continue
  end
  if $owner(B.d) \neq v \wedge owner(B.d) \notin CP$ **then**
    `// Backend domain not owned by developer or CP`
    $B.label \leftarrow$ "third-party";
    continue
  end
  if $B.ip \in CP$ **then**
    `// Backend IP hosted by cloud provider`
    $B.label \leftarrow$ "hybrid";
    continue
  end
  if $B.ip \in Colo$ **then**
    `// Backend IP hosted by collocation center`
    $B.label \leftarrow$ "first-party";
    continue
  end
  $B.label \leftarrow$ "unknown";
end

assigns "third-party" label if *lib*'s value is true or the backend domain belongs to the list of SDK backends.

If none of the previous statements are true about the domain, then SkyWalker checks the *IP* membership against the *CP* and *Colo* list. If the IP address belongs to a network on the *CP* list SkyWalker assigns "hybrid" label. If the IP address belongs to a network on the *Colo* list SkyWalker assigns "first-party" label. Otherwise, SkyWalker assigns an "unknown" label since it cannot be determined. It is important to note that SkyWalker's labeling approach relies on LibScout [21] to identify third-party backends based on the SDK libraries. SkyWalker performs an additional check before setting the *lib* flag to exclude *SDK* libraries built by the same vendor (Google, Facebook, etc.).

4.3 Service Discovery and Fingerprinting

Service discovery identifies internet-facing services on backends and fingerprinting identifies the software type, version, and configuration of each service. Our approach is a multi-tier approach that starts by remotely pinging the backend, then port scanning it, then interacting with the discovered service, and finally collecting service configurations. For instance, the scan first checks to see if the host is reachable, then it scans for all ports to identify available services, then it tries to connect to the service to collect its banner, and finally, if the services use TLS/SSL, it would collect their configurations and supported ciphers. For each step, our scanner is configured to be non-intrusive, throttled (slow scan speed and a light load on the remote server), and conservative (using techniques that yield low to no false positives).

First, SkyWalker groups all IP addresses into their network prefixes and in a random order picks a prefix and a random IP from the selected prefix to scan. Prefixes are grouped by the autonomous system number (ASN) for each network. If a network spans multiple ASNs, SkyWalker keeps each ASN as a separate prefix to distribute the scanning uniformly across different IP segments. SkyWalker does a TCP ping against common service ports (FTP, SSH, HTTP/S, IMAP, SMTP, RDP, etc.) by sending out a SYN packet followed by

Third-Party SDKs

ACRA	CleverTap	InMobi	Supersonic
AMoAd	Crashlytics	JSch	Syrup
AdColony	Crittercism	Joda-Time	Tapjoy
AdFalcon	Dagger	MdotM	Tremor Video
Adrally	EventBus	Millennial Media	Twitter4J
Amazon	ExoPlayer	Mixpanel	Urban-Airship
Android	Facebook	MoPub	Vungle
Apache	Firebase	New-Relic	WeChat
AppBrain	Flurry	OkHttp	flickrj
AppFlood	Fresco	Parse	heyZap
AppsFlyer	Fyber	Paypal	ironSource
BeaconsInSpace	Google	Picasso	jsoup
Bolts	Gson	Pollfish	roboguice
Brightroll	Guava	Retrofit	scribe
Butter-Knife	Guice	Segment	smaato
Chartboost	HockeyApp	Stetho	vkontakte

Table 3: A list of third-party SDKs extracted by LibScout from the top 5,000 apps, which is used to curate third-party backends.

a RST packet. TCP ping scans are more reliable in detecting the availability of the remote server (backend) because they are not filtered by firewalls like ICMP scans.

Once SkyWalker establishes the host is reachable, SkyWalker conducts a TCP SYN scan (SYN-SYN/ACK-RST) across *all* ports. This process identifies candidate ports on the target backend that will be used for a more thorough scan (TCP connect). To be efficient, SkyWalker uses the list of ports identified in the TCP SYN scan to conduct a TCP connect scan (SYN-SYN/ACK-ACK) i.e., establish a complete connection. Based on the port/service identified, SkyWalker interactively grabs the banner, the header response, and any available configuration. The retrieved information varies per service type, for example, HTTP will have header information unlike SSH, nonetheless, both help fingerprint the host. Moreover, SkyWalker looks for TLS/SSL connections on all candidate ports because many services like HTTP and IMAP can run over TLS/SSL. Finally, to obtain the backend IP address fronted by CDNs, SkyWalker looks up the IP address in a manually curated CDN list and uses passive DNS to find historical records that existed just before the current records. When SkyWalker cannot locate such record, the backend is excluded from fingerprinting.

Once SkyWalker discovers all the services running on a backend, SkyWalker uses the result to fingerprint the backend. The fingerprint identifies the *OS*, *SS*, and *CS* type (Linux, Windows; PHP, .NET, Python, Perl; FTP, SFTP, HTTP, HTTPS, SSH, IMAP, etc.), version, and configuration information if available. The fingerprinting uses open source and commercial Nessus Attack Scripting Language (NASL) scripts to identify the different layers of software on the backend. For example, to identify the *OS*, the NASL script inspects the banner string, analyzes the SSL certificate, checks additional running services (SMB, RDP, SSH), performs structured ICMP pings, inspects HTTP headers, and uses TCP/IP fingerprinting algorithms [24]. Based on these signals a confidence score is provided based on matching a set of pre-profiled *OS*es. For example, if 90% of the signals match a *Windows Server 2008 R2 Service Pack 1* profile, we consider the *OS* layer for that backend in the vulnerability analysis. Any confidence level below 90% or ambiguity between the same *OS* but different versions will not be considered for the vulnerability analysis phase.

Web Applications. Web apps (*AS*) are generally tailored per mobile app, unlike *OS*, *SS*, and *CS* layers. The binary phase performs in-context analysis for each API interface on the backend, which provides API information used for fingerprinting. We reference the OWASP's top 10 vulnerability issues [14] that can be passively tested within the ethical and legal bounds discussed in Section 8. Specifically, SkyWalker uses side-channel SQLi through time delay, reflective XSS, and XXE callback to identify *candidate* issues in web apps. It is important to note that other vulnerabilities such as authentication bypass, broken access control, and sensitive data exposure present a high risk that can violate legal obligations. Adding a module to SkyWalker to support additional vulnerabilities is trivial and can be easily implemented.

For each backend interface, a number of parameters (p) are associated with each request. SkyWalker tests each interface p times to check every parameter for SQLi and XSS. The XXE check is performed on all interfaces because some *AS* can accept JSON or XML requests. As mentioned earlier, the scan is slow and randomly done to avoid congestion and degradation of service on production backends. SkyWalker creates two queues, a job queue and a processing queue. SkyWalker generates p requests for a given backend interface and stores them in the job queue. The job queue contains all backend requests, which are shuffled and loaded into the processing queue in batches (128 requests per batch). Batches that contain requests with the same domain or IP address are removed and replaced by non-overlapping domains and IP address requests. There are 32 workers that ingest from the processing queue and store the results for vulnerability analysis.

4.4 Vulnerability Analysis

The vulnerability analysis is two parts, N-day analysis, and 0-day analysis. For the N-day analysis, SkyWalker correlates CVE entries with results from the fingerprinting to identify possible issues. The confidence level of the fingerprint results is also used to verify each vulnerability. SkyWalker uses NASL scripts that take the output of the service discovery, *OS* identification, *SS* identification, and *CS* identification as input and match them against known vulnerabilities (CVEs). The NASL results are considered if they have 90% confidence level or higher for *OS* detection, which provides high accuracy for vulnerability matching. Note, that the confidence level is calculated based on pre-profiled *OS*es by matching the fingerprint signals (collected from all layers) to the profile signals.

We manually verified all 983 N-days and found them to be all true positives. The zero false positive results are due to the Nessus configuration, which allows us to tune how the scans are done and how they should be reported. For example, we configure Nessus to perform the scan types described above, consider *OS* type and version detection of 90% or higher and consider *SS* that have banner information with version numbers. On the other hand, when we used UDP scanning techniques and consider generic service banner information we find over $6,500$ candidate N-day instances with a large false positive rate. In theory, the backend can be configured to lie about the banner information, which would make it hard for us to verify.

For the 0-day analysis, SkyWalker carefully triggers the candidate vulnerability to verify the findings. For each vulnerable parameter, SkyWalker generates a pair of request messages, the original message and the vulnerable message.

For *SQLi*, SkyWalker baselines the original request message several times throughout the week and at different times of the day. Then SkyWalker performs the same measurement on the vulnerable message in the same week but in non-overlapping time intervals by triggering the vulnerable parameter through an SQLi sleep injection. SkyWalker calculates the response time deviation based on the sleep parameter passed in the SQL statement and the average response time of the message pairs. If the deviation is equal to the time delay parameter in the SQL statement, SkyWalker concludes that the interface and parameter pair is vulnerable.

Similarly for *XSS*, SkyWalker triggers the vulnerable parameter and includes JavaScript code to creates a new *div* element with a unique *name* attribute. SkyWalker checks the returned content by parsing the document object model (DOM) to find the *div* element containing the unique *name* attribute. If the *div* element with the set *name* attribute exists SkyWalker concludes that the interface and parameter are vulnerable. Note that SkyWalker matches the returned content with parameters sent to ensure that the *XSS* candidate vulnerability is of type 2 (reflected). For *XXE*, SkyWalker generates a request message that contains an HTTP callback request to a server we operate. The request message is passed to the backend, which will parse the specially crafted XML document. If the parser is vulnerable to XXE, SkyWalker will log an HTTP request from the backend under analysis, which indicates the interface is vulnerable. In addition, we manually reviewed the request/return pairs for all 655 0-day instances and found no false positives.

4.5 Open Access for Developers

One of our primary goals for this work is to empower app developers with open access to SkyWalker via a free-to-use web-service. The service currently supports Android mobile apps but can be extended to support other mobile platforms, e.g., Apple iOS. The web-interface takes as input a link to an Android app in the Google Play store or a direct APK upload. SkyWalker then performs binary analysis to extract the backends, label them based on our curated dataset, fingerprint them, and identify vulnerabilities that affect them. In addition to the analysis, the output report provides guidelines on how to mitigate the identified issues using the strategies discussed earlier (upgrade, patch, block, report, and migrate).

SkyWalker summarizes vulnerability findings across all observed SDK and Java library backends, which developers can turn to *proactively* to make an informed decision when choosing third-party libraries to include in their future apps. It is important to note that attackers can abuse this system to attack mobile app backends. Therefore we require the developers to disclose their affiliation with the target app before the analysis results are provided. Once a user is manually vetted, they can only submit apps that they develop. We do not consider third-party *SDK*s in this process. The SkyWalker service can be found at: https://MobileBackend.vet.

5 Assessment Findings

5.1 Experiment Setup

Environmnet. We use a local workstation running Ubuntu 14.04 with 24GB memory and 16 x 2.393GHz Intel Xeon CPUs and four Nexus phones to run and instrument the mobile apps. We use an Amazon Web Service (AWS) Elastic Compute (EC2) instance with a reserved IP address to conduct the fingerprinting and run a web server with information about our study along with an email address for backend hosts to contact us if they want to opt-out.

Tools and Data Sets. For the binary analysis tool implementation, we relied on Soot [25], FlowDroid [26], Z3-str [27], and Xposed [28] with custom code written in Java (7,000 lines of code) and Python (900 lines of code). For our backend labeling implementation, we relied on Team Cymru IP-to-ASN [29], MaxMind Geolocation [23], Alexa ranking [30], ipcat list [20], and Domaintools WHOIS [31] with custom code written in Python (480 lines of code). For fingerprinting, we relied on the Nessus scanner and commercial plugins [32], sqlmap [33], and Acunetix [34]. We used Nessus plugins and custom Python code (1010 lines of code) to perform the vulnerability analysis. For internet measurements, we utilized honeypot scanning activity from Greynoise [35].

5.2 Software Vulnerability Details

Table 4 shows the distribution of 0-day and N-day instances across the software layers. We categorize the apps using the Google Play store groups and present the number of vulnerabilities and backend labels. Overall, we analyzed 4,980 apps with cloud-based backends and successfully extracted backends for 4,740 mobile apps. The remaining 240 mobile apps crashed and did not complete the full binary analysis.

Interestingly, the *OS* component reports the least vulnerabilities, while the *AS* component reports the most vulnerabilities, across all mobile app categories. Recall from Section 3.2, vulnerabilities affecting *AS* components are all considered *0-day*. The *OS*, *SS*, and *SC* components account for *N-day* vulnerabilities. Although the number of apps is not uniform across the categories, we use the *raw* vulnerability count for ranking. For *0-day* vulnerabilities, the top three mobile app categories are tools, entertainment, and games. For *N-day* vulnerabilities, the top three mobile app categories are entertainment, tools, and games.

Ownership. Table 4 presents the labels for the backends used by mobile apps. The most common label is *hybrid*, where 3,336 backends use hybrid infrastructure. The second

Category	# Mob. Apps	Vulnerabilties					Labels				
		# *OS*	# *SS*	# *AS*	# *CS*	**Total**	# B_{1st}	# B_{3rd}	# B_{hyb}	# B_{ukn}	**Total**
Books & Reference	332	15	49	55	71	190	365	653	501	354	1,873
Business	145	5	22	10	37	74	93	258	150	113	614
Entertainment	1,177	36	108	158	170	472	746	913	942	783	3,384
Games	1,283	34	81	147	106	368	290	804	651	444	2,189
Lifestyle	363	20	50	79	72	221	262	665	311	237	1,475
Misc	199	6	21	45	46	118	76	422	163	105	766
Tools	792	19	84	184	115	402	729	796	812	464	2,801
Video & Audio	689	24	46	89	98	257	267	648	434	357	1,706
Total	4,980	121	356	655	506	1,638	2,492	1,089	3,336	2,506	9,423

Table 4: An overview of the vulnerable mobile apps per genre along with the *raw* counts of vulnerabilities and labels.

Party	Vulnerable Component				Total
	OS	*SS*	*AS*	*CS*	
B_{1st}	37	87	155	211	490
B_{3rd}	6	21	200	42	269
B_{hyb}	47	150	154	184	535
B_{ukn}	55	135	146	173	509

Table 5: Count of *apps* affected by vulnerabilities per cloud layer and their corresponding labels.

Comp.	Vulnerability (Top 3)	#Apps
OS	Expired Lifecycle for Linux OS (various)	124
	Windows Server RCE (MS15-034)	64
	Expired Lifecycle for Windows Server	9
SS	Vulnerable PHP Version	357
	Expired Lifecycle for Web Server (various)	181
	Vulnerable Apache Version	76
AS	XSS (various)	262
	SQLi (various)	160
	XXE (various)	86
CS	Support for Vulnerable SSL Version 2 and 3	997
	OpenSSH Bypass (CVE-2015-5600)	16
	Vulnerable OpenSSL (various)	15

Table 6: The top three vulnerabilities found per cloud layer along with the number of affected mobile apps.

largest is *first-party* with 2,492 backends followed by *third-party* with 1,089 backends. There are 2,506 backends that we were not able to label due to ambiguities, but we labeled approximately 73% of all backends we encountered.

More important is providing remediation guidance to the responsible party. Table 5 shows the mapping between the backend labels and the vulnerable apps. We cannot say much about the *Unknown* category since the vulnerabilities may belong to either *first-party* or *hybrid* categories. We observe that for first-party backends, the highest number of vulnerable apps are found in *AS* and *CS* components with 155 and 211 instances, respectively. Similarly, the hybrid backends have 154 0-day vulnerability instances and 184 N-day instances. In general, we observe that the components that app developers are responsible for (*AS* and *CS* in the B_{1st} and B_{hyb}) have more vulnerabilities.

Operating System (*OS*). The *OS* component issues can be summarized into two categories: legacy unsupported *OS* or unpatched *OS*. The difference is that the legacy *OS* are no longer supported by the vendor, hence vulnerabilities will not be addressed. We see from Table 6 that both *Linux* (various flavors) and *Windows* backends use expired lifecycle versions and 133 apps use these backends. The second most common issue is the *Windows Server* vulnerability *MS15-034* affecting 64 apps, which have patches by the vendor. Overall, the top three *OS* vulnerabilities listed in Table 6 affect 197 mobile apps.

We found the MS15-034 vulnerability affecting hybrid backends (B_{hyb}) that run on *Amazon AWS*, *Akamai*, *OVH*, *Go Daddy*, *Digital Ocean*, and other smaller hosting providers. Further, some of the backends appear on CDN networks, like *Akamai*, *Fastly*, and *CloudFlare*, that offer "EdgeComputing" services [36] which provide web app accelerator services. This insight shows that some developers who deploy vanilla versions of Windows Server *OS* are not maintaining them. In Table 5 the first-party *OS* component has 37 vulnerable backends, which is much higher than third-party backends (6). App developers who run and maintain their own backends (B_{1st}) have to be mindful of these bugs, which in some cases require provisioning new backends with newer OSes causing incompatibilities with existing services (*SS*) and applications (*AS*). SkyWalker can inform the developer of these issues and report mitigation strategies.

Software Services (*SS*). SkyWalker identified multiple vulnerabilities affecting a range of PHP versions, which can be used to cause denial of service (*CVE-2017-6004*), disclose memory content (*CVE-2017-7890*), disclose sensitive information (*CVE-2016-1903*), and execute arbitrary code (*CVE-2017-11145*). Backends of 357 mobile apps affected by PHP vulnerabilities, significantly higher than the other two vulnerabilities. Further, even though some mobile app backends had no *0-day* vulnerabilities, an attacker can still craft special requests to trigger deep bugs within the interpreter to compromise the backend. Although this might be a difficult task, recent advancement in vulnerability fuzzing [37] can uncover these deep bugs.

The second most common *SS* vulnerability was unsupported versions of Apache web server (1.3.x and 2.0.x), Tomcat server (8.0.x), and Microsoft IIS web server (5.0). Similar to unsupported *OS*, web server vendors will not issue security patches for unsupported software, which affects backends of 181 mobile apps. For Apache web server versions less than 2.2.15, they are affected by several denial of service bugs (*CVE-2010-0408, CVE-2010-0434*) and TLS injection bug (*CVE-2009-3555*) affecting 76 mobile apps. Additionally, Apache servers that use Apache Struts versions 2.3.5 - 2.3.31 or 2.5.10.1 and lower are vulnerable to *CVE-2017-5638*, which allows remote code execution. The same Apache Struts vulnerability was reportedly used against Equifax's hack [38]. In total, the top three *SS* vulnerabilities affect 614 mobile apps.

Applications (*AS*). Table 7 has a breakdown of the number of mobile apps, their number of install categories, and the *instances* of *0-day* bugs affecting them. Although *XSS* is the largest category with 503 instances followed by *SQLi* (215) and *XXE* (46), we note that not all of the bugs have the same impact and some affect the same backend. For instance, an *SQLi* can be limited to an isolated instance of the app (e.g., a container), which would limit the attack to disclosing information from the application database or modifying preexisting records. Moreover, *XSS* vulnerabilities often have less impact than *SQLi* and *XXE*.

XXE vulnerabilities affect web apps that use XML for their API communication. The fundamental flaw that enables XXE vulnerabilities to exist is a faulty implementation of the XML parser. Based on our measurement, we found 1 *XXE* instance in the top 100M, 5 in the top 50M, 15 in the top 10M, 9 in the top 5M, and 17 in the top 1M. Table 7 shows the concentration of vulnerabilities found in lower ranking apps. For example: 1 *XXE* and 3 *XSS* vulnerabilities in the top 132 mobile apps; 4 *SQLi*, 10 *XSS*, and 5 *XXE* vulnerabilities in the next 131 mobile apps (though still representing over 50M+ installs each). However, *AS* vulnerabilities are not confined to lower ranking apps but do affect higher ranking apps.

# Installs	# Apps	# SQLi	# XSS	# XXE
1B	5	0	0	0
500M	11	0	0	0
100M	116	0	3	1
50M	131	4	10	5
10M	1,049	25	85	15
5M	1,047	54	89	9
1M	2,621	132	316	17

Table 7: The number of *0-day* vulnerabilities found per install category.

Table 8 shows the *AS* layer implementation language and associated vulnerabilities. *AS* implemented in *PHP* have the most *0-days* instances (284) affecting 108 different backends, followed by *ASP.NET* with 33 *0-day* instances affecting 13 different backends. We note that this trend does not mean causation. *PHP* is the most popular language used for web application development [39], hence it is expected to represent more vulnerabilities by being more popular. Furthermore, we found 9 *0-day* instances in *PERL*, 8 in *JavaScript (NodeJS)*, 5 in *JSP*, and the rest of the 316 could not be determined.

Language	# Backends	# 0-Days
PHP	108	284
ASP.NET	13	33
PERL	4	9
JS	4	8
JSP	2	5
Unknown	72	316

Table 8: The number of identified languages associated with *0-day* vulnerable backends.

Communication (*CS*). All mobile apps rely on the HTTP/HTTPS protocol for communication with their backends. The binary analysis phase extracted a total of 17,725 request messages from the 4,740 mobile apps. The request messages are split into HTTP (8,118) request messages and HTTPS (9,607) request messages. There are 446 mobile apps that only use HTTP communication and another set of 147 mobile apps that only use HTTPS communication. The remaining set of 4,147 apps mix between HTTP and HTTPS communication.

Despite using HTTPS, over 20% of the backends (1,012) have issues with TLS/SSL configuration (e.g., insecure session renegotiation and resumption) or unpatched software versions (e.g., SSL version 2 and 3). These flaws can be exploited by an attacker to carry out a MITM attack by downgrading the protocol negotiation using the POODLE [40] attack. Additionally, the OpenSSH Bypass vulnerability exposes the backend to compromise via SSH credential guessing or secret key leak. The mobile apps using these vulnerable backends do not use the *SSH* service and to remediate one can turn off, patch, or block the incoming internet traffic to it.

Those backends which only use HTTP expose users to eavesdropping and MITM attacks because it does not offer integrity or confidentiality. We manually inspected the request messages sent from 3,253 apps that use HTTP and found personally identifiable information (PII) such as name, gender, birth year, user ID, password, username, and country. Additionally, we found device information like MAC, IMEI, SDK version, make/model, SSID, Wifi signal, cell signal, screen resolution, carrier, root access, IP Address, and coordinate location. Combining this information, a network attacker can identify individuals and attribute behavior profiles to them. Furthermore, 6 apps we investigated perform a password reset over HTTP. Interestingly, the Apple iOS App Store enforces strict use of HTTPS through their *App Trans-*

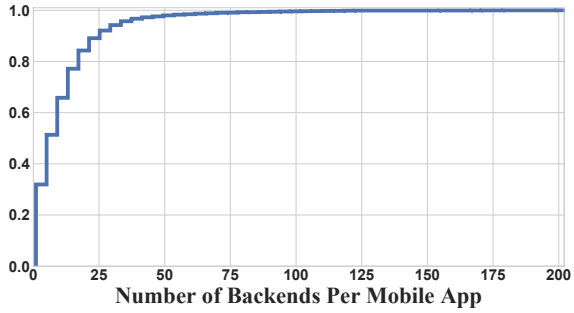


Figure 2: The figure shows the CDF of the number of backends per mobile app.

port Security [41] model. We recommend that the Android platform adopt the same restriction.

5.3 Impact on Mobile Application Users

The overall impact for each vulnerability varies based on the severity, the mobile app to backend usage, and the adversary capability/visibility. Although it is important to understand the impact of each vulnerability, it is not trivial to quantify the impact of each vulnerable backend on mobile apps. For N-day vulnerabilities, an attacker can perform an internet-wide scan to identify and attempt to compromise these backends. Even once identified, these N-days span many different components (*OS*, *SS*, and *CS*) that have varying impacts on the backend from basic information disclosure to a full system compromise. For 0-day vulnerabilities the attack impact varies based on the exploit type (*SQLi*, *XSS*, or *XXE*) and how the backend infrastructure is set up. Moreover, how the mobile app uses the backend directly impacts the severity of the vulnerability. For example, if a mobile app uses app slicing [3] or downloads additional libraries from the mobile backend, an attacker who compromises the backend can modify the content and attain code execution on the mobile device.

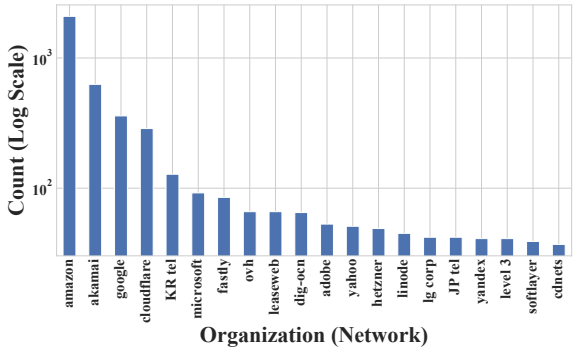


Figure 3: The figure shows the distribution of backends across internet networks.

In general, an attacker has a larger attack surface for apps that have many backends. Figure 2 shows a CDF of backends per mobile app. We can see that the majority of the 5,000 apps studied have between one and 25 different backends and in the worst case they have up to 203 different backends. We also observe that these backends reside in diverse networks as shown in Figure 3, which means the infrastructure set up for the backends will be different affecting the impact of the vulnerability.

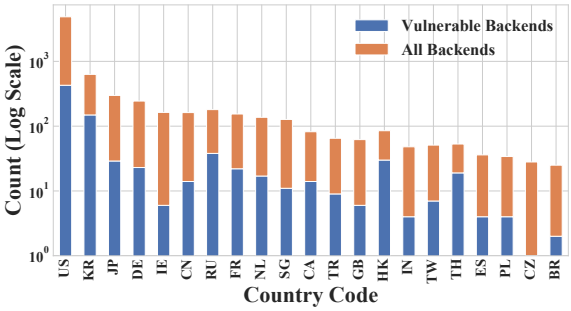


Figure 4: The figure shows the distribution of all mobile backends and vulnerable backends across the world.

Finally, the geographical distribution of the backends, shown in Figure 4, affect the impact on mobile apps. Many mobile apps deploy multiple backends that are geographically distributed to provide faster content for different user segments. In some cases, the different backends may not be fully synchronized in terms of the latest software patches for *OS*, *SS*, *AS*, and *CS* layers, which results in a vulnerable backend affecting only a segment of users for a particular mobile app. Directly quantifying the impact of each vulnerability is an involved task and depends on many variables such as the severity of the vulnerability, the mobile app to backend usage, the adversary capability, and other nuance factors (number of backends per app, network distribution, and geographical distribution). We plan to perform a comprehensive analysis to understand this impact as future work.

5.4 Vulnerability Disclosure, Bug Bounties, And In The Wild Threats

During our disclosure process, we identified two mobile platform vendors that have a bounty program, namely *Unity3D* [42] and *Simpli.fi* [43]. In addition, the top third-party platform providers, Google, Facebook, Crashlytics, and Flurry, all participate in or run their own bug bounty programs. Similarly, the cloud providers either run their own program or use a third-party bug bounty program like Bugcrowd [44] or Bounty Factory [45]. We submitted our vulnerability disclosures through their bounty management program (e.g., *HackerOne* [46]) and received confirmation of the bugs.

For smaller third-party and first-party developers, they did not have a formal way to contact them to report vulnerabilities. We followed a tiered approach in our notification by first notifying the app developer directly using the contact information in the Play Store. Our second attempt to report the vulnerability is by contacting the domain owner using the WHOIS information and following the mitigation strategy. Our third attempt to report the vulnerability is by contacting Google directly through their issue tracker portal. For parties that did not confirm or respond to our multiple attempts, we reported the vulnerabilities to US-CERT [47].

Component	# IP Scanners
Operating System (*OS*)	341,521
Services (*SS*)	445,908
Application (*AS*)	206,533

Table 9: Number of IPs observed scanning the internet for vulnerabilities reported by Greynoise.io [35] over a period of a year (Sept 2017 to Sept 2018).

The N-day vulnerabilities we found are discoverable and easy to exploit due to the availability of fast internet scanners like ZMap [48] and MASSCAN [49]. We argue that it is a matter of time until these vulnerabilities are found and exploited. Table 9 shows the number of active scans detected on the internet through Greynoise [35] honeypots over a period of one year (Sept 2017 to Sept 2018). There are 341,521 unique IPs scanning for *OS* related vulnerabilities, 445,908 unique IPs scanning for *SS* vulnerabilities, and 206,533 unique IPs scanning for *AS* vulnerabilities. Many of these scans target *N-day* vulnerabilities, while scans for *0-day* vulnerabilities cannot be accounted for. Nonetheless, past events demonstrate that attackers are prone to scan for and exploit 0-day vulnerable web apps like Wordpress [50], Drupel [51], and PHPMyAdmin [52] when publicly disclosed. Furthermore, a recent report [53] also pointed out that the number of vulnerabilities in web apps increased in 2018 and that support for PHP version 5.*x* and 7.*x* will end in 2019, which means we can anticipate more unpatched and exposed backends in the future.

6 Case Studies

6.1 Case Study 1: Vulnerable Web App

The mobile app "Dailyhunt" has more than 50M+ installs and is part of the "Books & Reference" category. The mobile app interacts with nine different backends as shown in Table 10. The backends are split into two labels, hybrid and third-party. The hybrid backends are hosted on Akamai's EdgeComputing [36] and run a custom web app to serve the mobile app. The hybrid backends are used for CDN, telemetry, and requesting app-specific data. Specifically, the

Label	Backend	Use	Vulns. *AS*	Vulns. *CS*
B_{hyb}	api-news.dailyhunt.in	App Data	0	1
	acq-news.dailyhunt.in	App Data & Telemetry	2	1
	bcdn.newshunt.com	CDN	0	1
	acdn.newshunt.com	CDN	0	1
B_{3rd}	fonts.gstatic.com	CDN	0	0
	e.crashlytics.com	Telemetry	0	0
	settings.crashlytics.com	Telemetry	0	0
	t.appsflyer.com	Ads	0	0
	api.appsflyer.com	Ads	0	0

Table 10: A list of backends and issues found for the mobile app Dailyhunt.

*api-news.** domain registers the device and requests content, where the *acq-news.** backend captures user behavior and offers promotion and the actual content is delivered by the two CDN domains *acdn.** and *bcdn.**.

We were not able to fingerprint the *OS* and the *SS* because the Akamai servers respond only to web app specific responses, i.e., minimal header and banner information. Nonetheless, we found two 0-day vulnerabilities in the *acq-news.** backend on the same API interface. Since this web application is specific to this mobile app, we looked for other apps published by the same developer. We found that the *eBooks by Dailyhunt* app (which has over 500K installs but does not rank in the top 5,000 apps) also uses the same vulnerable API interface. Additionally, the mobile apps use HTTP to communicate with the hybrid backends and HTTPS to communicate with third-party services.

As for the third-party services, we did not find any vulnerabilities. The third-party backends serve requests on port 443 (HTTPS). The *appsflyer.com* backend is a service for ad analytics that provides different functions using the same interface. The *t.appsflyer.com* backend is a telemetry endpoint for the ad network and the *api.appsflyer.com* backend authenticates and associates the app with its profile.

Takeaway. This case highlights several challenges to securing mobile app backends. First, backends are heterogeneous and differ across their software stack, topology setup, configuration, and custom application. Second, outsourcing cloud management and provisioning (e.g., to cloud providers and CDNs) benefits security but comes with a lack of visibility, limited per-app customization, and unclear incident liability. Third, vulnerabilities can exist (and be scanned for) in any software layer of the cloud and API interface on the web app, which makes them challenging to identify and fix. Unfortunately, app developers do not have the resources, time, or personnel to fulfill this task. Using SkyWalker, we aim to provide guidance to where the most pressing issues exist and map them to responsible parties as shown in Table 2.

App Name	# Reviews	# Installs
com.icegame.fruitlink	332,907	50M+
com.unbrained.wifipasswordgenerator	151,518	10M+
com.magdalm.wifimasterpassword	148,355	10M+
com.unbrained.wifipassgen.app	43,824	1M+
com.magdalm.freewifipassword	35,552	1M+
apps.ignisamerica.gamebooster	23,725	500K+
com.icegame.crazyfruit	23,631	1M+
com.magdalm.wifipasswordpro	22,113	1M+
apps.ignisamerica.bluelight	16,659	500K+
com.icegame.fruitsplash2	15,193	1M+

Table 11: A list of the top 10 mobile apps using the *appnext* platform.

Label	Backend	Usage	Vulns.		
			OS	*AS*	*CS*
B_{3rd}	admin.appnext.com	App Data	0	1	0
	global.appnext.com	App Data	0	0	0
	cdn.appnext.com	CDN	1	0	1
	cdn3.appnext.com	CDN	1	0	1

Table 12: List of backends and vulnerable layers found in the *appnext* platform.

6.2 Case Study 2: Vulnerable Platform

The *appnext* [54] platform integrates with mobile apps to ingest user behavior telemetry and provide predictive actions that users might perform. Developers use this to upsell subscription, ads, or recommend actions to app users. The *appnext* platform is used by 6 mobile apps from the top 5,000 free apps. We analyzed all apps by the same developers that are not in the top 5,000 and found 140 additional apps using the *appnext* platform. The top 10 most reviewed apps using the *appnext* platform can be found in Table 11. The top app has 332,907 reviews and over 50M+ installs. These numbers give us an indication of the the platform's significant popularity and daily use.

The *appnext* platform backends (shown in Table 12) are labelled as third-party, because the backends are found in an SDK library. We found two CDN domains that point to the same server IP, which are hosted on *Limelight Networks*, a CDN provider. This CDN backend is vulnerable to an *OS* integer overflow in the HTTP protocol stack (MS15-034) that can be remotely exploited. Further, the *CS* still offers SSLv2 and SSLv3, which are vulnerable to insecure padding scheme for *CBC* cipher. *appnext*'s *admin.** and *global.** domains run on Amazon AWS and provide app-specific data, like authentication, telemetry ingestion, predictive actions, and configuration. The infrastructures run Microsoft Windows Server 2008 R2 for the *OS*, Microsoft-IIS/7.5 for its web server (*SS*), and the *CS* uses *HTTPS*. The application (*AS*) backend is a custom web application that is written in *ASP* and uses the *ASP.NET* framework. The *AS* has a vulnerability that allows an attacker to run arbitrary SQL queries. We have notified the developers about these findings and awaiting remediation.

Takeaway. This case highlights multiple vulnerabilities, *0-day* and *N-day*, that affect three of the four software layers. This mobile platform collects sensitive information about user behavior, including PII and device information. Unfortunately, these backend vulnerabilities are inherited by multiple apps and developers, and the app developers cannot immediately remediate the vulnerabilities in third-party services. The mitigation strategy for the app developer is to report (*r*) these findings to *appnext* or migrate (*m*) their app to a different service. SkyWalker helps us label the backends, identify the vulnerability, and guide the developer to a clear action (report or migrate).

7 Mitigation

The goal of our empirical analysis was to bring attention to this overlooked problem in mobile backends, but also to provide guidance to app developers for building or choosing secure backends. In this section, we discuss the general mitigation strategies which SkyWalker recommends for app developers and to help improve the security posture of their app backends.

7.1 Remediation Strategies

App developers who rely on first-party backends have to **upgrade**, **patch**, and **block** as needed for each software layer on their backend. If they rely on third-party backends they can **report** the issue or **migrate** their backend to a more secure provider. Ambiguity arises when the backend is hosted by a cloud provider, a hybrid type backend. To resolve these issues we further generalize the hybrid backends into IaaS (cloud provider manages the virtual *HW*) and PaaS (cloud provider manages *HW*, *OS*, and *SS*).

	Hybrid				
Strategies	**HW**	**OS**	**SS**	**AS**	**CS**
Upgrade		✔	✔		✔✔
Patch		✔	✔	✔✔	✔✔
Block			✔		✔
Report	✔✔	✔	✔		
Migrate	✔✔	✔	✔		

Table 13: A mapping of mitigation strategies for developers hosting their hybrid backend on infrastructure (**IaaS**) or a platform (**PaaS**).

Table 13 provides developers with a guideline on how to mitigate vulnerable hybrid backends. For example: if the hybrid backend is using a cloud provider's platform offering, developers should report and/or migrate their backend if the vulnerabilities are found in *HW*, *OS*, *SS* and upgrade or patch if the vulnerabilities *CS* or *AS* related, respectively.

This matrix provides a starting point for app developers to explore their options, i.e., migrate or wait for a fix. In some cases, the offering from cloud providers includes *HW* and *OS* (as in the motivating example which uses Google Compute Engine Flexible Environment). In this case, developers have to make sure they use the latest *OS* images supported by their cloud provider.

7.2 Recommendations

The empirical analysis provides insight not only about insecure mobile backends, but also secure practices that developers can learn from. For developers who decide to build their own first-party backends, we recommend the following: First, developers should **delegate** as much of the backend functionality to reputable third-party backends and minimize the number of features and functions their backend needs to support. Second, developers should **dedicate** personnel to manage and maintain their backends including the routine maintenance of *OS*, *SS* and *CS*, and timely fixes of known vulnerabilities affecting their cloud backends and mobile apps using patching tools [55]–[57]. Third, developers should **develop** an audit plan and a mitigation plan and be familiar with it to execute during an incident or vulnerability disclosure. Finally, developers should utilize **defense** tools like web app firewalls (WAF), DDoS mitigation, and crawler/scanner blockers to protect from internet scanners, DDoS threats, and web app attacks (SQLi, authentication bypass, etc.). We identified over 730 backends using defense services, all of which had smaller footprints when fingerprinted and no vulnerabilities were detected.

8 Measurement Considerations

Ethical. Because our work does not require or implicate human subjects, no IRB approval was required by our institute. Our study identified a large number of *0-day* and *N-day* vulnerabilities in active mobile app backends through scanning and probing. Our techniques include service scans, banner grabs, and side-channel probes. We emphasize that no active exploitation, disruption, or sensitive data access was attempted against the mobile backends. Although there are no set guidelines for vulnerability measurements in the community, several previous works (e.g., [48], [58]–[60]) have set some precedent. Our measurements followed the best practices used in previous work using the following approach:

- **Good Internet Citizenship:** Similar to the work of Li et al. [58], we provided an opt-out page for our scanner IP that gives targets an option to be removed from the study. Further, we signal our benign intention by setting the user-agent string in the scans and provide a reverse DNS record for our IP to give targets additional information about our study. We were contacted by one app developer and requested that we remove their backends and related infrastructure from our study.
- **Non-Exploit Payloads:** Similar to the work of Durumeric et al. [59], our scanning and measurement techniques did not include any active exploits against the mobile app backends. We used side-channel measurements with time delay probes to infer vulnerabilities. The requests were carefully crafted to ensure that vulnerabilities are triggered for verification and not persistent or full system exploitation. Further, our scanning approach was throttled to ensure the availability of the backend is not affected by the additional load.
- **Responsible Disclosure:** Lastly, we notified affected mobile app developers and third-party mobile service providers through the appropriate channels. For developers and third-party service providers who did not respond to our communication, we reported the vulnerabilities through the US-CERT [47].

Legal. Similar to Ristenpart et al. [60], we operate within the legal bounds in conducting this study. In the US, the Computer Fraud and Abuse Act (CFAA) is the governing law that pertains to use and access of computer systems. The law states, in brief, that access to any computer system must be authorized, but does so in broad terms. The decision from the case of Moulton v VC3 (2000) sets a precedent that service discovery scanning does not cause damages or direct harm to target systems. Additionally, we assume any internet-facing service gives implicit permission to access the target computer system, in particular, we refer to how web crawlers and internet indexing services operate. As we outlined in our ethical section earlier, we provide subjects the option to opt-out, perform non-malicious measurement probes, and use responsible disclosure to notify affected parties.

9 Related Work

Cloud Security. Cloud security has been surveyed extensively [61]–[65]. Xiao et al. [9] performed a comprehensive analysis of the security issues in cloud services by surveying high-level *provider* and *tenant* issues for the cloud-based services in general. Singh et al. [66] presented a survey to identify common issues reported in third-party cloud services and summarize the work from the architecture framework, service and deployment, and cloud technologies perspective. Our work looks at "in-the-wild" deployment of cloud services from the *OS*, *SS*, *AS*, and *CS* perspectives to empirically study and uncover common issues in mobile backends.

Measurement Studies. Durumeric et al. [67] conducted a comprehensive internet-wide study of the HTTPS certificate ecosystem. Later, Durumeric et al. [59] carried out a similar internet-wide study for the Heartbleed vulnerability [68].

Perez-Botero et al. [69] presented an in-depth study characterizing hypervisor vulnerabilities in cloud services. Zuo et al. [19] proposed a system to identify mobile app URLs and examine their reputation with public blacklists to detect malicious apps. Our work differs from prior work by studying a range of vulnerabilities which may affect mobile app backends on the internet.

Empirical Backend Analysis. Zuo et al. [12] performed an assessment of mobile app backend services by investigating the cloud offerings of Google, Amazon, and Microsoft. Our work provides a wider analysis by going beyond just the third-party service backends and by examining a diverse set of cloud-based backends. Fernandes et al. [70] analyzed the top apps found in the Samsung SmartThings platform to identify permission issues. We follow a similar approach but focus on the mobile app integration with cloud services instead of IoT apps and cloud services. Alrawi et al. [71] presented a systematization security assessment of home-based IoT devices and their companion cloud and mobile apps. Our work encompasses a wider application, beyond only IoT mobile apps, and a more focused assessment by looking at the supporting backends provided by cloud services.

10 Conclusion

This paper presented SkyWalker, an analysis pipeline to study mobile app backends. We used SkyWalker to empirically analyze the top 5,000 mobile apps in the Google Play store and uncovered 655 0-days and 983 N-days instances affecting thousands of apps. Lastly, we offer SkyWalker as a public service to help app developers improve the security of their backends, give insight on what platforms are vulnerable, and guide developers to fix issues found in their backends: https://MobileBackend.vet.

Acknowledgement

We thank Manos Antonakakis, Yizheng Chen, Angelos Keromytis, Panagiotis Kintis, Chaz Lever, Frank Li, Xiaojing Liao, Yinqian Zhang, and the anonymous reviewers for their insightful comments. This work was partially supported by AFOSR under grant FA9550-14-1-0119, NSF awards 1834215, and 1834216.

References

[1] S. Ghosh, *British Airways customer data stolen from its website*, https://www.theguardian.com/business/2018/sep/06/british-airways-customer-data-stolen-from-its-website, 2018.

[2] Z. Whittaker, *Air Canada confirms mobile app data breach*, https://techcrunch.com/2018/08/29/air-canada-confirms-mobile-app-data-breach/, 2018.

[3] A. Martonik, *Epic's first Fortnite Installer allowed hackers to download and install anything on your Android phone silently*, https://www.androidcentral.com/epic-games-first-fortnite-installer-allowed-hackers-download-install-silently, 2018.

[4] K. Watkins, "HospitalGown: The Backend Exposure Putting Enterprise Data at Risk," Appthority, Tech. Rep., 2017.

[5] S. Subashini and v. Kavitha, "A survey on security issues in service delivery models of cloud computing," *Journal of Network and Computer Applications*, 2011.

[6] C. Höfer and G. Karagiannis, "Cloud computing services: Taxonomy and comparison," *Journal of Internet Services and Applications*, 2011.

[7] L. Youseff, M. Butrico, and D. Da Silva, "Toward a unified ontology of cloud computing," in *In Proc. IEEE Grid Computing Environments Workshop (GCE)*, 2008.

[8] D. Gonzales, J. M. Kaplan, E. Saltzman, Z. Winkelman, and D. Woods, "Cloud-trust-A security assessment model for infrastructure as a service (IaaS) clouds," *IEEE Transactions on Cloud Computing*, 2017.

[9] Z. Xiao and Y. Xiao, "Security and privacy in cloud computing," *IEEE Communications Surveys & Tutorials*, 2013.

[10] K. Watkins and S. M. Kywe, "Unsecured Firebase Databases: Exposing Sensitive Data via Thousands of Mobile Apps," Appthority, Tech. Rep., 2018.

[11] C. Zuo, Q. Zhao, and Z. Lin, "Authscope: Towards automatic discovery of vulnerable authorizations in online services," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct. 2017.

[12] C. Zuo, Z. Lin, and Y. Zhang, "Why does your data leak? uncovering the data leakage in cloud from mobile apps," in *Proceedings of the 40th Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[13] National Institute of Standards and Technology, *NATIONAL VULNERABILITY DATABASE*, https://nvd.nist.gov, 2019.

[14] OWASP, *OWASP Top 10 - 2017: The Ten Most Critical Web Application Security Risks*, https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf, 2018.

[15] Kony, *Kony Fabric*, https://www.kony.com/products/fabric/, 2018.

[16] OutSystems, *Build Mobile Apps*, https://www.outsystems.com/platform/build-mobile-apps/, 2018.

[17] Apache, *Architectural overview of Apache Cordova platform*, https://cordova.apache.org, 2018.

[18] Backbase, *Backbase Enterprise Integration Framework*, https://backbase.com/platform/integration/, 2018.

[19] C. Zuo and Z. Lin, "Smartgen: Exposing server urls of mobile apps with selective symbolic execution," in *Proceedings of the 26th International World Wide Web Conference (WWW)*, 2017.

[20] N. Galbreath, *Categorization of IP Addresses*, https://github.com/client9/ipcat, 2019.

[21] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[22] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee, "Identifying open-source license violation and 1-day security risk at large scale," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct. 2017.

[23] MaxMind, *About MaxMind*, https://www.maxmind.com/en/company, 2018.

[24] R. Beverly, "A robust classifier for passive TCP/IP fingerprinting," in *Workshop on Passive and Active Network Measurement*, Springer, 2004.

[25] E. Bodden, *A framework for analyzing and transforming java and an- droid apps*, https://sable.github.io/soot/, 2018.

[26] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Edinburgh, UK, Jun. 2014.

[27] Y. Zheng, X. Zhang, and V. Ganesh, "Z3-str: A z3-based string solver for web application analysis," in *Proceedings of the 18th European Software Engineering Conference (ESEC) / 21st ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Saint Petersburg, Russia, Aug. 2013.

[28] X. Framework, *Xposed Module Repository*, https://repo.xposed.info/module/de.robv.android.xposed.installer, 2018.

[29] T. Cymru, *IP TO ASN MAPPING*, http://www.team-cymru.com/IP-ASN-mapping.html, 2018.

[30] Alexa, *Find Website Traffic, Statistics, and Analytics*, https://www.alexa.com/siteinfo, 2018.

[31] DomainTools, *About Us*, https://www.domaintools.com/company/, 2018.

[32] T. Security, *Nessus Professional*, https://www.tenable.com/products/nessus/nessus-professional, 2018.

[33] S. Project, *sqlmap: automatic SQL injection and database takeover tool*, http://sqlmap.org, 2018.

[34] Acunetix, *Audit Your Web Security with Acunetix Vulnerability Scanner*, https://www.acunetix.com/vulnerability-scanner/, 2018.

[35] Geynoise, *About*, https://greynoise.io/about/, 2018.

[36] E. Nygren, R. K. Sitaraman, and J. Sun, "The akamai network: A platform for high-performance internet applications," in *Proceedings of the ACM SIGOPS Operating System Review*, vol. 44, Jul. 2010.

[37] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang, "Semfuzz: Semantics-based automatic generation of proof-of-concept exploits," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct. 2017.

[38] D. Goodin, *Failure to patch two-month-old bug led to massive Equifax breach*, https://arstechnica.com/information-technology/2017/09/massive-equifax-breach-caused-by-failure-to-patch-two-month-old-bug/, 2018.

[39] X. Li and Y. Xue, "A survey on server-side approaches to securing web applications," *ACM Computing Surveys (CSUR)*, 2014.

[40] B. Möller, T. Duong, and K. Kotowicz, "This poodle bites: Exploiting the ssl 3.0 fallback," *Security Advisory*, 2014.

[41] *App Transport Security*, https://forums.developer.apple.com/thread/6767, 2015.

[42] Unity3D, *Imagine, build and succeed with Unity*, https://unity3d.com, 2018.

[43] Simpli.fi, *About Us*, https://www.simpli.fi/about-us/, 2018.

[44] bugcrowd, *THE BUGCROWD DIFFERENCE*, https://www.bugcrowd.com/who-we-are/the-bugcrowd-difference/, 2018.

[45] B. Factory, *CREATE MY BUG BOUNTY PROGRAM*, https://bountyfactory.io/en/mybugbounty.html, 2018.

[46] HackerOne, *About HackerOne*, https://www.hackerone.com/about, 2018.

[47] US-CERT, *About Us*, https://www.us-cert.gov/about-us, 2018.

[48] Z. Durumeric, E. Wustrow, and J. A. Halderman, "Zmap: Fast internet-wide scanning and its security applications.," in *Proceedings of the 22th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2013.

[49] R. D. Graham, *MASSCAN*, https://github.com/robertdavidgraham/masscan, 2018.

[50] M. Veenstra, *Privilege Escalation Flaw In WP GDPR Compliance Plugin Exploited In The Wild*, https://www.wordfence.com/blog/2018/11/privilege-escalation-flaw-in-wp-gdpr-compliance-plugin-exploited-in-the-wild/, 2018.

[51] J. Mattsson, *Drupal core - Highly critical - Remote Code Execution - SA-CORE-2018-002*, https://www.drupal.org/sa-core-2018-002, 2018.

[52] C. Point, *Web servers PHPMyAdmin Misconfiguration Code Injection*, https://www.checkpoint.com/defense/advisories/public/2014/cpai-17-mar1.html, 2018.

[53] N. Avital, *The State of Web Application Vulnerabilities in 2018*, https://www.imperva.com/blog/the-state-of-web-application-vulnerabilities-in-2018/, 2019.

[54] Appnext, *The Appnext Discovery Platform*, https://www.appnext.com/platform/, 2018.

[55] SecurityFTW - cs-suite, *Cloud Security Suite - One stop tool for auditing the security posture of AWS/GCP/Azure infrastructure.* https://github.com/SecurityFTW/cs-suite, 2018.

[56] J. Arnold and M. F. Kaashoek, "Ksplice: Automatic rebootless kernel updates," in *Proceedings of the 4th European Conference on Computer Systems (EuroSys)*, Nuremberg, Germany, Mar. 2009.

[57] R. Duan, A. Bijlani, Y. Ji, O. Alrawi, Y. Xiong, M. Ike, B. Saltaformaggio, and W. Lee, "Automating patching of vulnerable open-source software versions in application binaries," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[58] F. Li, Z. Durumeric, J. Czyz, M. Karami, M. Bailey, D. McCoy, S. Savage, and V. Paxson, "You've got vulnerability: Exploring effective vulnerability notifications," in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[59] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, *et al.*, "The matter of heartbleed," in *Proceedings of the 14th Internet Measurement Conference (IMC)*, 2014.

[60] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, Chicago, Illinois, Nov. 2009.

[61] S. Subashini and V. Kavitha, "A survey on security issues in service delivery models of cloud computing," *Journal of network and computer applications*, vol. 34, no. 1, pp. 1–11, 2011.

[62] M. Almorsy, J. Grundy, and I. Müller, "An analysis of the cloud computing security problem," *arXiv preprint arXiv:1609.01107*, 2016.

[63] Y. Sun, G. Petracca, X. Ge, and T. Jaeger, "Pileus: Protecting user resources from vulnerable cloud services," in *Proceedings of the 32th Annual Computer Security Applications Conference (ACSAC)*, 2016.

[64] S. Iqbal, M. L. M. Kiah, B. Dhaghighi, M. Hussain, S. Khan, M. K. Khan, and K.-K. R. Choo, "On cloud security attacks: A taxonomy and intrusion detection and prevention as a service," *Journal of Network and Computer Applications*, vol. 74, pp. 98–120, 2016.

[65] N. V. Juliadotter and K.-K. R. Choo, "Cloud attack and risk assessment taxonomy," *IEEE Cloud Computing*, vol. 2, no. 1, pp. 14–20, 2015.

[66] A. Singh and K. Chatterjee, "Cloud security issues and challenges: A survey," *Journal of Network and Computer Applications*, 2017.

[67] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman, "Analysis of the https certificate ecosystem," in *Proceedings of the 13th Internet Measurement Conference (IMC)*, 2013.

[68] Codenomicon and Google, *The Heartbleed Bug*, https://heartbleed.com/, 2017.

[69] D. Perez-Botero, J. Szefer, and R. B. Lee, "Characterizing hypervisor vulnerabilities in cloud computing servers," in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, Oct. 2013.

[70] E. Fernandes, J. Jung, and A. Prakash, "Security analysis of emerging smart home applications," in *Proceedings of the 37th Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[71] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose, "Sok: Security evaluation of home-based iot deployments," in *Proceedings of the 40th Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

EnTrust: Regulating Sensor Access by Cooperating Programs via Delegation Graphs

Giuseppe Petracca
Penn State University, US
gxp18@cse.psu.edu

Yuqiong Sun
Symantec Research Labs, US
yuqiong_sun@symantec.com

Ahmad-Atamli Reineh
Alan Turing Institute, UK
atamli@turing.ac.uk

Patrick McDaniel
Penn State University, US
mcdaniel@cse.psu.edu

Jens Grossklags
Technical University of Munich, DE
jens.grossklags@in.tum.de

Trent Jaeger
Penn State University, US
tjaeger@cse.psu.edu

Abstract

Modern operating systems support a cooperating program abstraction that, instead of placing all functionality into a single program, allows diverse programs to cooperate to complete tasks requested by users. However, untrusted programs may exploit such interactions to spy on users through device sensors by causing privileged system services to misuse their permissions, or to forward user requests to malicious programs inadvertently. Researchers have previously explored methods to restrict access to device sensors based on the state of the user interface that elicited the user input or based on the set of cooperating programs, but the former approach does not consider cooperating programs and the latter approach has been found to be too restrictive for many cases. In this paper, we propose EnTrust, an authorization system that tracks the processing of input events across programs for eliciting approvals from users for sensor operations. EnTrust constructs *delegation graphs* by linking input events to cooperation events among programs that lead to sensor operation requests, then uses such *delegation graphs* for eliciting authorization decisions from users. To demonstrate this approach, we implement the EnTrust authorization system for Android OS. In a laboratory study, we show that attacks can be prevented at a much higher rate (47-67% improvement) compared to the first-use approach. Our field study reveals that EnTrust only requires a user effort comparable to the first-use approach while incurring negligible performance (<1% slowdown) and memory overheads (5.5 KB per program).

1 Introduction

Modern operating systems, such as Android OS, Apple iOS, Windows Phone OS, and Chrome OS, support a programming abstraction that enables programs to cooperate to perform user commands via input event delegations. Indeed, an emergent property of modern operating systems is that system services are relatively simple, provide a specific functionality, and often rely on the cooperation with other programs to perform tasks. For instance, modern operating systems now ship with voice-controlled personal assistants that may enlist apps and other system services to fulfill user requests, reaching for a new horizon in human-computer interaction.

Unfortunately, system services are valuable targets for adversaries because they often have more permissions than normal apps. In particular, system services are automatically granted access to device sensors, such as the camera, microphone, and GPS. In one recent case reported by Gizmodo [1], a ride-sharing app took advantage of Apple iOS system services to track riders. In this incident, whenever users asked their voice assistant "Siri, I need a ride", the assistant enlisted the ride-sharing app to process the request, which then leveraged other system services to record the users' device screens, even while running in the background. Other online magazines have reported cases of real-world evidence that apps are maliciously colluding with one another to collect and share users' personal data [2, 3, 4].

Such attacks are caused by system services being tricked into using their permissions on behalf of malicious apps (confused deputy attacks [5, 6]), or malicious apps exploiting their own privileges to steal data, and a combination of the two. Researchers have previously shown that such system services are prone to exploits that leverage permissions only available to system services [7]. Likewise, prior work has demonstrated that system services inadvertently or purposely (for functionality reasons) depend on untrusted and possibly malicious apps to help them complete tasks [8].

Such attacks are especially hard to prevent due to two information asymmetries. System services are being exploited when performing tasks on behalf of users, where: (1) users do not know what processing will result from their requests and (2) services do not know what processing users intended when making the request. Current systems employ methods to ask users to authorize program access to sensors, but to reduce users' authorization effort they only ask on a program's first use of that permission. However, once authorized, a program can utilize that permission at will, enabling programs

to spy on users as described above. To prevent such attacks, researchers have explored methods that bind input events, including facets of the user interface used to elicit those inputs, to permissions to perform sensor operations [9, 10, 12]. Such methods ask users to authorize permissions for those events and reuse those permissions when the same event is performed to reduce the user burden. Recent research extends the collection of program execution context (e.g., data flows and/or GUI flows between windows) more comprehensively to elicit user authorizations for sensitive operations [16, 11]. However, none of these methods addresses the challenge where an input event is delivered to one program and then a sensor operation, in response to that event, is requested by another program in a series of inter-process communications, a common occurrence in modern operating systems supporting the cooperating program abstraction.

Researchers have also explored methods to prevent unauthorized access by regulating inter-process communications (IPCs) and by reducing the permissions of programs that perform operations on behalf of other programs. First, prior work developed methods for blocking IPC communications that violate policies specified by app developers [8, 18, 19, 21, 22]. However, such methods may prevent programs from cooperating as expected. Decentralized information flow control [23, 24] methods overcome this problem by allowing programs with the authority to make security decisions and make IPCs that may otherwise be blocked. Second, DIFC methods, like capability-based systems in general [34], enable reduction of a program's permissions (i.e., callee) when performing operations on behalf of other programs (i.e., callers). Initial proposals for reducing permissions simply intersected the parties' permissions [7], which however was too restrictive because parties would have their permissions pruned after the interaction with less privileged parties. DIFC methods, instead, provide more flexibility [20], albeit with the added complexity of requiring programs to make non-trivial security decisions. Our insight to simplify the problem is that while DIFC methods govern information flows comprehensively to prevent the leakage of sensitive data available to programs, users instead want to prevent programs from abusing sensor access to obtain sensitive data in the first place.

In addition, prior work has also investigated the use of machine learning classifiers to analyze the contextuality behind user decisions to grant access to sensors automatically [14, 15]. Unfortunately, the effectiveness of the learning depends on the accuracy of the user decisions while training the learner. Therefore, we firmly believe that additional effort is necessary in improving user decision making before the user decisions can be used to train a classifier.

In this work, we propose the EnTrust authorization system to prevent malicious programs from exploiting cooperating system services to obtain unauthorized access to device sensors. At a high-level, our insight is to combine techniques that regulate IPC communications of programs of different privilege levels with techniques that enable users to be aware of the permissions associated with an input event and decide whether to grant such permissions for the identified flow context. The former techniques identify how a task is "delegated" among cooperating programs to restrict the permissions of the delegatee.[1] The latter techniques expose more contextual information to a user, which may be useful to make effective authorization decisions.

However, combining these two research threads results in several challenges. First, we must be able to associate input events with their resulting sensor operations in other programs to authorize such operations relative to the input events and sequence of cooperating programs. Prior work does not track how processing resulting from input events is delegated across programs [9, 10, 11, 12], but failing to do so results in attack vectors exploitable by an adversary. In EnTrust, we construct *delegation graphs* that associate input events with their resulting sensor operations across IPCs to authorize operations in other programs.

Second, multiple, concurrent input events and IPCs may create ambiguity in tracking delegations across processes that must be resolved to ensure correct enforcement. Prior work either makes assumptions that are often too restrictive or require manual program annotations to express such security decisions. EnTrust leverages the insights that input events are relatively infrequent, processed much more quickly than users can generate distinct events, and are higher priority than other processing. It uses these insights to ensure that an unambiguous *delegation path* can be found connecting each input event and sensor operation, if one exists, with little impact on processing overhead.

Third, we must develop a method to determine the permissions to be associated with an input event for other programs that may perform sensor operations. Past methods, including machine learning techniques [14, 15], depend on user decision making to select the permissions associated with input events, but we wonder whether the information asymmetries arising from delegation of requests across programs impair user decision making. In EnTrust, we elicit authorization decisions from users by using delegation paths. We study the impact of using delegation paths on users' decision making for both primed and unprimed user groups. Historically, there has been a debate on whether users should be considered a weak link in security [56, 57]. We examine this argument in a specific context by investigating if users can make informed security decisions given informative, yet precise, contextual information.

We implement and evaluate a prototype of the EnTrust authorization system for Android OS. We find that EnTrust significantly reduces exploits from three

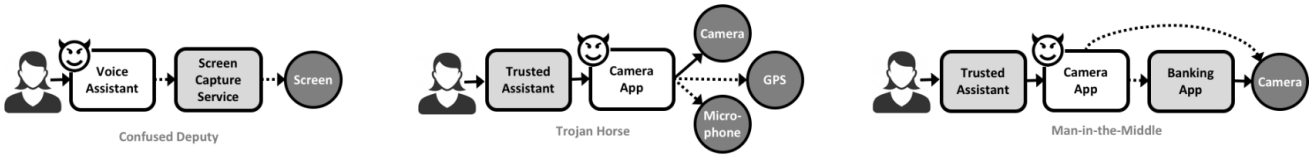


Figure 1: Possible attack vectors when diverse programs interact via input event delegations in a cooperating model. For consistency, we present the attack scenarios in terms of voice assistants receiving input events via voice commands; however, similar attack scenarios are possible for input events received by programs via Graphical User Interface (GUI) widgets rendered on the users' device screen.

canonical types of attack vectors possible in systems supporting cooperating programs, requires little additional user effort, and has low overhead in app performance and memory consumption. In a laboratory study involving 60 human subjects, EnTrust improves attack detection by 47-67% when compared to the first-use authorization approach. In a field study involving 9 human subjects, we found that - in the worst scenarios seen - programs required no more than four additional manual authorizations from users, compared to the less secure first-use authorization approach; which is far below the threshold that is considered at risk for user annoyance and habituation [33]. Lastly, we measured the overhead imposed by EnTrust via benchmarks and found that programs operate effectively under EnTrust, while incurring a negligible performance overhead (<1% slowdown) and a memory footprint of only 5.5 kilobytes, on average, per program.

In summary, we make the following contributions:

- We propose a method for authorizing sensor operations in response to input events performed by cooperating programs by building unambiguous *delegation graphs*. We track IPCs that delegate task processing to other programs without requiring system service or app code modifications.
- We propose EnTrust, an authorization system that generates delegation paths to enable users to authorize sensor operations, resulting from input events, and reuse such authorizations for repeated requests.
- We implement the EnTrust prototype and test its effectiveness with a laboratory study, the users' authorization effort with a field study, and performance and memory overhead via benchmarks.

2 Problem Statement

In current operating systems, users interact with programs that initiate actions targeting sensors, but users do not have control over *which programs* are going to service their requests, or *how such programs access sensors* while servicing such requests. Unfortunately, three well-studied attack vectors become critical in operating systems supporting a cooperating program abstraction.

Confused Deputy — First, a malicious program may leverage an input event as an opportunity to confuse a more privileged program into performing a sensitive operation. For example, a malicious voice assistant may invoke the screen capture service at each voice command (left side of Figure 1). The malicious voice assistant may therefore succeed in tricking the screen capture service into capturing and inadvertently leaking sensitive information (e.g., a credit card number written down in a note). In this scenario, the user only sees the new note created by the notes app, whereas the screen capture goes unnoticed. Currently, there are over 250 voice assistants available to the public on Google Play with over 1 million installs, *many by little known or unknown developers*.

Trojan Horse — Second, a program trusted by the user may delegate the processing of an input event to an untrusted program able to perform the requested task. For example, a trusted voice assistant may activate a camera app to serve the user request to take a selfie (middle of Figure 1). However, the camera app may be a Trojan horse app that takes a picture, but also records a short audio via the microphone, and the user location via GPS (e.g., a spy app[2] installed by a jealous boyfriend stalking on his girlfriend). Researchers reported over 3,500 apps available on Google Play Store that may be used as spyware apps for Intimate Partner Violence (IPV) [25]. In this scenario, the user only sees the picture being taken by the camera app, whereas the voice and location recordings go unnoticed, since a camera app is likely to be granted such permission. Also, the ride-sharing attack in the introduction is another example of this attack. Such attacks are possible because even trusted system services may inadvertently leverage malicious apps and/or rely on unknown apps by using implicit intents. An implicit intent enables any program registered to receive such intents to respond to IPCs when such intents are invoked. Researchers have reported several ways how programs can steal or spoof intents intended for other programs [26, 27, 28]. We performed an analysis of system services and applications distributed via the Android Open Source Project (AOSP), and found that 10 system programs out of a total of 69 (14%) use implicit intents.

Man-In-The-Middle — Third, a request generated by a program trusted by the user may be intercepted by a malicious program, which can behave as a man-in-the-middle in serving the input event in the attempt to obtain access to unauthorized data (right side of Figure 1). For example, a legitimate banking app may adopt the voice interaction intent mechanism to allow

customers to direct deposit a check via voice assistant with a simple voice command (e.g., "deposit check").[3] A malicious program may exploit such a service by registering itself with a voice assistant as able to service a similar voice interaction, such as "deposit *bank* check." Therefore, whenever the user instantiates the "deposit *bank* check" voice command, although the user expects the legitimate banking app to be activated, the malicious app is activated instead. The malicious app opens the camera, captures a frame with the check, and sends a spoofed intent to launch the legitimate banking app, all while running in the background. In this scenario, the user only sees the trusted banking app opening a camera preview to take a picture of the check. This is a realistic threat. We performed an analysis of 1,000 apps (among the top 2,000 most downloaded apps on Google Play Store) and found that 227 apps (23%) export at least a public service or a voice interaction intent. Apps were selected from the Google Play Store among those apps declaring at least one permission to access a sensitive sensor (e.g., camera, microphone, or GPS).

Security Guarantee. To mitigate such attack vectors, an authorization mechanism must provide the following guarantee, *for any sensor operation to be authorized, that operation must be: (1) initiated by an input event; (2) authorized for the input event to trigger the sensor operation; and (3) authorized for the sequence of programs receiving the input event directly or indirectly through IPCs leading to the program performing the sensor operation.* Such a guarantee ensures that any sensor operation must be initiated by an input event, the input event must imply authorization of the resultant sensor operation by the requesting program, and all programs associated with communicating the request for the sensor operation must be authorized to enable the sensitive data to be collected by the requesting program. To achieve the security guarantee above, we require a mechanism that accurately tracks the delegations leading from input events to resulting sensor operations, as well as a mechanism to authorize sensor operations to collect sensitive data given input events and delegations.

Regarding tracking delegations, a problem is that determining whether an IPC derives from an input event or receipt of a prior IPC depends on the data flows produced by the program implementations in general. Solving this problem requires data flow tracking, such as performed by taint tracking. However, taint tracking has downsides that we aim to avoid. Static taint tracking can be hard to use and be imprecise [30] and dynamic taint tracking has non-trivial overhead [29]. Instead, we aim to explore solutions that ensure all sensor operations resulting from an input event are detected (i.e., we overapproximate flows) without heavyweight analysis or program modifications.

Authorizing sensor operations to collect sensitive data, given an input event and one or more delegations, depends on determining the parties involved in the delegation as well as the user's intent when generating the event. Methods that restrict the permissions of an operation to the intersection of permissions granted to the parties involved [7], have been found to be too restrictive in practice. Decentralized information flow control [23, 24] (DIFC) prevents information leakage while allowing some privileged programs to make flexible security decisions to determine when to permit communications that are normally unauthorized, which has been applied to mobile systems [20, 13]. However, these information flow control techniques focus on preventing the leakage of sensitive information available to programs, whereas the main goal here is to prevent programs from obtaining access to sensitive information in the first place by abusing sensor access. To address this problem more directly, researchers have explored techniques that enable users to express the intent of their input events to authorize sensor operations, binding this intent to the context in which the input event was elicited, such as the graphical user interface (GUI) context [9, 10, 11]. In IoT environments, researchers have similarly explored gathering program execution context (e.g., data flows) to enable users to authorize IoT operations more accurately [16]. However, none of these techniques account for delegations of tasks to other processes. We aim to explore methods for eliciting user authorizations for sensor operations using contextual information related to the tracking of input events and subsequent delegations.

Further, researchers have explored learning methods to predict permissions for sensor operation based on prior user decisions [14, 15]. However, accurate user decision making is vital for improving the accuracy of these learning techniques.

3 Security Model

Trust Model – We assume that the system (e.g., Linux kernel, operating system, system services, and device drivers) is booted securely, runs approved code from device vendors, and is free of malice; user-level programs (e.g., applications) are isolated from each other via the sandboxing mechanism using separated processes [35, 36]; and, by default, user-level programs have no direct access to sensors due to the use of a Mandatory Access Control (MAC) policy [37, 38] enforced from boot time. We assume the use of *trusted paths*, protected by MAC, allowing users to receive unforgeable communications from the system, and providing unforgeable input events to the system. Our assumptions are in line with existing research on trusted paths and trusted user interfaces for browsers [39], X window systems [40, 41], and mobile operating systems [42].

Threat Model – We assume that users may install programs from unknown sources that may be malicious, then grant such programs access to sensors at first use. Despite the default isolation via sandboxing, programs may communicate via IPC mechanisms (i.e., intents or

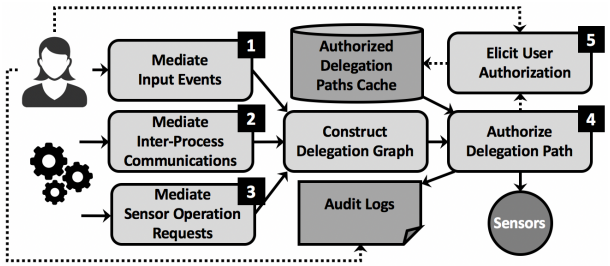


Figure 2: EnTrust Authorization Method – Input events, handoff events, and sensor operations are linked via delegation graphs to compute unambiguous delegation paths for user authorization of sensor operations.

broadcast messages). Thus, user-level programs (e.g., apps) may leverage such communication to exploit the attack vectors described in Section 2. Our objective is to provide a mechanism that helps users control how cooperating programs access sensors. *How programs manage and share the data collected from sensors is outside the scope of our research.* Researchers have already examined solutions to prevent data leakage based on taint analysis [29, 30, 31, 18] and Decentralized Information Flow Control (DIFC) [20, 23, 24, 32].

4 EnTrust Authorization Design

In this section, we describe our proposed framework, EnTrust, designed to restrict when programs may perform sensor operations by requiring each sensor operation to be unambiguously associated with an input event, even if the sensor operation is performed by a program different from the one receiving the input event. Figure 2 provides an overview of the EnTrust authorization system, which consists of five steps. In the first three steps, EnTrust mediates and records input events, inter-process communication events (handoff events), and sensor operation requests, respectively, to construct a *delegation graph* connecting input events to their handoff events and sensor operation requests. In the fourth step, EnTrust uses the constructed delegation graph to compute an unambiguous *delegation path* to a sensor operation request from its originating input event. Unless the authorization cache contains a user authorization for the constructed delegation path already, the fifth step elicits an authorization from the user for the delegation path, and caches the authorization for later use for the same delegation path. Optionally, users can review their prior decisions and correct them via an audit mechanism that logs past authorized and denied delegation graphs.

4.1 Building Delegation Graphs

The first challenge is to link input events to all the sensor operations that result from cooperating programs processing those events and then construct *delegation graphs* rooted at such input events.

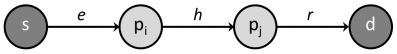


Figure 3: Delegation graphs connect input events with operation requests for sensors via handoff events.

First, for each input event received via a sensor s for a program p_i, EnTrust creates an *input event* tuple $e = (c, s, p_i, t_0)$, where c is the user interface context captured at the moment the input event occurred; s is the sensor through which the event was generated; p_i is the program displaying its graphical user interface on the screen and receiving the input event e; and t_0 is the time of the input event (step 1 in Figure 2). Note: EnTrust is designed to mediate both input events coming from input sensors (e.g., touch events on widgets rendered on the screen) as well as voice commands captured via the microphone. Voice commands are translated into text by the Google Cloud Speech-to-Text service.

Second, after receiving the input event, program p_i may hand off the event to another program p_j. EnTrust mediates handoff events by intercepting spawned intents and messages exchanged between programs [43] and models them as tuples $h = (p_i, p_j, t_i)$, where p_i is the program delegating the input event, p_j is the program receiving the event, and t_i is the time the event delegation occurred (step 2 in Figure 2).

Third, when the program p_j generates a request r for an operation o targeting a sensor d, EnTrust models the request as a tuple $r = (p_j, o, d, t_j)$, where p_j is the program requesting the sensor operation, o is the type of sensor operation requested, d is the destination sensor, and t_j is the time the sensor operation request occurred (step 3 in Figure 2).

Lastly, EnTrust connects sensor operation requests to input events via handoff events by constructing a delegation graph to regulate such operations, as shown in Figure 3. A *delegation graph* is a graph, $G = (V, E)$, where the edges $(u, v) \in E$ represent the flow of input events to programs and sensors, and the vertices, $v \in V$, represent the affected programs and sensors. Figure 3 shows a simple flow, whereby a source sensor s receives an input event e that is delivered to a program p_i, which performs a handoff event h to a program p_j that performs an operation request r for a destination sensor d. Thus, there are three types of edges: input event to program (user input delivery), program to program (handoff), and program to sensor operation request (request delivery).

Upon mediation of a sensor request r, EnTrust computes the associated *delegation path* by tracing backwards from the sensor request r to the original input event e. Hence, the operation request $r = (p_j, o, d, t_j)$ above causes a delegation path: $(c, s, p_i, t_0) \rightarrow (p_i, p_j, t_i) \rightarrow (p_j, o, d, t_j)$ to be reported in step 4 in Figure 2. Delegation paths are then presented to the user for authorization (see Section 4.3). The identified delegation path is shown to the user using natural language, in a



Figure 4: Two scenarios that create ambiguity. Multiple input events or handoff events delivered to the same program.

manner similar to first-use authorizations. We assess how effectively users utilize delegation paths to produce authorizations in a laboratory study in Section 6.2.

4.2 Computing Delegation Paths

Critical to computing delegation paths is the ability for EnTrust to find an unambiguous reverse path from the sensor operation request r back to an input event e. In particular, a delegation path is said to be *unambiguous* if and only if, given an operation request r by a program p_j for a sensor d, either there was a single input event e for program p_j that preceded the request r, or there was a single path $p_i \rightarrow p_j$ in the delegation graph, where program p_i received a single input event e.

To ensure unambiguous delegation paths without program modification, we need to define the conditions under which operations that create ambiguities cannot occur. First, ambiguity occurs if the same program p_i receives multiple input events and then performs a handoff, as depicted by the left side of Figure 4. In this case, it is unclear which one of the input events resulted in the handoff. To prevent this ambiguous case, we leverage the insight that input events are relatively infrequent, processed much more quickly than users can generate them, and have a higher priority than other processing. We observe that the time between distinct input events is much larger than the time needed to produce the operation request corresponding to the first input event. If every input event results in an operation request before the user can even produce another distinct input event, then there will be only one input event (edge) e from a source sensor (node) s to program (node) p_i, which received such input event. Therefore, there will be no ambiguous input event for program p_i. Thus, we propose to set a time limit for each input event, such that the difference between the time t_0 at which an input event e is generated and the time t_j for any sensor operation request r – based on that input event – must be below that limit for the event to be processed. Note that, once an input event is authorized (Section 4.3), repeated input events (e.g., pressing down a button multiple times) are not delayed. Indeed, repeated input events are expected to generate the same delegation path. Should the programs produce a different delegation path - in the middle of a sequence of operations spawned in this manner - then EnTrust would require a new authorization for the new delegation path, as described in Section 4.3.

Second, ambiguity is also possible if the same program p_j receives multiple handoff events before performing a sensor operation request, as depicted by the right side of Figure 4. Note that, handoff events may not be related to input events (e.g., intents not derived from input events). In this case, it is unclear which handoff is associated with a subsequent sensor operation request. Ambiguity prevention for handoff events is more subtle, but builds on the approach used to prevent ambiguity for input events. Figure 5 shows the key challenge. Suppose a malicious program p_k tries to "steal" a user authorization for a program p_j to perform a sensor operation by submitting a handoff event that will be processed concurrently to the handoff event from another program p_i, which received an input event. Should a sensor operation request occur, EnTrust cannot determine whether the sensor operation request from p_j was generated in response to the event handoff h_1 or to the event handoff h_2. So EnTrust cannot determine the delegation path unambiguously to authorize the operation request. If EnTrust knows the mapping between actions associated to handoff events and whether they are linked to sensor operations, EnTrust can block a handoff from p_k that states an action that requires an input event. EnTrust knows this mapping for system services, by having visibility of all inter-procedural calls for programs part of the operating system; however, EnTrust may not know such mapping for third-party apps whose inter-procedural control flow is not mediated to favor backward compatibility with existing apps.

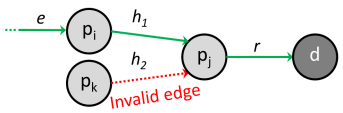


Figure 5: A program p_k attempts leveraging the input event received from program p_i to get program p_j to generate an operation request.

Thus, we extend the defense for input events to prevent ambiguity as follows: once the target program has begun processing a handoff associated with an input event, EnTrust delays the delivery of subsequent handoff events until this processing completes or until the assigned time limit ends. Conceptually, this approach is analogous to placing a readers-writers lock [44] over programs that may receive handoffs that result from input events. Note that, the use of time limits ensures no deadlock since it ensures preemption. To avoid starving input events (e.g., delaying them until the time limit), we prioritize delivery of handoffs that derive from input events ahead of other handoffs using a simple, two-level scheduling approach. We assess the impact of the proposed ambiguity prevention mechanisms on existing programs' functionality and performance in Section 7.

4.3 Authorizing Delegation Paths

For controlling when a sensor operation may be performed as the result of an input event, users are in the best position to judge the intent of their actions. This is

inline with prior work advocating that it is highly desirable to put the user in context when making permission granting decisions at runtime [16, 33, 17]. Therefore, users must be the parties to make the final authorization decisions. To achieve this objective, EnTrust elicits an explicit user authorization every time that a new delegation path is constructed (step 5 in Figure 2). Hence, to express a delegation path comprehensively, EnTrust builds an authorization request that specifies that delegation path to the user. Prior work presented users with information about the Graphical User Interface (GUI) used to elicit the input event, including GUI components [9, 10, 11], user interface workflows [13], and Application Programming Interface (API) calls made by applications [11, 16]. EnTrust, instead, presents the delegation path that led to the sensor operation, which includes the GUI context (c in the input event) and the handoffs and sensor operations. As a result, EnTrust ensures that all the programs receiving sensor data are clearly identified and reported in the authorization request presented to the user, along with the input event, handoff events, and the resulting sensor operation.

To reduce users' authorization effort, EnTrust caches authorized delegation paths for reuse. After storing an authorized delegation path, EnTrust proceeds in allowing the authorized sensor operation. For subsequent instances of the same input event that results in exactly the same delegation path, EnTrust omits step 5 and automatically authorizes the sensor operation by leveraging the cached authorization. Note that, EnTrust requires an explicit user's authorization *only* the first time a delegation path is constructed for a specific input event, similarly to the first-use permission approach. As long as the program receiving an input event does not change the way it processes that event (i.e., same handoffs and operation request), no further user authorization will be necessary. In Section 6.2, we show that such an approach does not prohibitively increase the number of access control decisions that users have to make, thus avoiding decision fatigue [45].

Further, EnTrust evicts cached authorizations in two scenarios. First, if a new delegation path is identified for an input event that already has a cached delegation path, then EnTrust evicts the cached authorization and requires a new user authorization for the newly constructed delegation path, before associating it to such an input event and caching it. Second, users can leverage an audit mechanism, similar to proposals in related work [11, 15], to review previous authorizations and correct them if needed. Denied authorizations are also logged for two reasons. First, they allow users to have a complete view of their past decisions; but more importantly, they allow EnTrust to prevent malicious programs from annoying users by generating authorization requests over a give threshold, for operations already denied by the user in the past. Also, users may set the lifetime of cached authorizations, after which they are evicted. We discuss utilizing such logs for revoking mistaken authorizations and denials in Section 8.

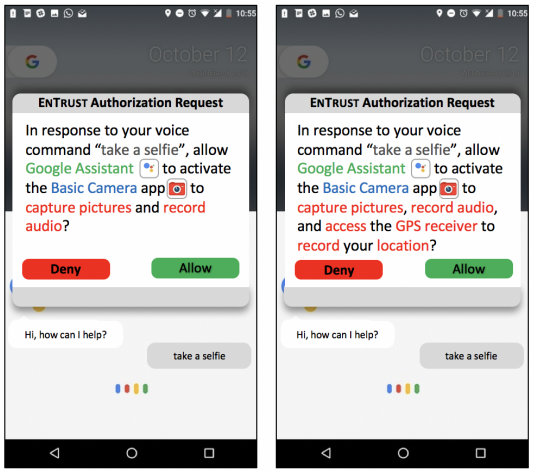


Figure 6: Authorization requests prompted to users by EnTrust upon *delegation paths* creation. Screenshots showing benign (left) and attack (right) scenarios by the Basic Camera app.

5 Implementation

We implemented a prototype of the EnTrust authorization system by modifying a recent release of the Android OS (Android-7.1.1_r3) available via the Android Open Source Project (AOSP).[4] The choice of implementing the EnTrust prototype for the Android OS was guided by its open-source nature and its wide adoption. EnTrust's footprint is 170 SLOC in C for the Linux kernel (bullhead 3.10), plus 380 SLOC in C, 830 SLOC in C++, and 770 SLOC in Java for components in the Android OS.

In this section, we provide the implementation details for event scheduling and authorization management. In Appendices C-F, we provide further implementation details regarding event authentication and mediation.

In Android, the Event Hub (part of the Input Manager server) reads raw input events from the input device driver files (`/dev/input/*`) and delivers them to the Input Reader. The Input Reader then formats the raw data and creates input event data that is delivered to the Input Dispatcher. The Input Dispatcher then consults the Window Manager to identify the target program based on the activity window currently displayed on the screen. Hence, we enhanced the Input Dispatcher to hold - for the duration of a time window - incoming input events for a target program should there be already a delivered input event for such a program that has not been processed, yet. For handoff events, instead, the Binder is the single point of mediation for inter-process communication (IPC) between isolated programs. It has the knowledge of all the pending messages exchanged as well as knowledge of the identity

of the two communicating parties. Hence, we also enhanced the Binder to hold - for the duration of a time window - incoming handoff events for a target program should the program be already involved in another communication with a third program.

EnTrust prompts users with *authorization messages* for explicit authorizations of delegation paths, as shown in Figure 6. Users are made *aware of all the programs cooperating in serving their requests* as well as of the *entire delegation path*. Also, users are prompted with programs' names and identity marks to ease their identification. EnTrust crosschecks developers' signatures and apps' identity (i.e., names and logos) by pulling information from the official Google Play Store to prevent identity spoofing. Also, EnTrust prevents programs from creating windows that overlap the *authorization messages* by leveraging the Android screen overlay protection mechanism. Finally, EnTrust prevents unauthorized modification of *authorization messages* by other programs by using isolated per-window processes forked from the Window Manager to implement a *Compartmented Mode Workstation* model [48].

6 EnTrust Evaluation

We investigated the following research questions:

▶ *To what degree is the* EnTrust *authorization assisting users in avoiding confused deputy, Trojan horse, and man-in-the-middle attacks?* We performed a *laboratory study* and found that EnTrust significantly increased (from 47-67% improvement) the ability of participants in avoiding attacks.

▶ *What is the decision overhead imposed by* EnTrust *on users due to explicit authorization of constructed delegation graphs?* We performed a *field study* and found that the number of decisions imposed on users by EnTrust remained confined - in worst case scenarios - to no more than 4 explicit authorizations per program.

▶ *Is* EnTrust *backward compatible with existing programs? How many operations from legitimate programs are incorrectly blocked by* EnTrust*?* We used a well-known compatibility test suite to evaluate the compatibility of EnTrust with 1,000 apps (selected among the most popular apps on Google Play Store) and found that EnTrust does not cause the failure of any program.

▶ *What is the performance overhead imposed by* EnTrust *for delegation graph construction and enforcement?* We used a well-known software exerciser to measure the performance overhead imposed by EnTrust. We found that EnTrust introduced a negligible overhead (order of milliseconds) unlikely noticeable to users.

6.1 Study Preliminaries

We designed our user studies following suggested practices for human subject studies in security to avoid common pitfalls in conducting and writing about security and privacy human subject research [49]. An Institutional Review Board (IRB) approval was obtained from our institution. The data collected did not contain Personally Identifiable Information (PII) and was securely stored and accessible only to authorized researchers. We recruited study participants via local mailing lists, Craigslist, Twitter, and local groups on Facebook. We compensated them with a $5 gift card. We excluded acquaintances from participating in the studies to avoid acquiescence bias. Before starting the study, participants had to sign our consent form and complete an entry survey containing demographic questions. We made sure to get a wide diversity of subjects, both in terms of age and experience with technology (details available in Appendix A). For all the experiments, we configured the test environment on LG Google Nexus 5X phones running the Android 7.1 Nougat OS. We used a background service, automatically relaunched at boot time, to log participants' responses to system messages and alerts, input events generated by participants while interacting with the testing programs, as well as system events and inter-process communications between programs. Furthermore, during the experiments, the researchers took note of comments made by participants to ease the analysis of user decision making.

6.2 Laboratory Study

We performed a *laboratory study* to evaluate the effectiveness of EnTrust in supporting users in avoiding all the three attack vectors previously identified in Section 2. We compared EnTrust with the *first-use* authorization used in commercial systems. We could not compare mechanisms proposed in related work [9, 10], because they are unable to handle handoff events. We divided participants into four groups, participants in `Group-FR-U` and `Group-FR-P` interacted with a *stock* Android OS implementing the *first-use* authorization mechanism. Participants in `Group-EN-U` and `Group-EN-P` interacted with a *modified* version of the Android OS integrating the EnTrust authorization system. To account for the priming effect, we avoided influencing subjects in `Group-FR-U` and `Group-EN-U` and advertised the test as a generic "voice assistants testing" study without mentioning security implications. On the other hand, to assess the impact of priming, subjects in `Group-FR-P` and `Group-EN-P` were informed that attacks targeting sensors (e.g., camera, microphone, and GPS receiver) were possible during the interaction with programs involved in the experimental tasks, but without specifying what program performed the attacks or what attacks were performed.

Experimental Procedures: For our experiment, we used a test assistant developed in our research lab called Smart Assistant, which provides basic virtual assistant functionality, such as voice search, message composition, and note keeping. However, Smart Assistant is also designed to perform confused deputy attacks on

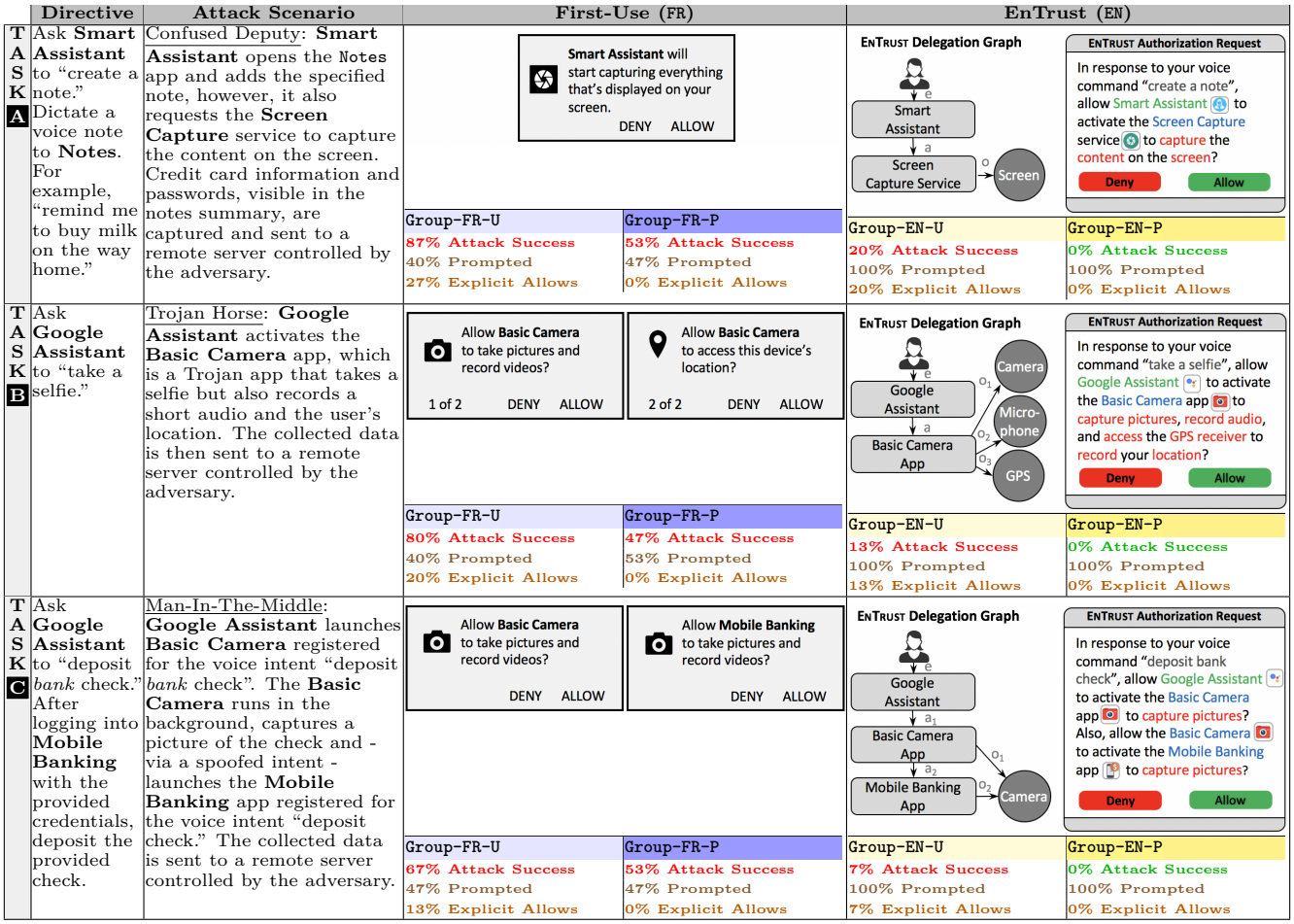


Table 1: Experimental tasks for the laboratory study, derived from the attack vectors described in Section 2. We report the authorization messages shown to subjects in the four groups as well as the delegation graphs used by EnTrust to construct such authorization messages. In the group names, the suffix U indicates *unprimed* subjects, whereas P indicates *primed* subjects. Notice that, authorization requests prompted by EnTrust include programs' identity marks (i.e., apps' icon and unique id).

system services, such as the Screen Capture service. We also used a test app, Basic Camera, developed in our research lab. It provides basic camera functionality, such as capturing pictures or videos and applying photographic filters. However, Basic Camera is also designed to perform man-in-the-middle and Trojan horse attacks for requests to capture photographic frames. Lastly, we used a legitimate Mobile Banking app, from a major international bank, available on Google Play Store. Apart from the testing apps and voice assistant, the smartphone provided to participants had pre-installed both the Google Assistant and the Android Camera app.

Our laboratory study was divided into two phases. A *preliminary phase* during which no attacks were performed. This phase enabled participants to familiarize themselves with the provided smartphone, the pre-installed apps and the voice assistants. This phase avoided a "cold start" and approximated a more realistic scenario in which users have some experience using relevant apps and voice assistants. Furthermore, this preliminary phase enabled capturing how malicious programs may leverage pre-authorized operations in the first-use approach to then perform operations not expected by the users; a malicious behavior that is instead prevented by EnTrust via the construction of per-delegation authorizations. The preliminary phase was then followed by an *attack phase*, during which participants interacted with programs performing attacks. Participants were not made aware of the existence of the two experimental phases nor of the difference between the two phases.

All instructions regarding experimental tasks to be performed were provided to participants in writing via a handout at the beginning of each experimental task. During the *preliminary phase* the participants performed the following three tasks: (1) asked a voice assistant to "take a screenshot;" (2) asked a voice assistant to "record a memo;" and (3) used a camera app to "record a video." During the *attack phase*, instead, the participants performed the three tasks described in Table 1. In each phase, each participant was presented with a different randomized order of the above tasks.

Experimental Results: In total, 60 subjects participated in and completed our laboratory study. We *randomly* assigned 15 participants to each group. In this study, *we did not observe denials of legitimate operations* for sensitive sensors for non-attack tasks performed during the preliminary phase, but we discuss the need for more study on preventing and resolving mistaken denials in Section 8. Table 1 summarizes the results of the three experimental tasks for the *attack phase*. Our focus was to study the effectiveness of EnTrust in reducing the success rate of attacks when compared to the first-use approach. During the *preliminary phase* and the *experimental tasks*, *all the participants were prompted* with the corresponding authorization messages depending on the group to which they were assigned,[5] as reported in Table 1. Prompted authorizations included legitimate operations, see left side of Figure 6 for an example of what a prompt for a legitimate operation looked like. Our analysis reports that each participant of each group was prompted at least 4 times for non-attack operations. Note that, as per definition of first-use authorization, participants in Group-FR-U and Group-FR-P were not prompted with authorization messages once again should they have already authorized the program in a previous task or during the preliminary phase. Instead, participants in Group-EN-U and participants in Group-EN-P were presented with a new authorization message any time a new delegation path was identified by EnTrust. This explains the lower percentage of subjects prompted, with an authorization request, in the first-use groups.

TASK A: The analysis of subjects' responses revealed that 9 subjects from Group-FR-U and 8 subjects from Group-FR-P interacted with Smart Assistant during the preliminary phase, or during another task, to "take a screenshot" and granted the app permission to capture their screen; thus, they were not prompted once again with an authorization message during this task, as per default in first-use permissions. In addition, 4 subjects from Group-FR-U explicitly allowed Smart Assistant to capture their screen, therefore, resulting in a 87% and 53% attack success, respectively, as reported in Table 1. On the contrary, only 3 subjects from Group-EN-U and no subject from Group-EN-P allowed the attack (20% and 0% attack success, respectively). Also, similarly to what happened in Group-FR-U and Group-FR-P, 8 subjects from Group-EN-U and 8 subjects from Group-EN-P interacted with Smart Assistant during the preliminary phase and asked to "take a screenshot." However, since the voice command "create a note" was a different command, EnTrust prompted all subjects with a new authorization message, as shown in Table 1.

TASK B: The analysis of subjects' responses revealed that 9 subjects from Group-FR-U and 7 subjects from Group-FR-P interacted with Basic Camera to take a picture or record a video, either during the preliminary phase or during another task, and authorized it to capture pictures, audio, and access the device's location. Thus, they were not prompted once again during this task as per default in first-use permissions. Also, we found that 3 subjects from Group-FR-U explicitly authorized Basic Camera to access the camera, as well as the microphone, and the GPS receiver; therefore, resulting in 80% and 47% attack success, respectively. In contrast, 2 subjects from Group-EN-U and no subject from Group-EN-P authorized access to the camera, microphone, and GPS receiver (13% and 0% attack success, respectively). Also, we found that 8 subjects from Group-EN-U and 6 subjects from Group-EN-P interacted with Basic Camera during the preliminary phase or during another task. However, none of them asked to "take a selfie" before, so all subjects were prompted by EnTrust with a new authorization message. At the end of the experiment, among all the subjects, when asked why they authorized access to the GPS receiver, the majority said that they expected a camera app to access location to create geo-tag metadata when taking a picture. In contrast, the subjects who denied access stated not feeling comfortable sharing their location *when taking a selfie*.

TASK C: The analysis of subjects' responses revealed that 8 subjects from Group-FR-U and 8 subjects from Group-FR-P interacted with Basic Camera, either during the preliminary phase or during another task, and authorized the app to capture pictures. Thus, during this task, they were not prompted with an authorization message once again as per default in first-use permissions. They were only prompted to grant permission to Mobile Banking, explaining why even the primed subjects were not able to detect the attack. In addition, 2 subjects from Group-FR-U explicitly authorized Basic Camera to capture a frame with the bank check; therefore, resulting in 67% and 53% attack success, respectively. On the other hand, only 1 subject from Group-EN-U and no subject from Group-EN-P authorized Basic Camera to capture a frame with the bank check, resulting in a 7% and 0% attack success, respectively. Notice that all subjects from Group-EN-U and Group-EN-P were prompted with a new authorization message by EnTrust for the new command "deposit *bank* check." Interestingly, the one subject from Group-EN-U, who allowed Basic Camera to capture a frame with the bank check, verbally expressed his concern about the permission notification presented on the screen. The subject stated observing that two apps asked permission to access the camera to take pictures. This is reasonable for an unprimed subject not expecting a malicious behavior.

Discussion: Comparing the results from Group-FR-U versus those from Group-FR-P, and those from Group-EN-U versus those from Group-EN-P, we observe - as expected - that primed subjects allowed fewer attacks. We find that users primed for security problems still fall victim to attacks due to first-use authorization,

even when rejecting all the malicious operations they see. On the other hand, unprimed users fail to detect attacks between 7-20% with EnTrust. So while this is a marked improvement, over the 67-87% failure for users with first-use authorization, there is room for further improvement. However, it is apparent that the delegation graphs constructed by EnTrust aided the subjects in avoiding attacks even when unprimed. EnTrust performed slightly better than first-use authorization in terms of explicit authorizations (explicit allows in Table 1); which suggests that the additional information provided by EnTrust in authorization messages (i.e., programs' name and identity mark as well as delegation information, as shown in Figure 6) may be helpful to users in avoiding unexpected program behaviors. We verified the hypothesis that the information in EnTrust authorizations helps *unprimed* users identify attacks by calculating the difference in *explicit allows*, across the three experimental tasks, for subjects in Group-FR-U versus subjects in Group-EN-U. Our analysis indeed revealed a statistically significant difference ($\chi^2 = 19.3966$; p = 0.000011).

Also, EnTrust was significantly more effective than first-use in keeping users "on guard" independently of whether subjects were primed (47-67% lower attack success with EnTrust). Indeed, different from the first-use approach, EnTrust was able to highlight whether pre-authorized programs attempted accessing sensors via unauthorized delegation paths. If so, EnTrust prompted users for an explicit authorization for the newly identified delegation path. We verified the hypothesis that EnTrust better helps *primed* and *unprimed* users in preventing attacks than first-use, by calculating the difference in *successful attacks*, across the three experimental tasks, for subjects in Group-FR-U and Group-FR-P, versus subjects in Group-EN-U and Group-EN-P. Our analysis indeed revealed a statistically significant difference ($\chi^2 = 65.5603$; p = 0.00001). Normally, the standard Bonferroni correction would be applied for multiple testing, but due to the small p-values such a correction was not necessary.

6.3 Field Study

We performed a *field study* to evaluate whether EnTrust increases the decision-overhead imposed on users. We measured the number of explicit authorizations users had to make when interacting with EnTrust under realistic and practical conditions, and compared it with the first-use approach adopted in commercial systems (i.e., Android OS and Apple iOS). We also measured the number of authorizations handled by EnTrust via the cache mechanism that, transparently to users, granted authorized operations.

Experimental Procedures: Participants met with one of our researchers to set up the loaner device, an LG Nexus 5X smartphone running a *modified* version of the Android OS integrating the EnTrust authorization framework. The loaner device had pre-installed 5 voice assistants and 10 apps selected among the most popular[6] with up to millions of downloads from the official Google Play store. For such programs, to ensure the confidentiality of participants' personal information, mock accounts were set up instead of real accounts for all apps requiring a log-in. To facilitate daily use of the loaner device, the researcher transferred participants' SIM cards and data, as well as participants' apps in the loaner device, however no data was collected from such apps. The above protocol was a requirement for the IRB approval by our Institution and it is compliant with the protocol followed in related work [33, 15, 11]. Before loaning the device, the researcher asked each participant to use the loaner device for their everyday tasks for a period of 7 days. In addition to their everyday tasks, participants were asked to explore each of the pre-installed voice assistants and apps, at least once a day, by interacting as they would normally do. Particularly, we asked the participants to interact with each voice assistant by asking the following three questions: (1) "capture a screenshot," (2) "record a voice note," (3) "how long does it take to drive back home." Additionally, we asked participants to be creative and ask three additional questions of their choice. Table 2 summarizes all the assistants and apps pre-installed on the smartphones for the field study. Because the mere purpose of our field study was to measure the decision-overhead imposed to users by EnTrust and to avoid participants' bias, the researcher advertised the study as a generic "voice assistants and apps testing" study without mentioning security implications or training the users about the features provided by EnTrust. The smartphones provided to participants were running a background service with runtime logging enabled, automatically restarted at boot time, to monitor the number of times each program was launched, the users' input events, the constructed delegation graphs, the authorization decisions made by the participants, and the number of authorizations automatically granted by EnTrust. The background service also measured the gaps between consecutive input events and handoff events, as well as the time required by each program to service each event. This data was used to perform the time constraints analysis reported in Appendix B.

Experimental Results: Nine subjects participated and completed the field study. The data collected during our experiment indicates that all user authorizations were obtained within the first 72 hours of interaction with the experimental device, after which we observed only operations automatically granted by EnTrust via the caching mechanism.

The first subject allowed us to discover two implementation issues that affected the number of explicit authorizations required by EnTrust. First, changing the orientation of the screen (portrait versus landscape) was causing EnTrust to request a new explicit user autho-

	Expl. Authorizations		Impl. Authorizations
	First-Use	EnTrust	in s 7 Days Period
Snapchat	3	3	276
YouTube	3	3	84
Facebook Messenger	2	2	93
Instagram	3	3	393
Facebook	3	3	117
Whatsapp	2	2	76
Skype	3	3	100
WeChat	2	2	101
Reddit	1	1	18
Bitmoji	3	3	127
Google Assistant	1	**4**	72
Microsoft Cortana	1	**3**	49
Amazon Alexa	1	**4**	84
Samsung Bixby	1	**4**	63
Lyra Virtual Assistant	1	**3**	56

Table 2: Apps and voice assistants tested in the field study. The last column shows the number of operations automatically authorized by EnTrust after user's authorization.

rization for an already authorized widget whenever the screen orientation changed. This inconvenience was due to the change in some of the features used to model the context within which the widget was presented. To address this shortcoming, we modified our original prototype to force the Window Manager to generate in memory two graphical user interfaces for both screen orientations to allow EnTrust to bind them with a specific widget presented on the screen. Second, for the voice commands, we noticed that differently phrased voice commands with the same meaning would be identified as different input events. For instance, "take a selfie" and "take a picture of me". This shortcoming was causing EnTrust to generate a new delegation graph for each differently phrased voice command. To address this issue, we leveraged the *Dialogflow* engine by Google, part of the AI.API.[7] *Dialogflow* is a development suite for building conversational interfaces and provides a database of synonyms to group together voice commands with the same meaning. We fixed the issues and continued our experiment with other subjects.

Table 2 reports the average number of explicit authorizations performed by the subjects. We compared them with the number of explicit authorizations that would be necessary if the *first-use* permission mechanism was used instead. The results show that EnTrust required the same number of explicit authorizations by users for all the tested apps. For all voice assistants, instead, EnTrust may require up to 3 additional explicit authorizations, when compared with the *first-use* approach; which is far below the 8 additional explicit authorizations used in prior work, which are considered likely not to introduce significant risk of habituation or annoyance [33]. These additional authorizations are due to the fact that with the *first-use* approach the programs (activated by the voice assistant to serve the user request) may have already received the required permissions to access the sensitive sensors. EnTrust instead captures the entire sequence of events, from the input event to any subsequent action or operation request, and then ties them together. Therefore, EnTrust constructs a new graph for each novel interaction. Nonetheless, the number of decisions imposed on the users remains very modest. Indeed, on average, three additional explicit user authorizations are required per voice assistant. Also, the number of explicit authorizations made by the users remained a constant factor compared to the number of operations implicitly authorized by EnTrust, which instead grew linearly over time. We measured an average of 16 operations implicitly authorized by EnTrust during a 24-hour period (last column of Table 2). Therefore, if we consider such a daily average number of implicitly authorized operations for a period of one year, we will have on the order of thousands of operations automatically authorized by EnTrust, which would not require additional explicit effort for the users.

6.4 Backward Compatibility Analysis

To verify that EnTrust is backward compatible with existing programs, we used the Compatibility Test Suite (CTS),[8] an automated testing tool released by Google via the AOSP.[9] In particular, this analysis verified that possible delays in the delivery of events introduced by EnTrust or the change in scheduling of events did not impact applications' functionality. We tested the compatibility of EnTrust with 1,000 existing apps, among the top 2,000 most downloaded apps on Google Play Store, selected based on those declaring permissions to access sensitive sensors in their manifest. The experiment took 19 hours and 45 minutes to complete, and EnTrust passed 132,681 tests without crashing the operating system and without incorrectly blocking any legitimate operation. Among the 1,000 tested apps, we also included 5 popular augmented reality multiplayer gaming app (`InGress`, `Pokémon Go`, `Parallel Kingdom`, `Run An Empire`, and `Father.io`), which typically have a high rate of input events and are very sensitive to delays. The set of tests targeting these 5 gaming apps ran for 16 minutes, during which we continuously observed the device screen to identify possible issues in terms of responsiveness to input events or glitches in the rendering of virtual objects on the screen. However, we did not identify any discernible slowdown, glitch, or responsiveness issue.

7 Performance Measurements

We performed four micro-benchmarks on a standard Android developer smartphone, the LG Nexus 5X, powered by 1.8GHz hexa-core 64-bit Qualcomm Snapdragon 808 Processor and Adreno 418 GPU, 2GB of RAM, and 16GB of internal storage. All of our benchmarks are measured using Android 7.1 Nougat pulled from the Android Open Source Project (AOSP) repository.

Delegation Graph Construction – Our first micro-benchmark of EnTrust measured the overhead incurred for constructing delegation graphs of varying sizes. To do this, we had several programs interacting

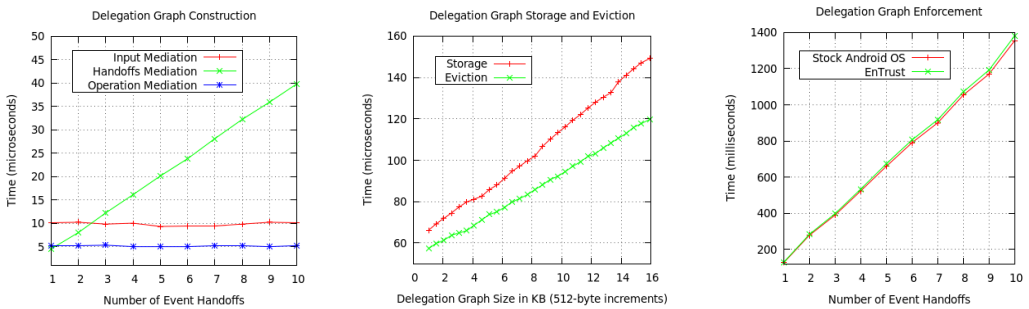


Figure 7: Overheads for Delegation Graphs Construction, Storage, Eviction, and Enforcement.

and generating a handoff-events chain varying from 1 to 10 handoffs in length and measured the time to mediate the input event, the handoff event, and the operation request. We repeated the measurements 100 times. Each set of measurements was preceded by a priming run to remove any first-run effects. We then took an average of the middle 8 out of 10 such runs for each number of handoff events. The results in Figure 7 show that the input mediation requires an overhead of 10 μs, the handoff event mediation requires an additional overhead of 4 μs per event handoff, whereas the operation mediation requires a fixed overhead of 5 μs. The overheads are within our expectations and do not cause noticeable performance degradation.

Delegation Graph Caching – Our second micro-benchmark of EnTrust measures the overhead incurred for caching delegation graphs constructed at runtime. We measured the overhead introduced by EnTrust in the authorization process for both storing a new delegation graph, as well as evicting from cache a stale one. To do this, we simulated the creation and eviction of delegation graphs of different sizes varying from 1 to 16 Kilobytes in 512-byte increments.[10] We repeated the measurement 5 times for each random size and took an average of the middle 3 out of 5 such runs. The results in Figure 7 show that the storing of delegation graphs in the cache required a base overhead of 66 μs with an additional 3 μs per 512-byte increment. The eviction instead required a base overhead of 57 μs with an additional 2.5 μs for each 512-byte increment.

Delegation Graph Enforcement – Our third micro-benchmark was designed to compare the unmodified version of the Android Nougat build for control measurement with a modified build integrating our EnTrust features for the delegation graph enforcement during authorization. To guarantee fairness in the comparison between the two systems, we used the Android UI/Application Exerciser Monkey[11] to generate the same sequence of events for the same set of programs. For both systems, we measured the total time needed to authorize a sensor operation as the time from the input event to the authorization of the resulting operation request, corresponding to the last node of the delegation graph for EnTrust. We repeated the measurement 100 times for each system by varying the number of handoff events from 1 to 10. Each set of measurements was preceded by a priming run to remove any first-run effects. We then took an average of the middle 8 out of 10 such runs for each number of handoff events. Figure 7 shows that the overhead introduced by EnTrust for the delegation graph enforcement is negligible, with the highest overhead observed being below 0.02%. Thus, the slowdown is likely not to be noticeable by users. Indeed, none of our study participants raised any concerns about discernible performance degradation or system slowdown.

Ambiguity Prevention – Our fourth micro-benchmark was designed to measure the performance implications, in terms of delayed events, due to the ambiguity prevention mechanism. For this micro-benchmark, we selected the system UI (User Interface) process, which is one of the processes receiving the highest number of input events, and the media server process that receives the highest number of handoff events and performs sensor operations with higher frequency than any other process. The time window for the construction of each delegation path was set to 150 ms. We generated 15,000 input events with gaps randomly selected in the range [140-1,500][12] ms. The time window and the gaps were selected based on data reported in Appendix B. The generated input events caused 2,037 handoff events and 5,252 operation requests targeting sensors (22,289 total scheduled events). The results indicated a total of 256 delayed events (1.15% of the total events), with a maximum recorded delay of 9 ms. Thus, the performance overhead introduced is negligible.

Memory Requirement – We also recorded the average cache size required by EnTrust to store both event mappings and authorized delegation graphs to be about 5.5 megabytes, for up to 1,000 programs.[13] Therefore, EnTrust required about 5.5 kilobytes of memory per program, which is a small amount of memory when compared to several gigabytes of storage available in modern systems. We ran the measurement 10 times and then took an average of the middle 8 out of 10 of such runs.

8 Discussion of Limitations

Evaluating mechanisms that prevent abuse of sensitive sensors while trading off privacy and usability is chal-

lenging. In this section, we discuss the limitations of our study and provide guidance on future work.

Authorization Comprehension – In designing our authorization messages, we have used the language adopted in current permission systems (e.g., Android OS and Apple iOS) and prior research work [47, 11, 15, 16] as references. However, such language may not be as effective in eliciting access control decisions from users as desired. Further improvements may be possible by studying NLP techniques and how access control questions may be phrased using such techniques. Also, a combination of text, sound, and visuals may be useful in conveying access questions to users. EnTrust is largely orthogonal to any specific way how access control questions are presented, enabling it to be used as a platform for further study.

Decision Revocation – Users may make mistakes when allowing or denying authorizations. EnTrust caches user decisions to reduce users' authorization effort, allowing such mistakes to persist. Mistakes in authorizing access to sensor operations may permit malicious applications to abuse access, albeit limited to that delegation path only. Mistakes in denying access to sensor operations prevents legitimate use of sensor operations silently as a result of caching. One possible solution to these problems is to invalidate the cache periodically to prevent stale authorization decisions. However, frequent authorization prompts negatively affect user experience. Currently, EnTrust enables users to review authorization decisions via an audit mechanism, as suggested elsewhere [53, 15]. However, to improve the effectiveness of such mechanisms, further laboratory studies will be necessary to examine how to present audit results (or other new approaches) to help users to investigate and resolve mistaken authorizations.

Study Scenarios – In this project, we focused on whether users would be able to deny attack scenarios effectively. Another problem is that users may not evaluate non-attack scenarios correctly once they become aware of possible attacks. In our study, we did not observe that users denied any legitimate sensor operations during the lab study, but it would be beneficial to extend the laboratory study to include more subtle non-attack scenarios, where we push the boundaries of what is perceived as benign, to evaluate whether these scenarios may cause false denials due to users being unable to identify that the request was indeed benign. Also, we recognize that all attacks were generated by programs unfamiliar to participants, even though they were given the opportunity to familiarize themselves with such programs during the preliminary phase of our lab study.

Study Size – The number of subjects recruited for this project, 60 for the laboratory study and 9 for the field study, is comparable with the number of subjects in similar studies [33, 14, 15, 11]. Other related work [47] had a higher number of subjects, but subjects were not required to be physically present in the laboratory during the experimental tasks having been recruited via online tools (e.g., Mechanical Turk). However, research has shown, in the context of password study, that a laboratory study may produce more realistic results than an online study [58].

Study Comprehensiveness – Our study does not focus explicitly on long-term habituation, user annoyance, and users' attitudes toward privacy. Researchers have already extensively studied users' general level of privacy concerns [51, 52, 53, 15]. Other researchers have studied users' habituation for first-use authorization systems extensively [33, 45, 50]. Our field study (Section 6.3) shows that our approach is comparable to first-use in terms of the number of times users are prompted, and the number of explicit authorizations from users is far below the 8 additional explicit authorizations used in prior work, which are considered likely not to introduce significant risk of habituation or annoyance [33].

9 Related Work

Researchers have extensively demonstrated that IPC mechanisms allow dangerous interactions between programs, such as unauthorized use of intents, where adversaries can hijack activities and services by stealing intents [18, 21, 22, 26]. Prior work has also shown that such interactions can be exploited by adversaries to cause permission re-delegations [7] in the attempt to leverage capabilities available to trusted programs (e.g., system services). Also, related work has demonstrated how trusted programs inadvertently or purposely (for functionality reasons) expose their interfaces to other programs [8], thus exposing attack vectors to adversaries. In this paper, we have demonstrated that dangerous interactions among programs can lead to critical attack vectors related to input event delegations.

Researchers have tried to regulate such interactions with automated tools for IPC-related vulnerability analysis. For instance, *ComDroid* is a tool that parses the disassembled applications' code to analyze intent creation and transition for the identification of unauthorized intent reception and intent spoofing [26]. *Efficient and Precise ICC discovery* (EPICC) is a more comprehensive static analysis technique for Inter-Component Communication (ICC)[14] calls [19]. It can identify ICC vulnerabilities due to intents that may be intercepted by malicious programs, or scenarios where programs expose components that can be launched via malicious intents. *Secure Application INTeraction* (*Saint*) [54] extends the existing Android security architecture with policies that would allow programs to have more control to whom permissions for accessing their interfaces are granted and used at runtime. *Quire* provides context in the form of provenance to programs communicating via Inter-Procedure Calls (IPC) [55]. It annotates IPCs occurring within a system, so that the recipient of an

IPC request can observe the full call sequence associated with it, before committing to any security-relevant decision. Although effort has been made to analyze and prevent IPC-related vulnerabilities, none of the proposed approaches above tackled the problem from our perspective, i.e., instead of giving control to application developers, we must give control to users who are the real target for privacy violations by malicious programs.

In line with our perspective of giving control to users, *User-Driven Access Control* [9, 10] proposes the use of access control gadgets, predefined by the operating systems and embedded into applications' code, to limit what operation can be associated with a specific input event. Also, *AWare* [11] proposes to bind each operation request targeting sensitive sensors to an input event and to obtain explicit authorization from the user for each event-operation combination. Similarly, *ContexIoT* [16] is a context-based permission system for IoT platforms which leverages runtime prompts with rich context information including the program execution flow that allows users to identify how a sensitive operation is triggered. Unfortunately, all of these mechanisms only control how the input event is consumed by the program receiving the input event. The proposed mechanisms to enable mediation do not mediate inter-process communication and the operations resulting from such communication (e.g., *event delegations between programs*), which is necessary to prevent the attack vectors discussed in this paper. Also, differently from prior work on permission re-delegation [7], we do not rely on an over-restrictive defense mechanism that totally forbids programs from using their additional privileges. Such an over-restrictive defense would block necessary interactions between programs even when the interactions are benign and expected by users.

Prior work has also investigated the use of machine learning classifiers to analyze the contextuality behind user decisions to automatically grant access to sensors [14, 15]. Unfortunately, such classifiers only model the context relative to the single program that the user is currently interacting with, and the API calls that are made by such a program during the interaction. However, the context modeled by these classifiers does not account for inter-process communications, which allow programs to enlist other programs to perform sensor operations via input event delegation. Furthermore, the effectiveness of the learning depends on the accuracy of the user decisions used in training the learner. In other words, if the user's decisions suffer from inadequate information during the training phase, the learner will as well. Therefore, we firmly believe that an additional effort is necessary to support user decision making before the user decisions can be used to train a classifier.

Lastly, mechanisms based on taint analysis [29, 30, 31] or Decentralized Information Flow Control (DIFC) [13, 20] have been proposed by researchers to, respectively, track and control how sensitive data is used by or shared between programs. However, such mechanisms solve the orthogonal problem of controlling sensitive data leakage or accidental disclosure, rather than enabling users to control *how*, *when*, and *which* programs can access sensors for the collection of sensitive data.

10 Conclusion

While a collaborative model allows the creation of useful, rich, and creative applications, it also introduces new attack vectors that can be exploited by adversaries. We have shown that three well-studied attack vectors become critical, in operating systems supporting a cooperating program abstraction, and proposed the EnTrust authorization system to help mitigate them. EnTrust demonstrates that it is possible to prevent programs from abusing the collaborative model – in the attempt to perform delegated confused deputy, delegated Trojan horse, or delegated man-in-the-middle attacks – by binding together, input event, handoff events, and sensor operation requests made by programs, and by requiring an explicit user authorization for the constructed delegation path. Our results show that existing systems have room for improvement and permission-based systems, as well as machine learning classifiers, may significantly benefit from applying our methodology.

Acknowledgements

Thanks to our shepherd, Sascha Fahl, and the anonymous reviewers. The effort described in this article was partially sponsored by the U.S. Army Research Laboratory Cyber Security Collaborative Research Alliance under Contract Number W911NF-13-2-0045. The views and conclusions contained in this document are those of the authors, and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation hereon. The research work of Jens Grossklags was supported by the German Institute for Trust and Safety on the Internet (DIVSI).

References

[1] Conger, K. Researchers: Uber's iOS app had secret permissions that allowed it to copy your phone screen. *Gizmodo*, (2017).

[2] Lieberman, E. Hackers are gunning for your personal data by tracking you. *The Daily Caller*, (2016).

[3] Sulleyman, A. Android apps secretly steal users' data by colluding with each other, finds research. *Independent*, (2017).

[4] Revel, T. Android apps share data between them without your permission. *NewScientist*, (2017).

[5] Norm, H., The Confused Deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, (1988).

[6] Petracca, G., Sun, Y., Jaeger, T., and Atamli, A. AuDroid: Preventing attacks on audio channels in mobile devices. In *ACSAC*, (2015), ACM.

[7] Felt, A. P., Wang, H., Moshchuk, A., Hanna, S., and Chin, E. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, (2011).

[8] Aafer, Y., Zhang, N., Zhang, Z., Zhang, X., Chen, K., Wang, X., Zhou, X., Du, W., and Grace, M. Hare hunting in the wild Android: A study on the threat of hanging attribute references. In *CCS*, (2015), ACM.

[9] Roesner, F., Kohno, T., Moshchuk, A., Parno, B., Wang, H., and Cowan, C. User-driven access control: Rethinking permission granting in modern operating systems. In *S&P*, (2012), IEEE.

[10] Ringer, T., Grossman, D., and Roesner, F. Audacious: User-driven access control with unmodified operating systems. In *CCS* (2016), ACM.

[11] Petracca, G., Reineh, A.-A., Sun, Y., Grossklags, J., and Jaeger, T. AWare: Preventing abuse of privacy-sensitive sensors via operation bindings. In *USENIX Security Symposium*, (2017).

[12] Onarlioglu, K., Robertson, W., and Kirda, E. Overhaul: Input-driven access control for better privacy on traditional operating systems. In *DSN*, (2016), IEEE/IFIP.

[13] Nadkarni, A., and Enck, W. Preventing accidental data disclosure in modern operating systems. In *CCS*, (2013), ACM.

[14] Wijesekera, P., Baokar, A., Tsai, L., Reardon, J., Egelman, S., Wagner, D., and Beznosov, K. The feasibility of dynamically granted permissions: Aligning mobile privacy with user preferences. In *S&P* (2017), IEEE.

[15] Olejnik, K., Dacosta, I., Machado, J.S., Huguenin, K., Khan, M.E., and Hubaux, J.P. Smarper: Context-aware and automatic runtime-permissions for mobile devices. In *S&P*, (2017), IEEE.

[16] Jia, Y. J., Chen, Q. A., Wang, S., Rahmati, A., Fernandes, E., Mao, Z. M., and Prakash, A. ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms. In *NDSS*, (2017).

[17] Acar, Y., Backes, M., Bugiel, S., Fahl, S., McDaniel, P. and Smith, M., Sok: Lessons learned from android security research for appified software platforms. In *S&P*, (2017), IEEE.

[18] Li, L., Bartel, A., Bissyandé, T. F., Klein, J., Le Traon, Y., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., and McDaniel, P. Iccta: Detecting inter-component privacy leaks in Android apps. In *ICSE*, (2015), IEEE.

[19] Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J., and Le Traon, Y. Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis. In *USENIX Security Symposium*, (2013).

[20] Nadkarni, A., Andow, B., Enck, W., and Jha, S. Practical DIFC enforcement on Android. In *USENIX Security Symposium*, (2016).

[21] Octeau, D., Luchaup, D., Dering, M., Jha, S., and McDaniel, P. Composite constant propagation: Application to Android inter-component communication analysis. In *ICSE*, (2015), IEEE.

[22] Octeau, D., Jha, S., Dering, M., McDaniel, P., Bartel, A., Li, L., Klein, J., and Le Traon, Y. Combining static analysis with probabilistic models to enable market-scale Android inter-component analysis. In *ACM SIGPLAN Notices*, (2016).

[23] Krohn, M.N., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M.F., Kohler, E., and Morris R. Information flow control for standard OS abstractions. In *SOSP*, (2007).

[24] Zeldovich, N., Boyd-Wickizer, S., Kohler, E., and Mazières, D. Making information flow explicit in HiStar. In *OSDI*, (2006).

[25] Chatterjee, R., Doerfler, P., Orgad, H., Havron, S., Palmer, J., Freed, D., Levy, K., Dell, N., McCoy, D., and Ristenpart, T. The Spyware Used in Intimate Partner Violence. In *S&P*, (2018), IEEE.

[26] Chin, E., Felt, A. P., Greenwood, K., and Wagner, D. Analyzing inter-application communication in Android. In *MobiSys* (2011), ACM.

[27] Huang, L.-S., Moshchuk, A., Wang, H. J., Schecter, S., and Jackson, C. Clickjacking: Attacks and defenses. In *USENIX Security Symposium*, (2012).

[28] Luo, T., Jin, X., Ananthanarayanan, A., and Du, W. Touchjacking attacks on web in Android, iOS, and Windows phone. In *FPS* (2012).

[29] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *USENIX OSDI* (2010).

[30] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *ACM Sigplan Notices*, (2014), pp. 259–269.

[31] Tang, Y., Ames, P., Bhamidipati, S., Bijlani, A., Geambasu, R., and Sarda, N. CleanOS: Limiting mobile data exposure with idle eviction. In *USENIX OSDI* (2012).

[32] Sun, Y., Petracca, G., Ge, X., and Jaeger, T. Pileus: Protecting user resources from vulnerable cloud services. In *ACSAC*, (2016), ACM.

[33] Wijesekera, P., Baokar, A., Hosseini, A., Egelman, S., Wagner, D., and Beznosov, K. Android permissions remystified: A field study on contextual integrity. *USENIX Security Symposium*, (2015).

[34] Levy, H. M. *Capability-Based Computer Systems*. Digital Press. Available at http://www.cs.washington.edu/homes/levy/capabook/, (1984).

[35] Prevelakis, V., and Spinellis, D. Sandboxing applications. In *USENIX Annual Technical Conference, FREENIX Track*, (2001).

[36] Chang, F., Itzkovitz, A., and Karamcheti, V. User-level resource-constrained sandboxing. In *USENIX Windows Systems Symposium*, (2000).

[37] Smalley, S., Vance, C., and Salamon, W. Implementing SELinux as a Linux security module. *NAI Labs Report #01-043*, (2001).

[38] Smalley, S., and Craig, R. Security Enhanced (SE) Android: Bringing flexible MAC to Android. In *NDSS*, (2013).

[39] Ye, Z., Smith, S., and Anthony, D. Trusted paths for browsers. *ACM Transactions on Information and System Security*, (2005).

[40] Zhou, Z., Gligor, V., Newsome, J., and McCune, J. Building verifiable trusted path on commodity x86 computers. In *S&P*, (2012), IEEE.

[41] Shapiro, J., Vanderburgh, J., Northup, E., and Chizmadia, D. Design of the EROS trusted window system. In *USENIX Security Symposium*, (2004).

[42] Li, W., Ma, M., Han, J., Xia, Y., Zang, B., Chu, C.-K., and Li, T. Building trusted path on untrusted device drivers for mobile devices. In *Asia-Pacific Workshop on Systems*, (2014), ACM.

[43] Eugster, P., Felber, P., Guerraoui, R., and Kermarrec, A.-M. The many faces of publish/subscribe. *ACM Computing Surveys*, (2003).

[44] Mellor-Crummey, J. M., and Scott, M. L. Scalable reader-writer synchronization for shared-memory multiprocessors. *ACM SIGPLAN Notices*, (1991).

[45] Felt, A. P., Ha, E., Egelman, S., Haney, A., Chin, E., and Wagner, D. Android permissions: User attention, comprehension, and behavior. In *SOUPS*, (2012), ACM.

[46] Rivest, R., Shamir, A., and Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, (1978).

[47] Bianchi, A., Corbetta, J., Invernizzi, L., Fratantonio, Y., Kruegel, C., and Vigna, G. What the App is that? Deception and countermeasures in the Android user interface. *S&P*, (2015), IEEE.

[48] Cummings, P., Fullan, D.A., Goldstien, M.J., Gosse, M.J., Picciotto, J., Woodward, J.P., and Wynn, J. Compartmented Model Workstation: Results through prototyping. *S&P*, (1987), IEEE.

[49] Schechter, S. Common pitfalls in writing about security and privacy human subjects experiments, and how to avoid them. Microsoft Tech. Rep. (2013).

[50] Felt, A. P., Egelman, S., Finifter, M., Akhawe, D., and Wagner, D. How to ask for permission. In *USENIX Workshop on Hot Topics in Security* (2012).

[51] Sheehan, K.B. Toward a typology of Internet users and online privacy concerns. *The Information Society*, (2012).

[52] Debatin, B., Lovejoy, J.P., Horn, A.K., and Hughes, B.N. Facebook and online privacy: Attitudes, behaviors, and unintended consequences. *Journal of Computer-Mediated Communication*, (2009).

[53] Petracca, G., Atamli-Reineh, A., Sun, Y., Grossklags, J., and Jaeger, T. Aware: Controlling app access to I/O devices on mobile platforms. *CoRR abs/1604.02171*, (2016).

[54] Ongtang, M., McLaughlin, S., Enck, W., and McDaniel, P. Semantically rich application-centric security in Android. *Security and Communication Networks*, (2012).

[55] Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., and Wallach, D. Quire: Lightweight provenance for smart phone operating systems. In *USENIX Security Symposium* (2011).

[56] Sasse, M. A., Brostoff, S., and Weirich, D. Transforming the 'Weakest Link' — a Human/Computer Interaction Approach to Usable and Effective Security *BT Technology Journal*, (2001).

[57] Arce, I. The weakest link revisited [information security]. In *IEEE Security & Privacy*, (2003).

[58] Fahl, S., Harbach, M., Acar, Y., and Smith, M. On the Ecological Validity of a Password Study In *Ninth Symposium on Usable Privacy and Security*, (2013).

Appendices

Appendix A - Study Demographics: In total, from the 69 recruited subjects that completed our study, 34 (49%) were female; 36 (52%) were in the 18-25 years old range, 27 (39%) in the 26-50 range, and 6 (9%) were in above the 51 range; 33 (48%) were students from our Institution, 9 of them (13%) were undergraduate and 24 (35%) were graduate students, 2 (3%) were Computer Science Majors; 11 (16%) worked in Public Administration, 9 (13%) worked in Hospitality, 6 (9%) in Human Services, 6 (9%) in Manufacturing, and 4 (6%) worked in Science or Engineering. All participants reported being active smartphone users (1-5 hours/day). Also, 42 (61%) of the subjects were long-term Android users (3-5 years), others were long-term iOS users. For our laboratory and field studies, we redistributed the available participants as evenly as possible. Each lab group had 9 long-term Android users, the remaining 6 long-term Android users participated in our field study.

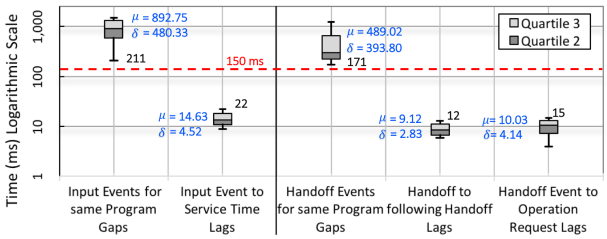


Figure 8: Time analysis used to study the possibility of ambiguous events delegation paths, as discussed in Section 4.2.

Appendix B - Time Constraints Analysis: We leveraged data collected via the field study to perform an analysis of time constraints for input events and action/operation requests to calibrate the time window

for the event ambiguity prevention mechanism (Section 4.2). Figure 8 reports the measurements of the gaps[15] between consecutive input events and consecutive handoff events, as well as the lags between each event and the corresponding response from the serving program. From the measurements, we observed: (1) the minimum gap between subsequent input events targeting the same program (211 ms) is an order of magnitude larger than the maximum lag required by the program to serve each incoming event (22 ms); and (2) the minimum gap (171 ms) between subsequent handoff events targeting the same program is an order of magnitude larger than the maximum lag required by the program to serve incoming requests (15 ms). Hence, to avoid ambiguity, we may set the time window to 150 ms to guarantee that the entire delegation path can be identified before the next event for the same program arrives. Lastly, we observed that 87% of the delegation paths had a total length of three edges (one input event, one handoff event, and one sensor operation request). The remaining 13% of the delegation paths had a maximum length of four edges (one additional handoff event), which further supports our claim that we can hold events without penalizing concurrency of such events.

Appendix C - Program Identification: To prove the programs' identity to users, EnTrust specifies both the programs' name and visual identity mark (e.g., icon) in every *delegation request* as shown in Figure 6. EnTrust retrieves programs' identity by accessing the `AndroidManifest.xml`, which must contain a unique name and a unique identity mark (e.g., icon) for the program package. EnTrust verifies programs' identity via the crypto-checksum[16] of the program's binary signed with the developer's private key and verifiable with the developer's public key [46], similarly to what proposed in prior work [47, 11].

Appendix D - Input Event Authentication: EnTrust leverages SEAndroid [38] to ensure that programs cannot inject input events by directly writing into input device files (i.e., `/dev/input/*`) corresponding to hardware and software input interfaces attached to the mobile platform. Hence, only device drivers can write into input device files and only the Android Input Manager, a trusted system service, can read such device files and dispatch input events to programs. Also, EnTrust leverages the Android screen overlay mechanism to block overlay of graphical user interface components and prevent hijacking of input events. Lastly, EnTrust accepts only voice commands that are processed by the Android System Voice Actions module.[17] EnTrust authenticates input events by leveraging sixteen mediation hooks placed inside the stock Android Input Manager and six mediation hooks placed inside the System Voice Actions module.

Appendix E - Handoff Event Mediation: Programs communicate with each other via Inter-Component Communication (ICC) that, in Android, is implemented as part of the Binder IPC mechanisms. The ICC includes both *intent* and *broadcast* messages that can be exchanged among programs. The Binder and the Activity Manager regulate messages exchanged among programs via the intent API.[18] Programs can also send intents to other programs or services by using the broadcast mechanism that allows sending intents as arguments in broadcast messages. The Activity Manager routes intents to broadcast receivers based on the information contained in the intents and the broadcast receivers that have registered their interest in the first place. To mediate intents and broadcast messages exchanged between programs completely, EnTrust leverages mediation hooks placed inside the Activity Manager and the Binder.

Notice that, other operating systems support mechanisms similar to Android's Intents. For instance, MacOS and iOS adopt the Segue mechanism, while Chrome OS supports Web Intents, thus EnTrust can be also implemented for other modern systems supporting the cooperating program abstraction.

Appendix F - Sensor Operation Mediation: Android uses the Hardware Abstraction Layer (HAL) interface to allow only system services and privileged processes to access system sensors indirectly via a well-defined API exposed by the kernel. Moreover, SEAndroid [38] is used to ensure that only system services can communicate with the HAL at runtime. Any other programs (e.g., apps) must interact with such system services to request execution of operations targeting sensors. EnTrust leverages such a mediation layer to identify operation requests generated by programs, by placing 12 hooks inside the stock Android Audio System, Media Server, Location Services, and Media Projection.

Notes

[1] In this paper, we use the term "delegate" to refer to the use of IPCs to request help in task processing, not the granting of permissions to other processes.

[2] One of the surveillance mobile apps available online (e.g., `flexispy`).

[3] Several banks are now offering these services to their clients.

[4] `https://source.android.com`

[5] The runtime permission mechanism enabled users to revoke permissions at any time.

[6] Source: `https://fortune.com`

[7] `https://dialogflow.com`

[8] `https://source.android.com/compatibility/cts/`

[9] `Android Open Source Project - https://source.android.com`

[10] This range was selected based on the size of the delegation graphs created during our experiments, which should be representative of real scenarios.

[11] `https://developer.android.com/studio/test/monkey.html`

[12] To stress test our system, we selected a lower bound that is considerably lower than the maximum speed at which a user can possibly keep tapping on the screen (~210 ms).

[13] Chosen among the most-downloaded Android apps from the Google Play Store and including all apps and system services shipped with the stock Android OS.

[14] Equivalent of IPCs for Android OS.

[15] Gaps higher than 1,500 ms were excluded because not relevant to the analysis.

[16] Android requires all apps and services to be signed by their developers.

[17] `https://developers.google.com/voice-actions/`

[18] `https://developer.android.com`

PolicyLint: Investigating Internal Privacy Policy Contradictions on Google Play

Benjamin Andow*, Samin Yaseer Mahmud*, Wenyu Wang†, Justin Whitaker*
William Enck*, Bradley Reaves*, Kapil Singh‡, Tao Xie†
*North Carolina State University
†University of Illinois at Urbana-Champaign
‡IBM T.J. Watson Research Center

Abstract

Privacy policies are the primary mechanism by which companies inform users about data collection and sharing practices. To help users better understand these long and complex legal documents, recent research has proposed tools that summarize collection and sharing. However, these tools have a significant oversight: they do not account for contradictions that may occur within an individual policy. In this paper, we present PolicyLint, a privacy policy analysis tool that identifies such contradictions by simultaneously considering negation and varying semantic levels of data objects and entities. To do so, PolicyLint automatically generates ontologies from a large corpus of privacy policies and uses sentence-level natural language processing to capture both positive and negative statements of data collection and sharing. We use PolicyLint to analyze the policies of 11,430 apps and find that 14.2% of these policies contain contradictions that may be indicative of misleading statements. We manually verify 510 contradictions, identifying concerning trends that include the use of misleading presentation, attempted redefinition of common understandings of terms, conflicts in regulatory definitions (e.g., US and EU), and "laundering" of tracking information facilitated by sharing or collecting data that can be used to derive sensitive information. In doing so, PolicyLint significantly advances automated analysis of privacy policies.

1 Introduction

Mobile apps collect, manage, and transmit some of the most sensitive information that exists about users—including private communications, fine-grained location, and even health measurements. These apps regularly transmit this information to first or third parties [1, 7, 14]. Such data collection/sharing by an app is often considered (legally) acceptable if it is described in the privacy policy for the app. Privacy policies are sophisticated legal documents that are typically long, vague, and difficult for novices, experts, and algorithms to interpret. Accordingly, it is difficult to determine whether app developers adhere to privacy policies, which can help app markets and other analysts identify privacy violations, or help end users choose more-privacy-friendly apps.

Recent work has begun studying whether or not mobile app behavior matches statements in privacy policies [26,28,29,32]. However, the prior work fails to account for contradictions within privacy policies; these contradictions may lead to incorrect interpretation of sharing and collection practices. Identifying contradictions requires overcoming two main challenges. First, privacy policies refer to information at different semantic granularities. For example, a policy may discuss its practices using broad terms (e.g., "personal information") in one place in the policy, but later discuss its practices using more specific terms (e.g., "email address"). Prior approaches have tackled this issue by crowdsourcing data object ontologies [26, 28],[1] but such crowdsourced information is not complete, accurate, or easily collected. Second, prior approaches have struggled to accurately detect negative statements, relying on bi-grams (e.g., "not share") [32] or detecting only verb modifiers [29] while neglecting the more-complicated statements (e.g., "will share X except Y") that are common in privacy policies. Modeling negative statements is required to determine the correct meaning of a policy statement (i.e., "not sharing" versus "sharing" information). Fully characterizing contradictions requires addressing both preceding challenges.

In this paper, we present PolicyLint for automatically identifying potential contradictions of sharing and collection practices in software privacy policies. Contradictions make policies unclear, confusing both humans and any automated system that rely on interpreting the policies. Considering these uses cases, PolicyLint defines two contradiction groupings. *Logical contradictions* are contradictory policy statements that are more likely to cause harm if users and analysts are not aware of the contradictory statements. One example is a policy that initially claims not to collect personal information, but later in fine print discloses collecting a user's name and email address for advertisers. *Narrowing definitions* may cause automated techniques that reason over policy statements

[1] Ontologies are graph data structures that capture relationships among entities. For example, "personal information" subsumes "your email address."

to make incorrect or inconsistent decisions and may result in vague policies. PolicyLint is the first tool to have the sophistication necessary to reason about both negative sentiments and statements covering varying levels of specificity, being necessary for uncovering contradictions.

PolicyLint is inspired by other security `lint` tools [6, 8–11, 19], which analyze code for indicators of potential bugs. Like any static approach, not every `lint` finding is necessarily a real bug. For example, potential bug conditions could be mitigated by an external control or other context that the tool cannot verify. In many cases, only a human can verify the outputs of a `lint` finding. In the case of PolicyLint, we note that privacy policies are complex legal documents that may be intentionally vague, ambiguous, or misleading even for human interpretation. Despite these challenges, PolicyLint condenses long, complicated policies into a small set of candidate issues of interest to human or algorithmic analysis.

This paper makes the following main contributions:

- **Automated generation of ontologies from privacy policies.** PolicyLint uses an expanded set of Hearst patterns [16] to extract ontologies for both data objects and entities from a large corpus of privacy policies (e.g., "W such as X, Y, and Z"). PolicyLint is more comprehensive and scalable than crowdsourced efforts [26, 28].

- **Automated sentence-level extraction of privacy practices.** PolicyLint uses sentence-level NLP and leverages parts-of-speech and type-dependency information to capture data collection and sharing as a four-tuple: (actor, action, data object, entity). For example, "We [actor] share [action] personal information [data object] with advertisers [entity]." Sentence-level NLP is critically important for the correct identification of negative statements. We also show that prior attempts at analyzing negation would fail on 28.2% of policies.

- **Automated analysis of contradictions in privacy policies.** We formally model nine types of contradictions that result from semantic relationships between terms, providing an algorithmic technique to detect contradictory policy statements. Our groupings of *narrowing definitions* and *logical contradictions* lay the foundation for ensuring the soundness of automated policy tools and identifying potentially misleading policy statements. In a study of 11,430 privacy policies from mobile apps, we are the first to find that logical contradictions and narrowing definitions are *rampant*, affecting 17.7% of policies, with logical contradictions affecting 14.2%.

- **Manual analysis of contradictions to identify trends.** The high ratio of policy contradictions is surprising. We manually review 510 contradictions across 260 policies, finding that many contradictions are indeed indicators of misleading or problematic policies. These contradictions include making broad claims to protect personal information early in a policy, yet later carving out exceptions for data that the authors attempt to redefine as not personal, that could be used to derive sensitive information (e.g., IP addresses and location), or that are considered sensitive by some regulators but not others.

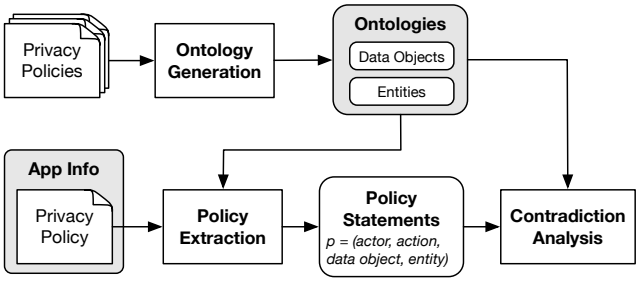


Figure 1: Overview of PolicyLint

PolicyLint has four main potential use cases. First, policy writers can leverage PolicyLint to reduce the risk of releasing misleading policies. In fact, when we contacted parties responsible for the contradictory policies, several fixed their policies (Section 3.4). Second, regulators can use PolicyLint's definition of logical contradictions to identify deceptive policies. While the FTC has identified contradictory statements as problem areas within privacy policies [3], to our knowledge, there is no legal precedent regarding whether regulatory agencies would take action as a result of contradictory policies. However, we believe that some of our findings in Section 3 potentially fall under the FTC's definition of deceptive practices [13]. We envision that regulators could deploy PolicyLint to audit companies' privacy policies for misleading statements at large scale. Third, app markets, such as Google Play, can deploy PolicyLint similarly to ensure that apps posted in the store do not have misleading statements in their privacy policies. Furthermore, they can also use PolicyLint's extraction of policy statements to automatically generate privacy labels to display on the markets to nudge users to less-privacy-invasive apps. Finally, automated techniques for analyzing privacy policies can use PolicyLint's fine-grained extraction of policy statements and formalization of logical contradictions and narrowing definitions to help ensure tool soundness.

The rest of this paper is organized as follows. Section 2 describes PolicyLint's design. Section 3 reports on our study using PolicyLint. Section 4 discusses limitations and future work. Section 5 describes related work. Section 6 concludes.

2 PolicyLint

PolicyLint seeks to identify contradictions within individual privacy policies for software. It provides privacy *warnings* based on contradictory sharing and collection statements within policies, but similar to lint tools for software, these warnings require manual verification. PolicyLint identifies

"candidate contradictions" within policies. A candidate contradiction is a pair of contradictory policy statements when considered in the most conservative interpretation (i.e., context-insensitive). Candidate contradictions that are validated by analysts are termed as "validated contradictions." Manual verification is required due to the fundamental problems of ambiguity when interpreting the meaning of natural language sentences (i.e., multiple interpretations of the same sentence).

For example, consider the privacy policy for a popular recipe app (com.omniluxtrade.allrecipes). One part of the policy states "We do not collect personally identifiable information from our users." It is clear from this sentence that the app does not collect any personal information. However, later the policy states, "We may collect your email address in order to send information, respond to inquiries, and other requests or questions." Such sentence is a clear contradiction to the earlier sentence, as email address is considered personal information. As discussed in detail in Section 3, the cause for this underlying contradiction is that the developer does not consider email address as personal information.

To our knowledge, we are the first to characterize and automatically analyze contradictions within privacy policies. While PolicyLint is not the first NLP tool to analyze privacy policies, identifying contradictions requires addressing two broad challenges.

- *References to information are expressed at different semantic levels.* Prior work [26, 28] uses ontologies to capture subsumptive (i.e., "is-a") relationships between terms; however, such ontologies are crowdsourced and subsumptive relationships are manually defined by the authors, leaving concerns of comprehensiveness and scalability. For example, prior work [26, 28] builds their ontology using only 50 and 30 policies, respectively. While crowdsourced ontologies could be comprehensive given unlimited time and manpower, crowdsourcing at large scale is infeasible due to limited resources. Furthermore, existing general-purpose ontologies do not capture all of the specific relationships required to reason over data types and entities mentioned within privacy policies.

- *Privacy policies include negative sharing and collection statements.* Most prior work [26, 28] operates at paragraph level and cannot capture negative sharing statements. Prior work [29, 32] that does capture negative statements misses complex statements (e.g., "will share personal information except your email address"). Such prior work extracts coarse-grained summaries of policy statements (paragraph-level [26, 28], document-level [32]) and can never precisely model negative statements or the entities involved. Their imprecision may result in incorrectly reasoning about 28.2% of policies due to their negation modeling (Finding 1 in Section 3).

We tackle these challenges using two key insights.

Sentence structure informs semantics: Sharing and collection statements generally follow a learnable set of templates. PolicyLint uses these templates to extract a four tuple from such statements: (actor, action, data object, entity). For example, "We [actor] share [action] personal information [data object] with advertisers [entity]." The sentence structure also provides greater insight into more complex negative sharing. For example, "We share personal information except your email address with advertisers." PolicyLint extracts such semantics from policy statements by building on top of existing parts-of-speech and dependency parsers.

Privacy policies encode ontologies: Due to the legal nature of privacy policies, general terms are often defined in terms of examples or their constituent parts. While each policy might not define semantic relationships for all of the terms used in the policy, those relationships should exist in some other policies in our dataset. By processing a large number of privacy policies, PolicyLint automatically generates an ontology specific to policies (one for data objects and one for entities). PolicyLint extracts term definitions using Hearst patterns [16], which we have extended for our domain.

Figure 1 depicts the data flow within PolicyLint. There are three main components of PolicyLint: ontology generation, policy extraction, and contradiction analysis. The following sections describe these components. Readers interested in policy-preprocessing considerations can refer to Appendix A.

2.1 Ontology Generation

The goal of ontology generation is to define subsumptive ("is-a") relationships between terms in privacy policies to allow reasoning over different granularities of language. PolicyLint operates on the intuition that subsumptive relationships are often embedded within the text of a privacy policy, e.g., an example of the types of data considered to be a specific class of information. The following example identifies that demographic information subsumes age and gender.

Example 1. *We may share demographic information, such as your age and gender, with advertisers.*

PolicyLint uses such sentences to automatically discover subsumptive relationships across a large set of privacy policies. It focuses on data objects and the entities receiving data.

PolicyLint uses a semi-automated and data-driven technique for ontology generation. It breaks ontology generation into three main parts. First, PolicyLint performs domain adaptation of an existing model of statistical-based named entity recognition (NER). NER is used to label data objects and entities within sentences, capturing not only terms, but also surrounding context in the sentence. Second, PolicyLint learns subsumptive relationships for labeled data objects and entities by using a set of 11 lexicosyntactic patterns with enforced named-entity label constraints. Third, PolicyLint takes a set

Table 1: NER Performance: Comparison of spaCy's stock en_core_web_lg model versus our adapted domain model

	Overall		Data Objects		Entities	
	Default	Adapted	Default	Adapted	Default	Adapted
Precision	43.48%	84.12%	-	82.20%	61.22%	86.75%
Recall	8.33%	81.67%	-	79.84%	17.75%	85.21%
F1-Score	13.99%	82.88%	-	81.00%	27.52%	85.97%

of seed words as input and generates data-object/entity ontologies using the subsumptive relationships discovered in the prior step. It iteratively adds relationships to the ontology until a fixed point is reached. We next describe this process.

2.1.1 NER Domain Adaptation

To identify subsumptive relationships for data objects and entities, PolicyLint must identify which sentence tokens represent a data object or entity. For Example 1, we seek to identify "demographic information," "age," and "gender" as data objects, and "we" and "advertisers" as entities. PolicyLint uses a statistical-based technique of named-entity recognition (NER) to label data objects and entities within sentences. Prior research [26, 28, 29, 32] proposed keyphrase-based techniques for identifying data objects. However, keyphrase-based techniques are less versatile in practice: they cannot handle term ambiguity and variability, and they can identify only terms in their pre-defined list. For example, "internet service provider" can be both a data object and entity, which keyphrase-based techniques cannot differentiate. In contrast, statistical-based NER both resolves ambiguity and discovers "unseen" terms.

Unfortunately, existing NER models are not trained for our problem domain (data objects and collective terms describing entities, e.g., "advertisers"). Training an NER model from scratch is time-consuming due to the large amount of training data required to achieve reasonable performance. Therefore, PolicyLint takes an existing NER model and updates it using annotated training data from our problem domain. Specifically, PolicyLint adopts spaCy's NER engine [17], which uses deep convolutional neural networks. We adapt the *en_core_web_lg* model to the privacy policy domain.

To perform domain adaptation, we gather 500 sentences as training data. Our training data is selected as follows. First, we randomly select 50 unique sentences from our policy dataset. Second, for each of the 9 lexicosyntactic patterns described in Section 2.1.2, we randomly select 50 sentences that contain the pattern (450 in total). We run the existing NER model on the training sentences to prevent the model from "forgetting" old annotations. We then manually annotate the sentences with data objects and entities.

When updating the existing NER model, we perform multiple passes over the annotated training data, shuffling at each epoch, and using minibatch training with a batch size of 4. To perform the domain adaptation, the current model predicts the NER labels for each word in the sentence and adjusts the synaptic weights in the neural network accordingly if the prediction does not match the annotation. We stop making passes over the training data when the loss rate begins to converge. We annotate an additional 100 randomly selected sentences as holdout data for testing the model. Table 1 shows the NER model performance before and after domain adaptation for our holdout dataset. PolicyLint achieves 82.2% and 86.8% precision for identifying data objects and entities, respectively.

Table 2: Lexicosyntatic patterns for subsumptive relationships

#	Pattern
H1	X, *such as* $Y_1, Y_2, \ldots, Y_n$
H2	*such* X *as* $Y_1, Y_2, \ldots Y_n$
H3	X $[or\|and]$ *other* $Y_1, Y_2, \ldots Y_n$
H4	X, *including* $Y_1, Y_2, \ldots Y_n$
H5	X, *especially* $Y_1, Y_2, \ldots Y_n$
H′1	X, $[e.g.\|i.e.]$, $Y_1, Y_2, \ldots Y_n$
H′2	X ($[e.g.\|i.e.]$, $Y_1, Y_2, \ldots Y_n$)
H′3	X, *for example,* $Y_1, Y_2, \ldots Y_n$
H′4	X, *which may include* $Y_1, Y_2, \ldots Y_n$

* H* = Hearst Pattern; H′* = Custom Pattern

2.1.2 Subsumptive Relationship Extraction

PolicyLint uses a set of 9 lexicosyntactic patterns to discover subsumptive relationships within sentences, as shown in Table 2. The first 5 are Hearst Patterns [16], and the last 4 are custom deviations based on observations of text in privacy policies. For each pattern, PolicyLint ensures that named-entity labels are consistent across the pattern (i.e., PolicyLint uses Hearst patterns enforcing constraints on named-entity labels). For example, Example 1 is recognized by the pattern "X, such as $Y_1, Y_2, \cdots, Y_n$" where X is a noun, $Y_1, Y_2, \cdots, Y_n$ are all nouns, and the NER labels for X and each Y_i are all data objects. Note that PolicyLint merges noun phrases before applying the lexicosyntactic patterns to ease extraction.

Given the set of extracted relationships, PolicyLint normalizes the relationships by lemmatizing the text and substituting terms with their synonym. For example, consider that "blood sugar levels" is a synonym for "blood glucose level." Lemmatization turns "blood sugar levels" into "blood sugar level," and synonym substitution turns it into "blood glucose level." To identify synonyms, we output non-terminal (i.e., X value of the Hearst patterns) data objects and entities in the subsumptive relationships. We manually scan through the list and mark synonyms. We repeat the process with the terminal nodes that are included after constructing the ontology. We then output the data objects and entities labeled from all policies and sort the terms by frequency. We mark synonyms for the most frequent terms by keyword searching for related terms based on sub-strings and domain knowledge. For example, if "location" appears as a frequent term, we output all data objects that contain the word "location," read through the list, and mark synonyms (e.g., "geographic location"). Next, we use domain knowledge to identify that "latitude and longitude" is a synonym of "location," output the terms that contain those words, and manually identify synonyms.

Table 3: Seed terms used for ontology construction

Ontology	Seeds
Data Ontology	information, personal information, non-personal information, information about you, biometric information, financial information, device sensor information, government-issue identification information, vehicle usage information
Entity Ontology	third party

2.1.3 Ontology Construction

PolicyLint generates ontologies by combining the subsumptive relationships extracted from policies with a set of seed terms (Table 3). For each ontology, PolicyLint iterates through each of the seeds, selecting relationships that contain it. PolicyLint then expands the term list from the relationships in that iteration. PolicyLint continues iterating over the relationships until no new relationships are added to the ontology. If there exists any inconsistent relationship where X is subsumed under Y and Y is subsumed under X, PolicyLint uses the relationship that has a higher frequency (i.e., appearing in more privacy policies). Once a fixed point is reached, PolicyLint ensures that there is only one root node by creating connections between any nodes that do not contain inward-edges with the root of the ontology (i.e., "information" for the data ontology, and "public" for the entity ontology). Finally, PolicyLint ensures that no cycles exist in the ontology by identifying simple cycles in the graph and removing an edge between nodes to break the cycle. PolicyLint chooses which edge to remove by finding the edge that appears least frequently in the subsumptive relationships and ensures that the destination node has more than one in-edge to ensure that a new root node is not created.

2.2 Policy Statement Extraction

The goal of policy statement extraction is to extract a concise representation of a policy statement to allow for automated reasoning over this statement. We represent data sharing and collection statements as a tuple (*actor*, *action*, *data object*, *entity*) where the *actor* performs some *action* (i.e., share, collect, not share, not collect) on the *data object*, and the *entity* represents the entity receiving the data object. For example, the statement, "We will share your personal information with advertisers," can be represented by the tuple of (we, share, personal information, advertisers). PolicyLint extracts complete policy statements from privacy policy text by using patterns of the grammatical structures between data objects, entities, and verbs that represent sharing or collection (for brevity we call these verbs SoC verbs). This section describes the steps in policy statement extraction.

2.2.1 DED Tree Construction

The goal of constructing the data and entity dependency (DED) trees is to extract a concise representation of the grammatical relationships between the data objects, entities, and SoC verbs (i.e., the verbs that represent sharing or collection). The main intuition behind constructing these trees is to allow PolicyLint to infer semantics of the sentence based on the grammatical relationships between the tokens (i.e., *who* collects/shares *what* with *whom*). The DED tree for a sentence is derived from the sentence's dependency-based parse tree. However, the DED tree removes nodes and paths that are not relevant to the data objects, entities, or SoC verbs, and performs a set of simplifications to generalize the representation. The transformation for Example 2 is shown in Figure 2.

Table 4: SoC verbs used by PolicyLint

Type	Word
Sharing	disclose, distribute, exchange, give, provide, rent, report, sell, send, share, trade, transfer, transmit
Collection	access, check, collect, gather, know, obtain, receive, save, store, use

Example 2. *If you register for our cloud-based services, we will collect your email address.*

To construct DED trees, PolicyLint parses a sentence and uses its custom-trained NER model to label data objects and entities within the sentence (Section 2.1). PolicyLint merges noun phrases and iterates over sentence tokens to label SoC verbs by ensuring that the PoS (part-of-speech) tag of the token is a verb and the lemma of the verb is in PolicyLint's manually curated list of terms (Table 4). PolicyLint also labels the pronouns, "we," "I," "you," "me," and "us," as entities during this step. PolicyLint then extracts the sentence's dependency-based parse tree whose nodes are labeled with the data object, entity, and SoC verb labels as discussed earlier.

Negated Verbs: PolicyLint identifies negated verbs by checking for negation modifiers in the dependency-based parse tree. If the verb is negated, PolicyLint labels the node as negative sentiment. PolicyLint propagates the negative sentiment to descendant verb nodes in three cases. First, if a descendant verb is part of a conjunctive verb phrase with the negated verb, negative sentiment is propagated. For example, "*We do not sell, rent, or trade your personal information,*" means "not sell," "not rent," and "not trade." Second, if the descendant verb has an open clausal complement to the negated verb, negative sentiment is propagated. For example, "*We do not require you to disclose any personal information,*" initially has "require" marked with negative sentiment. Since "disclose" is an open clausal complement to "require," it is marked with negative sentiment. Third, if the descendant verb is an adverbial clause modifier to the negated verb, negative sentiment is propagated. For example, "*We do not collect your information to share with advertisers,*" initially has "collect" marked with negative sentiment. Since "share" is an adverbial clause modifier to "collect," "share" is marked with negative sentiment.

Exception Clauses: PolicyLint identifies exception clauses by traversing the parse tree and finding terms that represent exceptions to a prior statement, such as "except," "unless,"

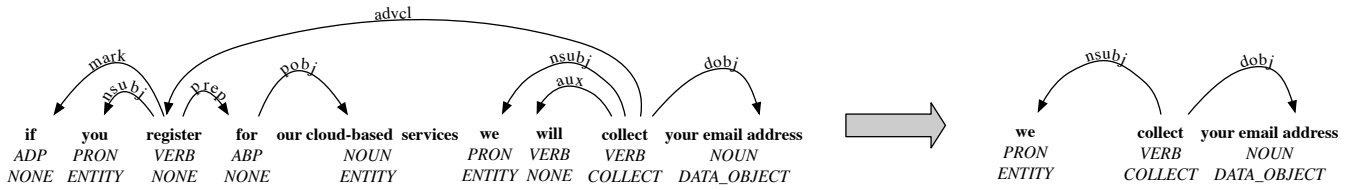


Figure 2: Transformation of Example 2 from its dependency-based parse tree to its DED tree.

"aside/apart from," "with the exception of," "besides," "without," and "not including." For each identified exception clause, PolicyLint traverses down the parse tree from the exception clause to identify verb phrases (subject-verb-object) and noun phrases related to that exception. PolicyLint then traverses upward from the exception term to identify the nearest verb node and appends as a node attribute the list of noun phrases and verb phrases identified in the downward traversal.

In certain cases, the term may not have a subtree. For example, the exception term may be a *marker* that introduces a subordinate clause. In the sentence, "*We will not share your personal information unless consent is given,*" the term "unless" is a marker that introduces the subordinate clause "your consent is given." For empty sub-trees, PolicyLint attempts the downward traversal from its parent node.

DED Tree construction: Finally, PolicyLint constructs the DED tree by computing the paths between labeled nodes on the dependency-based parse tree, copying labels and attributes described above. Note that PolicyLint also copies over all unlabeled subjects and direct objects from the parse tree, as they are needed to extract the information. PolicyLint further simplifies the tree by merging conjuncts of SoC verbs into one node if the coordinating conjunction is "and" or "or." For example, "*We will not sell, rent, or trade your personal information,*" can be simplified by collapsing "sell," "rent," and "trade" into one node. The resulting node's label is a union of all of the tags of the merged verbs (i.e., {share} + {collect} = {share, collect}. Similarly, PolicyLint repeats the same process for conjuncts of data objects and entities.

PolicyLint then prunes the DED tree by iterating through the nodes labeled as verbs in the graph and performing the following process. First, for a verb node labeled as an SoC verb, PolicyLint ensures that its sub-tree contains at least one other node labeled as an SoC verb, data object, or entity. If the node's sub-tree does not meet this condition, PolicyLint removes the subtree rooted at the node labeled as an SoC verb. Second, for verb nodes not labeled as SoC verbs, PolicyLint ensures that at least one SoC verb is contained in its sub-tree and that it meets the preceding conditions for an SoC verb. Similarly, if these conditions are not met, PolicyLint also removes the sub-tree rooted at that non-labeled verb node. For example, this pruning step causes the sub-tree rooted at the verb "register" to be removed in Figure 2.

2.2.2 SoC Sentence Identification

To identify sentences that describe sharing and collection practices, PolicyLint takes a set of positive examples of sentences as input and then extracts their DED trees to use as known patterns for sharing and collection phrases. In particular, we start by feeding PolicyLint a set of 560 example sentences that describe sharing and collect practices. PolicyLint generates the DED trees from these sentences and learns 82 unique patterns. The example sentences are auto-generated from a set of 16 sentence templates (Appendix B). We choose to auto-generate the sentences, because it is challenging to manually select a set of sentences with diverse grammatical structures. Our auto-generation does not adversely impact PolicyLint's extensibility, as adding a new pattern is as simple as feeding PolicyLint a new sentence for reflecting this new pattern.

PolicyLint iterates through each sentence of a given privacy policy. If the sentence contains at least one SoC verb and data object (labeled by NER), PolicyLint constructs the DED tree. PolicyLint then compares the sentence's DED tree to the DED tree of each known pattern. A pattern is matched if (1) the label types of the sentence's DED tree are equivalent to the ones of the known pattern's DED tree (e.g., {*entity*, *SoC_verb*, *data*}), and (2) the known pattern's DED tree is a subtree of the sentence's DED tree.

For a tree t_1 to be a subtree of tree t_2, (1) the tree structure must be equivalent, (2) the dependency labels on edges between nodes must match, and (3) the following three node conditions must hold. First, for SoC verb nodes to match, they must have a common lemma. For example, a node with the lemmas {*sell*, *rent*} matches a node with lemma {*rent*}. Second, if the node's part-of-speech is an apposition, the tags, dependency label, and lemmas must be equal. Third, for all other nodes, the tags and dependencies must be equal.

On sub-tree match, PolicyLint records the nodes in the sub-tree match and continues the process until either (1) each pattern is checked, or (2) the entire DED tree has been covered by prior sub-tree matches. If at least one sub-tree match is found, PolicyLint identifies the sentence as a potential SoC sentence and begins extracting the policy statement tuple.

2.2.3 Policy Extraction

The goal of policy extraction is to transform the DED tree into a (*actor*, *action*, *data object*, *entity*) tuple for a policy state-

ment. PolicyLint performs extraction starting with the SoC nodes present in the sub-tree matches. If multiple SoC nodes exist in the sub-tree matches, multiple tuples are generated. However, multiple sub-tree matches over the same SoC node result in the generation of only one tuple. The SoC determines the action (e.g., collect, not_collect). The action's sentiment is determined based on whether the node is labeled with positive or negative sentiment, as discussed in Section 2.2.1.

Actor Extraction: To extract the actor, PolicyLint starts from the matching SoC verb node. The actor is a labeled entity chosen from the (1) subject, (2) prepositional object, or (3) direct object (in that order). However, if the dependency is an open clausal complement or adverbial clause modifier, PolicyLint prioritizes the direct object and prepositional object over the subject. For example, "*We do not require you to disclose any personal information,*" has "disclose" as an open clausal complement to "require." In this case, the correct actor of this policy statement is the user (i.e., "you") rather than the vendor (i.e., "we"), which is correctly captured due to PolicyLint's dependency-based prioritization rules. If no match is found, PolicyLint traverses up one level in the DED tree and repeats. Finally, if no match is found, PolicyLint assumes that the actor is the implicit first party.

Data Object Extraction: To extract the data objects, PolicyLint starts from the matching SoC verb node. It traverses down the DED tree to extract all nodes labeled as data objects. The traversal continues until another SoC verb is reached. If no data objects are found, and the verb's subject and direct object are not labeled as a data object, PolicyLint extracts the data objects from the nearest ancestor SoC verb.

Entity Extraction: To extract the entities, PolicyLint starts from the matching SoC verb node. It traverses down the DED tree extracting all nodes labeled as entities that are not actors. The traversal continues until another SoC verb is reached.

Exception Clauses: PolicyLint considers exception clauses if the verb is marked with negative sentiment (e.g., not collect, not share), creating a cloned policy statement with the sentiment to change. We do not handle exception clauses for positive sentiment. For example, "*We might also share personal information without your consent to carry out your own requests,*" still shares personal information.

For negative sentiment verbs, there are three cases. First, if the exception clause's node attribute contains only data objects, PolicyLint replaces the data objects of the new policy with the data objects under the exception clause. For example, "*We will not collect your personal information except for your name and phone number,*" produces policies: (we, not_collect, personal information, NULL), (we, collect, [name, phone number], NULL). Second, if all noun phrases have an entity label, PolicyLint replaces the entities of the new policies with the entities under the exception attribute. For example, "*We do not share your demographics with advertisers except for AdMob,*" produces policies: (we, not_share, demographics, advertisers) and (we, share, demographics, AdMob). Third, if the labels are not data objects or entities, PolicyLint removes the initial policy statement. For example, "*We will not collect your personal information without your consent,*" produces the policy: (we, collect, personal information, NULL).

Policy Expansion: PolicyLint may extract multiple actors, actions, data objects, and entities when creating policy statements. These complex tuples are expanded. For example, ([we], [share, sell], [location, age], [Google, Facebook]) is expanded to (we, share, location, Google), (we, share, location, Facebook), (we, share, age, Google), etc.

2.3 Policy Contradictions

PolicyLint's components of ontology generation and policy extraction identify the sharing and collection statements in privacy policies. This section formally defines a logic for characterizing different contradictions. It then describes how PolicyLint uses this logic to identify candidate contradictions within privacy policies. We note that contradictions may occur between an app's privacy policy and the privacy policies of third-party libraries (e.g., advertisement libraries). While our study focuses specifically on contradictions within an individual privacy policy, the formal logic and subsequent analysis tools may also be used to include the privacy policies for third-party libraries with minimal modification.

2.3.1 Policy Simplification

PolicyLint simplifies policy statements for contradiction analysis. We refer to the (*actor*, *action*, *data object*, *entity*) tuple defined in Section 2.2 as a Complete Policy Statement (CPS). We simplify CPS statements about the sharing of data (i.e., *action* is share or not share) by capturing sharing as collection.

Definition 1 (Simplified Policy Statement: SPS)**.** *An SPS is a tuple, $p = (e, c, d)$, where d is the data object discussed by the statement, $c \in \{collect, not_collect\}$ represents whether the object is collected or not collected, and e is the entity receiving the data object.*

To transform a CPS into an SPS, we leverage three main insights. First, policies do not typically disclose whether the sharing of the data occurs at the client side or server side. Therefore, an actor sharing a data object with an entity may imply that the actor is collecting the data and performing the data sharing at the server side. In this case, a new policy statement would need to be generated for allowing the actor to collect the data object (Rule T1, Table 5). Second, a data object being shared with an entity may imply that the entity is collecting the information from the mobile device (Rule T2, Table 5). Similarly, a policy for stating that the actor does not share a data object with an entity implies that the entity is not collecting the data from the mobile device (Rule T3, Table 5). Finally, a policy for stating that the actor does not share a data object implies that the actor collects the data object, because

Table 5: Rules that transform a CPS into an SPS

Rule	Transformation Rules	Rationale
T1	(actor, share, data object, entity) $\implies$ (actor, collect, data object)	Unknown whether sharing occurs at the client side or server side
T2	(actor, share, data object, entity) $\implies$ (entity, collect, data object)	Can observe only client-side behaviors
T3	(actor, not_share, data object, entity) $\implies$ (entity, not_collect, data object)	Can observe only client-side behaviors
T4	(actor, not_share, data object, entity) $\implies$ (actor, collect, data object)	If mention not share, assume implicit collection

the policy would likely have not mentioned not sharing data that was never collected (Rule T4, Table 5).

However, there are two special cases. First, PolicyLint treats only verb lemmas "save" and "store" with positive sentiment ("not saving/storing" does not mean "not collecting"). Second, PolicyLint ignores negative statements with verb lemma "use." This case leads to false positives, as PolicyLint does not extract the collection purpose. For example, "*We do not use your location for advertising*," means that it is not collected for the specific purpose of advertising.

2.3.2 Contradiction Types

We model an app's privacy policy as a set of simplified policy statements P. Let D represent the total set of data objects and E represent the total set of entities, as represented by ontologies for data objects and entities, respectively. A policy statement $p \in P$ is a tuple, $p = (e, c, d)$ where $d \in D$, $e \in E$, and $c \in \{collect, not_collect\}$ (Definition 1).

Language describing policy statements may use different semantic granularities. One policy statement may speak in generalizations over data objects and entities while another statement may discuss specific types. For example, consider the policies $p_1 =$ (advertiser, not_collect, demographics) and $p_2 =$ (Google Admob, collect, age). If we want to identify contradictions, we need to know that Google AdMob is an advertiser and age is demographic information. These relationships are commonly referred to as *subsumptive relationships* where a more specific term is subsumed under a more general term (e.g., AdMob is subsumed under advertisers and age is subsumed under demographics).

We use the following notation to describe binary relationships between terms representing data objects and entities.

Definition 2 (Semantic Equivalence)**.** *Let x and y be terms partially ordered by an ontology o. $x \equiv_o y$ is true if x and y are synonyms, defined with respect to an ontology o.*

Definition 3 (Subsumptive Relationship)**.** *Let x and y be terms partially ordered by "is-a" relationships in an ontology o. $x \sqsubset_o y$ is true if term x is subsumed under the term y such that $x \not\equiv_o y$. Similarly, $x \sqsubseteq_o y \implies x \sqsubset_o y \vee x \equiv_o y$.*

Note that Definitions 2-3 parameterize the operators with an ontology o. PolicyLint operates on two ontologies: data objects and entities. Therefore, the following discussion parameterizes the operators with δ for the data object ontology and ε for the entity ontology. For example, $x \equiv_\delta y$ and $x \equiv_\varepsilon y$.

A contradiction occurs if two policy statements suggest that entities both may and may not collect or share a data object.

Table 6: Contradictions (C) and Narrowing Definitions (N) $P = \{(e_i, \text{collect}, d_k), (e_j, \text{not_collect}, d_l)\}$

Rule	Logic	Example
C_1	$e_i \equiv_\varepsilon e_j \wedge d_k \equiv_\delta d_l$	(companyX, collect, email address) (companyX, not_collect, email address)
C_2	$e_i \equiv_\varepsilon e_j \wedge d_k \sqsubset_\delta d_l$	(companyX, collect, email address) (companyX, not_collect, personal info)
C_3	$e_i \sqsubset_\varepsilon e_j \wedge d_k \equiv_\delta d_l$	(companyX, collect, email address) (advertiser, not_collect, email address)
C_4	$e_i \sqsubset_\varepsilon e_j \wedge d_k \sqsubset_\delta d_l$	(companyX, collect, email address) (advertiser, not_collect, personal info)
C_5	$e_i \sqsupset_\varepsilon e_j \wedge d_k \sqsubset_\delta d_l$	(advertiser, collect, email address) (companyX, not_collect, personal info)
N_1	$e_i \equiv_\varepsilon e_j \wedge d_k \sqsupset_\delta d_l$	(companyX, collect, personal info) (companyX, not_collect, email address)
N_2	$e_i \sqsubset_\varepsilon e_j \wedge d_k \sqsupset_\delta d_l$	(companyX, collect, personal info) (advertiser, not_collect, email address)
N_3	$e_i \sqsupset_\varepsilon e_j \wedge d_k \equiv_\delta d_l$	(advertiser, collect, email address) (companyX, not_collect, email address)
N_4	$e_i \sqsupset_\varepsilon e_j \wedge d_k \sqsupset_\delta d_l$	(advertiser, collect, personal info) (companyX, not_collect, email address)

Contradictions can occur at the same or different semantic levels. For example, the simplest form of contradiction is an exact contradiction where a policy states that an entity will both collect and not collect the same data object, e.g., (advertiser, collect, age) and (advertiser, not_collect, age). Due to subsumptive relationships (Definitions 3), there are 3 relationships between terms ($x \equiv_o y$, $x \sqsubset_o y$, and $x \sqsupset_o y$). Each binary relationship applies to both entities and data objects. Thus, there are $3^2 = 9$ types of contradictions, as shown in Table 6.

Contradictions have two primary impacts when privacy policies are analyzed. First, all contradictions impact analysis techniques that seek to automatically reason over policies and may result in these techniques making incorrect or inconsistent decisions. For example, unlike firewall rules that have a specific evaluation sequence, privacy policy statements do not have a specific pre-defined sequence of evaluation. Therefore, analysis techniques may make incorrect or inconsistent decisions based on the order in which they evaluate policy statements. Second, contradictions may impact a human analyst's understanding of a privacy policy, such as by containing misleading statements. We define two groupings of contradictions based on whether they may impact a human's comprehension of privacy policies or solely impact automated analysis.

Logical Contradictions (C_{1-5}): Logical contradictions are contradictory policy statements that are more likely to cause harm if users and analysts are not aware of the contradictory statements. Logical contradictions may cause difficulties when humans attempt to comprehend or interpret the sharing

Table 7: First Party Synonyms

First Party Synonyms
we, I, us, me, our app, our mobile application, our mobile app, our application, our service, our website, our web site, our site
app, mobile application, mobile app, application, service, company, business, web site, website, site

and collection practices discussed in the policy. They can be characterized as either exact contradictions (C_1) or those that discuss *not* collecting broad types of data and later discuss collecting exact or more specific types (C_{2-5}). One example is a policy that initially claims not to collect personal information, but later in fine print discloses collecting a user's name and email address for advertisers. Similar to the goal of software `lint` tools, PolicyLint flags logical contradictions as problems within policies to allow a human analyst to manually inspect the statements and analyze intent. As shown in Section 3, these contradictions may lead to the identification of intentionally deceptive policy statements or those that may result in ambiguous interpretations.

Narrowing Definitions (N_{1-4}): Narrowing definitions are contradictory policy statements where broad information is stated to be collected, and specific data types are stated not to be collected. These statements narrow the scope of the types of data that are collected, such as stating that personal information is collected while your name is *not* collected. Note that narrowing definitions are not necessarily an undesirable property of policies, because saying that a broad data type is collected does not necessarily imply that every specific subtype is collected, but narrowing definitions may result in vague policies. There may be clearer ways for policy writers to convey this information, such as explicitly stating the exact data types collected and shared. For example, if the app collects your email address, the policy could directly state, "We collect your email address," instead of including a narrowing definition, such as "We collect personal information. We do not collect your name." However, policies that narrow the scope of their data sharing and collection practices can be seen as more desirable in contrast to policies that just disclose practices over broad categories of data. Nonetheless, narrowing definitions impact the logic behind analysis techniques, as they must consider prioritization of data objects and entities.

2.3.3 Contradiction Identification

PolicyLint uses the contradiction types from Table 6 to determine a set of candidate contradictions. It then uses a set of heuristics to reduce the set of candidate contradictions that are potentially low-quality indicators of underlying problems. Next, PolicyLint prepares the contradictions for presentation by collapsing duplicate contradictions, linking other metadata (e.g., download counts of apps), and by using a set of filtering heuristics to allow the regulator or privacy analysts to focus on specific subclasses of candidate contradictions. The remainder of this section describes this process.

Initial Candidate Set Selection: Given policy statements p_1 and p_2, PolicyLint ensures that $p_1.c$ does not equal $p_2.c$, as contradictions require opposing sentiments. PolicyLint then compares entities $p_1.e$ and $p_2.e$, determining whether they are equal or have a subsumptive relationship. A subsumptive relationship occurs if there is a path between the entities in the entity ontology. When comparing entities, PolicyLint treats the terms in Table 7 as synonyms for the first party (i.e., "we"). If an entity match is found, PolicyLint then performs the same steps for data objects $p_1.d$ and $p_2.d$ using the data object ontology. If a data object match is found, PolicyLint adds the candidate contradiction to the candidate set. Note that in policy statements PolicyLint ignores entities and data objects that are not contained in the ontologies, as it cannot reason about those relationships. However, if PolicyLint cannot find a direct match for a term in the ontologies, it will try to find submatches by splitting the term on the coordinating conjunction terms (e.g., "and," "or") and checking for their existence in the ontologies. Furthermore, PolicyLint does not identify policy statements as contradictions if they are generated from the same sentence due to the semantics of exception clauses. For example, "*We do not collect your personal information except for your name*," produces simplified policy statements (we, not_collect, personal information) and (we, collect, name). While the semantics of this statement is clear, our definition of contradictions would incorrectly identify these statements as a C_2 contradiction. Therefore, PolicyLint ignores same-sentence contradictions to reduce false positives.

Candidate Set Reduction: PolicyLint uses heuristics to prune candidate contradictions that are likely low-quality indicators of underlying problems. PolicyLint removes contradictions that occur based on potentially poor relationships discovered in the ontologies. For example, PolicyLint filters out contradictions that occur between certain data object pairs, such as "usage information" and "personal information." Contradictions whose entities refer to the user (e.g., "user," "customer," "child") or involve terms for general data objects (e.g., "information," "content") or entities (e.g., "individual," "public") are also removed. Finally, PolicyLint removes candidate contradictions where a negative-sentiment policy statement may be conditioned with age restrictions or based on user choice by searching for common phrases in the sentences that are used to generate the policy statements (e.g., "under the age of," "from children," "you do not need to provide"). Note that some of these reductions may occur during candidate set construction to reduce complexity of the analysis.

Candidate Set Filtering: PolicyLint further filters the set of candidate contradictions into subsets based on the data objects involved in the contradictions to allow for targeted exploration during verification. For example, all of the contradictions with statements involving collecting email address but not collecting personal information are placed into one subset (e.g., (*, collect, email address) and (*, not_collect, PII)).

Contradiction Validation: Given the filtered subsets of can-

didate contradictions, the next step is to explore certain subsets and validate candidate contradictions. To validate a candidate contradiction, the analyst reads through the policy statements and sentences that are used to generate them in context of the entire policy, and makes a decision.

3 Privacy Study

Our primary motivation for creating PolicyLint is to analyze contradictory policy statements within privacy policies. In this section, we use PolicyLint to perform a large-scale study on 11,430 privacy policies from top Android apps.

Dataset Collection: To select our dataset, we scrape Google Play for the privacy policy links of the top 500 free apps for each of Google Play's 35 app categories in September 2017. Note that the "top free apps" are based on the ranking provided by Google Play ("topselling_free" collection per app category). We download the HTML privacy policies using the Selenium WebDriver in a headless Google Chrome browser to allow for the execution of dynamic content (e.g., JavaScript). We exclude apps that do not have a privacy policy link on Google Play, those whose pages are unreachable at the time of collection, and privacy policies where the majority of the document is not English, as discussed in Appendix A. We convert the HTML policies to plaintext documents. Our final dataset consists of 11,430 privacy policies.

3.1 General Policy Characteristics

PolicyLint extract sharing and collection policy statements from 91% of the policies in our dataset (10,397/11,430). From those policies, PolicyLint extract 438,667 policy statements from 177,169 sentences that PolicyLint identifies as a sharing or collection sentence. Of those policy statements, 32,876 have negative sentiment and 405,789 have positive sentiment. In particular, 60.5% (6,912/11,430) of the policies have at least one negative-sentiment policy statement and 89.6% (10,239/11,430) of the policies have at least one positive sentiment policy statement. We explore why PolicyLint does not extract policy statements for 9% of the policies by analyzing a random subset of 100 policies and find that it is mainly due to dataset collection or preprocessing errors (Appendix C).

Finding 1: *Policies frequently contain negative sentiment policy statements that discuss broad categories of data.* For 60.5% of the policies with at least one negative sentiment policy statement, the data object "personal information" appears in 67.7% of those policies (4,681/6,912). This result demonstrates the importance of handling negative policy statements, as around 41.0% of the policies contain a negative sentiment policy statement for claiming that a broad type of data (i.e., "personal information") is not collected. Further, we measure the distance from the negation (i.e., "not") to the verb that the negation modifies, and find that 28.2% (3,234/11,430) of the policies have a distance greater than one word away. This result calls into question prior approaches [26,28] that assume only positive sentiment when considering sharing and collection statements, as the prior approaches could be incorrect up to 60.5% of the time when reasoning over sharing and collection statements. Further, approaches [32] that handle negations using bigrams would have failed to reason about 28.2% of the policies.

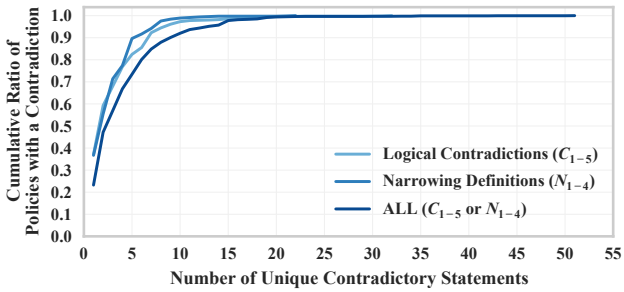


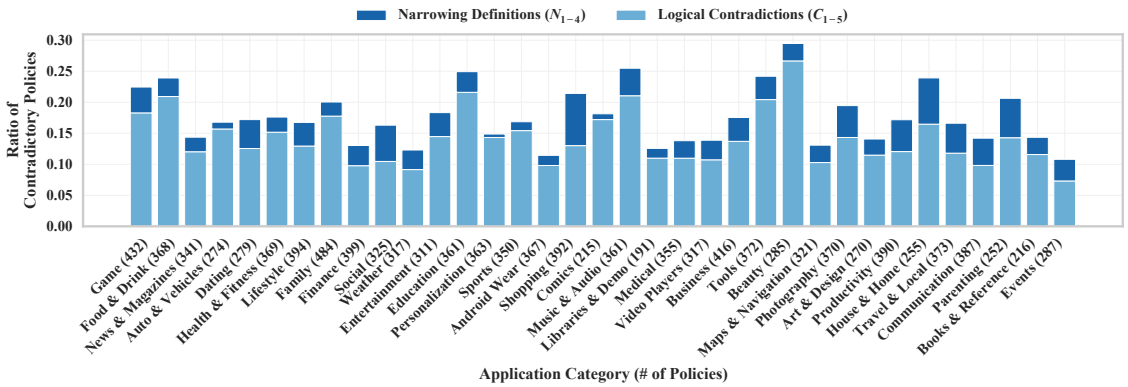
Figure 3: CDF of Contradictory Policy Statements: 50% of contradictory policies have 2 or fewer logical contradictions.

3.2 Candidate Contradictions

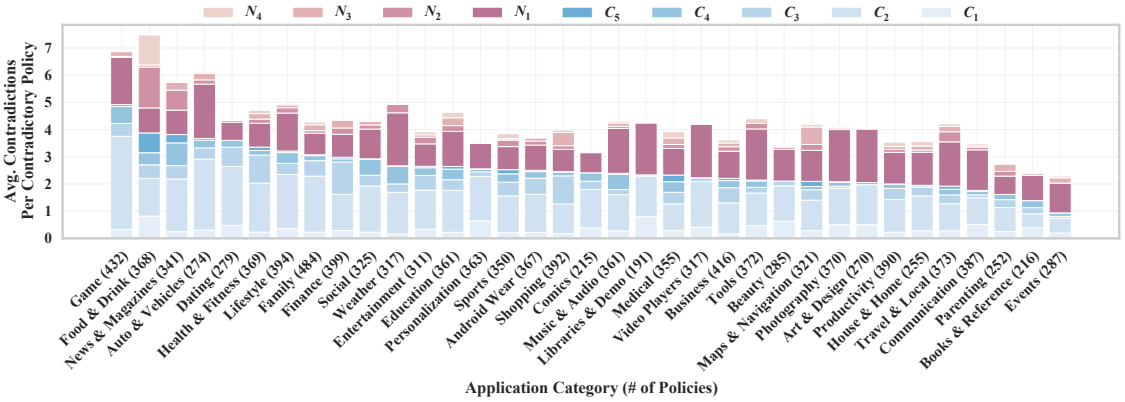
Based on PolicyLint's fine-grained policy statement extraction, we find that 59.1% (6,754/11,430) of the policies are candidates for contradiction analysis, as they contain at least one positive and one negative sentiment policy statements. Among these policies, there are 13,871 and 129,575 policy statements with negative and positive sentiment, respectively.

Finding 2: *For candidate contradictions, 14.2% of the privacy policies contain logical contradictions (C_{1-5}).* PolicyLint identifies 9,906 logical contradictions across around 14.2% (1,618/11,430) of the policies. Therefore, 14.2% of the policies may contain potentially misleading statements. Figure 3 shows that around three-fifths (59.2%) of the policies with at least one logical contradiction have 2 or fewer unique contradictions. The relatively low number of candidate contradictions per policy indicates that manual validation is feasible. As roughly 6 in 7 policies are not contradictory, writing policies without logical contradictions is possible.

Finding 3: *Contradiction prevalence and frequency do not substantially vary across Google Play app categories.* Figure 4a shows the ratio of policies containing candidate contradictions per each Google Play category. The categories with policies most and least prone to contradiction are *Beauty* and *Events*, respectively. However, when analyzing the policies within those categories, we find that their means are skewed by contradictory policies for apps by the same developer. When we recompute the means without the outliers, these categories follow the general trend. Policies with logical contradictions accompany 7.3% to 20.9% of apps across all categories. We find that policies with logical contradictions are not substantially more prevalent in particular categories of apps, but instead occur consistently in apps from every category. We also find that prevalence of logical contradiction



(a) Ratio of Contradictory Policies per Category: 79.7% of contradictory policies have at least one or more logical contradictions (C_{1-5}) that may indicate potentially deceptive statements.



(b) Average number of unique candidate contradictions per category: Logical contradictions (C_{1-5}) and narrowing definitions (N_{1-4}) are both widely prevalent across Google Play categories.

Figure 4: Distribution of Candidate Contradictions across Google Play Categories.

does not substantially vary by download count as well.

Figure 4b displays the average number of candidate contradictions for policies containing one or more contradictions. We find that frequency of logical contradiction for contradictory policies does not substantially vary across Google Play categories. Initial analysis indicates that contradictory policies for apps in the *Games* category contain around 4.9 logical contradictions on average. Further analysis reveals that this result is due to policies with 19 unique logical contradictions in 9 apps produced by the same developer, and one app that has 31 unique logical contradictions. Excluding these outliers brings the category's average to 3.16 logical contradictions per app, fitting the trend of the rest of the categories. This result may indicate that poor policies are linked to problematic developers. Similar analysis on categories *Food & Drink*, *Auto & Vehicles*, and *News & Magazines* produces similar results. We find that the number of logical contradictions per policy is roughly equivalent across app categories, indicating that one app category is not necessarily more contradictory on average than another.

Finding 4: *Negative sentiment policy statements that discuss broad categories of data are problematic.* Figure 5 shows the frequency of the most common data-type pairs referred to in contradictory policy statements. The contradictory policy statements in the topmost row are most problematic. This row represents logical contradictions, which are either (1) exact contradictions or (2) discussion of not collecting broad types of data and collecting more specific data types (C_{1-5}). As we demonstrate in Section 3.3, logical contradictions can lead to a myriad of problems when one interprets the policy including making interpretation ambiguous in certain cases. The leftmost column corresponds to narrowing definitions (N_{1-4}), which solely impact automated analysis techniques, as discussed in Section 2.3.

Finding 5: *For candidate contradictions, 17.7% of the privacy policies contain at least one or more logical contradictions (C_{1-5}) or narrowing definitions (N_{1-4}).* PolicyLint identifies 17,986 logical contradictions and narrowing definitions across around 17.7% (2,028/11,430) of the policies. Figure 3 shows that slightly more than half (57.0%) of the contradictory policies have 3 or fewer unique logical contradictions and narrowing definitions. As discussed in Section 2.3, logical contradictions and narrowing definitions impact approaches that seek to automatically reason over privacy policies. To

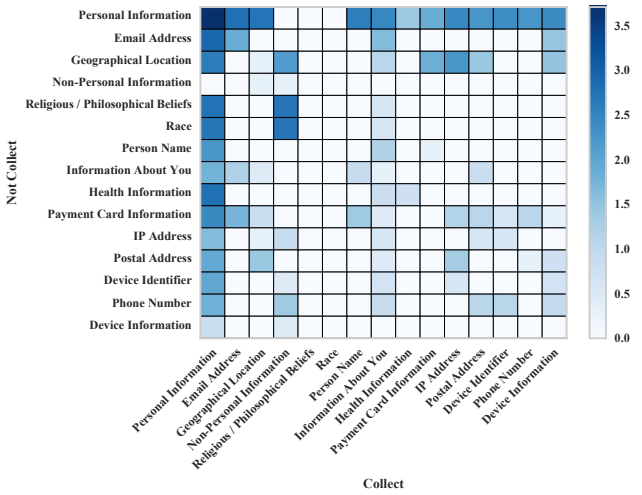


Figure 5: Log 10 Frequency of Data-Type Pairs in Contradictions: Negative statements that discuss broad categories of data are problematic (i.e., C_{1-5}).

correctly reason over contradictions, analysis techniques must include logic to prioritize specificity of data types and entities, and be able to identify potentially undecidable cases, such as exact contradictions (C_1). No prior approaches [26, 28, 29, 32] that attempt to reason over policies operate at the required granularity to identify contradictions or contain the logic to correctly reason over them. Therefore, these prior approaches could make incorrect or inconsistent decisions around 17.7% of the time around 1–3 times per policy on average. We perform similar analysis across categories as Finding 3 and find that logical contradictions and narrowing definitions do not substantially vary across Google Play categories.

3.3 Deeper Findings

In this section, we describe the findings from validating candidate contradictions. We limit our scope to logical contradictions (C_{1-5}), as they may be indicative of misleading policy statements. Due to resource constraints, we do not validate all 9,906 candidate logical contradictions from the 1,618 policies. Instead, we narrow the scope of our study by choosing categories of candidate contradictions to focus on. Our selection and validation methodology is described below.

Selection Methodology: To select the categories of candidate contradictions to focus on, we analyze Figure 5 for the data objects involved in the contradictory statements. We limit our scope to logical contradictions (C_{1-5}) that discuss not collecting "personal information" and collecting "email address," "device identifier," or "personal information." We also explore two other categories of candidate contradictions in which one type of data can be derived from the other type; these categories caught our attention when analyzing the heat map. Within each category of candidate contradictions, we choose which candidate contradictions to validate by sorting the contradictions based on the belonging app's popularity and working down the list. We spend around one week validating contradictions where our cutoffs are due to time constraints and attempting to achieve coverage across categories.

Validation Methodology: To validate candidate contradiction, one of three student authors reads through the sentences that are used to generate each policy statement for the candidate contradiction to ensure correctness of policy statement extraction. If there is an error with policy statement extraction, we record the candidate contradiction as a false positive and stop analysis. Next, we locate the sentences within the policy and view the context in which they appear (i.e., section, surrounding sentences) to determine whether the policy statements are contradictory. We try to determine why the contradiction occurs if possible and record any observations. If the author is uncertain about his/her decision, a second author analyzes it. The two authors discuss and resolve conflicts, with no conflicts left unresolved after discussion.

3.3.1 Personal Information and Email Addresses

For candidate contradictions with negative statements about "personal information" and with positive statements about "email address," we find 618 candidate contradictions across 333 policies (C_2, C_4, C_5) We validate 204 candidate contradictions from 120 policies. We find that 5 candidate contradictions are false positives due to inaccuracies of labeling data objects by the NER model. From the 199 remaining candidate contradictions across 118 policies, we have the following main findings. Note that for the findings discussed below, the terms "personally identifiable information" and "personal information" are commonly used synonymously in USA regulations, and "personal data" is considered the EU equivalent albeit covering a broader range of information.

Finding 6: *Policies are stating certain types of common personally identifiable information, such as email addresses, as non-personally identifiable.* When validating 14 candidate contradictions, we find 14 policies for explicitly stating that they do NOT consider email address as personally identifiable information. 11 of those policies are released by the same developer (OmniDroid) where the most popular app in the set (com.omniluxtrade.allrecipes) has over 1M+ downloads. OmniDroid's policy explicitly lists email address when defining non-personally identifiable information. The remaining 3 policies belong to another app developer, PlayToddlers. The apps are explicitly targeted toward children from 2-8 years old and have between 500K-1M+ downloads for each app. Their policy states the following sentence verbatim, "When the user provides us with an email address to subscribe to the "PlayNews" mailing list, the user confirms that this address is not a personal data, nor does it contain any personal data."

The fact that *any* privacy policies are declaring email addresses as non-personal information is surprising, as it goes against the norms of *what* data is considered personal information as defined by regulations (e.g., CalOPPA, GDPR),

standards bureaus (NIST), and common sense.

Finding 7: *Services that auto-generate template-based policies for app developers are producing contradictory policies.* During our validation process, we notice that many policies have similar structural compositions and contain a lot of the same text in paragraphs. When validating 78 candidate contradictions, we find 59 contradictory policies that are automatically generated or used templates. Identical policy statements from various developers suggest that some policies may be generated automatically or acquired from a template. We investigate these cases and identify 59 policies that use 3 unique templates. We check that these policies are not ones for apps created by the developers or organization. Findings 8 and 11 discuss the problems caused by the templates. This result demonstrates that poor policy generators can be a contributing factor for numerous contradictory policies.

Finding 8: *Policies use blanket statements affirming that personal information is not collected and contradict themselves by stating that subtypes of personal information are collected, such as email addresses.* When validating 182 candidate contradictions, we find 104 policies for broadly making blanket statements that personal information is not collected in one part of the policy and then directly contradicting their prior statements by disclosing that they collect email addresses. We find 69 of those policies (127 validated contradictions) for stating that they do not collect personal information, but later stating that they collect email addresses for some purpose. Of those 69 policies, 32 policies define email address as personal information in one part of their policy. Due to the lack of definition of what they consider personal information in the other 37 policies, it is unclear whether they do not consider email address as personal information or are just contradictory.

The same organization (emoji-keyboard.com) produces 20 of those policies that explicitly define email addresses as personal information, but contradict themselves. The most popular app in that group has 50M+ downloads (emoji.keyboard.emoticonkeyboard). The following two sentences are in the policy verbatim: (1) *"Since we do not collect Personal Information, we may not use your personal information in any way."*; (2) *"For users that opt in to Emoji Keyboard Cloud, we will collect your email address, basic demographic information and information concerning the words and phrases that you use ("Language Modeling Data") to enable services such as personalization, prediction synchronization and backup."* This case is clearly a contradictory statement and arguably a misleading practice.

A policy for a particular app with 1M+ downloads (com.picediting.haircolorchanger) appears to have been potentially trying to mislead users by using bold text to highlight desirable properties and then contradicting themselves. For example, the following excerpt is in the policy verbatim including the bold typography: *"**We do not collect any Personal information** but it may be collected in a number of ways. We may collect certain information that you voluntarily provide to us which may contain personal information. For example, we may collect your name, email address you provide us when you contact us by e-mail or use our services..."* The use of bold typography and general presentation of these policy statements could potentially be considered as attempting to deceive the reader, who may not perform a close read of the text in fine-print. This finding validates PolicyLint's value in flagging problematic areas in policies to aid in the identification of deceptive statements.

Finding 9: *Policies consider hashed email addresses as pseudonymized non-personal information and share it with advertisers.* When validating three candidate contradictions, we find that two policies discuss sharing hashed email addresses with third parties, such as advertisers. One candidate contradiction is a false positive due to misclassifying a sentence discussing opt-out choices as a sharing or collection sentence. The other policy belongs to an app named Tango (com.sgiggle.production). Tango is a messaging and video call app, which has over 100M+ downloads on Google Play and according to their website has 390M+ users globally. Their policy states the following sentences verbatim ,*"For example, we may tell our advertisers the number of users our app receives or share anonymous identifiers (such as device advertising identifiers or hashed email addresses) with advertisers and business partners."* Tango explicitly states that they consider hashed email addresses as anonymous identifiers. It is arguable whether hashing is sufficient for pseudonymization as defined by GDPR, as it is likely that advertisers are using hashed email addresses to identify individuals.

3.3.2 Personal Information and Device Identifiers

For the candidate contradictions with negative statements about "personal information" and with positive statements about "device identifiers," we find 234 candidate contradictions across 155 policies. We investigate this group of candidate contradictions as there are differing regulations across countries on whether device identifiers are considered personal information. For example, various court cases within the US (Robinson v. Disney Online, Ellis v. Cartoon Network, Eichenberger v. ESPN) rule that device identifiers are not personal information. However, the GDPR defines device identifiers as personal information. Therefore, our goal is to check whether policies are complying to the stricter GDPR definition of personal information or to the US definition, as the outcome could hint toward problems with complying to regulations across country boundaries. In total, we validate 10 candidate contradictions across 9 policies.

Finding 10: *Policies are considering device identifiers as non-personal information, raising concerns regarding globalization of their policies.* When validating 10 candidate contradictions, we find 9 policies for stating that they do not collect personal information, but later state that they collect device identifiers. We find that classification of device identifiers

varies across policies. We find 4 policies that explicitly describe device identifiers as non-personal information. The most popular app is Tango (com.sgiggle.production), which boasts of 390M+ global users on their website. It is likely a safe assumption that some of those users are in the EU, which is subject to GDPR. As their current policy still contains this statement, it may hint that they may not be GDPR compliant.

To reduce the threats to the validity of our claims, we re-request the 9 policies using a proxy to route the traffic through an EU country (Germany) to ensure that an EU-specific policy is not served based on the origin of the request. We request the English version of the policy where applicable and find similarly problematic statements in regard to not treating device identifiers as personal information.

3.3.3 Personal Information

We find 5100 candidate contradictions across 1061 policies where the data type of both the negative statement and positive statement is "personal information." We validate 254 candidate contradictions across 153 policies.

Finding 11: *Policies directly contradict themselves.* When validating the 254 candidate contradictions, we find that the 153 policies directly contradict themselves on their data practices on "personal information." For example, the policy for an app with 1M+ downloads states "We may collect personal information from our users in order to provide you with a personalized, useful and efficient experience." However, later in the policy they state, "We do not collect Personal Information, and we employ administrative, physical and electronic measures designed to protect your Non-Personal Information from unauthorized access and use." These scenarios are clearly problematic, as the policies state both cases and it makes it difficult, if not, impossible to determine their actual data sharing and collection practices.

3.3.4 Derived Data

In this section, we explore cases of candidate contradictions where the negative statements discuss data that can be derived from the data discussed in the positive statement. In particular, we explore two cases: (1) coarse location from IP address; and (2) postal address from precise location.

For "coarse location from IP," we find 170 candidate contradictions from 167 policies that represent collecting IP address and not collecting location. We remove candidate contradictions whose statements discuss precise location, as IP address does not provide a precise location. This filtering results in 18 candidate contradictions from 18 different policies. We validate 15 candidate contradictions across 15 different policies for this case. We note that 3 candidate contradictions from 3 policies are false positives due to incorrect negation handling.

For "postal address from precise location," we find 27 candidate contradictions across 20 policies. Note that we remove candidate contradictions that discuss coarse location, as they are not precise enough to derive postal addresses. We validate 22 candidate contradictions across 17 apps, as 5 candidate contradictions are false positives due to sentence misclassification (4 cases) or errors of handling negations (1 case).

Finding 12: *Policies state that they do not collect certain data types, but state that they collect other data types in which the original can be derived.* When validating the 15 candidate contradictions for "coarse location from IP," we find that all 15 policies are stating that they do not collect location information, but state that they automatically collect IP addresses. As coarse location information can generally be derived from the user's IP address, it can be argued that the organization is technically collecting the user's location information. Interestingly, two of the policies discuss that if users disable location services, then location will not be collected. It is highly unlikely that companies cease IP address collection based on device privacy settings. However, as IP address collection typically occurs passively at the server side, we cannot claim with 100% certainty that the companies still collect IP addresses when location services are disabled.

When validating 20 candidate contradictions for "postal address from precise location," we find that 15 policies discuss not collecting postal addresses, but then state that they collect locations. Similar to the preceding case, postal addresses can be derived from location data (i.e., latitude and longitude). Again, the argument can be made that they are collecting data precise enough to be considered a postal address, causing a contradiction. For the other two candidate contradictions from two policies, it is not clear whether it is actually a contradiction, as they state that they do not collect addresses from the user's address book, which is more specific than a general statement about not collecting addresses.

3.4 Notification to Vendors

For the 510 contradictions that are validated across 260 policies, we contact each vendor via the email address listed for the privacy policy's corresponding app on Google Play. We disclose the exact statements that we find to be contradictory and explain our rationale. We ask whether they consider the statements to be contradictory and request clarifications on their policy. Figure 6 (Appendix) shows a template of the email. Overall, 244 emails are successfully delivered, as 16 email addresses are either invalid or unreachable. In total, we receive a 4.5% response rate (11/244), which is relatively substantial when considering that responding to our emails could raise liability concerns. All of the responses are received within a week or less after we send the initial emails. We have not received additional responses in the 4 months that have passed before publication. The remainder of this section discusses the responses.

Fixed Policy: Three vendors agree with our findings and update their policy to remove the contradiction. One ven-

dor states that there is an "error" in their privacy policy and updates it accordingly. We confirm that the policy has been updated. The remaining two vendors state that the self-contradictory portion of the policy is a leftover remnant from a prior update and should have been removed. They clarify that they do not collect email addresses.

Disagreed with our Findings: One vendor explicitly disagrees with our findings. Their policy states that they do not collect personal information, but also states that they collect email addresses. The vendor responds by claiming that email addresses are only personal information if it contains identifiable information, such as your name (e.g., john.doe@gmail.com), but are not personal information if it does not contain identifiable information (e.g., wxyz@gmail.com). They state that they explicitly tell users not to submit email addresses that contain personal information and thus their policy is not contradictory. This interpretation is surprising, because it goes against the definition of email addresses as personal information, as defined by regulations (e.g., CalOPPA, GDPR) and standards bureaus (NIST).

Claimed Outdated Policy: Four vendors respond that they do not find the reported statements in their current policy and that we analyzed an older version of their policy. One of the vendors sent us their updated policy for their mobile app. However, the policy has a link to refer to their full privacy policy, which links to the policy analyzed by us. We request clarifications on their "full policy," but receive no response. We analyze the remaining three policies and find that they have the contradictory statements removed from their policy.

No Comment: Two vendors respond without providing a comment or clarification of their policy. One developer simply replies back "Thanks for the observation." The other responds that their app is removed from Google Play.

4 Limitations

PolicyLint provides a set of techniques to extract a concise structured representation of both collection and sharing statements from the unstructured natural language text in privacy policies. In so doing, it provides unprecedented analysis depth, and our findings in Section 3 demonstrate the utility and value of such analysis. However, extracting structured information from unstructured natural language text continues to be an open and active area of NLP research, and there are currently no perfect techniques for this challenging problem. PolicyLint is thus limited by the current state of NLP techniques, such as the limitations of NLP parsers and named-entity recognition. Its performance also depends on its verb lists and policy statement patterns, which may be incomplete despite our best efforts, reducing overall recall. We note that as a `lint` tool, our goal is to provide high precision at a potential cost to high recall. We note that PolicyLint achieves 97.3% precision (496/510) based on the 14 false positives identified during our validation of contradictions.

Another limitation is that PolicyLint cannot extract the conditions or purposes behind collection and sharing statements. Extracting such information would allow for a more holistic analysis, but doing so would require advances in decades-old NLP problems, including semantic role labeling, coreference resolution, and natural language understanding.

Finally, our analysis focuses on policies of Android apps. While we cannot claim that our findings would certainly generalize to policies from other domains (e.g., iOS, web), we hypothesize that self-contradictions likely occur across the board, as policies are written for all platforms in largely the same way to describe data types collected by all platforms.

5 Related Work

Recent research has increasingly focused on automated analysis of privacy policies. Various approaches have used NLP for deriving answers to a limited number of binary questions [31] from privacy policies, applied topic modeling to reduce ambiguity in privacy policies [27], and used data mining [30] or deep learning [15] models to extract summaries from policies of what and how information is used. Some other approaches [26, 28] have used *crowdsourced* ontologies for policy analysis. These approaches are often limited by lack of accuracy, completeness, and collection complexity. Other approaches [12, 18] identify patterns to extract subsumptive relationships. However, they do not provide methodology to generate usable ontologies from such extracted information and rely on a fixed lexicon. Prior research has also attempted to infer negative statements in privacy policies with limited success. Zimmeck et al. [32] and Yu et al. [29] rely on keyword-based approaches of using bi-grams and verb modifiers, respectively, to detect the negative statements. In contrast to these previous approaches, our work provides a more comprehensive analysis with an automatically constructed ontology and accounting for negations and exceptions in text.

Prior approaches [2, 5, 29] identify the possibility of conflicting policy statements. However, we are the first to characterize and automatically analyze self-contradictions resulting from interactions of varying semantic levels and sentiments.

Analyzing the usability and effectiveness of privacy policies is another well-researched focus area. Research has shown that privacy policies are hard to comprehend by users [23] and proposals have been made to simplify their understanding [24]. Cranor et al. [5] perform a large-scale study of privacy notices of US financial institutions to highlight a number of concerning practices. Their policy analysis relies on standardized models used for such notices; in contrast, privacy policies in mobile apps follow no such standards making the analysis more challenging. Other approaches [20, 21, 25] have attempted to bridge the gap between users' privacy expectations and app policies. While standardizing privacy policy specification has been attempted [4] with limited success [22], mobile apps' privacy policies have generally failed

to adhere to any standards. The findings in our study highlight the need to renew standardization discussion.

6 Conclusion

In this paper, we have presented PolicyLint, a privacy policy analysis tool that conducts natural language processing to identify contradictory sharing and collection practices within privacy policies. PolicyLint reasons about contradictory policy statements that occur at different semantic levels of granularity by auto-generating domain ontologies. We apply PolicyLint on 11,430 privacy policies from popular apps on Google Play and find that around 17.7% of the policies contain logical contradictions and narrowing definitions, with 14.2% containing logical contradictions. Upon deeper inspection, we find a myriad of concerning issues with privacy policies, including misleading presentations and re-defining common terms. PolicyLint's fine-grained extraction techniques and formalization of narrowing definitions and logical contradictions lay the foundation to help ensure the soundness of automated policy analysis and identify potentially deceptive policies.

Acknowledgment

We thank our shepherd, David Evans, and the anonymous reviewers for their valuable comments. This work is supported in part by NSF grants CNS-1513690, CNS-1513939, and CCF-1816615. Any findings and opinions expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[2] Travis D. Breaux and Ashwini Rao. Formal Analysis of Privacy Requirements Specifications for Multi-tier Applications. In *Proceedings of the IEEE International Requirements Engineering Conference (RE)*, 2013.

[3] Federal Trade Commission. Privacy Online: Fair Information Practices in the Electronic Marketplace: A Federal Trade Commission Report to Congress, May 2000.

[4] Lorrie Faith Cranor, Marc Langheinrich, Massimo Marchiori, Martin Presler-Marshall, and Joseph Reagle. The Platform for Privacy Preferences 1.0 (P3P1.0) Specification. *W3C Recommendation*, 16, April 2002.

[5] Lorrie Faith Cranor, Pedro Giovanni Leon, and Blase Ur. A Large-Scale Evaluation of US Financial Institutions' Standardized Privacy Notices. *ACM Transactions on the Web*, 2016.

[6] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security (CCS)*, 2013.

[7] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2010.

[8] David Evans. Annotation-Assisted Lightweight Static Checking. In *Proceedings of the International Workshop on Automated Program Analysis, Testing and Verification*, 2000.

[9] David Evans, John Guttag, Jim Horning, , and Yang Meng Tan. LCLint: A Tool for Using Specifications to Check Code. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 1994.

[10] David Evans and David Larochelle. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Proceedings of the USENIX Security Symposium*, 2001.

[11] David Evans and David Larochelle. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software*, January 2002.

[12] Morgan C. Evans, Jaspreet Bhatia, Sudarshan Wadkar, and Travis D. Breaux. An Evaluation of Constituency-based Hyponymy Extraction from Privacy Policies. In *Proceedings of the IEEE International Requirements Engineering Conference (RE)*, 2017.

[13] Federal Trade Commission Act: Section 5: Unfair or Deceptive Acts or Practices. https://www.federalreserve.gov/boarddocs/supmanual/cch/ftca.pdf.

[14] Michael Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe Exposure Analysis of Mobile In-App Advertisements. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2012.

[15] Hamza Harkous, Kassem Fawaz, Rémi Lebret, Florian Schaub, Kang G. Shin, and Karl Aberer. Polisis: Automated Analysis and Presentation of Privacy Policies Using Deep Learning. In *Proceedings of the USENIX Security Symposium*, 2018.

[16] Marti A. Hearst. Automatic Acquisition of Hyponyms from Large Text Corpora. In *Proceedings of the Conference on Computational Linguistics (COLING)*, 1992.

[17] Matthew Honnibal and Ines Montani. spaCy 2: Natural Language Understanding with Bloom Embeddings, Convolutional Neural Networks, and Incremental Parsing, 2017.

[18] Mitra Bokaei Hosseini, Travis D. Breaux, and Jianwei Niu. Inferring Ontology Fragments from Semantic Role Typing of Lexical Variants. In *Proceedings of the International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ)*, 2018.

[19] S. C. Johnson. Lint, a C Program Checker. In *Computer Science Technical Report*, pages 78–1273, 1978.

[20] Jialiu Lin, Norman Sadeh, and Jason I. Hong. Modeling Users' Mobile App Privacy Preferences: Restoring Usability in a Sea of Permission Settings. In *Proceedings of the Symposium on Usable Privacy and Security (SOUPS)*, 2014.

[21] Fei Liu, Rohan Ramanath, Norman Sadeh, and Noah A. Smith. A Step Towards Usable Privacy Policy: Automatic Alignment of Privacy Statements. In *Proceedings of the International Conference on Computational Linguistics (COLING)*, 2014.

[22] Aditya Marella, Chao Pan, Ziwei Hu, Florian Schaub, Blase Ur, and Lorrie Faith Cranor. Assessing Privacy Awareness from Browser Plugins. In *Proceedings of the Symposium on Usable Privacy and Security (SOUPS)*, 2014.

[23] Aleecia M. McDonald and Lorrie Faith Cranor. The Cost of Reading Privacy Policies. *I/S Journal of Law and Policy for the Information Society (ISJLP)*, 4, 2008.

[24] Thomas B. Norton. Crowdsourcing Privacy Policy Interpretation. In *Proceedings of the Research Conference on Communications, Information, and Internet Policy (TPRC)*, 2015.

[25] Ashwini Rao, Florian Schaub, Norman Sadeh, Alessandro Acquisti, and Ruogu Kang. Expecting the Unexpected: Understanding Mismatched Privacy Expectations Online. In *Proceedings of the Symposium on Usable Privacy and Security (SOUPS)*, 2016.

[26] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D. Breaux, and Jianwei Niu. Toward a Framework for Detecting Privacy Policy Violations in Android Application Code. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2016.

[27] John W. Stamey and Ryan A. Rossi. Automatically Identifying Relations in Privacy Policies. In *Proceedings of the ACM International Conference on Design of Communication (SIGDOC)*, 2009.

[28] Xiaoyin Wang, Xue Qin, Mitra Bokaei Hosseini, Rocky Slavin, Travis D. Breaux, and Jianwei Niu. GUILeak: Tracing Privacy Policy Claims on User Input Data for Android Applications. In *Proceedings of the International Conference of Software Engineering (ICSE)*, 2018.

[29] Le Yu, Xiapu Luo, Xule Liu, and Tao Zhang. Can We Trust the Privacy Policies of Android Apps? In *Proceedings of the IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, 2016.

[30] Razieh Nokhbeh Zaeem, Rachel L. German, and K. Suzanne Barber. PrivacyCheck: Automatic Summarization of Privacy Policies Using Data Mining. *ACM Transactions on Internet Technology (TOIT)*, 2013.

[31] Sebastian Zimmeck and Steven M. Bellovin. Privee: An Architecture for Automatically Analyzing Web Privacy Policies. In *Proceedings of the USENIX Security Symposium*, 2014.

[32] Sebastian Zimmeck, Ziqi Wang, Lieyong Zou, Roger Iyengar, Bin Liu, Florian Schaub, Shomir Wilson, Norman Sadeh, Steven M. Bellovin, and Joel Reidenberg. Automated Analysis of Privacy Requirements for Mobile Apps. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2017.

A Preprocessing Privacy Policies

Privacy policies are commonly made available via a link on Google Play to the developer's website and hosted in HTML. Most NLP parsers expect plaintext input; therefore, PolicyLint begins by converting the HTML privacy policy into plaintext. We next describe how PolicyLint achieves this conversion.

Removing Non-relevant and Non-displayed Text: Privacy policies are frequently embedded as main content on a webpage containing navigational elements and other non-relevant text. Additionally, non-displayed text should also be stripped from the HTML, as we want to analyze what is actually displayed to users. PolicyLint extracts the privacy policy portion of the webpage by iterating over the elements in the HTML document. To remove non-relevant text, PolicyLint strips comment, `style`, `script`, `nav`, and `video` HTML tags. PolicyLint also strips HTML links containing phrases commonly used for page navigation (e.g., "learn more," "back to top," "return to top"). Finally, PolicyLint removes HTML `span` and `div` tags using the "`display:none`" style attribute.

Converting HTML to Flat Plaintext Documents: Certain HTML elements, such as pop-up items, result in a non-flat structure. When flattening the HTML documents, PolicyLint must ensure that the plaintext document has a formatting style similar to the text displayed on the webpage (e.g., same paragraph and sentence breaks). PolicyLint handles pop-up elements by relocating the text within the pop-up element to the end of the document. Pop-up elements often provide additional context, explanation, clarification, or a definition of a term. Therefore, relocating these elements should not have a significant effect on processing the referencing paragraph. To ensure that the formatting style is maintained, PolicyLint converts the HTML document to markdown using `html2text`.

Merging Formatted Lists: Formatted lists within text can cause NLP parsers to incorrectly detect sentence breaks or incorrectly tag parts-of-speech and typed dependencies. These parsing errors can negatively impact the semantic reasoning of sentences. Therefore, PolicyLint merges the text within list items with the preceding clauses before the list begins. PolicyLint also uses a set of heuristics for nesting list structures to ensure that list items propagate to the correct clause.

PolicyLint merges formatted lists in two phases. The first phase occurs before the aforementioned conversion to markdown. In this phase, PolicyLint iterates over HTML elements using list-related HTML tags (i.e., `ol`, `ul`, `li`) to annotate list structures and nesting depth. The second phase occurs after the conversion to markdown. In this phase, PolicyLint searches for paragraphs ending in a colon where the next sentence is a list item (e.g., starts with bullets, roman numerals, formatted numbers, or contains annotations from the first phase). It then forms complete sentences by merging the list item text with the preceding text.

PolicyLint iterates over the paragraphs in the markdown document to find those that end in a colon. For each paragraph that ends in a colon, PolicyLint checks whether the proceeding paragraph is a list item. If the line of text is a list item, PolicyLint creates a new paragraph by appending the list item text to the preceding text that ends with the colon. If the list item ends in another colon, PolicyLint repeats the same preceding process but by prepending the nested list items to the new paragraph created in the last step. PolicyLint then leverages the symbols that denote list items to predict the next list item's expected symbol, which is useful for detecting boundaries of nested lists. For example, if the current list item starts with "(1)," then we would expect the next list item to start with "(2)." If the item symbol matches the expected symbol, PolicyLint merges the list item text as discussed above and continues this process. If the item symbol does not match the expected symbol, PolicyLint stops this process and returns.

Final Processing: The final step converts markdown to plaintext. PolicyLint normalizes Unicode characters and strips markdown formatting, such as header tags, bullet points, list item numbering, and other format characters. PolicyLint then uses `langid` to determine whether the majority of the document is written in English. If not, PolicyLint discards the document. If so, PolicyLint outputs the plaintext document.

B Training Sentence Generation

PolicyLint requires a training set of sharing and collection sentences to learn underlying patterns from in order to iden-

Table 8: Sentence Generation Templates

1	*ENT* may *VERB_PRESENT DATA*	We may share your personal information.
2	We may *VERB_PRESENT DATA PREP ENT*	We may share your personal information with advertisers.
3	We may *VERB_PRESENT ENT DATA*	We may send advertisers your personal information.
4	We may *VERB_PRESENT PREP ENT DATA*	We may share with advertisers your personal information.
5	*DATA* may be *VERB_PAST PREP ENT*	Personal information may be shared with advertisers.
6	*DATA* may be *VERB_PAST*	Personal information may be shared.
7	*DATA* may be *VERB_PAST* by *ENT*	Personal information may be shared by advertisers.
8	We may choose to *VERB_PRESENT DATA*	We may choose to share personal information.
9	We may choose to *VERB_PRESENT DATA PREP ENT*	We may choose to share personal information with advertisers.
10	You may be required by us to *VERB_PRESENT DATA*	You may be required by us to share personal information.
11	You may be required by us to *VERB_PRESENT DATA PREP ENT*	You may be required by us to share personal information with advertisers.
12	We are requiring you to *VERB_PRESENT DATA*	We are requiring you to share personal information.
13	We are requiring you to *VERB_PRESENT DATA PREP ENT*	We are requiring you to share personal information with advertisers.
14	We require *VERB_PRESENT_PARTIC DATA*	We require sharing personal information.
15	We require *VERB_PRESENT_PARTIC DATA PREP ENT*	We require sharing personal information with advertisers.
16	We may *VERB_PRESENT ENT* with *DATA*	We may provide advertisers with your personal information.

tify "unseen" sharing and collection sentences. As it is tedious to manually select a set of sharing and collection sentences with diverse grammatical structures, we opt to auto-generate the training sentences instead. Note that auto-generating sentences does not adversely impact the extensibility of PolicyLint, as adding a new pattern is as simple as feeding PolicyLint a new sentence for reflecting this new pattern. To identify the templates, we use our domain expertise to identify different sentence compositions that could describe sharing and collection sentences. We identify 16 sentence templates, as shown in Table 8.

To fill the templates, we substitute an entity (*ENT*), data object (*DATA*), the correct tense of an SoC verb (*VERB_PRESENT*, *VERB_PAST*, *VERB_PRESENT_PARTICIPLE*), and a preposition that describes with *whom* the sharing occurs for sharing verbs (*PREP*). We begin by identifying the present tense, past tense, and present participle forms of all of the SoC verbs (e.g., "share," "shared," "sharing," respectively). We then identify common prepositions for each of the sharing verbs that describe with *whom* the sharing occurs. For example, for the terms *share, trade, and exchange*, the preposition is "with" and for the terms *sell, transfer, distribute, disclose, rent, report, transmit, send, give, provide*, the preposition is "to."

For each template, we fill the placeholders accordingly. If the template has a placeholder for prepositions (*PREP*) and the verb is a collect verb, we skip the template. We also skip the templates for "send," "give," and "provide" if they do not contain a placeholder for a preposition (T1, T6, T8, T10, T12, T14), as those template sentences do not make sense without specifying to *whom* the data is being sent/given/provided. We set *DATA* to the phrases "your personal information" and "your personal information, demographic information, and financial information." Similarly, we set *ENT* to the phrases "advertiser" and "advertisers, analytics providers, and our business partners." Note that we include conjuncts of the *DATA* and *ENT* placeholders to account for deviations in the parse tree due to syntactic ambiguity (i.e., a sentence can have multiple interpretations). Therefore, we generate two sentences for each template: one with a singular *DATA* and *ENT*, and second with the plural *DATA* and plural *ENT*. In total, we generate 560 sentences, which are used by PolicyLint to learn patterns to identify sharing and collection sentences.

C Policy Statement Extraction

For the 9% of the policies from which PolicyLint does not extract policy statements, we randomly select 100 policies to explore why this situation occurs. We find that 88% (88/100) are due to an insufficient crawling strategy (i.e., privacy policy links being pointed at home pages, navigation pages, or 404 error pages). Among the remaining 12 policies, the links in 3 policies point at a PDF privacy policy while PolicyLint handles only HTML, and 2 policies are not written in English but are not caught by our language classifier. Finally, 7 policies are due to errors in extracting policy statements, such as incomplete verb lists (3 cases), tabular format policies (1 case), and policies for describing permission uses (3 cases).

D Email Template

Subject: Contradictory Privacy Policy Statements in <APP_NAME>
To Whom It May Concern:
We are a team of security researchers from the WSPR Lab in the Department of Computer Science at North Carolina State University. We created a tool to analyze privacy policies and found that the privacy policy for your "<APP_NAME>" application (<PACKAGE>) that we downloaded in September 2017 may contain the following potential contradictory statements:
(1) The following statements claim that personal information is both collected and not collected.
(A) <CONTRADICTORY_SENTENCE_1>
(B) <CONTRADICTORY_SENTENCE_2>
...
Do you believe that these are contradictory statements? Any additional information that you may have to clarify this policy would be extremely helpful. If you have any questions or concerns, please feel free to contact us, preferably by February 11th.
Thank you for your time!
Best Regards,

<EMAIL_SIGNATURE>

Figure 6: Email Template

50 Ways to Leak Your Data: An Exploration of Apps' Circumvention of the Android Permissions System

Joel Reardon
University of Calgary
AppCensus, Inc.

Álvaro Feal
IMDEA Networks Institute
Universidad Carlos III de Madrid

Primal Wijesekera
U.C. Berkeley / ICSI

Amit Elazari Bar On
U.C. Berkeley

Narseo Vallina-Rodriguez
IMDEA Networks Institute / ICSI
AppCensus, Inc.

Serge Egelman
U.C. Berkeley / ICSI
AppCensus, Inc.

Abstract

Modern smartphone platforms implement permission-based models to protect access to sensitive data and system resources. However, apps can circumvent the permission model and gain access to protected data without user consent by using both covert and side channels. Side channels present in the implementation of the permission system allow apps to access protected data and system resources without permission; whereas covert channels enable communication between two colluding apps so that one app can share its permission-protected data with another app lacking those permissions. Both pose threats to user privacy.

In this work, we make use of our infrastructure that runs hundreds of thousands of apps in an instrumented environment. This testing environment includes mechanisms to monitor apps' runtime behaviour and network traffic. We look for evidence of side and covert channels being used in practice by searching for sensitive data being sent over the network for which the sending app did not have permissions to access it. We then reverse engineer the apps and third-party libraries responsible for this behaviour to determine how the unauthorized access occurred. We also use software fingerprinting methods to measure the static prevalence of the technique that we discover among other apps in our corpus.

Using this testing environment and method, we uncovered a number of side and covert channels in active use by hundreds of popular apps and third-party SDKs to obtain unauthorized access to both unique identifiers as well as geolocation data. We have responsibly disclosed our findings to Google and have received a bug bounty for our work.

1 Introduction

Smartphones are used as general-purpose computers and therefore have access to a great deal of sensitive system resources (*e.g.*, sensors such as the camera, microphone, or GPS), private data from the end user (*e.g.*, user email or contacts list), and various persistent identifiers (*e.g.*, IMEI). It is crucial to protect this information from unauthorized access. Android, the most-popular mobile phone operating system [75], implements a permission-based system to regulate access to these sensitive resources by third-party applications. In this model, app developers must explicitly request *permission* to access sensitive resources in their Android Manifest file [5]. This model is supposed to give users control in deciding which apps can access which resources and information; in practice it does not address the issue completely [30, 86].

The Android operating system sandboxes user-space apps to prevent them from interacting arbitrarily with other running apps. Android implements isolation by assigning each app a separate *user ID* and further mandatory access controls are implemented using SELinux. Each running process of an app can be either code from the app itself or from SDK libraries embedded within the app; these SDKs can come from Android (*e.g.*, official Android support libraries) or from third-party providers. App developers integrate third-party libraries in their software for things like crash reporting, development support, analytics services, social-network integration, and advertising [16, 62]. By design, any third-party service bundled in an Android app inherits access to all permission-protected resources that the user grants to the app. In other words, if an app can access the user's location, then all third-party services embedded in that app can as well.

In practice, security mechanisms can often be circumvented; *side channels* and *covert channels* are two common techniques to circumvent a security mechanism. These channels occur when there is an alternate means to access the protected resource that is not audited by the security mechanism, thus leaving the resource unprotected. A *side channel* exposes a path to a resource that is outside the security mechanism; this can be because of a flaw in the design of the security mechanism or a flaw in the implementation of the design. A classic example of a side channel is that power usage of hardware when performing cryptographic operations can leak the particulars of a secret key [42]. As an example in the physical world, the frequency of pizza deliveries to government buildings may leak information about political crises [69].

A *covert channel* is a more deliberate and intentional effort between two cooperating entities so that one with access to some data provides it to the other entity without access to the data in violation of the security mechanism [43]. As an example, someone could execute an algorithm that alternates between high and low CPU load to pass a binary message to another party observing the CPU load.

The research community has previously explored the potential for covert channels in Android using local sockets and shared storage [49], as well as other unorthodox means, such as vibrations and accelerometer data to send and receive data between two coordinated apps [3]. Examples of side channels include using device sensors to infer the gender of the user [51] or uniquely identify the user [72]. More recently, researchers demonstrated a new permission-less device fingerprinting technique that allows tracking Android and iOS devices across the Internet by using factory-set sensor calibration details [90]. However, there has been little research in detecting and measuring at scale the prevalence of covert and side channels in apps that are available in the Google Play Store. Only isolated instances of malicious apps or libraries inferring users' locations from WiFi access points were reported, a side channel that was abused in practice and resulted in about a million dollar fine by regulators [82].

In fact, most of the existing literature is focused on understanding personal data collection using the system-supported access control mechanisms (*i.e.*, Android permissions). With increased regulatory attention to data privacy and issues surrounding user consent, we believe it is imperative to understand the effectiveness (and limitations) of the permission system and whether it is being circumvented as a preliminary step towards implementing effective defenses.

To this end, we extend the state of the art by developing methods to detect actual circumvention of the Android permission system, at scale in real apps by using a combination of dynamic and static analysis. We automatically executed over 88,000 Android apps in a heavily instrumented environment with capabilities to monitor apps' behaviours at the system and network level, including a TLS man-in-the-middle proxy. In short, we ran apps to see when permission-protected data was transmitted by the device, and scanned the apps to see which ones *should not* have been able to access the transmitted data due to a lack of granted permissions. We grouped our findings by *where* on the Internet *what* data type was sent, as this allows us to attribute the observations to the actual app developer or embedded third-party libraries. We then reverse engineered the responsible component to determine exactly how the data was accessed. Finally, we statically analyzed our entire dataset to measure the prevalence of the channel. We focus on a subset of the *dangerous* permissions that prevent apps from accessing location data and identifiers. Instead of imagining new channels, our work focuses on tracing evidence that suggests that side- and covert-channel abuse is occurring in practice.

We studied more than 88,000 apps across each category from the U.S. Google Play Store. We found a number of side and covert channels in active use, responsibly disclosed our findings to Google and the U.S. Federal Trade Commission (FTC), and received a bug bounty for our efforts.

In summary, the contributions of this work include:

- We designed a pipeline for automatically discovering vulnerabilities in the Android permissions system through a combination of dynamic and static analysis, in effect creating a scalable honeypot environment.
- We tested our pipeline on more than 88,000 apps and discovered a number of vulnerabilities, which we responsibly disclosed. These apps were downloaded from the U.S. Google Play Store and include popular apps from all categories. We further describe the vulnerabilities in detail, and measure the degree to which they are in active use, and thus pose a threat to users. We discovered covert and side channels used in the wild that compromise both users' location data and persistent identifiers.
- We discovered companies getting the MAC addresses of the connected WiFi base stations from the ARP cache. This can be used as a surrogate for location data. We found 5 apps exploiting this vulnerability and 5 with the pertinent code to do so.
- We discovered Unity obtaining the device MAC address using `ioctl` system calls. The MAC address can be used to uniquely identify the device. We found 42 apps exploiting this vulnerability and 12,408 apps with the pertinent code to do so.
- We also discovered that third-party libraries provided by two Chinese companies—Baidu and Salmonads—independently make use of the SD card as a covert channel, so that when an app can read the phone's IMEI, it stores it for other apps that cannot. We found 159 apps with the potential to exploit this covert channel and empirically found 13 apps doing so.
- We found one app that used picture metadata as a side channel to access precise location information despite not holding location permissions.

These deceptive practices allow developers to access users' private data without consent, undermining user privacy and giving rise to both legal and ethical concerns. Data protection legislation around the world—including the General Data Protection Regulation (GDPR) in Europe, the California Consumer Privacy Act (CCPA) and consumer protection laws, such as the Federal Trade Commission Act—enforce transparency on the data collection, processing, and sharing practices of mobile applications.

This paper is organized as follows: Section 2 gives more background information on the concepts discussed in the introduction. Section 3 describes our system to discover vulnerabilities in detail. Section 4 provides the results from our

study, including the side and covert channels we discovered and their prevalence in practice. Section 5 describes related work. Section 6 discusses their potential legal implications. Section 7 discusses limitations to our approach and concludes with future work.

2 Background

The Android permissions system has evolved over the years from an ask-on-install approach to an ask-on-first-use approach. While this change impacts when permissions are granted and how users can use contextual information to reason about the appropriateness of a permission request, the backend enforcement mechanisms have remained largely unchanged. We look at how the design and implementation of the permission model has been exploited by apps to bypass these protections.

2.1 Android Permissions

Android's permissions system is based on the security principle of *least privilege*. That is, an entity should only have the minimum capabilities it needs to perform its task. This standard design principle for security implies that if an app acts maliciously, the damage will be limited. Developers must declare the permissions that their apps need beforehand, and the user is given an opportunity to review them and decide whether to install the app. The Android platform, however, does not judge whether the set of requested permissions are all strictly necessary for the app to function. Developers are free to request more permissions than they actually need and users are expected to judge if they are reasonable.

The Android permission model has two important aspects: obtaining user consent before an app is able to access any of its requested permission-protected resources, and then ensuring that the app cannot access resources for which the user has not granted consent. There is a long line of work uncovering issues on how the permission model interacts with the user: users are inadequately informed about why apps need permissions at installation time, users misunderstand exactly what the purpose of different permissions are, and users lack context and transparency into how apps will ultimately use their granted permissions [24, 30, 78, 86]. While all of these are critical issues that need attention, the focus of our work is to understand how apps are circumventing system checks to verify that apps have been granted various permissions.

When an app requests a permission-protected resource, the resource manager (*e.g.*, `LocationManager`, `WiFiManager`, etc.) contacts the `ActivityServiceManager`, which is the *reference monitor* in Android. The resource request originates from the sandboxed app, and the final verification happens inside the Android platform code. The platform is a Java operating system that runs in system space and acts as an interface for a customized Linux kernel, though apps can interact with the kernel directly as well. For some permission-protected resources, such as network sockets, the reference monitor is the kernel, and the request for such resources bypasses the platform framework and directly contacts the kernel. Our work discusses how real-world apps circumvent these system checks placed in the kernel and the platform layers.

The Android permissions system serves an important purpose: to protect users' privacy and sensitive system resources from deceptive, malicious, and abusive actors. At the very least, if a user denies an app a permission, then that app should not be able to access data protected by that permission [24, 81]. In practice, this is not always the case.

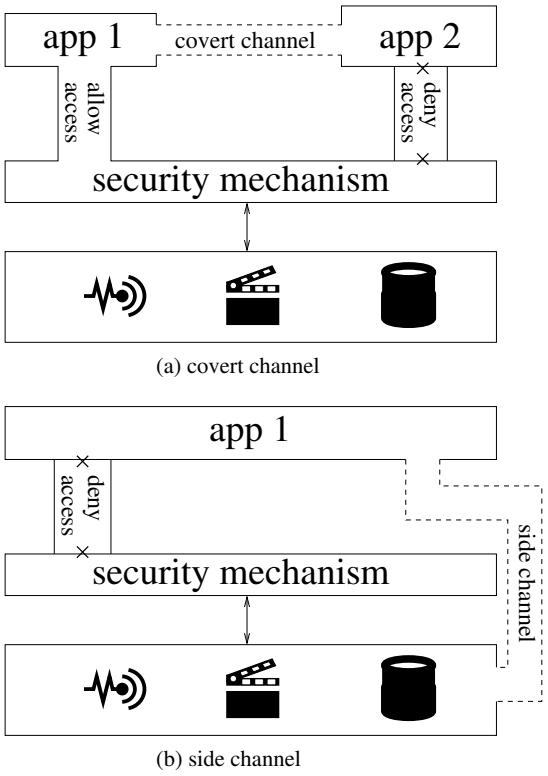
2.2 Circumvention

Apps can circumvent the Android permission model in different ways [3, 17, 49, 51, 52, 54, 70, 72, 74]. The use of covert and side channels, however, is particularly troublesome as their usage indicates deceptive practices that might mislead even diligent users, while underscoring a security vulnerability in the operating system. In fact, the United State's Federal Trade Commission (FTC) has fined mobile developers and third-party libraries for exploiting side channels: using the MAC address of the WiFi access point to infer the user's location [82]. Figure 1 illustrates the difference between covert and side channels and shows how an app that is denied permission by a security mechanism is able to still access that information.

Covert Channel A covert channel is a communication path between two parties (*e.g.*, two mobile apps) that allows them to transfer information that the relevant security enforcement mechanism deems the recipient unauthorized to receive [18]. For example, imagine that AliceApp has been granted permission through the Android API to access the phone's IMEI (a persistent identifier), but BobApp has been denied access to that same data. A covert channel is created when AliceApp legitimately reads the IMEI and then gives it to BobApp, even though BobApp has already been denied access to this same data when requesting it through the proper permission-protected Android APIs.

In the case of Android, different covert channels have been proposed to enable communication between apps. This includes exotic mediums such as ultrasonic audio beacons and vibrations [17, 26]. Apps can also communicate using an external network server to exchange information when no other opportunity exists. Our work, however, exposes that rudimentary covert channels, such as shared storage, are being used in practice at scale.

Side Channel A side channel is a communication path that allows a party to obtain privileged information without relevant permission checks occurring. This can be due to non-conventional unprivileged functions or features, as well as ersatz versions of the same information being available without



(a) covert channel

(b) side channel

Figure 1: Covert and side channels. (a) A security mechanism allows `app1` access to resources but denies `app2` access; this is circumvented by `app2` using `app1` as a facade to obtain access over a communication channel not monitored by the security mechanism. (b) A security mechanism denies `app1` access to resources; this is circumvented by accessing the resources through a side channel that bypasses the security mechanism.

being protected by the same permission. A classical example of a side channel attack is the timing attack to exfiltrate an encryption key from secure storage [42]. The system under attack is an algorithm that performs computation with the key and unintentionally leaks timing information—*i.e.*, how long it runs—that reveals critical information about the key.

Side channels are typically an unintentional consequence of a complicated system. ("Backdoors" are intentionally-created side channels that are meant to be obscure.) In Android, a large and complicated API results in the same data appearing in different locations, each governed by different access control mechanisms. When one API is protected with permissions, another unprotected method may be used to obtain the same data or an ersatz version of it.

2.3 App Analysis Methods

Researchers use two primary techniques to analyze app behaviour: static and dynamic analysis. In short, static analysis studies *software as data* by reading it; dynamic analysis studies *software as code* by running it. Both approaches have the goal of understanding the software's ultimate behaviour, but they offer insights with different certainty and granularity: static analysis reports instances of hypothetical behaviour; dynamic analysis gives reports of observed behaviour.

Static Analysis Static analysis involves scanning the code for all possible combinations of execution flows to understand potential execution behaviours—the behaviours of interest may include various privacy violations (*e.g.*, access to sensitive user data). Several studies have used static analysis to analyze different types of software in search of malicious behaviours and privacy leaks [4, 9–11, 19–22, 32, 37, 39, 41, 45, 92]. However, static analysis does not produce actual observations of privacy violations; it can only suggest that a violation may happen if a given part of the code gets executed at runtime. This means that static analysis provides an upper bound on hypothetical behaviours (*i.e.*, yielding false positives).

The biggest advantage of static analysis is that it is easy to perform automatically and at scale. Developers, however, have options to evade detection by static analysis because a program's runtime behaviour can differ enormously from its superficial appearance. For example, they can use code obfuscation [23, 29, 48] or alter the flow of the program to hide the way that the software operates in reality [23, 29, 48]. Native code in unmanaged languages allow pointer arithmetic that can skip over parts of functions that guarantee pre-conditions. Java's reflection feature allows the execution of dynamically created instructions and dynamically loaded code that similarly evades static analysis. Recent studies have shown that around 30% of apps render code dynamically [46], so static analysis may be insufficient in those cases.

From an app analysis perspective, static analysis lacks the contextual aspect, *i.e.*, it fails to observe the circumstances surrounding each observation of sensitive resource access and sharing, which is important in understanding when a given privacy violation is likely to happen. For these reasons, static analysis is useful, but is well complemented by dynamic analysis to augment or confirm findings.

Dynamic analysis Dynamic analysis studies an executable by running it and auditing its runtime behaviour. Typically, dynamic analysis benefits from running the executable in a controlled environment, such as an instrumented mobile OS [27, 85], to gain observations of an app's behaviour [16, 32, 46, 47, 50, 65, 66, 73, 85, 87–89].

There are several methods that can be used in dynamic analysis, one example is taint analysis [27, 32] which can be inefficient and prone to control flow attacks [68, 71]. A challenge to performing dynamic analysis is the logistical burden of performing it at scale. Analyzing a single Android app in isolation is straightforward, but scaling it to run automatically for tens of thousands of apps is not. Scaling dynamic analysis is facilitated with automated execution and creation of behavioural reports. This means that effective dynamic analysis

requires building an instrumentation framework for possible behaviours of interest *a priori* and then engineering a system to manage the endeavor.

Nevertheless, some apps are resistant to being audited when run in virtual or privileged environments [12, 68]. This has led to new auditing techniques that involve app execution on real phones, such as by forwarding traffic through a VPN in order to inspect network communications [44, 60, 63]. The limitations of this approach are the use of techniques robust to man-in-the-middle attacks [28, 31, 61] and scalability due to the need to actually run apps with user input.

A tool to automatically execute apps on the Android platform is the UI/Application Exerciser Monkey [6]. The Monkey is a UI fuzzer that generates synthetic user input, ensuring that some interaction occurs with the app being automatically tested. The Monkey has no context for its actions with the UI, however, so some important code paths may not be executed due to the random nature of its interactions with the app. As a result, this gives a lower bound for possible app behaviours, but unlike static analysis, it does not yield false positives.

Hybrid Analysis Static and dynamic analysis methods complement each other. In fact, some types of analysis benefit from a hybrid approach, in which combining both methods can increase the coverage, scalability, or visibility of the analyses. This is the case for malicious or deceptive apps that actively try to defeat one individual method (*e.g.*, by using obfuscation or techniques to detect virtualized environments or TLS interception). One approach would be to first carry out dynamic analysis to triage potential suspicious cases, based on collected observations, to be later examined thoroughly using static analysis. Another approach is to first carry out static analysis to identify interesting code branches that can then be instrumented for dynamic analysis to confirm the findings.

3 Testing Environment and Analysis Pipeline

Our instrumentation and processing pipeline, depicted and described in Figure 2, combines the advantages of both static and dynamic analysis techniques to triage suspicious apps and analyze their behaviours in depth. We used this testing environment to find evidence of covert- and side-channel usage in 252,864 versions of 88,113 different Android apps, all of them downloaded from the U.S. Google Play Store using a purpose-built Google Play scraper. We executed each app version individually on a physical mobile phone equipped with a customized operating system and network monitor. This testbed allows us to observe apps' runtime behaviours both at the OS and network levels. We can observe how apps request and access sensitive resources and their data sharing practices. We also have a comprehensive data analysis tool to de-obfuscate collected network data to uncover potential deceptive practices.

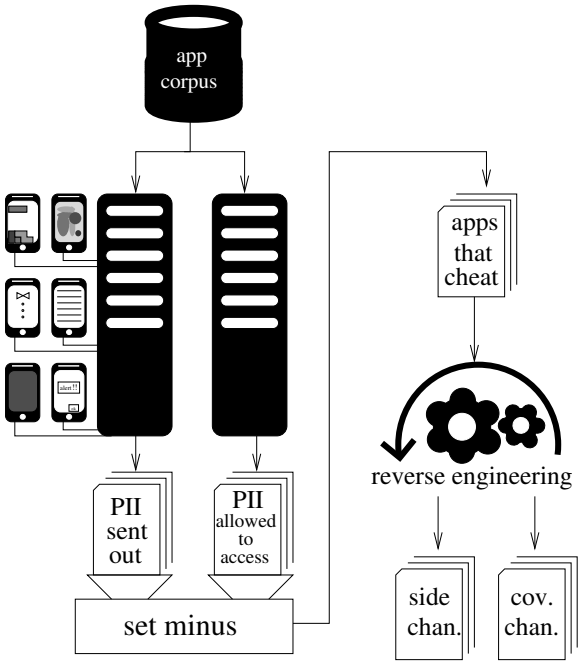


Figure 2: Overview of our analysis pipeline. Apps are automatically run and the transmissions of sensitive data are compared to what would be allowed. Those suspected of using a side or covert channel are manually reverse engineered.

Before running each app, we gather the permission-protected identifiers and data. We then execute each app while collecting all of its network traffic. We apply a suite of decodings to the traffic flows and search for the permission-protected data in the decoded traffic. We record all transmissions and later filter for those containing permission-protected data sent by apps not holding the requisite permissions. We hypothesize that these are due to the use of side and covert channels; that is, we are not looking for these channels, but rather looking for evidence of their use (*i.e.*, transmissions of protected data). Then, we group the suspect transmissions by the data type sent and the destination where it was sent, because we found that the same data-destination pair reflects the same underlying side or covert channel. We take one example per group and manually reverse engineer it to determine how the app gained permission-protected information without the corresponding permission.

Finally, we fingerprint the apps and libraries found using covert- and side-channels to identify the static presence of the same code in other apps in our corpus. A fingerprint is any string constant, such as specific filename or error message, that can be used to statically analyze our corpus to determine if the same technique exists in other apps that did not get triggered during our dynamic analysis phase.

3.1 App Collection

We wrote a Google Play Store scraper to download the most-popular apps under each category. Because the popularity distribution of apps is long tailed, our analysis of the 88,113 most-popular apps is likely to cover most of the apps that people currently use. This includes 1,505 non-free apps we purchased for another study [38]. We instrumented the scraper to inspect the Google Play Store to obtain application executables (APK files) and their associated metadata (*e.g.*, number of installs, category, developer information, etc.).

As developers tend to update their Android software to add new functionality or to patch bugs [64], these updates can also be used to introduce new side and covert channels. Therefore, it is important to examine different versions of the same app, because they may exhibit different behaviours. In order to do so, our scraper periodically checks if a new version of an already downloaded app is available and downloads it. This process allowed us to create a dataset consisting of 252,864 different versions of 88,113 Android apps.

3.2 Dynamic Analysis Environment

We implemented the dynamic testing environment described in Figure 2, which consists of about a dozen Nexus 5X Android phones running an instrumented version of the Android Marshmallow platform.[1] This purpose-built environment allows us to comprehensively monitor the behaviour of each of 88,113 Android apps at the kernel, Android-framework, and network traffic levels. We execute each app automatically using the Android Automator Monkey [6] to achieve scale by eliminating any human intervention. We store the resulting OS-execution logs and network traffic in a database for offline analysis, which we discuss in Section 3.3. The dynamic analysis is done by extending a platform that we have used in previous work [66].

Platform-Level Instrumentation We built an instrumented version of the Android 6.0.1 platform (Marshmallow). The instrumentation monitored resource accesses and logged when apps were installed and executed. We ran apps one at a time and uninstalled them afterwards. Regardless of the obfuscation techniques apps use to disrupt static analysis, no app can avoid our instrumentation, since it executes in the system space of the Android framework. In a sense, our environment is a honeypot allowing apps to execute as their true selves. For the purposes of preparing our bug reports to Google for responsible disclosure of our findings, we retested our findings on a stock Pixel 2 running Android Pie—the most-recent version at the time—to demonstrate that they were still valid.

[1] While as of this writing Android Pie is the current release [35], Marshmallow and older versions were used by a majority of users at the time that we began data collection.

Kernel-Level Instrumentation We built and integrated a custom Linux kernel into our testing environment to record apps' access to the file system. This module allowed us to record every time an app opened a file for reading or writing or unlinked a file. Because we instrumented the system calls to open files, our instrumentation logged both regular files and special files, such as device and interface files, and the `proc/` filesystem, as a result of the *"everything is a file"* UNIX philosophy. We also logged whenever an `ioctl` was issued to the file system. Some of the side channels for bypassing permission checking in the Android platform may involve directly accessing the kernel, and so kernel-level instrumentation provides clear evidence of these being used in practice.

We ignored the special device file `/dev/ashmem` (Android-specific implementation of asynchronous shared memory for inter-process communication) because it overwhelmed the logs due to its frequent use. As Android assigns a separate *user* (*i.e.*, uid) to each app, we could accurately attribute the access to such files to the responsible app.

Network-Level Monitoring We monitored all network traffic, including TLS-secured flows, using a network monitoring tool developed for our previous research activities [63]. This network monitoring module leverages Android's VPN API to redirect all the device's network traffic through a localhost service that inspects all network traffic, regardless of the protocol used, through deep-packet inspection and in user-space. It reconstructs the network streams and ascribes them to the originating app by mapping the app owning the socket to the UID as reported by the `proc` filesystem. Furthermore, it also performs TLS interception by installing a root certificate in the system trusted certificate store. This technique allows it to decrypt TLS traffic unless the app performs advanced techniques, such as certificate pinning, which can be identified by monitoring TLS records and proxy exceptions [61].

Automatic App Execution Since our analysis framework is based on dynamic analysis, apps must be executed so that our instrumentation can monitor their behaviours. In order to scale to hundreds of thousands of apps tested, we cannot rely on real user interaction with each app being tested. As such, we use Android's UI/Application Exerciser Monkey, a tool provided by Android's development SDK to automate and parallelize the execution of apps by simulating user inputs (*i.e.*, taps, swipes, etc.).

The Monkey injects a pseudo-random stream of simulated user input events into the app, *i.e.*, it is a UI fuzzer. We use the Monkey to interact with each version of each app for a period of ten minutes, during which the aforementioned tools log the app's execution as a result of the random UI events generated by the Monkey. Apps are rerun if the operation fails during execution. Each version of each app is run once in this manner; our system also reruns apps if there is unused capacity.

After running the app, the kernel, platform, and network logs are collected. The app is then uninstalled along with any other app that may have been installed through the process of automatic exploration. We do this with a white list of allowed apps; all other apps are uninstalled. The logs are then cleared and the device is ready to be used for the next test.

3.3 Personal Information in Network Flows

Detecting whether an app has legitimately accessed a given resource is straightforward: we compare its runtime behaviour with the permissions it had requested. Both users and researchers assess apps' privacy risks by examining their requested permissions. This presents an incomplete picture, however, because it only indicates what data an app *might* access, and says nothing about with whom it may share it and under what circumstances. The only way of answering these questions is by inspecting the apps' network traffic. However, identifying personal information inside network transmissions requires significant effort because apps and embedded third-party SDKs often use different encodings and obfuscation techniques to transmit data. Thus, it is a significant technical challenge to be able to de-obfuscate all network traffic and search it for personal information. This subsection discusses how we tackle these challenges in detail.

Personal Information We define "personal information" as any piece of data that could potentially identify a specific individual and distinguish them from another. Online companies, such as mobile app developers and third-party advertising networks, want this type of information in order to track users across devices, websites, and apps, as this allows them to gather more insights about individual consumers and thus generate more revenue via targeted advertisements. For this reason, we are primarily interested in examining apps' access to the persistent identifiers that enable long-term tracking, as well as their geolocation information.

We focus our study on detecting apps using covert and side channels to access specific types of highly sensitive data, including persistent identifiers and geolocation information. Notably, the unauthorized collection of geolocation information in Android has been the subject of prior regulatory action [82]. Table 1 shows the different types of personal information that we look for in network transmissions, what each can be used for, the Android permission that protects it, and the subsection in this paper where we discuss findings that concern side and covert channels for accessing that type of data.

Decoding Obfuscations In our previous work [66], we found instances of apps and third-party libraries (SDKs) using obfuscation techniques to transmit personal information over the network with varying degrees of sophistication. To identify and report such cases, we automated the decoding of a standard suite of standard HTTP encodings to identify personal information encoded in network flows, such as gzip, base64, and ASCII-encoded hexadecimal. Additionally, we search for personal information directly, as well as the MD5, SHA1, and SHA256 hashes of it.

After analyzing thousands of network traces, we still find new techniques SDKs and apps use to obfuscate and encrypt network transmissions. While we acknowledge their effort to protect users' data, the same techniques could be used to hide deceptive practices. In such cases, we use a combination of reverse engineering and static analysis to understand the precise technique. We frequently found a further use of AES encryption applied to the payload before sending it over the network, often with hard-coded AES keys.

A few libraries followed best practices by generating random AES session keys to encrypt the data and then encrypt the session key with a hard-coded RSA public key, sending both the encrypted data and encrypted session key together. To de-cipher their network transmissions, we instrumented the relevant Java libraries. We found two examples of third-party SDKs "encrypting" their data by XOR-ing a keyword over the data in a Viginère-style cipher. In one case, this was *in addition* to both using standard encryption for the data *and* using TLS in transmission. Other interesting approaches included reversing the string after encoding it in base64 (which we refer to as "46esab"), using base64 multiple times (base-base6464), and using a permuted-alphabet version of base64 (sa4b6e). Each new discovery is added to our suite of decodings and our entire dataset is then re-analyzed.

3.4 Finding Side and Covert Channels

Once we have examples of transmissions that suggest the permission system was violated (*i.e.*, data transmitted by an app that had not been granted the requisite permissions to do so), we then reverse engineer the app to determine how it circumvented the permissions system. Finally, we use static analysis to measure how prevalent this practice is among the rest of our corpus.

Reverse Engineering After finding a set of apps exhibiting behaviour consistent with the existence of side and covert channels, we manually reverse engineered them. While the reverse engineering process is time consuming and not easily automated, it is necessary to determine how the app actually obtained information outside of the permission system. Because many of the transmissions are caused by the same SDK code, we only needed to reverse engineer each unique *circumvention technique*: not every app, but instead for a much smaller number of unique SDKs. The destination endpoint for the network traffic typically identifies the SDK responsible.

During the reverse engineering process, our first step was to use apktool [7] to decompile and extract the smali bytecode for each suspicious app. This allowed us to analyse and identify where any strings containing PII were created and from

Table 1: The types of personal information that we search for, the permissions protecting access to them, and the purpose for which they are generally collected. We also report the subsection in this paper where we report side and covert channels for accessing each type of data, if found, and the number of apps exploiting each. The dynamic column depicts the number of apps that we directly observed inappropriately accessing personal information, whereas the static column depicts the number of apps containing code that exploits the vulnerability (though we did not observe being executed during test runs).

Data Type	Permission	Purpose/Use	Subsection	Nº of Apps		Nº of SDKs		Channel Type	
				Dynamic	Static	Dynamic	Static	Covert	Side
IMEI	READ_PHONE_STATE	Persistent ID	4.1	13	159	2	2	2	0
Device MAC	ACCESS_NETWORK_STATE	Persistent ID	4.2	42	12,408	1	1	0	1
Email	GET_ACCOUNTS	Persistent ID	Not Found						
Phone Number	READ_PHONE_STATE	Persistent ID	Not Found						
SIM ID	READ_PHONE_STATE	Persistent ID	Not Found						
Router MAC	ACCESS_WIFI_STATE	Location Data	4.3	5	355	2	10	0	2
Router SSID	ACCESS_WIFI_STATE	Location Data	Not Found						
GPS	ACCESS_FINE_LOCATION	Location Data	4.4	1	1	0	0	0	1

which data sources. For some particular apps and libraries, our work also necessitated reverse engineering C++ code; we used IdaPro [1] for that purpose.

The typical process was to search the code for strings corresponding to destinations for the network transmissions and other aspects of the packets. This revealed where the data was already in memory, and then static analysis of the code revealed where that value first gets populated. As intentionally-obfuscated code is more complicated to reverse engineer, we also added logging statements for data and stack traces as new bytecode throughout the decompiled app, recompiled it, and ran it dynamically to get a sense of how it worked.

Measuring Prevalence The final step of our process was to determine the prevalence of the particular side or covert channel in practice. We used our reverse engineering analysis to craft a unique fingerprint that identifies the presence of an exploit in an embedded SDK, which is also robust against false positives. For example, a fingerprint is a string constant corresponding to a fixed encryption key used by one SDK, or the specific error message produced by another SDK if the operation fails.

We then decompiled all of the apps in our corpus and searched for the string in the resulting files. Within smali bytecode, we searched for the string in its entirety as a `const-string` instruction. For shared objects libraries like Unity, we use the `strings` command to output its printable strings. We include the path and name of the file as matching criteria to protect against false positives. The result is a set of all apps that may also exploit the side or covert channel in practice but for which our instrumentation did not flag for manual investigation, *e.g.*, because the app had been granted the required permission, the Monkey did not explore that particular code branch, etc.

4 Results

In this section, we present our results grouped by the type of permission that should be held to access the data; first we discuss covert and side channels enabling the access to persistent user or device IDs (particularly the IMEI and the device MAC address) and we conclude with channels used for accessing users' geolocation (*e.g.*, through network infrastructure or metadata present in multimedia content).

Our testing environment allowed us to identify five different types of side and covert channels in use among the 88,113 different Android apps in our dataset. Table 1 summarizes our findings and reports the number of apps and third-party SDKs that we find exploiting these vulnerabilities in our dynamic analysis and those in which our static analysis reveals code that *can* exploit these channels. Note that this latter category—those that *can* exploit these channels—were not seen as doing so by our instrumentation; this may be due to the Automator Monkey not triggering the code to exploit it or because the app had the required permission and therefore the transmission was not deemed suspicious.

4.1 IMEI

The International Mobile Equipment Identity (IMEI) is a numerical value that identifies mobile phones uniquely. The IMEI has many valid and legitimate operational uses to identify devices in a 3GPP network, including the detection and blockage of stolen phones.

The IMEI is also useful to online services as a persistent device identifier for tracking individual phones. The IMEI is a powerful identifier as it takes extraordinary efforts to change its value or even spoof it. In some jurisdictions, it is illegal to change the IMEI [56]. Collection of the IMEI by third parties facilitates tracking in cases where the owner tries to protect their privacy by resetting other identifiers, such as the advertising ID.

Android protects access to the phone's IMEI with the READ_PHONE_STATE permission. We identified two third-party online services that use different covert channels to access the IMEI when the app does not have the permission required to access the IMEI.

Salmonads and External Storage Salmonads is a "third party developers' assistant platform in Greater China" that offers analytics and monetization services to app developers [67]. We identified network flows to `salmonads.com` coming from five mobile apps that contained the device's IMEI, despite the fact that the apps did not have permission to access it.

We studied one of these apps and confirmed that it contained the Salmonads SDK, and then studied the workings of the SDK closer. Our analysis revealed that the SDK exploits covert channels to read this information from the following hidden file on the SD card: `/sdcard/.googlex9/.xamdecoq0962`. If not present, this file is created by the Salmonads SDK. Then, whenever the user installs another app with the Salmonads SDK embedded and with legitimate access to the IMEI, the SDK—through the host app—reads and stores the IMEI in this file.

The covert channel is the apps' shared access to the SD card. Once the file is written, all other apps with the same SDK can simply read the file instead of obtaining access through the Android API, which is regulated by the permission system. Beyond the IMEI, Salmonads also stores the advertising ID—a resettable ID for advertising and analytics purposes that allows opting out of interest-based advertising and personalization—and the phone's MAC address, which is protected with the ACCESS_NETWORK_STATE permission. We modified the file to store new random values and observed that the SDK faithfully sent them onwards to Salmonads' domains. The collection of the advertising ID alongside other non-resettable persistent identifiers and data, such as the IMEI, undermines the privacy-preserving feature of the advertising ID, which is that it can be *reset*. It also may be a violation of Google's Terms of Service [36],

Our instrumentation allowed us to observe five different apps sending the IMEI without permission to Salmonads using this technique. Static analysis of our entire app corpus revealed that six apps contained the `.xamdecoq0962` filename hardcoded in the SDK as a string. The sixth app had been granted the required permission to access the IMEI, which is why we did not initially identify it, and so it may be the app responsible for having initially written the IMEI to the file. Three of the apps were developed by the same company, according to Google Play metadata, while one of them has since been removed from Google Play. The lower bound on the number of times these apps were installed is 17.6 million, according to Google Play metadata.

Baidu and External Storage Baidu is a large Chinese corporation whose services include, among many others, an online search engine, advertising, mapping services [14], and geocoding APIs [13]. We observed network flows containing the device IMEI from Disney's Hong Kong Disneyland park app (com.disney.hongkongdisneyland_goo) to Baidu domains. This app helps tourists to navigate through the Disney-themed park, and the app makes use of Baidu's Maps SDK. While Baidu Maps initially only offered maps of mainland China, Hong Kong, Macau and Taiwan, as of 2019, it now provides global services.

Baidu's SDK uses the same technique as Salmonads to circumvent Android's permission system and access the IMEI without permission. That is, it uses a shared file on the SD card so one Baidu-containing app with the right permission can store it for other Baidu-containing apps that do not have that permission. Specifically, Baidu uses the following file to store and share this data: `/sdcard/backups/.SystemConfig/.cuid2`. The file is a base64-encoded string that, when decoded, is an AES-encrypted JSON object that contains the IMEI as well as the MD5 hash of the concatenation of "com.baidu" and the phone's Android ID.

Baidu uses AES in CBC mode with a static key and the same static value for the initialization vector (IV). These values are, in hexadecimal, `33303231323130326469637564696162`. The reason why this value is not superficially representative of a random hexadecimal string is because Baidu's key is computed from the binary representation of the ASCII string `30212102dicudiab`—observe that when reversed, it reads as `baidu cid 2012 12 03`. As with Salmonads, we confirmed that we can change the (encrypted) contents of this file and the resulting identifiers were faithfully sent onwards to Baidu's servers.

We observed eight apps sending the IMEI of the device to Baidu without holding the requisite permissions, but found 153 different apps in our repository that have hardcoded the constant string corresponding to the encryption key. This includes two from Disney: one app each for their Hong Kong and Shanghai (com.disney.shanghaidisneyland_goo) theme parks. Out of that 153, the two most popular apps were Samsung's Health (com.sec.android.app.shealth) and Samsung's Browser (com.sec.android.app.sbrowser) apps, both with more than 500 million installations. There is a lower bound of 2.6 billion installations for the apps identified as containing Baidu's SDK. Of these 153 apps, all but 20 have the READ_PHONE_STATE permission. This means that they have legitimate access to the IMEI and can be the apps that actually create the file that stores this data. The 20 that do not have the permission can only get the IMEI through this covert channel. These 20 apps have a total lower bound of 700 million installations.

4.2 Network MAC Addresses

The Media Access Control Address (MAC address) is a 6-byte identifier that is uniquely assigned to the Network Interface Controller (NIC) for establishing link-layer communications. However, the MAC address is also useful to advertisers and analytics companies as a hardware-based persistent identifier, similar to the IMEI.

Android protects access to the device's MAC address with the ACCESS_NETWORK_STATE permission. Despite this, we observed apps transmitting the device's MAC address without having permission to access it. The apps and SDKs gain access to this information using C++ native code to invoke a number of unguarded UNIX system calls.

Unity and IOCTLs Unity is a cross-platform game engine developed by Unity Technologies and heavily used by Android mobile games [77]. Our traffic analysis identified several Unity-based games sending the MD5 hash of the MAC address to Unity's servers and referring to it as a `uuid` in the transmission (*e.g.*, as an HTTP GET parameter key name). In this case, the access was happening inside of Unity's C++ native library. We reverse engineered `libunity.so` to determine how it was obtaining the MAC address.

Reversing Unity's 18 MiB compiled C++ library is more involved than Android's bytecode. Nevertheless, we were able to isolate where the data was being processed precisely because it hashes the MAC address with MD5. Unity provided its own unlabelled MD5 implementation that we found by searching for the constant numbers associated with MD5; in this case, the initial state constants.

Unity opens a network socket and uses an `ioctl` (UNIX "input-output control") to obtain the MAC address of the WiFi network interface. In effect, `ioctl`s create a large suite of "numbered" API calls that are technically no different than well-named system calls like `bind` or `close` but used for infrequently used features. The behaviour of an `ioctl` depends on the specific "request" number. Specifically, Unity uses the SIOCGIFCONF[2] `ioctl` to get the network interfaces, and then uses the SIOCGIFHWADDR[3] `ioctl` to get the corresponding MAC address.

We observed that 42 apps were obtaining and sending to Unity servers the MAC address of the network card without holding the ACCESS_NETWORK_STATE permission. To quantify the prevalence of this technique in our corpus of Android apps, we fingerprinted this behaviour through an error string that references the `ioctl` code having just failed. This allowed us to find a total of 12,408 apps containing this error string, of which 748 apps do not hold the ACCESS_NETWORK_STATE permission.

[2] Socket ioctl get interface configuration
[3] Socket ioctl get interface hardware address

4.3 Router MAC Address

Access to the WiFi router MAC address (BSSID) is protected by the ACCESS_WIFI_STATE permission. In Section 2, we exemplified side channels with router MAC addresses being ersatz location data, and discussed the FTC enacting millions of dollars in fines for those engaged in the practice of using this data to deceptively infer users' locations. Android Nougat added a requirement that apps hold an additional location permission to scan for nearby WiFi networks [34]; Android Oreo further required a location permission to get the SSID and MAC address of the connected WiFi network. Additionally, knowing the MAC address of a router allows one to link different devices that share Internet access, which may reveal personal relations by their respective owners, or enable cross-device tracking.

Our analysis revealed two side channels to access the connected WiFi router information: reading the ARP cache and asking the router directly. We found no side channels that allowed for scanning of other WiFi networks. Note that this issue affects *all* apps running on recent Android versions, not just those without the ACCESS_WIFI_STATE permission. This is because it affects apps *without* a location permission, and it affects apps *with* a location permission that the user has not granted using the ask-on-first-use controls.

Reading the ARP Table The Address Resolution Protocol (ARP) is a network protocol that allows discovering and mapping the MAC layer address associated with a given IP address. To improve network performance, the ARP protocol uses a cache that contains a historical list of ARP entries, *i.e.*, a historical list of IP addresses resolved to MAC address, including the IP address and the MAC address of the wireless router to which the device is connected (*i.e.*, its BSSID).

Reading the ARP cache is done by opening the pseudo file `/proc/net/arp` and processing its content. This file is not protected by any security mechanism, so any app can access and parse it to gain access to router-based geolocation information without holding a location permission. We built a working proof-of-concept app and tested it for Android Pie using an app that requests *no permissions*. We also demonstrated that when running an app that requests both the ACCESS_WIFI_STATE and ACCESS_COARSE_LOCATION permissions, when those permissions are denied, the app will access the data anyway. We responsibly disclosed our findings to Google in September, 2018.

We discovered this technique during dynamic analysis, when we observed one library using this method in practice: OpenX [57], a company that according to their website "creates programmatic marketplaces where premium publishers and app developers can best monetize their content by connecting with leading advertisers that value their audiences." OpenX's SDK code was not obfuscated and so we observed that they had named the responsible function

Table 2: SDKs seen sending router MAC addresses and also containing code to access the ARP cache. For reference, we report the number of apps and a lower bound of the total number of installations of those apps. We do this for all apps containing the SDK; those apps that *do not* have ACCESS_WIFI_STATE, which means that the side channel circumvents the permissions system; and those apps which *do* have a location permission, which means that the side channel circumvents location revocation.

SDK Name	Contact Domain	Incorporation Country	Total Prevalance (Apps)	Total Prevalance (Installs)	Wi-Fi Permission (Apps)	Wi-Fi Permission (Installs)	No Location Permission (Apps)	No Location Permission (Installs)
AIHelp	cs30.net	United States	30	334 million	3	210 million	12	195 million
Huq Industries	huq.io	United Kingdom	137	329 million	0	0	131	324 million
OpenX	openx.net	United States	42	1072 million	7	141 million	23	914 million
xiaomi	xiaomi.com	China	47	986 million	0	0	44	776 million
jiguang	jpush.cn	China	30	245 million	0	0	26	184 million
Peel	peel-prod.com	United States	5	306 million	0	0	4	206 million
Asurion	mysoluto.com	United States	14	2 million	0	0	14	2 million
Cheetah Mobile	cmcm.com	China	2	1001 million	0	0	2	1001 million
Mob	mob.com	China	13	97 million	0	0	6	81 million

`getDeviceMacAddressFromArp`. Furthermore, a close analysis of the code indicated that it would first *try* to get the data legitimately using the permission-protected Android API; this vulnerability is only used after the app has been explicitly denied access to this data.

OpenX did not directly send the MAC address, but rather the MD5 hash of it. Nevertheless, it is still trivial to compute a MAC address from its corresponding hash: they are vulnerable to a brute-force attack on hash functions because of the small number of MAC addresses (*i.e.*, an upper bound of 48 bits of entropy).[4] Moreover, insofar as the router's MAC address is used to resolve an app user's geolocation using a MAC-address-to-location mapping, one need only to hash the MAC addresses in this mapping (or store the hashes in the table) and match it to the received value to perform the lookup.

While OpenX was the only SDK that we observed exploiting this side channel, we searched our entire app corpus for the string `/proc/net/arp`, and found multiple third-party libraries that included it. In the case of one of them, `igexin`, there are existing reports of their predatory behaviour [15]. In our case, log files indicated that after `igexin` was denied permission to scan for WiFi, it read `/system/xbin/ip`, ran `/system/bin/ifconfig`, and then ran `cat /proc/net/arp`. Table 2 shows the prevalence of third-party libraries with code to access the ARP cache.

Router UPnP One SDK in Table 2 includes another technique to get the MAC address of the WiFi access point: it uses UPnP/SSDP discovery protocols. Three of Peel's smart remote control apps (tv.peel.samsung.app, tv.peel.smartremote, and tv.peel.mobile.app) connected to `192.168.0.1`, the IP address of the router that was their gateway to the Internet. The router in this configuration was a commodity home router that supports universal plug-and-play; the app requested the `igd.xml` (Internet gateway device configuration) file through port 1900 on the router. The router replied with, among other manufacturing details, its MAC address as part of its UUID. These apps also sent WiFi MAC addresses to their own servers and a domain hosted by Amazon Web Services.

The fact that the router is providing this information to devices hosted in the home network is not a flaw with Android *per se*. Rather it is a consequence of considering every app on every phone connected to a WiFi network to be on the trusted side of the firewall.

4.4 Geolocation

So far our analysis has showed how apps circumvent the permission system to gain access to persistent identifiers and data that can be used to infer geolocation, but we also found suspicious behaviour surrounding a more sensitive data source, *i.e.*, the actual GPS coordinates of the device.

We identified 70 different apps sending location data to 45 different domains without having any of the location permissions. Most of these location transmissions were not caused by circumvention of the permissions system, however, but rather the location data was provided within incoming packets: ad mediation services provided the location data embedded within the ad link. When we retested the apps in a different location, however, the returned location was no longer as precise, and so we suspect that these ad mediators were using IP-based geolocation, though with a much higher degree of precision than is normally expected. One app explicitly used `www.googleapis.com`'s IP-based geolocation and we found that the returned location was accurate to within a few meters; again, however, this accuracy did not replicate when we retested elsewhere [59]. We did, however, discover one genuine side channel through photo EXIF data.

Shutterfly and EXIF Metadata We observed that the Shutterfly app (com.shutterfly) sends precise geolocation data to its own server (`apcmobile.thislife.com`) without holding a location permission. Instead, it sent photo metadata

[4] Using commodity hardware, the MD5 for every possible MAC address can be calculated in a matter of minutes [40].

from the photo library, which included the phone's precise location in its exchangeable image file format (EXIF) data. The app actually processed the image file: it parsed the EXIF metadata—including location—into a JSON object with labelled `latitude` and `longitude` fields and transmitted it to their server.

While this app may not be intending to circumvent the permission system, this technique can be exploited by a malicious actor to gain access to the user's location. Whenever a new picture is taken by the user with geolocation enabled, any app with read access to the photo library (*i.e.*, READ_EXTERNAL_STORAGE) can learn the user's precise location when said picture was taken. Furthermore, it also allows obtaining historical geolocation fixes with timestamps from the user, which could later be used to infer sensitive information about that user.

5 Related Work

We build on a vast literature in the field of covert- and side-channel attacks for Android. However, while prior studies generally only reported isolated instances of such attacks or approached the problem from a theoretical angle, our work combines static and dynamic analysis to automatically detect real-world instances of misbehaviours and attacks.

Covert Channels Marforio *et al.* [49] proposed several scenarios to transmit data between two Android apps, including the use of UNIX sockets and external storage as a shared buffer. In our work we see that the shared storage is indeed used in the wild. Other studies have focused on using mobile noises [26, 70] and vibrations generated by the phone (which could be inaudible to users) as covert channels [3, 17]. Such attacks typically involve two physical devices communicating between themselves. This is outside of the scope of our work, as we focus on device vulnerabilities that are being exploited by apps and third parties running in user space.

Side Channels Spreitzer *et al.* provided a good classification of mobile-specific side-channels present in the literature [74]. Previous work has demonstrated how unprivileged Android resources could be to used to infer personal information about mobile users, including unique identifiers [72] or gender [51]. Researchers also demonstrated that it may be possible to identify users' locations by monitoring the power consumption of their phones [52] and by sensing publicly available Android resources [91]. More recently, Zhang *et al.* demonstrated a sensor calibration fingerprinting attack that uses unprotected calibration data gathered from sensors like the accelerometer, gyroscope, and magnetometer [90]. Others have shown that unprotected system-wide information is enough to infer input text in gesture-based keyboards [72]. Research papers have also reported techniques that leverage lowly protected network information to geolocate users at the network level [2, 54, 82]. We extend previous work by reporting third-party libraries and mobile applications that gain access to unique identifiers and location information in the wild by exploiting side and covert channels.

6 Discussion

Our work shows a number of side and covert channels that are being used by apps to circumvent the Android permissions system. The number of potential users impacted by these findings is in the hundreds of millions. In this section, we discuss how these issues are likely to defy users' reasonable expectations, and how these behaviours may constitute violations of various laws.

We note that these exploits may not necessarily be malicious and intentional. The Shutterfly app that extracts geolocation information from EXIF metadata may not be doing this to learn location information about the user or may not be using this data later for any purpose. On the other hand, cases where an app contains both code to access the data through the permission system and code that implements an evasion do not easily admit an innocent explanation. Even less so for those containing code to legitimately access the data and then store it for others to access. This is particularly bad because covert channels can be exploited by *any app* that knows the protocol, not just ones sharing the same SDK. The fact that Baidu writes user's IMEI to publicly accessible storage allows any app to access it without permission—not just other Baidu-containing apps.

6.1 Privacy Expectations

In the U.S., privacy practices are governed by the "notice and consent" framework: companies can give *notice* to consumers about their privacy practices (often in the form of a privacy policy), and consumers can *consent* to those practices by using the company's services. While website privacy policies are canonical examples of this framework in action, the permissions system in Android (or in any other platform) is another example of the notice and consent framework, because it fulfills two purposes: (i) providing transparency into the sensitive resources to which apps request access (notice), and (ii) requiring explicit user consent before an app can access, collect, and share sensitive resources and data (consent). That apps can and do circumvent the notice and consent framework is further evidence of the framework's failure. In practical terms, though, these app behaviours may directly lead to privacy violations because they are likely to defy consumers' expectations.

Nissenbaum's "Privacy as Contextual Integrity" framework defines privacy violations as data flows that defy contextual information norms [55]. In Nissenbaum's framework, data flows are modeled by senders, recipients, data subjects, data

types, and transmission principles in specific contexts (*e.g.*, providing app functionality, advertising, etc.). By circumventing the permissions system, apps are able to exfiltrate data to their own servers and even third parties in ways that are likely to defy users' expectations (and societal norms), particularly if it occurs after having just denied an app's explicit permission request. That is, regardless of context, were a user to explicitly be asked about granting an app access to personal information and then explicitly declining, it would be reasonable to expect that the data then would not be accessible to the app. Thus, the behaviours that we document in this paper constitute clear privacy violations. From a legal and policy perspective, these practices are likely to be considered deceptive or otherwise unlawful.

Both a recent CNIL decision (France's data protection authority), with respect to GDPR's notice and consent requirements, and various FTC cases, with respect to unfair and deceptive practices under U.S. federal law—both described in the next section—emphasize the notice function of the Android permissions system from a consumer expectations perspective. Moreover, these issues are also at the heart of a recent complaint brought by the Los Angeles County Attorney (LACA) under the California State Unfair Competition Law. The LACA complaint was brought against a popular mobile weather app on related grounds. The case further focuses on the permissions system's notice function, while noting that, "users have no reason to seek [geolocation data collection] information by combing through the app's lengthy [privacy policy], buried within which are opaque discussions of [the developer's] potential transmission of geolocation data to third parties and use for additional commercial purposes. Indeed, on information and belief, the vast majority of users do not read those sections at all" [76].

6.2 Legal and Policy Issues

The practices that we highlight in this paper also highlight several legal and policy issues. In the United States, for example, they may run afoul of the FTC's prohibitions against deceptive practices and/or state laws governing unfair business practices. In the European Union, they may constitute violations of the General Data Protection Regulation (GDPR).

The Federal Trade Commission (FTC), which is charged with protecting consumer interests, has brought a number of cases under Section 5 of the Federal Trade Commission (FTC) Act [79] in this context. The underlying complaints have stated that circumvention of Android permissions and collection of information absent users' consent or in a manner that is misleading is an unfair and deceptive act [84]. One case suggested that apps requesting permissions beyond what users expect or what are needed to operate the service were found to be "unreasonable" under the FTC Act. In another case, the FTC pursued a complaint under Section 5 alleging that a mobile device manufacturer, HTC, allowed developers to collect information without obtaining users' permission via the Android permission system, and failed to protect users from potential third-party exploitation of a related security flaw [81]. Finally, the FTC has pursued cases involving consumer misrepresentations with respect to opt-out mechanisms from tailored advertising in mobile apps more generally [83].

Also in the United States, state-level Unfair and Deceptive Acts and Practices (UDAP) statutes may also apply. These typically reflect and complement the corresponding federal law. Finally, with growing regulatory and public attention to issues pertaining to data privacy and security, data collection that undermines users' expectations and their informed consent may also be in violation of various general privacy regulations, such as the Children's Online Privacy Protection Act (COPPA) [80], the recent California Privacy Protection Act (CCPA), and potentially data breach notification laws that focus on unauthorized collection, depending on the type of personal information collected.

In Europe, these practices may be in violation of GDPR. In a recent landmark ruling, the French data regulator, CNIL, levied a 50 million Euro fine for a breach of GDPR's transparency requirements, underscoring informed consent requirements concerning data collection for personalized ads [25]. This ruling also suggests that—in the context of GDPR's consent and transparency provisions—permission requests serve a key function of both informing users of data collection practices and as a mechanism for providing informed consent [81].

Our analysis brings to light novel permission circumvention methods in actual use by otherwise legitimate Android apps. These circumventions enable the collection of information either without asking for consent or after the user has explicitly refused to provide consent, likely undermining users' expectations and potentially violating key privacy and data protection requirements on a state, federal, and even global level. By uncovering these practices and making our data public,[5] we hope to provide sufficient data and tools for regulators to bring enforcement actions, industry to identify and fix problems before releasing apps, and allow consumers to make informed decisions about the apps that they use.

7 Limitations and Future Work

During the course of performing this research, we made certain design decisions that may impact the comprehensiveness and generalizability of this work. That is, all of the findings in this paper represent lower bounds on the number of covert and side channels that may exist in the wild.

Our study considers a subset of the permissions labeled by Google as *dangerous*: those that control access to user identifiers and geolocation information. According to Android's documentation, this is indeed the most concerning and privacy intrusive set of permissions. However, there may be

[5] https://search.appcensus.io/

other permissions that, while not labeled as *dangerous*, can still give access to sensitive user data. One example is the BLUETOOTH permission; it allows apps to discover nearby Bluetooth-enabled devices, which may be useful for consumer profiling, as well as physical and cross-device tracking. Additionally, we did not examine all of the dangerous permissions, specifically data guarded by content providers, such as address book contacts and SMS messages.

Our methods rely on observations of network transmissions that suggest the existence of such channels, rather than searching for them directly through static analysis. Because many apps and third-party libraries use obfuscation techniques to disguise their transmissions, there may be transmissions that our instrumentation does not flag as containing permission-protected information. Additionally, there may be channels that are exploited, but during our testing the apps did not transmit the accessed personal data. Furthermore, apps could be exposing channels, but never abuse them during our tests. Even though we would not report such behavior, this is still an unexpected breach of Android's security model.

Many popular apps also use certificate pinning [28, 61], which results in them rejecting the custom certificate used by our man-in-the-middle proxy; our system then allows apps to continue without interference. Certificate pinning is reasonable behaviour from a security standpoint; it is possible, however, that it is being used to thwart attempts to analyse and study the network traffic of a user's mobile phone.

Our dynamic analysis uses the Android Exerciser Monkey as a UI fuzzer to generate random UI events to interact with the apps. While in our prior work we found that the Monkey explored similar code branches as a human for 60% of the apps tested [66], it is likely that it still fails to explore some code branches that may exploit covert and side channels. For example, the Monkey fails to interact with apps that require users to interact with login screens or, more generally, require specific inputs to proceed. Such apps are consequently not as comprehensively tested as apps amenable to automated exploration. Future work should compare our approaches to more sophisticated tools for automated exploration, such as Moran et al.'s Crashscope [53], which generates inputs to an app designed to trigger crash events.

Ultimately, these limitations only result in the possibility that there are side and covert channels that we have not yet discovered (*i.e.*, false negatives). It has no impact on the validity of the channels that we did uncover (*i.e.*, there are no false positives) and improvements on our methodology can only result in the discovery of more of these channels.

Moving forward, there has to be a collective effort coming from all stakeholders to prevent apps from circumventing the permissions system. Google, to their credit, have announced that they are addressing many of the issues that we reported to them [33]. However, these fixes will only be available to users able to upgrade to Android Q—those with the means to own a newer smartphone. This, of course, positions privacy as a luxury good, which is in conflict with Google's public pronouncements [58]. Instead, they should treat privacy vulnerabilities with the same seriousness that they treat security vulnerabilities and issue hotfixes to all supported Android versions.

Regulators and platform providers need better tools to monitor app behaviour and hold app developers accountable by ensuring apps comply with applicable laws, namely by protecting users' privacy and respecting their data collection choices. Society should support more mechanisms, technical and other, that empower users' informed decision-making with greater transparency into what apps are doing on their devices. To this end, we have made the list of all apps that exploit or contain code to exploit the side and covert channels we discovered available online [8].

Acknowledgments

This work was supported by the U.S. National Security Agency's Science of Security program (contract H98230-18-D-0006), the Department of Homeland Security (contract FA8750-18-2-0096), the National Science Foundation (grants CNS-1817248 and grant CNS-1564329), the Rose Foundation, the European Union's Horizon 2020 Innovation Action program (grant Agreement No. 786741, SMOOTH Project), the Data Transparency Lab, and the Center for Long-Term Cybersecurity at U.C. Berkeley. The authors would like to thank John Aycock, Irwin Reyes, Greg Hagen, René Mayrhofer, Giles Hogben, and Refjohürs Lykkewe.

References

[1] IDA: About. Ida pro. https://www.hex-rays.com/products/ida/.

[2] J. P. Achara, M. Cunche, V. Roca, and A. Francillon. WifiLeaks: Underestimated privacy implications of the access wifi state Android permission. Technical Report EURECOM+4302, Eurecom, 05 2014.

[3] A. Al-Haiqi, M. Ismail, and R. Nordin. A new sensors-based covert channel on android. *The Scientific World Journal*, 2014, 2014.

[4] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Software*, 32(5):53–59, 2015.

[5] Android Documentation. App Manifest Overview. https://developer.android.com/guide/topics/manifest/manifest-intro, 2019. Accessed: February 12, 2019.

[6] Android Studio. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.html, 2017. Accessed: October 12, 2017.

[7] Apktool. Apktool: A tool for reverse engineering android apk files. https://ibotpeaches.github.io/Apktool/.

[8] AppCensus Inc. Apps using Side and Covert Channels. https://blog.appcensus.mobi/2019/06/01/apps-using-side-and-covert-channels/, 2019.

[9] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proc. of PLDI*, pages 259–269, 2014.

[10] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.

[11] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden. Mining apps for abnormal usage of sensitive data. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 426–436. IEEE Press, 2015.

[12] G. S. Babil, O. Mehani, R. Boreli, and M. A. Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *2013 International Conference on Security and Cryptography (SECRYPT)*, pages 1–8, July 2013.

[13] Baidu. Baidu Geocoding API. https://geocoder.readthedocs.io/providers/Baidu.html, 2019. Accessed: February 12, 2019.

[14] Baidu. Baidu Maps SDK. http://lbsyun.baidu.com/index.php?title=androidsdk, 2019. Accessed: February 12, 2019.

[15] Bauer, A. and Hebeisen, C. Igexin advertising network put user privacy at risk. https://blog.lookout.com/igexin-malicious-sdk, 2019. Accessed: February 12, 2019.

[16] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall. Brahmastra: Driving Apps to Test the Security of Third-Party Components. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 1021–1036, San Diego, CA, 2014. USENIX Association.

[17] K. Block, S. Narain, and G. Noubir. An autonomic and permissionless android covert channel. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 184–194. ACM, 2017.

[18] S. Cabuk, C. E. Brodley, and C. Shields. IP covert channel detection. *ACM Transactions on Information and System Security (TISSEC)*, 12(4):22, 2009.

[19] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In *Proc. of NDSS*, 2015.

[20] B. Chess and G. McGraw. Static analysis for security. *IEEE Security & Privacy*, 2(6):76–79, 2004.

[21] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. Technical report, Wisconsin Univ-Madison Dept of Computer Sciences, 2006.

[22] M. Christodorescu, S. Jha, S. A Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Security and Privacy, 2005 IEEE Symposium on*, pages 32–46. IEEE, 2005.

[23] A. Continella, Y. Fratantonio, M. Lindorfer, A. Puccetti, A. Zand, C. Kruegel, and G. Vigna. Obfuscation-resilient privacy leak detection for mobile apps through differential analysis. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, pages 1–16, 2017.

[24] Commission Nationale de l'Informatique et des Libertés (CNIL). Data Protection Around the World. https://www.cnil.fr/en/data-protection-around-the-world, 2018. Accessed: September 23, 2018.

[25] Commission Nationale de l'Informatique et des Libertés (CNIL). The CNIL's restricted committee imposes a financial penalty of 50 Million euros against Google LLC, 2019.

[26] Luke Deshotels. Inaudible sound as a covert channel in mobile devices. In *USENIX WOOT*, 2014.

[27] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[28] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why eve and mallory love android: An analysis of android ssl (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 50–61. ACM, 2012.

[29] P. Faruki, A. Bharmal, V. Laxmi, M. S. Gaur, M. Conti, and M. Rajarajan. Evaluation of android anti-malware techniques against dalvik bytecode obfuscation. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2014 IEEE 13th International Conference on*, pages 414–421. IEEE, 2014.

[30] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: user attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, page 3, New York, NY, USA, 2012. ACM.

[31] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: validating ssl certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 38–49. ACM, 2012.

[32] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Proc. of the 5th Intl. Conf. on Trust and Trustworthy Computing*, TRUST'12, pages 291–307, Berlin, Heidelberg, 2012. Springer-Verlag.

[33] Google, Inc. Android Q privacy: Changes to data and identifiers. https://developer.android.com/preview/privacy/data-identifiers#device-identifiers. Accessed: June 1, 2019.

[34] Google, Inc. Wi-Fi Scanning Overview. https://developer.android.com/guide/topics/connectivity/wifi-scan#wifi-scan-permissions. Accessed: June 1, 2019.

[35] Google, Inc. Distribution dashboard. https://developer.android.com/about/dashboards, May 7 2019. Accessed: June 1, 2019.

[36] Google Play. Usage of Google Advertising ID. https://play.google.com/about/monetization-ads/ads/ad-id/, 2019. Accessed: February 12, 2019.

[37] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, volume 15, page 110, 2015.

[38] C. Han, I. Reyes, A. Elazari Bar On, J. Reardon, Á. Feal, S. Egelman, and N. Vallina-Rodriguez. Do You Get What You Pay For? Comparing The Privacy Behaviors of Free vs. Paid Apps. In *Workshop on Technology and Consumer Protection*, ConPro '19, 2019.

[39] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, pages 40–52. ACM, 2004.

[40] Jeremi M. Gosney. Nvidia GTX 1080 Hashcat Benchmarks. https://gist.github.com/epixoip/6ee29d5d626bd8dfe671a2d8f188b77b, 2016. Accessed: June 1, 2019.

[41] J. Kim, Y. Yoon, K. Yi, J. Shin, and SWRD Center. Scandal: Static analyzer for detecting privacy leaks in android applications. *MoST*, 12, 2012.

[42] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Annual International Cryptology Conference*, pages 388–397. Springer, 1999.

[43] Butler W Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[44] A. Le, J. Varmarken, S. Langhoff, A. Shuba, M. Gjoka, and A. Markopoulou. AntMonitor: A System for Monitoring from Mobile Devices. In *Workshop on Crowdsourcing and Crowdsharing of Big (Internet) Data*, pages 15–20, 2015.

[45] I. Leontiadis, C. Efstratiou, M. Picone, and C. Mascolo. Don't kill my ads! Balancing Privacy in an Ad-Supported Mobile Application Market. In *Proc. of ACM HotMobile*, page 2, 2012.

[46] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer. Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *badgers*, pages 3–17, 2014.

[47] M. Liu, H. Wang, Y. Guo, and J. Hong. Identifying and Analyzing the Privacy of Apps for Kids. In *Proc. of ACM HotMobile*, 2016.

[48] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security*, 51:16–31, 2015.

[49] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun. Analysis of the communication between colluding applications on modern smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 51–60. ACM, 2012.

[50] E. McReynolds, S. Hubbard, T. Lau, A. Saraf, M. Cakmak, and F. Roesner. Toys That Listen: A Study of Parents, Children, and Internet-Connected Toys. In *Proc. of ACM CHI*, 2017.

[51] Y. Michalevsky, D. Boneh, and G. Nakibly. Gyrophone: Recognizing speech from gyroscope signals. In *USENIX Security Symposium*, pages 1053–1067, 2014.

[52] Y. Michalevsky, A. Schulman, G. A. Veerapandian, D. Boneh, and G. Nakibly. Powerspy: Location tracking using mobile device power analysis. In *USENIX Security Symposium*, pages 785–800, 2015.

[53] K. Moran, M. Linares-Vasquez, C. Bernal-Cardenas, C. Vendome, and D. Poshyvanyk. Crashscope: A practical tool for automated testing of android applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 15–18, May 2017.

[54] L. Nguyen, Y. Tian, S. Cho, W. Kwak, S. Parab, Y. Kim, P. Tague, and J. Zhang. Unlocin: Unauthorized location inference on smartphones without being caught. In *2013 International Conference on Privacy and Security in Mobile Systems (PRISMS)*, pages 1–8. IEEE, 2013.

[55] Helen Nissenbaum. Privacy as contextual integrity. *Washington Law Review*, 79:119, February 2004.

[56] United Kingdom of Great Britain and Northern Ireland. Mobile telephones (re-programming) act. http://www.legislation.gov.uk/ukpga/2002/31/introduction, 2002.

[57] OpenX. Why we exist. https://www.openx.com/company/, 2019.

[58] Sundar Pichai. Privacy Should Not Be a Luxury Good. The New York Times, May 7 2019. https://www.nytimes.com/2019/05/07/opinion/google-sundar-pichai-privacy.html.

[59] I. Poese, S. Uhlig, M. A. Kaafar, B. Donnet, and B. Gueye. IP geolocation databases: Unreliable? *ACM SIGCOMM Computer Communication Review*, 41(2):53–56, 2011.

[60] A. Rao, J. Sherry, A. Legout, A. Krishnamurthy, W. Dabbous, and D. Choffnes. Meddle: middleboxes for increased transparency and control of mobile traffic. In *Proceedings of the 2012 ACM conference on CoNEXT student workshop*, pages 65–66. ACM, 2012.

[61] A. Razaghpanah, A. A. Niaki, N. Vallina-Rodriguez, S. Sundaresan, J. Amann, and P. Gill. Studying TLS usage in Android apps. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 350–362. ACM, 2017.

[62] A. Razaghpanah, R. Nithyanand, N. Vallina-Rodriguez, S. Sundaresan, M. Allman, C. Kreibich, and P. Gill. Apps, Trackers, Privacy, and Regulators: A Global Study of the Mobile Tracking Ecosystem. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[63] A. Razaghpanah, N. Vallina-Rodriguez, S. Sundaresan, C. Kreibich, P. Gill, M. Allman, and V. Paxson. Haystack: In Situ Mobile Traffic Analysis in User Space. *arXiv preprint arXiv:1510.01419*, 2015.

[64] J. Ren, M. Lindorfer, D. J. Dubois, A. Rao, D. Choffnes, and N. Vallina-Rodriguez. Bug fixes, improvements,... and privacy leaks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[65] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. Choffnes. ReCon: Revealing and Controlling Privacy Leaks in Mobile Network Traffic. In *Proceedings of the ACM SIGMOBILE MobiSys*, pages 361–374, 2016.

[66] I. Reyes, P. Wijesekera, J. Reardon, A. Elazari Bar On, A. Razaghpanah, N. Vallina-Rodriguez, and S. Egelman. "Won't Somebody Think of the Children?" Examining COPPA Compliance at Scale. *Proceedings on Privacy Enhancing Technologies*, 2018(3):63–83, 2018.

[67] Salmonads. About us. http://publisher.salmonads.com, 2016.

[68] G. Sarwar, O. Mehani, R. Boreli, and M. A. Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *SECRYPT*, volume 96435, 2013.

[69] Sarah Schafer. With capital in panic, pizza deliveries soar. The Washington Post, December 19 1998. https://www.washingtonpost.com/wp-srv/politics/special/clinton/stories/pizza121998.htm.

[70] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*, volume 11, pages 17–33, 2011.

[71] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 317–331, Washington, DC, USA, 2010. IEEE Computer Society.

[72] L. Simon, W. Xu, and R. Anderson. Don't interrupt me while i type: Inferring text entered through gesture typing on android keyboards. *Proceedings on Privacy Enhancing Technologies*, 2016(3):136–154, 2016.

[73] Y. Song and U. Hengartner. PrivacyGuard: A VPN-based Platform to Detect Information Leakage on Android Devices. In *Proc. of ACM SPSM*, 2015.

[74] R. Spreitzer, V. Moonsamy, T. Korak, and S. Mangard. Systematic classification of side-channel attacks: a case study for mobile devices. *IEEE Communications Surveys & Tutorials*, 20(1):465–488, 2017.

[75] Statista. Global market share held by the leading smartphone operating systems in sales to end users from 1st quarter 2009 to 2nd quarter 2018. https://www.statista.com/statistics/266136, 2019. Accessed: February 11, 2019.

[76] COUNTY OF LOS ANGELES SUPERIOR COURT OF THE STATE OF CALIFORNIA. Complaint for injunctive relief and civil penalties for violations of the unfair competition law. http://src.bna.com/EqH, 2019.

[77] Unity Technologies. Unity 3d. https://unity3d.com, 2019.

[78] L. Tsai, P. Wijesekera, J. Reardon, I. Reyes, S. Egelman, D. Wagner, N. Good, and J.W. Chen. Turtle guard: Helping android users apply contextual privacy preferences. In *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*, pages 145–162. USENIX Association, 2017.

[79] U.S. Federal Trade Commission. The federal trade commission act. (ftc act). https://www.ftc.gov/enforcement/statutes/federal-trade-commission-act.

[80] U.S. Federal Trade Commission. Children's online privacy protection rule ("coppa"). https://www.ftc.gov/enforcement/rules/rulemaking-regulatory-reform-proceedings/childrens-online-privacy-protection-rule, November 1999.

[81] U.S. Federal Trade Commission. In the Matter of HTC America, Inc. https://www.ftc.gov/sites/default/files/documents/cases/2013/07/130702htcdo.pdf, 2013.

[82] U.S. Federal Trade Commission. Mobile Advertising Network InMobi Settles FTC Charges It Tracked Hundreds of Millions of Consumers' Locations Without Permission. https://www.ftc.gov/news-events/press-releases/2016/06/mobile-advertising-network-inmobi-settles-ftc-charges-it-tracked, June 22 2016.

[83] U.S. Federal Trade Commission. In the Matter of Turn Inc. https://www.ftc.gov/system/files/documents/cases/152_3099_c4612_turn_complaint.pdf, 2017.

[84] U.S. Federal Trade Commission. Mobile security updates: Understanding the issues. https://www.ftc.gov/system/files/documents/reports/mobile-security-updates-understanding-issues/mobile_security_updates_understanding_the_issues_publication_final.pdf, 2018.

[85] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov. Android permissions remystified: A field study on contextual integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 499–514, Washington, D.C., August 2015. USENIX Association.

[86] P. Wijesekera, A. Baokar, L. Tsai, J. Reardon, S. Egelman, D. Wagner, and K. Beznosov. The feasibility of dynamically granted permissions: Aligning mobile privacy with user preferences. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1077–1093, May 2017.

[87] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1043–1054. ACM, 2013.

[88] B. Yankson, F. Iqbal, and P.C.K. Hung. Privacy preservation framework for smart connected toys. In *Computing in Smart Toys*, pages 149–164. Springer, 2017.

[89] S. Yong, D. Lindskog, R. Ruhl, and P. Zavarsky. Risk Mitigation Strategies for Mobile Wi-Fi Robot Toys from Online Pedophiles. In *Proc. of IEEE SocialCom*, pages 1220–1223. IEEE, 2011.

[90] J. Zhang, A. R. Beresford, and I. Sheret. Sensorid: Sensor calibration fingerprinting for smartphones. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2019.

[91] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt. Identity, location, disease and more: Inferring your secrets from android public resources. In *Proc. of 2013 ACM SIGSAC conference on Computer & Communications Security*. ACM, 2013.

[92] S. Zimmeck, Z. Wang, L. Zou, R. Iyengar, B. Liu, F. Schaub, S. Wilson, N. Sadeh, S. M. Bellovin, and J. Reidenberg. Automated analysis of privacy requirements for mobile apps. In *24th Network & Distributed System Security Symposium*, 2017.

SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks

Saad Islam[1], Ahmad Moghimi[1], Ida Bruhns[2], Moritz Krebbel[2], Berk Gulmezoglu[1], Thomas Eisenbarth[1, 2], and Berk Sunar[1]

[1]Worcester Polytechnic Institute, Worcester, MA, USA
[2] University of Lübeck, Lübeck, Germany

Abstract

Modern microarchitectures incorporate optimization techniques such as *speculative loads* and *store forwarding* to improve the memory bottleneck. The processor executes the `load` speculatively before the `stores`, and forwards the data of a preceding `store` to the `load` if there is a potential dependency. This enhances performance since the `load` does not have to wait for preceding `stores` to complete. However, the dependency prediction relies on partial address information, which may lead to false dependencies and stall hazards.

In this work, we are the first to show that the dependency resolution logic that serves the speculative load can be exploited to gain information about the physical page mappings. Microarchitectural side-channel attacks such as Rowhammer and cache attacks like Prime+Probe rely on the reverse engineering of the virtual-to-physical address mapping. We propose the SPOILER attack which exploits this leakage to speed up this reverse engineering by a factor of 256. Then, we show how this can improve the Prime+Probe attack by a 4096 factor speed up of the eviction set search, even from sandboxed environments like JavaScript. Finally, we improve the Rowhammer attack by showing how SPOILER helps to conduct DRAM row conflicts deterministically with up to 100% chance, and by demonstrating a double-sided Rowhammer attack with normal user's privilege. The later is due to the possibility of detecting contiguous memory pages using the SPOILER leakage.

1 Introduction

Microarchitectural attacks have evolved over the past decade from attacks on weak cryptographic implementations [5] to devastating attacks breaking through layers of defenses provided by the hardware and the Operating System (OS) [52]. These attacks can steal secrets such as cryptographic keys [4, 44] or keystrokes [33]. More advanced attacks can entirely subvert the OS memory isolation to read the memory content from more privileged security domains [35], and to bypass defense mechanisms such as Kernel Address Space Layout Randomization (KASLR) [11, 17]. Rowhammer attacks can further break the data and code integrity by tampering with memory contents [29, 47]. While most of these attacks require local access and native code execution, various efforts have been successful in conducting them remotely [50] or from within a remotely accessible sandbox such as JavaScript [42].

Memory components such as DRAM [29] and cache [43] are not the only microarchitectural attack surfaces. *Spectre* attacks on the branch prediction unit [30, 38] imply that side channels such as caches can be used as a primitive for more advanced attacks on speculative engines. Speculative engines predict the outcome of an operation before its completion, and they enable execution of the following dependent instructions ahead of time based on the prediction. As a result, the pipeline can maximize the instruction level parallelism and resource usage. In rare cases where the prediction is wrong, the pipeline needs to be flushed resulting in performance penalties. However, this approach suffers from a security weakness, in which an adversary can fool the predictor and introduce arbitrary mispredictions that leave microarchitectural footprints in the cache. These footprints can be collected through the cache side channel to steal secrets.

Modern processors feature further speculative behavior such as *memory disambiguation* and speculative loads [10]. A `load` operation can be executed speculatively before preceding `store` operations. During the speculative execution of the `load`, false dependencies may occur due to the unavailability of physical address information. These false dependencies need to be resolved to avoid computation on invalid data. The occurrence of false dependencies and their resolution depend on the actual implementation of the memory subsystem. Intel uses a proprietary memory disambiguation and *dependency resolution logic* in the processors to predict and resolve false dependencies that are related to the speculative load. In this work, we discover that the dependency resolution logic suffers from an unknown false dependency independent of the 4K aliasing [40, 49]. The discovered false dependency happens during the 1 MB aliasing of speculative memory accesses

which is exploited to leak information about physical page mappings.

The state-of-the-art microarchitectural attacks [25, 45] either rely on knowledge of physical addresses or are significantly eased by that knowledge. Yet, knowledge of the physical address space is only granted with root privileges. Cache attacks such as *Prime+Probe* on the Last-Level Cache (LLC) are challenging due to the unknown mapping of virtual addresses to cache sets and slices. Knowledge about the physical page mappings enables more attack opportunities using the Prime+Probe technique. Rowhammer [29] attacks require efficient access to rows within the same bank to induce fast row conflicts. To achieve this, an adversary needs to reverse engineer layers of abstraction from the virtual address space to DRAM cells. Availability of physical address information facilitates this reverse engineering process. In sandboxed environments, attacks are more limited, since in addition to the limited access to the address space, low-level instructions are also inaccessible [18]. Previous attacks assume special access privileges only granted through weak software configurations [25, 34, 55] to overcome some of these challenges. In contrast, SPOILER only relies on simple operations, `load` and `store`, to recover crucial physical address information, which in turn enables Rowhammer and cache attacks, by leaking information about physical pages without assuming any weak configuration or special privileges.

1.1 Our Contribution

We have discovered a novel microarchitectural leakage which reveals critical information about physical page mappings to user space processes. The leakage can be exploited by a limited set of instructions, which is visible in all Intel generations starting from the 1st generation of Intel Core processors, independent of the OS and also works from within virtual machines and sandboxed environments. In summary, this work:

1. exposes a previously unknown microarchitectural leakage stemming from the false dependency hazards during speculative load operations.

2. proposes an attack, SPOILER, to efficiently exploit this leakage to speed up the reverse engineering of virtual-to-physical mappings by a factor of 256 from both native and JavaScript environments.

3. demonstrates a novel eviction set search technique from JavaScript and compares its reliability and efficiency to existing approaches.

4. achieves efficient DRAM row conflicts and the first *double-sided Rowhammer* attack with normal user-level privilege using the contiguous memory detection capability of SPOILER.

5. explores how SPOILER can track nearby load operations from a more privileged security domain right after a context switch.

1.2 Related Work

Kosher et al. [30] and Maisuradze et al. [38] have exploited vulnerabilities in the speculative branch prediction unit. Transient execution of instructions after a fault, as exploited by Lipp et al. [35] and Bulck et al. [52], can leak memory content of protected environments. Similarly, transient behavior due to the lazy store/restore of the FPU and SIMD registers can leak register contents from other contexts [48]. New variants of both Meltdown and Spectre have been systematically analyzed [7]. The Speculative Store Bypass (SSB) vulnerability [21] is a variant of the Spectre attack and relies on the stale sensitive data in registers to be used as an address for speculative loads which may then allow the attacker to read this sensitive data. In contrast to previous attacks on speculative and transient behaviors, we discover a new leakage on the undocumented memory disambiguation and dependency resolution logic. SPOILER is **not** a Spectre attack. The root cause for SPOILER is a weakness in the address speculation of Intel's proprietary implementation of the memory subsystem which directly leaks timing behavior due to physical address conflicts. Existing spectre mitigations would therefore not interfere with SPOILER.

The timing behavior of the 4K aliasing false dependency on Intel processors have been studied [12, 61]. *MemJam* [40] uses this behavior to perform a side-channel attack, and Sullivan et al. [49] demonstrate a covert channel. These works only mention the 4K aliasing as documented by Intel [24], and the authors conclude that the address aliasing check is a two stage approach: Firstly, it uses page offset for the initial guess. Secondly, it performs the final resolution based on the exact physical address. On the contrary, we discover that the undocumented *address resolution* logic performs additional partial address checks that lead to an unknown, but observable aliasing behavior based on the physical address.

Several microarchitectural attacks have been discovered to recover virtual address information and break KASLR by exploiting the Translation Lookaside Buffer (TLB) [22], Branch Target Buffer (BTB) [11] and Transactional Synchronization Extensions (TSX) [27]. Additionally, Gruss et al. [17] exploit the timing information obtained from the `prefetch` instruction to leak the physical address information. The main obstacle to this approach is that the `prefetch` instruction is not accessible in JavaScript, and it can be disabled in native sandboxed environments [62], whereas SPOILER is applicable to sandboxed environments including JavaScript.

Knowledge of the physical address enables adversaries to bypass OS protections [28] and ease other microarchitectural attacks [34]. For instance, the `procfs` filesystem exposes physical addresses [34], and *Huge pages* allocate contiguous

physical memory [25, 36]. *Drammer* [55] exploits the Android *ION memory allocator* to access contiguous memory. However, access to the aforementioned primitives is restricted on most environments by default. We do not have any assumption about the OS and software configuration, and we exploit a hardware leakage with minimum access rights to find virtual pages that have the same least significant 20 physical address bits. *GLitch* [13] detects contiguous physical pages by exploiting row conflicts through the GPU interface. In contrast, our attack does not rely on a specific integrated GPU configuration, and it is widely applicable to any system running on an Intel CPU. We use SPOILER to find contiguous physical pages with a high probability and verify it by producing row conflicts. SPOILER is particularly helpful for attacks in sandboxed low-privilege environments such as JavaScript, where previous methods require a time-consuming brute forcing of the memory addresses [18, 42, 47].

2 Background

2.1 Memory Management

The virtual memory manager shares the DRAM across all running tasks by assigning isolated virtual address spaces to each task. The assigned memory is allocated in pages, which are typically 4 kB each, and each virtual page will be stored as a physical page in DRAM through a virtual-to-physical page mapping. Memory instructions operate on virtual addresses, which are translated within the processor to the corresponding physical addresses. The page offset comprising the least significant 12 bits of the virtual address is not translated. The processor only translates the bits in the rest of the virtual address, the virtual page number. The OS is the reference for this translation, and the processor stores the translation results inside the TLB. As a result, repeated translations of the same address are performed more efficiently.

2.2 Cache Hierarchy

Modern processors incorporate multiple levels of caches to avoid the DRAM access latency. The cache memory on Intel processors is organized into sets and slices. Each set can store a certain number of lines, where the line size is 64 bytes. The 6 Least Significant Bits (LSBs) of the physical address are used to determine the offset within a line and the remaining bits are used to determine which set to store the cache line in. The number of physical address bits that are used for mapping is higher for the LLC, since it has a large number of sets, e.g., 8192 sets. Hence, the untranslated part of the virtual address bits which is the page offset, cannot be used to index the LLC sets. Instead, higher physical address bits are used. Further, each set of LLC is divided into multiple slices, one slice for each logical processor. The mapping of the physical addresses to the slices uses an undocumented function [26]. When the processor accesses a memory address, a cache hit or miss occurs. If a miss occurs in all cache levels, the memory line has to be fetched from DRAM. Accesses to the same memory address would be served from the cache unless other memory accesses evict that cache line. In addition, we can use the `clflush` instruction, which follows the same memory access check as other memory operations, to evict our own cache lines from the entire cache hierarchy.

2.3 Prime+Probe Attack

In the Prime+Probe attack, the attacker first fills an entire cache set by accessing memory addresses that are mapped to the same set, an *eviction set*. Later, the attacker checks whether the victim program has displaced any entry in the cache set by accessing the eviction set again and measuring the execution time. If this is the case, the attacker can detect congruent addresses, since the displaced entries cause an increased access time. However, finding the eviction sets is difficult due to the unknown translation of virtual addresses to physical addresses. Since an unprivileged attacker has no access to hugepages [23] or the virtual-to-physical page mapping such as the `pagemap` file [34], knowledge about the physical address bits greatly speeds up the eviction set search.

2.4 Rowhammer Attack

DRAM consists of multiple memory banks, and each bank is subdivided into rows. When the processor accesses a memory location, the corresponding row needs to be activated and loaded into the row buffer. If the processor accesses the same row again, it is called a row hit, and the request will be served from the row buffer. Otherwise, it is called a row conflict, and the previous row will be deactivated and copied back to the original row location, after which the new row is activated. DRAM cells leak charge over time and need to be refreshed periodically to maintain the data. A Rowhammer [29] attack causes cells of a victim row to leak faster by activating the neighboring rows repeatedly. If the refresh cycle fails to refresh the victim row fast enough, that leads to bit flips. Once bit flips are found, they can be exploited by placing any security-critical data structure or code page at that particular location and triggering the bit flip again [16, 47, 60]. The Rowhammer attack requires fast access to the same DRAM cells by bypassing the CPU cache, e.g., using `clflush` [29]. Additionally, cache eviction based on an eviction set can also result in access to DRAM cells when `clflush` is not available [3, 18]. Efficiently building eviction sets may thus also enhance Rowhammer attacks. For a successful Rowhammer attack, it is essential to collocate multiple memory pages within the same bank and adjacent to each other. A number of physical address bits, depending on the hardware configuration, are used to map memory pages to banks [45]. Since the rows are generally placed sequentially within the banks,

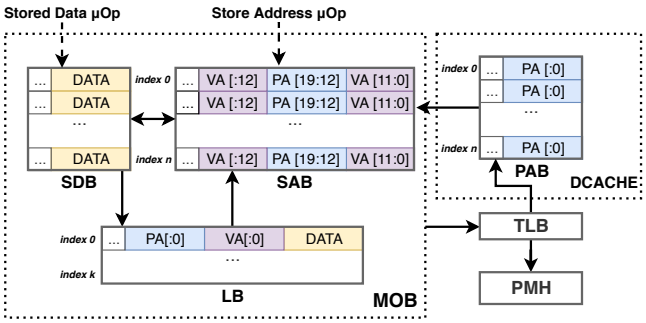


Figure 1: The Memory Order Buffer includes circular buffers SDB, SAB and LB. SDB, SAB and PAB of the DCACHE have the same number of entries. SAB may initially hold the virtual address and the partial physical address. MOB requests the TLB to translate the virtual address and update the PAB with the translated physical address.

access to adjacent rows within the same bank can be achieved if we have access to contiguous physical pages.

2.5 Memory Order Buffer

The processor manages memory operations using the Memory Order Buffer (MOB). MOB is tightly coupled with the data cache. The MOB assures that memory operations are executed efficiently by following the Intel memory ordering rule [39] in which memory `stores` are executed in-order and memory `loads` can be executed out-of-order. These rules have been enforced to improve the efficiency of memory accesses, while guaranteeing their correct commitment. Figure 1 shows the MOB schematic according to Intel [1, 2]. The MOB includes circular buffers, *store buffer*[1] and *load buffer* (LB). A `store` will be decoded into two micro ops to store the address and data, respectively, to the store buffer. The store buffer enables the processor to continue executing other instructions before commitment of the `stores`. As a result, the pipeline does not have to stall for the `stores` to complete. This further enables the MOB to support out-of-order execution of the `load`.

Store forwarding is an optimization mechanism that sends the `store` data to a `load` if the `load` address matches any of the store buffer entries. This is a speculative process, since the MOB cannot determine the true dependency of the `load` on `stores` based on the store buffer. Intel's implementation of the store buffer is undocumented, but a potential design suggests that it will only hold the virtual address, and it may include part of the physical address [1, 2, 31]. As a result, the processor may falsely forward the data, although the physical addresses do not match. The complete resolution will be delayed until the `load` commitment, since the MOB needs to ask the TLB for the complete physical address information, which is time consuming. Additionally, the data cache (DCACHE) may hold the translated store addresses in a Physical Address Buffer (PAB) with equal number of entries as the store buffer.

[1] Store buffer consists of *Store Address Buffer (SAB)* and *Store Data Buffer (SDB)*. For simplicity, we use *Store Buffer* to mention the logically combined SAB and SDB units.

3 Speculative Load Hazards

As we mentioned earlier, memory `loads` can be executed out-of-order and before the preceding memory `stores`. If one of the preceding `stores` modifies the content of a location in memory, the memory `load` address is referring to, out-of-order execution of the `load` will operate on stale data, which results in invalid execution of a program. This out-of-order execution of the memory `load` is a speculative behavior, since there is no guarantee during the execution time of the `load` that the virtual addresses corresponding to the memory `stores` do not conflict with the `load` address after translation to physical addresses. Figure 2 demonstrates this effect on a hypothetical processor with 7 pipeline stages. As multiple `stores` may be blocked due to limited resources, the execution of the `load` and dependent instructions in the pipeline, the *load block*, will bypass the `stores` since the MOB assumes the load block to be independent of the `stores`. This speculative behavior improves the memory bottleneck by letting other instructions continue their execution. However, if the dependency of the `load` and preceding `stores` is not verified, the load block may be computed on incorrect data which is either falsely forwarded by store forwarding (false dependency), or loaded from a stale cache line (unresolved true dependency). If the processor detects a false dependency before committing the `load`, it has to flush the pipeline and re-execute the load block. This will cause observable performance penalties and timing behavior.

3.1 Dependency Resolution

Dependency checks and resolution occur in multiple stages depending on the availability of the address information in the store buffer. A `load` instruction needs to be checked against all preceding `stores` in the store buffer to avoid false dependencies and to ensure the correctness of the data. A potential design [20, 31],[2] suggests the following stages for the dependency check and resolution, as shown in Figure 3:

1. **Loosenet**: The first stage is the *loosenet* check where the page offsets of the `load` and `stores` are compared[3]. In case of a loosenet hit, the compared `load` and `store` may be dependent and the processor will proceed to the next check stage.

[2] The implementation of the MOB used in Intel processors is unpublished and therefore we cannot be certain about the precise architecture. Our results agree with some of the possible designs that are described in the Intel patents.

[3] According to `Ld_Blocks_Partial:Address_Alias` Hardware Performance Counter (HPC) event [24], loosenet is defined by Intel as the mechanism that only compare the page offsets.

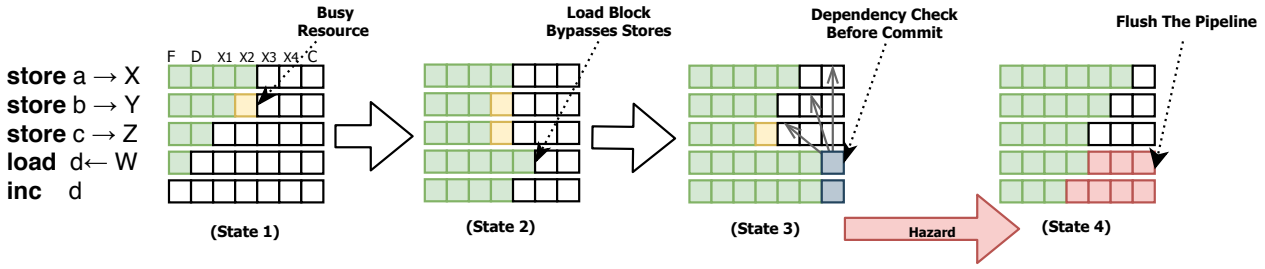


Figure 2: The speculative load is demonstrated on a hypothetical processor with 7 pipeline stages: F = Fetch, D = Decode, X_{1-4} = Executions, and C = Commit. When the memory `stores` are blocked competing for resources (State 1), the `load` will bypass the `stores` (State 2). The load block including the dependent instructions will not be committed until the dependency of the address W versus X,Y,Z are resolved (State 3). In case of a dependency hazard (State 4), the pipeline is flushed and the `load` is restarted.

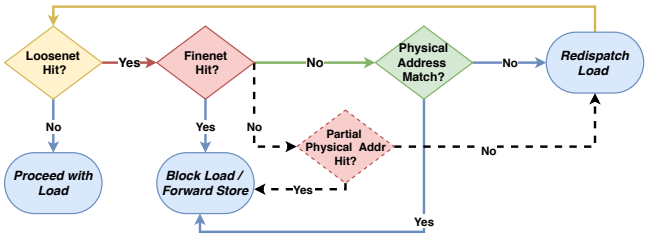


Figure 3: The dependency check logic: *loosenet* initially checks the least 12 significant bits (page offset) and the *finenet* checks the upper address bits, related to the page number. The final dependency using the physical address matching might still fail due to partial physical address checks.

2. **Finenet**: The next stage, called *finenet*, uses upper address bits. The *finenet* can be implemented to check the upper virtual address bits [20], or the physical address tag [31]. Either way, it is an intermediate stage, and it is not the final dependency resolution. In case of a *finenet* hit, the processor blocks the `load` and/or forwards the store data, otherwise, the dependency resolution will go into the final stage.

3. **Physical Address Matching**: At the final stage, the physical addresses will be checked. Since this stage is the final chance to resolve potential false dependencies, we expect the full physical address to be checked. However, one possible design suggests that if the physical addresses are not available, the physical address matching returns true and continues with the store forwarding [20].

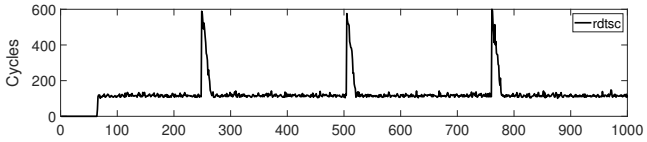
Since the page offset is identical between the virtual and physical address, loosenet can be performed as soon as the `store` is decoded. [2] suggests that the store buffer only holds bit 19 to 12 of the physical address. Although the PAB holds the full translated physical address, it is not clear in which stage this information can be available to the MOB. As a result, the *finenet* check may be implemented based on checking the partial physical address bits. As we verify later, the dependency resolution logic may fail to resolve the dependency at multiple intermediate stages due to unavailability of the full physical address.

4 The SPOILER Attack

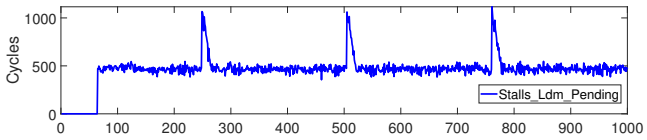
The attack model for SPOILER is the same as Rowhammer and cache attacks where the attacker's code is needed to be executed on the same underlying hardware as of the victim. As described in Section 3, speculative loads may face other aliasing conditions in addition to the 4K aliasing, due to the partial checks on the higher address bits. To confirm this, we design an experiment to observe timing behavior of a speculative load based on higher address bits. For this purpose, we propose Algorithm 1 that executes a speculative load after multiple `stores` and further make sure to fill the store buffer with addresses that cause 4K aliasing during the execution of the `load`. Having w as the window size, the algorithm iterates over a number of different memory pages, and for each page, it performs `stores` to that page and all previous w pages within a *window*. Since the size of the store buffer varies between different processor generations, we choose a big enough window ($w = 64$) to ensure that the `load` has 4K aliasing with the maximum number of entries in the store buffer and hence maximum potential conflicts. Following the `stores`, we measure the timing of a `load` operation from a different memory page, as defined by x. Since we want the `load` to be executed speculatively, we can not use a store fence such as `mfence` before the `load`. As a result, our measurements are an estimate of execution time for the speculatively `load` and nearby microarchitectural events. This may include a negligible portion of overhead for the execution of `stores`, and/or any delay due to the dependency resolution. If we iterate over a diverse set of addresses with different virtual and physical page numbers, but the same page offset, we should be able to monitor any discrepancy.

Algorithm 1 Address Aliasing

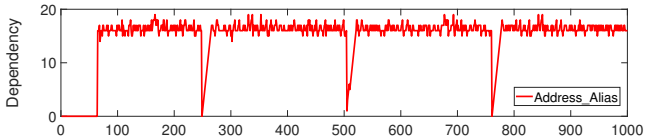
```
for p from w to PAGE_COUNT do
    for i from w to 0 do
        data --store--> buffer[(p − i) × PAGE_SIZE]
    end for
    t1 = rdtscp()
    data <--load-- buffer[x × PAGE_SIZE]
    t2 = rdtscp()
    measure[p] ← t2 − t1
end for
return measure
```



(a) Step-wise peaks with a very high latency can be observed on some of the virtual pages



(b) Affected HPC event: `Cycle_Activity:Stalls_Ldm_Pending`



(c) Affected HPC event: `Ld_Blocks_Partial:Address_Alias`

Figure 4: SPOILER's timing measurements and hardware performance counters recorded simultaneously.

4.1 Speculative Dependency Analysis

In this section, we use Algorithm 1 and Hardware Performance Counters (HPC) to perform an empirical analysis of the dependency resolution logic. HPCs can keep track of low-level hardware-related events in the CPU. The counters are accessible via special purpose registers and can be used to analyze the performance of a program. They provide a powerful tool to detect microarchitectural components that cause bottlenecks. Software libraries such as Performance Application Programming Interface (PAPI) [51] simplifies programming and reading low-level HPC on Intel processors. Initially, we execute Algorithm 1 for 1000 different virtual pages. Figure 4(a) shows the cycle count for each iteration with a set of 4 kB aliased store addresses. Interestingly, we observe multiple step-wise peaks with a very high latency. Then, we use PAPI to monitor 30 different performance counters listed in Table 5 in the appendix while running the same

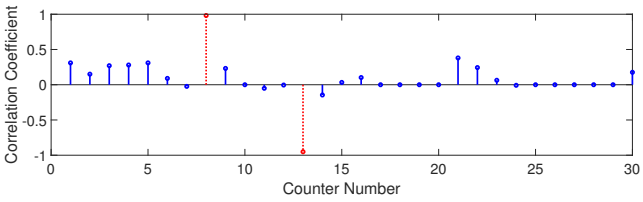


Figure 5: Correlation with HPCs listed in Table 5 in the appendix. `Ld_Blocks_Partial:Address_Alias` and `Cycle_Activity:Stalls_Ldm_Pending` (both dotted red) have strong positive and negative correlations, respectively.

experiment. At each iteration, only one performance counter is monitored alongside the aforementioned timing measurement. After each speculative load, the performance counter value and the `load` time are both recorded. Finally, we obtain the timings and performance counter value pairs as depicted in Figure 4.

To find any relation between the observed high latency and a particular event, we compute correlation coefficients between counters and the timing measurements. Since the latency only occurs in the small region of the trace where the timing increases, we only need to compute the correlation on these regions. When an increase of at least 200 clock cycles is detected, the next s values from timing and the HPC traces are used to calculate the correlations, where s is the number of steps from Table 1 and 200 is the average execution time for a load.

As shown in Figure 5, two events have a high correlation with the leakage: `Cycle_Activity:Stalls_Ldm_Pending` has the highest correlation of 0.985. This event shows the number of cycles for which the execution is stalled and no instructions are executed due to a pending `load`. `Ld_Blocks_Partial:Address_Alias` has an inverse correlation with the leakage. This event counts the number of false dependencies in the MOB when loosenet resolves the 4K aliasing condition. Separately, `Exe_Activity:Bound_on_Stores` increases with more number of `stores` within the inner window loop in Algorithm 1, but it does not have a correlation with the leakage. The reason behind this behavior is that the store buffer is full, and additional store operations are pending. However, since there is no correlation with the leakage, this shows that the timing behavior is not due to the `stores` delay. We also attempt to profile any existing counters related to the memory disambiguation. However, the events `Memory_Disambiguation.Success` and `Memory_Disambiguation.Reset` are not available on the modern architectures that are tested.

CPU Model	Architecture	Steps	SB Size
Intel Core i7-8650U	Kaby Lake R	22	56
Intel Core i7-7700	Kaby Lake	22	56
Intel Core i5-6440HQ	Skylake	22	56
Intel Xeon E5-2640v3	Haswell	17	42
Intel Xeon E5-2670v2	Ivy Bridge EP	14	36
Intel Core i7-3770	Ivy Bridge	12	36
Intel Core i7-2670QM	Sandy Bridge	12	36
Intel Core i5-2400	Sandy Bridge	12	36
Intel Core i5 650	Nehalem	11	32
Intel Core2Duo T9400	Core	N/A	20
Qualcomm Kryo 280	ARMv8-A	N/A	*
AMD A6-4455M	Bulldozer	N/A	*

Table 1: 1 MB aliasing on various architectures: The tested AMD and ARM architectures, and Intel Core generation do not show similar effects. The Store Buffer (SB) sizes are gathered from Intel Manual [24] and *wikichip.org* [57–59].

4.2 Leakage of the Physical Address Mapping

In this experiment, we evaluate whether the observed step-wise latency has any relationship with the physical page numbers by observing the `pagemap` file. As shown in Figure 6, we observe step-wise peaks with a very high latency which appear once in every 256 pages on average.The 20 least significant bits of physical address for the `load` matches with the physical addresses of the `stores` where high peaks for virtual pages are observed. In our experiments, we always detect peaks with different virtual addresses, which have the matching least 20 bits of physical address. This observation clearly discovers the existence of 1 MB aliasing effect based on the physical addresses. This 1 MB aliasing leaks information about 8 bits of mapping that were unknown to the user space processes.

Matching this observation with the previously observed `Cycle_Activity:Stalls_Ldm_Pending` with a high correlation, the speculative load has been stalled to resolve the dependency with conflicting store buffer entries after the occurrence of a 1 MB aliased address. This observation verifies that the latency is due to the pending `load`. When the latency is at the highest point, `Ld_Blocks_Partial:Address_Alias` drops to zero, and it increments at each down step of the peak. This implies that the loosenet check does not resolve the rest of the store dependencies whenever there is a 1 MB aliased address in the store buffer.

4.3 Evaluation

In the previous experiment, the execution time of the `load` operation that is delayed by 1 MB aliasing decreases gradually in each iteration (Figure 6). The number of steps to reach the normal execution time is consistent on the same processor. When the first store in the window loop accesses a memory address with the matching 1 MB aliased address, the latency is at its highest point, marked as "1" in Figure 6. As the window loop accesses this address later in the loop, it appears closer to the `load` with a lower latency like the steps marked as 5, 15 and 22. This observation matches the *carry chain algorithm* described by Intel [20] where the aliasing check starts from the most recent `store`. As shown in Table 1, experimenting with various processor generations shows that the number of steps has a linear correlation with the size of the store buffer which is architecture dependent. While the leakage exists on all Intel Core processors starting from the first generation, the timing effect is higher for the more recent generations with a bigger store buffer size. The analyzed ARM and AMD processors do not show similar behavior[4].

As our time measurement for speculative load suggests, it is not possible to reason whether the high timing is due to a very slow `load` or commitment of store operations. If the step-wise delay matches the store buffer entries, this delay may be either due to the the dependency resolution logic performing a pipeline flush and restart of the `load` for each 4 kB aliased entry starting from the 1 MB aliased entry, or due to the `load` waiting for all the remaining `stores` to commit because of an unresolved hazard. To explore this further, we perform an additional experiment with all store addresses replaced with non-aliased addresses except for one. This experiment shows that the peak disappears if there is only a single 4 kB and 1 MB aliased address in the store buffer.

Lastly, we run the same experiments on a shuffled set of virtual addresses to assure that the contiguous virtual addresses may not affect the observed leakage. Our experiment with the shuffled virtual addresses exactly match the same step-wise behavior suggesting that the upper bits in virtual addresses do not affect the leakage behavior, and the leakage is solely due to the aliasing on physical address bits.

4.3.1 Comparison of Address Aliasing Scenarios

We further test other address combinations to compare additional address aliasing scenarios using Algorithm 1. As shown by Figure 7, when `stores` and the `load` access different cache sets without aliasing, the `load` is executed in 30 cycles, which is the typical timing for an L1 data cache load including the `rdtscp` overhead. When the `stores` have different memory addresses with the same page offset, but the `load` has a different offset, the `load` takes 100 cycles to execute. This shows that even memory addresses in the store buffer having 4K Aliasing conditions with each other that are totally unrelated to the speculative load create a memory bottleneck for the `load`. In the next scenario, 4K aliasing between the `load` and all `stores`, the average load time is about 200 cycles. While the aforementioned 4K aliasing scenarios may leak cross domain information about memory accesses

[4] We use `rdtscp` for Intel and AMD processors and the `clock_gettime` for ARM processors to perform the time measurements.

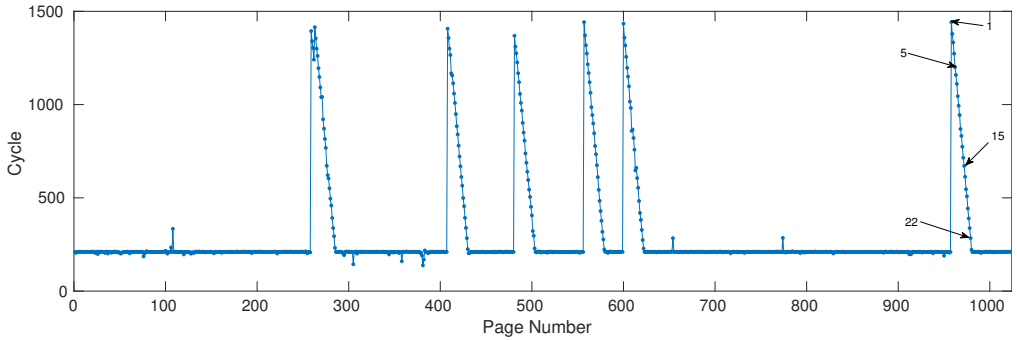


Figure 6: Step-wise peaks with 22 steps and a high latency can be observed on some of the pages (*Core i7-8650U* processor).

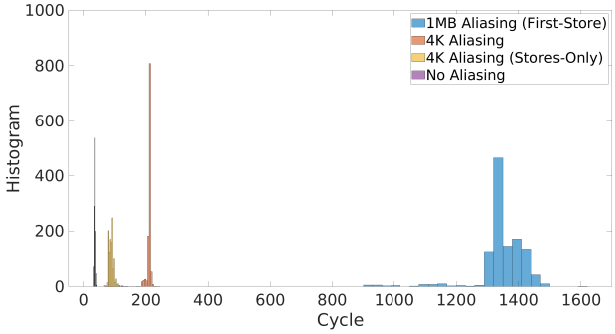


Figure 7: Histogram of the measurement for the speculative load with various store addresses. `Load` will be fast, 30 cycles, without any dependency. If there exists 4K aliasing only between the `stores`, the average is 100. The average is 200 when there is 4K aliasing of `load` and `stores`. The 1 MB aliasing has a distinctive high latency.

(Section 7), the most interesting scenario is the 1 MB aliasing which takes more than 1200 cycles for the highest point in the peak. For simplicity, we refer to the 1 MB aliased address as *aliased address*, in the rest of the paper.

4.4 Discussion

4.4.1 The Curious Case of Memory Disambiguation

The processor uses an additional speculative engine, called the *memory disambiguator* [10, 32], to predict memory false dependencies and reduce the chance of their occurrences. The main idea is to predict if a `load` is independent of preceding `stores` and proceed with the execution of the `load` by ignoring the store buffer. The predictor uses a hash table that is indexed with the address of the `load`, and each entry of the hash table has a saturating counter. If the pre-commitment dependency resolution does not detect false dependencies, the counter is incremented, otherwise it will be reset to zero. After multiple successful executions of the same `load` instruction, the predictor assumes that the `load` is safe to execute. Every time the counter resets to zero, the next iteration of the `load` will be blocked to be checked against the store buffer entries. Mispredictions result in performance overhead due to pipeline flushes. To avoid repeated mispredictions, a watchdog mechanism monitors the success rate of the prediction, and it can temporarily disable the memory disambiguator.

The predictor of the memory disambiguator should go into a stable state after the first few iterations, since the memory `load` is always truly independent of any aliased store. Hence the saturating counter for the target speculative load address passes the threshold, and it never resets due to a false prediction. As a result, the memory disambiguator should always fetch the data into the cache without any access to the store buffer. However, since the memory disambiguation performs speculation, the dependency resolution at some point verifies the prediction. The misprediction watchdog is also supposed to only disable the memory disambiguator when the misprediction rate is high, but in this case we should have a high prediction rate. Accordingly, the observed leakage occurs after the disambiguation and during the last stages of dependency resolution, i.e., the memory disambiguator only performs prediction on the 4K aliasing at the initial loosenet check, and it cannot protect the pipeline from 1 MB aliasing that appears at a later stage.

4.4.2 Hyperthreading Effect

Similar to the 4K Aliasing [40, 49], we empirically test whether the 1 MB aliasing can be used as a covert/side channel through logical processors. Our observation shows that when we run our experiments on two logical processors on the same physical core, the number of steps in the peaks is exactly halved. This matches the description by Intel [24] where it is stated that the store buffer is split between the logical processors. As a result, the 1 MB aliasing effect is not visible and exploitable across logical cores. [31] suggests that loosenet checks mask out the `stores` on the opposite thread.

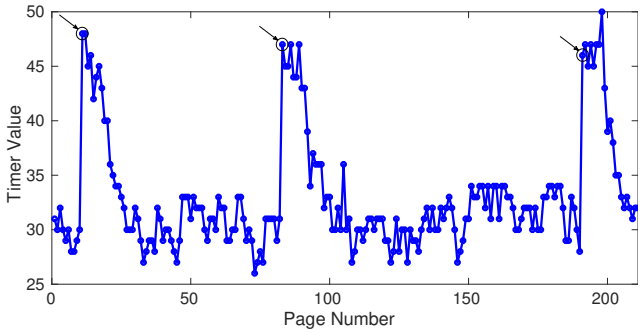


Figure 8: Reverse engineering physical page mappings in JavaScript. The markers point to addresses having same 20 bits of physical addresses being part of the same eviction set.

5 SPOILER from JavaScript

Microarchitectural attacks from JavaScript have a high impact as drive-by attacks in the browser can be accomplished without any privilege or physical proximity. In such attacks, co-location is automatically granted by the fact that the browser loads a website with malicious embedded JavaScript code. The browsers provide a sandbox where some instructions like `clflush` and `prefetch` and file systems such as `procfs` are inaccessible, limiting the opportunity for attack. Genkin et al. [14] showed that side-channel attacks inside a browser can be performed more efficiently and with greater portability through the use of *WebAssembly*.Yet, WebAssembly introduces an additional abstraction layer, i.e. it emulates a 32-bit environment that translates the internal addresses to virtual addresses of the host process (the browser). WebAssembly only uses addresses of the emulated environment and similar to JavaScript, it does not have direct access to the virtual addresses. Using SPOILER from JavaScript opens the opportunity to puncture these abstraction layers and to obtain physical address information directly. Figure 8 shows the address search in JavaScript using SPOILER. Compared to native implementations, we replace the `rdtscp` measurement with a timer based on a shared array buffer [19]. We cannot use any fence instruction such as `lfence`, and as a result, there remains some negligible noise in the JavaScript implementation. However, the aliased addresses can still be clearly seen, and we can use this information to improve the state-of-the art eviction set creation for both Rowhammer and cache attacks.

5.1 Efficient Eviction Set Finding

We use the algorithm proposed in [14]. It is a slight improvement to the former state-of-the-art brute force method [42] and consists of three phases:

- *expand*: A large pool of addresses P is allocated with the last twelve bits of all addresses being zero. A random address is picked as a witness t and tested against a candidate set C. If t is not evicted by C, it is added to C and a new witness will be picked. As soon as t gets evicted by C, C forms an eviction set for t.
- *contract*: Addresses are subsequently removed from the eviction set. If the set still evicts t, the next address is removed. If it does not evict t anymore, the removed address is added back to the eviction set. At the end of this phase, we have a minimal eviction set of the size of the set associativity.
- *collect*: All addresses mapping to the already found eviction set are removed from P by testing if they are evicted by the found set. After finding 128 initial cache sets, this approach utilizes the linearity property of the cache: For each found eviction set, the bits 6-11 are enumerated instead. This provides 63 more eviction sets for each found set, leading to full cache coverage.

We test this approach on an Intel Core i7-4770 with four physical cores and a shared 8MB 16-way L3 cache with Chromium 68.0.3440.106, Firefox 62 and Firefox Developer Edition 63. The approach yields an 80% accuracy rate to find all 8192 eviction sets when starting with a pool of 4096 pages. The entire eviction set creation process takes an average of $46\,s$. We improve the algorithm by **1)** using the addresses removed from the eviction set in the contract phase as a new candidate set and **2)** removing more than one address at a time from the eviction set during the contract phase. The improved eviction set creation process takes $35\,s$ on average.

5.1.1 Evaluation

The probability of finding a congruent address is $P(C) = 2^{\gamma-c-s}$, where c is the number of bits determining the cache set, γ is the number of bits attackers know, and s is the number of slices [56]. Since SPOILER allows us to control $\gamma \geq c$ bits, we are only left with uncertainty about a few address bits that influence the slice selection algorithm [26]. In theory, the eviction set search is sped up by a factor of 4096 by using aliased addresses in the pool, since on average one of 2^8 instead of one of 2^{20} addresses is an aliased address. Additionally, the address pool is much smaller, where 115 addresses are enough to find all the eviction sets. In native code, the overhead involved in finding the aliased addresses is negligible, less than a second in our experiments. However, in JavaScript, due to the noise, it takes 9s for finding aliased addresses and then 3s for eviction set as compared to the baseline of 46s for classic method in Table 2. Success rate however is 100% with SPOILER as compared to 80% for the classic method. Besides, success rate of the classical method can be affected by the availability and consumption of memory on the system.

From each aliased address pool, 4 eviction sets can be found (corresponding to the 4 slices which are the only unknown

Algorithm	R	t_{total}	t_{AAS}	t_{ESS}	Success
Classic [42]	3	46s	-	100%	80%
Improved [14]	3	35s	-	100%	80%
AA (ours)	10	10s	54%	46%	67%
AA (ours)	20	12s	75%	25%	100%

Table 2: Comparison of different eviction set finding algorithms on an Intel Core i7-4770. Classic is the method from [42], improved is the same method with slight improvement, *Aliased Address (AA)* uses SPOILER. t_{AAS} is the time percentage used for finding aliased addresses. t_{ESS} is the time percentage for finding eviction sets. R is the number of Rounds.

part in the mapping). These can be enumerated again to form 63 more eviction sets since we still kept the bits 6-11 fixed. To accomplish full cache coverage, the aliased address pool has to be constructed 32 times. The SPOILER variant for finding eviction sets is more susceptible to system noise, which is why it needs more repetitions i.e. R rounds to get reliable values. On the other hand, it is less prone to values deviating largely from the mean, which is a problem in the classic eviction set creation algorithm. The classic method does not succeed about one out of five times in our experiments, as shown in Table 2. The unsuccessful attempts occur due to aborts if the algorithm takes much longer than statistically expected. As a result, SPOILER can be incorporated in an end-to-end attack such as drive-by key-extraction cache attacks by Genkin et al. [14]. SPOILER increases both speed and reliability of the eviction set finding and therefore the entire attack.

6 Rowhammer Attack using SPOILER

To perform a Rowhammer attack, the adversary needs to efficiently access DRAM rows adjacent to a victim row. In a single-sided Rowhammer attack, only one row is activated repeatedly to induce bit flips on one of the nearby rows. For this purpose, the attacker needs to make sure that multiple virtual pages co-locate on the same bank. The probability of co-locating on the same bank is low without the knowledge of physical addresses and their mapping to memory banks. In a double-sided Rowhammer attack, the attacker tries to access two different rows $n+1$ an $n-1$ to induce bit flips in the row n placed between them. While double-sided Rowhammer attacks induce bit flips faster due to the extra charge on the nearby cells of the victim row n, they further require access to contiguous memory pages. In this section, we show that SPOILER can help boosting both single and double-sided Rowhammer attacks by its additional 8-bit physical address information and resulting detection of contiguous memory.

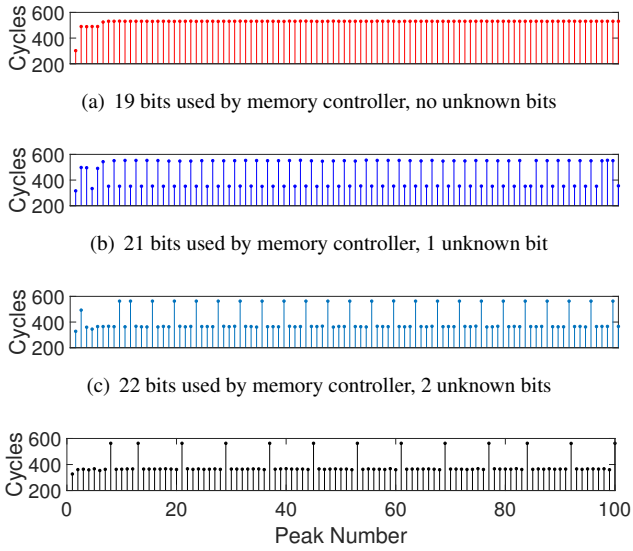
System Model	DRAM Configuration	# of Bits
Dell XPS-L702x (Sandy Bridge)	1 x (4GB 2Rx8)	21
	2 x (4GB 2Rx8)	22
Dell Inspiron-580 (Nehalem)	1 x (2GB 2Rx8) **(b)**	21
	2 x (2GB 2Rx8) **(c)**	22
	4 x (2GB 2Rx8) **(d)**	23
Dell Optiplex-7010 (Ivy Bridge)	1 x (2GB 1Rx8) **(a)**	19
	2 x (2GB 1Rx8)	20
	1 x (4GB 2Rx8) **(e)**	21
	2 x (4GB 2Rx8)	22

Table 3: Reverse engineering the DRAM memory mappings using DRAMA tool, *# of Bits* represents the number of physical address bits used for the bank, rank and channel [45].

6.1 DRAM Bank Co-location

DRAMA [45] reverse engineered the memory controller mapping. This requires elevated privileges to access physical addresses from the `pagemap` file. The authors have suggested that prefetch side-channel attacks [17] may be used to gain physical address information instead. SPOILER is an alternative way to obtain partial address information and is still feasible when the `prefetch` instruction is not available, e.g. in JavaScript. In our approach, we use SPOILER to detect aliased virtual memory addresses where the 20 LSBs of the physical addresses match. The memory controller uses these bits for mapping the physical addresses to the DRAM banks [45]. Even though the memory controller may use additional bits, the majority of the bits are known using SPOILER. An attacker can directly hammer such aliased addresses to perform a more efficient single-sided Rowhammer attack with a significantly increased probability of hitting the same bank. As shown in Table 3, we reverse engineer the DRAM mappings for different hardware configurations using the DRAMA tool, and only a few bits of physical address entropy beyond the 20 bits will remain unknown.

To verify if our aliased virtual addresses co-locate on the same bank, we use the row conflict side channel as proposed in [13] (timings in the appendix, Section 10.2). We observe that whenever the number of physical address bits used by the memory controller to map data to physical memory is equal to or less than 20, we always hit the same bank. For each additional bit the memory controller uses, the probability of hitting the same bank is divided by 2 as there is one more bit of entropy. In general, we can formulate that our probability p to hit the same bank is $p = 1/2^n$, where n is the number of unknown physical address bits in the mapping. We experimentally verify the success rate for the setups listed in Table 3, as depicted in Figure 9. In summary, SPOILER drastically improves the efficiency of finding addresses mapping to the same bank without administrative privilege or reverse engineering the memory controller mapping.



(a) 19 bits used by memory controller, no unknown bits

(b) 21 bits used by memory controller, 1 unknown bit

(c) 22 bits used by memory controller, 2 unknown bits

(d) 23 bits used by memory controller, 3 unknown bits

Figure 9: Bank co-location for various DRAM configurations (a), (b), (c) & (d) from Table 3. The regularity of the peaks shows that the allocated memory was contiguous, which is coincidental.

6.2 Contiguous Memory

For a double-sided Rowhammer attack, we need to hammer rows adjacent to the victim row in the same bank. This requires detecting contiguous memory pages in the allocated memory, since the rows are written to the banks sequentially. Without contiguous memory, the banks will be filled randomly and we will not be able to locate neighboring rows. We show that an attacker can use SPOILER to detect contiguous memory using 1 MB aliasing peaks. For this purpose, we compare the physical frame numbers to the SPOILER leakage for 10000 different virtual pages allocated using `malloc`. Figure 10 shows the relation between 1 MB aliasing peaks and physical page frame numbers. When the distance between the peaks is random, the trend of frame numbers also change randomly. After around 5000 pages, we observe that the frame numbers increase sequentially. The number of pages between the peaks remains constant at 256 where this distance comes from the 8 bits of physical address leakage due to 1 MB aliasing.

We also compare the accuracy of obtaining contiguous memory detected by SPOILER by analyzing the actual physical addresses from the `pagemap` file. By checking the difference between physical page numbers for each detected virtual page, we can determine the accuracy of our detection method: the success rate for finding contiguous memory is above 99% disregarding the availability of the contiguous pages. For detailed experiment on the availability of the contiguous pages, see Section 10.3 in the appendix.

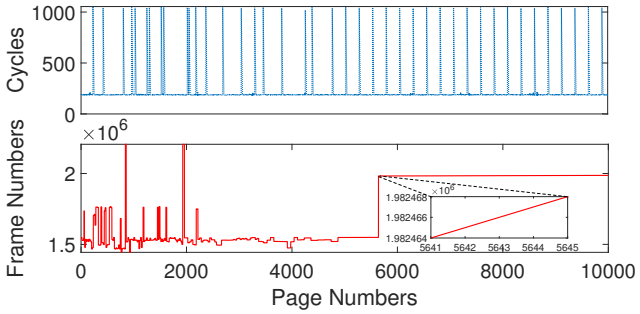


Figure 10: Relation between leakage peaks and the physical page numbers. The dotted plot shows the leakage peaks from SPOILER. The solid plot shows the decimal values of the physical frame numbers from the `pagemap` file. Once the peaks in the dotted plot become regular, the solid plot is linearly increasing, which shows contiguous memory allocation.

6.3 Double-Sided Rowhammer with SPOILER

As double-sided Rowhammer attacks are based on the assumption that rows within a bank are contiguous, we mount a practical double-sided Rowhammer attack on several DRAM modules using SPOILER without any root privileges. First, we use SPOILER to detect a suitable amount of contiguous memory. If enough contiguous memory is available in the system, SPOILER finds it, otherwise a double-sided Rowhammer attack is not feasible. In our experiments, we empirically configure SPOILER to detect 10 MB of contiguous memory. Second, we apply the row conflict side channel only to the located contiguous memory, and get a list of virtual addresses which are contiguously mapped within a bank. Finally, we start performing a double-sided Rowhammer attack by selecting 3 consecutive addresses from our list. While we have demonstrated the bit flips in our own process, we can free that memory which can then be assigned to a victim process by using previously known techniques like spraying and memory waylaying [16]. As the bit flips are highly reproducible, we can again flip the same bits in the victim process to demonstrate a full attack. Table 4 shows some of the DRAM modules susceptible to Rowhammer attack.

The native version of Rowhammer in this work is also applicable in JavaScript. The JavaScript-only variant implementation of Rowhammer by Gruss et al. [18], named rowhammer.js[5], can be combined with SPOILER to implement an end-to-end attack. In the original rowhammer.js, 2MB huge pages were assumed to get a contiguous chunk of physical memory. With SPOILER, this assumption is no longer required as explained in Section 6.3.

Figure 11 shows the number of hammers compared to the the amount of bit flips for configuration (e) in Table 3. We

[5] https://github.com/IAIK/rowhammerjs

DRAM Model	Architecture	Flippy
M378B5273DH0-CK0	Ivy Bridge	✓
M378B5273DH0-CK0	Sandy Bridge	✓
M378B5773DH0-CH9	Sandy Bridge	✓
M378B5173EB0-CK0	Sandy Bridge	×
NT2GC64B88G0NF-CG	Sandy Bridge	×
KY996D-ELD	Sandy Bridge	×
M378B5773DH0-CH9	Nehalem	✓
NT4GC64B8HG0NS-CG	Sandy Bridge	×
HMA41GS6AFR8N-TF	Skylake	×

Table 4: DRAM modules susceptible to double-sided Rowhammer attack using SPOILER.

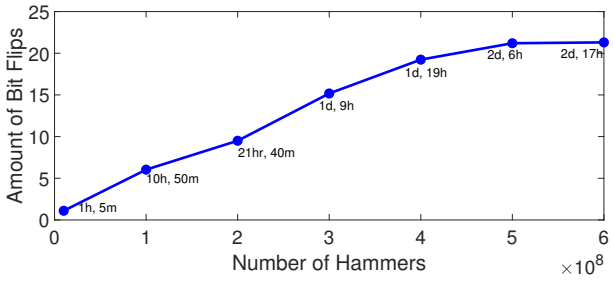


Figure 11: Amount of bit flips increases with the increase in number of hammerings. The timings do not include the time taken for reboots and 1 minute sleep time.

repeat this experiment 30 times for every measurement and the results are then averaged out. On every experiment, the system is rebooted using a script because once the memory becomes fragmented, no more contiguous memory is available. The number of bit flips increases with more number of hammerings. Hammering for 500 million times is found to be an optimal number for this DRAM configuration, as the continuation of hammering is not increasing bit flips.

7 Tracking Speculative Loads With SPOILER

Single-threaded attacks can be used to steal information from other security contexts running before/after the attacker code on the same thread [8, 41]. Example scenarios are I) context switches between processes of different users, or II) between a user process and a kernel thread, and III) Intel Software Guard eXtensions (SGX) secure enclaves [41, 54]. In such attacks, the adversary puts the microarchitecture to a particular state, waits for the context switch and execution of the victim thread, and then tries to observe the microarchitectural state after the victim's execution. We propose an attack where the adversary **1)** fills the store buffer with arbitrary addresses, **2)** issues the victim context switch and lets the victim perform a secret-dependent memory access, and **3)** measures the execution time of the victim. Any correlation between the victim's

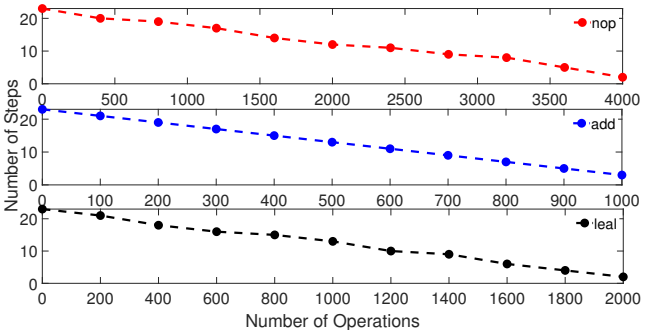


Figure 12: The depth of SPOILER leakage with respect to different instructions and execution units.

timing and the load address can leak secrets [61]. Due to the nature of SPOILER, the victim should access the memory while there are aliased addresses in the store buffer, i.e. if the `stores` are committed before the victim's speculative load, there will be no dependency resolution hazard.

We first perform an analysis of the depth of the operations that can be executed between the `stores` and the `load` to investigate the viability of SPOILER. In this experiment, we repeat a number of instructions between `stores` and the `load` that are free from memory operations. Figure 12 shows the number of stall steps due to the dependency hazard with the added instructions. Although `nop` is not supposed to take any cycle, adding 4000 `nop` will diffuse the timing latency. Then, we test `add` and `leal`, which use the Arithmetic Logic Unit (ALU) and the Address Generation Unit (AGU), respectively. Figure 12 shows that only 1000 `adds` can be executed between the `stores` and `load` before the SPOILER effect is lost. Since each `add` typically takes about 1 cycle to execute, this roughly gives a 1000 cycle depth for SPOILER. Considering the observed depth, we discuss potential attacks that can track the speculative load in the following two scenarios.

7.1 SPOILER Context Switch

In this attack, we are interested in tracking a memory access in the privileged kernel environment after a context switch. First, we fill the store buffer with addresses that have the same page offset, and then execute a system call. During the execution of the system call, we expect to observe a delayed execution if a secret `load` address has aliasing with the `stores`. We utilize SPOILER to iterate over various virtual pages, thus some of the pages have more noticeable latency due to the 1 MB aliasing. We analyze multiple `syscalls` with various execution times. For instance, Figure 13 shows the execution time for `mincore`. In the first experiment (red/1 MB Conflict), we fill the store buffer with addresses that have aliasing with a memory `load` operation in the kernel code space. The 1 MB aliasing delay with 7 steps suggests that we can track the

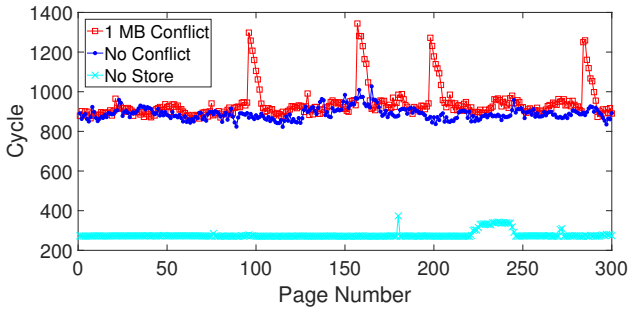


Figure 13: Execution time of `mincore` system call. When a kernel `load` address has aliasing with the attacker's `stores` (red/1MB Conflict), the step-wise delay will appear. These timings are measured with Kernel Page Table Isolation disabled.

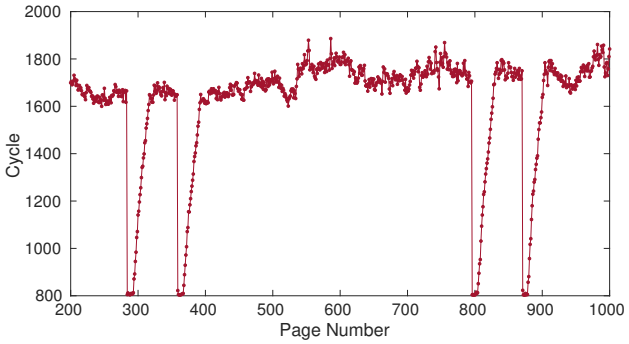


Figure 14: The effect of SPOILER on TLB flush. The execution cycle always increases for 4 kB aliased addresses, except for some of the virtual pages inside in the store buffer where we observe step-wise hills.

address of a kernel memory `load` by the knowledge of our arbitrary filled store addresses. The blue (No Conflict) line shows the timing when there is no aliasing between the target memory `load` and the attackers `store`. Surprisingly, only by filling the store buffer, the system call executes much slower: the normal execution time for `mincore` should be around 250 cycles (cyan/No Store). This proof of concept shows that SPOILER can be used to leak information from more privileged contexts, however this is limited only to `loads` that appear at the beginning of the next context.

7.2 Negative Result: SPOILER SGX

In this experiment, we try to combine SPOILER with the *CacheZoom* [41] approach to create a novel single-threaded side-channel attack against SGX enclaves with high temporal and spatial resolution (4-byte) [40]. We use *SGX-STEP* [53] to precisely interrupt every single instruction. *Nemesis* [54] shows that the interrupt handler context switch time is dependent on the execution time of the currently running instruction. On our test platform, *Core i7-8650U*, each context switch on an enclave takes about 12000 cycles to execute. If we fill the store buffer with memory addresses that match the page offset of a `load` inside the enclave in the interrupt handler, the context switch timing is increased to about 13500 cycles. While we cannot observe any correlation between the matched 4 kB or 1 MB aliased addresses, we do see unexpected periodic downward peaks with a similar step-wise behavior as SPOILER(Figure 14). We later reproduce a similar behavior by running SPOILER before an `ioctl` routine that flushes the TLB on each call. Intel SGX also performs an implicit TLB flush during each context switch. We can thus infer that the downward peaks occur due to the TLB flush, especially since the addresses for the downward peaks do not have any address correlation with the `load` address. This suggests that the TLB flush operation itself is affected by SPOILER. This effect eliminates the opportunity to observe any potential correlation due to the speculative load. As a result, we can not use SPOILER to track memory accesses inside an enclave. Further exploration of the root cause of the TLB flush effect can be carried out as a future work.

8 Mitigations

Software Mitigations The attack exploits the fact that when there is a `load` instruction after a number of `store` instructions, the physical address conflict causes a high timing behavior. This happens because of the speculatively executed `load` before all the `stores` are finished executing. There is no software mitigation that can completely erase this problem. While the timing behavior can be removed by inserting store fences between the `loads` and `stores`, this cannot be enforced to the user's code space, i.e., the user can always leak the physical address information. Another yet less robust approach is to execute other instructions between the `loads` and `stores` to decrease the depth of the attack. However, both of the approaches are only applicable to defend against attacks such as the one described in Section 7.

As for most attacks on JavaScript, removing accurate timers from the browser would be effective against SPOILER. Indeed, some timers have been removed or distorted by jitters as a response to attacks [35]. There is however a wide range of timers with varying precision available, and removing all of them seems impractical [13, 46].

When it is not possible to mitigate the microarchitectural attacks, developers can use dynamic tools to at least detect the presence of such leakage [6, 9, 63]. One of the dynamic approaches is gained by monitoring hardware performance counters in real-time. As explained in Section 4.1, two of the counters `Ld_Blocks_Partial:Address_Alias` and `Cycle_Activity:Stalls_Ldm_Pending` have high correlations with the leakage.

Hardware Mitigations The hardware design for the memory disambiguator may be revised to prevent such physical address leakage, but modifying the speculative behavior may cause performance impacts. For instance, partial address comparison was a design choice for performance. Full address comparison may address this vulnerability, but will also impact performance. Moreover, hardware patches are difficult to be applied to legacy systems and take years to be deployed.

9 Conclusion

We introduced SPOILER, a novel approach for gaining physical address information by exploiting a new information leakage due to speculative execution. To exploit the leakage, we used the speculative load behavior after jamming the store buffer. SPOILER can be executed from user space and requires no special privileges. We exploited the leakage to reveal information on the 8 least significant bits of the physical page number, which are critical for many microarchitectural attacks such as Rowhammer and cache attacks. We analyzed the causes of the discovered leakage in detail and showed how to exploit it to extract physical address information.

Further, we showed the impact of SPOILER by performing a highly targeted Rowhammer attack in a native user-level environment. We further demonstrated the applicability of SPOILER in sandboxed environments by constructing efficient eviction sets from JavaScript, an extremely restrictive environment that usually does not grant any access to physical addresses. Gaining even partial knowledge of the physical address will make new attack targets feasible in browsers even though JavaScript-enabled attacks are known to be difficult to realize in practice due to the limited nature of the JavaScript environment. Broadly put, the leakage described in this paper will enable attackers to perform existing attacks more efficiently, or to devise new attacks using the novel knowledge. The source code for SPOILER is available on GitHub[6].

Responsible Disclosure We informed the *Intel Product Security Incident Response Team* (iPSIRT) of our findings. iPSIRT thanked for reporting the issue and for the coordinated disclosure. iPSIRT then released the public advisory and CVE. Here is the time line for the responsible disclosure:

- **12/01/2018:** We informed our findings to iPSIRT.
- **12/03/2018:** iPSIRT acknowledged the receipt.
- **04/09/2019:** iPSIRT released public advisory (INTEL-SA-00238) and assigned CVE (CVE-2019-0162).

Acknowledgments

We thank Yuval Yarom, our shepherd Eric Wustrow and the anonymous reviewers for their valuable comments for improving the quality of this paper.

This work is supported by U.S. Department of State, Bureau of Educational and Cultural Affairs' Fulbright Program and National Science Foundation under grant CNS-1618837 and CNS-1814406. We also thank Cloudflare for their generous gift to support our research.

[6] https://github.com/UzL-ITS/Spoiler

References

[1] Jeffery M Abramson, Haitham Akkary, Andrew F Glew, Glenn J Hinton, Kris G Konigsfeld, and Paul D Madland. Method and apparatus for performing a store operation, April 23 2002. US Patent 6,378,062.

[2] Jeffrey M Abramson, Haitham Akkary, Andrew F Glew, Glenn J Hinton, Kris G Konigsfeld, Paul D Madland, David B Papworth, and Michael A Fetterman. Method and apparatus for dispatching and executing a load operation to memory, February 10 1998. US Patent 5,717,882.

[3] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. Anvil: Software-based protection against next-generation rowhammer attacks. *ACM SIGPLAN Notices*, 51(4):743–755, 2016.

[4] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "ooh aah... just a little bit" : A small amount of side channel can go a long way. In *Cryptographic Hardware and Embedded Systems – CHES 2014*, pages 75–92, Berlin, Heidelberg, 2014. Springer.

[5] Daniel J Bernstein. Cache-timing attacks on aes, 2005.

[6] Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. Cacheshield: Detecting cache attacks through self-observation. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, CODASPY '18, pages 224–235, New York, NY, USA, 2018. ACM.

[7] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. *arXiv preprint arXiv:1811.05441*, 2018.

[8] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre attacks: Stealing intel secrets from sgx enclaves via speculative execution. *arXiv preprint arXiv:1802.09085*, 2018.

[9] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing*, 49:1162–1174, 2016.

[10] Jack Doweck. Inside intel® core microarchitecture. In *Hot Chips 18 Symposium (HCS), 2006 IEEE*, pages 1–35. IEEE, 2006.

[11] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, pages 40:1–40:13, Piscataway, NJ, USA, 2016. IEEE Press.

[12] Agner Fog. The microarchitecture of intel, amd and via cpus: An optimization guide for assembly programmers and compiler makers. *Copenhagen University College of Engineering*, pages 02–29, 2012.

[13] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand pwning unit: Accelerating microarchitectural attacks with the gpu. In *Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU*, page 0, Washington, DC, USA, 2018. IEEE, IEEE Computer Society.

[14] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by key-extraction cache attacks from portable code. In *International Conference on Applied Cryptography and Network Security*, pages 83–102. Springer, 2018.

[15] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall, London, 2004.

[16] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another flip in the wall of rowhammer defenses. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 245–261. IEEE, 2018.

[17] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 368–379, New York, NY, USA, 2016. ACM.

[18] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in javascript. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 300–321. Springer, 2016.

[19] Lars T Hansen. Shared memory: Side-channel information leaks, 2016.

[20] Sebastien Hily, Zhongying Zhang, and Per Hammarlund. Resolving false dependencies of speculative load instructions, October 13 2009. US Patent 7,603,527.

[21] Jann Horn. speculative execution, variant 4: speculative store bypass, 2018.

[22] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205. IEEE, 2013.

[23] Mehmet Sinan İnci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery on the cloud. In *Cryptographic Hardware and Embedded Systems – CHES 2016*, pages 368–388, Berlin, Heidelberg, 2016. Springer.

[24] Intel. Intel® 64 and IA-32 Architectures Optimization Reference Manual.

[25] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$a: A shared cache attack that works across cores and defies vm sandboxing – and its application to aes. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 591–604, Washington, DC, USA, 2015. IEEE Computer Society.

[26] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic reverse engineering of cache slice selection in intel processors. In *2015 Euromicro Conference on Digital System Design (DSD)*, pages 629–636. IEEE, 2015.

[27] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 380–392. ACM, 2016.

[28] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. ret2dir: Rethinking kernel isolation. In *USENIX Security Symposium*, pages 957–972, 2014.

[29] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 361–372. IEEE Press, 2014.

[30] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, January 2018.

[31] Steffen Kosinski, Fernando Latorre, Niranjan Cooray, Stanislav Shwartsman, Ethan Kalifon, Varun Mohandru, Pedro Lopez, Tom Aviram-Rosenfeld, Jaroslav Topp, Li-Gao Zei, et al. Store forwarding for data caches, November 29 2016. US Patent 9,507,725.

[32] Evgeni Krimer, Guillermo Savransky, Idan Mondjak, and Jacob Doweck. Counter-based memory disambiguation techniques for selectively predicting load/store conflicts, October 1 2013. US Patent 8,549,263.

[33] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical keystroke timing attacks in sandboxed javascript. In *Computer Security – ESORICS 2017*, pages 191–209. Springer, 2017.

[34] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, Austin, TX, 2016. USENIX Association.

[35] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, 2018. USENIX Association.

[36] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 605–622, Washington, DC, USA, 2015. IEEE Computer Society.

[37] Errol L. Lloyd and Michael C. Loui. On the worst case performance of buddy systems. *Acta Informatica*, 22(4):451–473, Oct 1985.

[38] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2109–2122. ACM, 2018.

[39] Intel 64 Architecture Memory Ordering White Paper. http://www.cs.cmu.edu/~410-f10/doc/Intel_Reordering_318147.pdf, 2008. Accessed: 2018-11-26.

[40] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. Memjam: A false dependency attack against constant-time crypto implementations in SGX. In *Topics in Cryptology - CT-RSA 2018 - The Cryptographers' Track at the RSA Conference 2018, San Francisco, CA, USA, April 16-20, 2018, Proceedings*, pages 21–44, 2018.

[41] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How sgx amplifies the power of cache attacks. In *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 69–90. Springer, 2017.

[42] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1406–1418, New York, NY, USA, 2015. ACM.

[43] Colin Percival. Cache missing for fun and profit, 2005.

[44] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. "make sure dsa signing exponentiations really are constant-time". In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1639–1650, New York, NY, USA, 2016. ACM.

[45] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. Drama: Exploiting dram addressing for cross-cpu attacks. In *USENIX Security Symposium*, pages 565–581, 2016.

[46] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: high-resolution microarchitectural attacks in javascript. In *International Conference on Financial Cryptography and Data Security*, pages 247–267. Springer, 2017.

[47] Mark Seaborn and Thomas Dullien. Exploiting the dram rowhammer bug to gain kernel privileges. *Black Hat*, 15, 2015.

[48] Julian Stecklina and Thomas Prescher. Lazyfp: Leaking fpu register state using microarchitectural side-channels. *arXiv preprint arXiv:1806.07480*, 2018.

[49] Dean Sullivan, Orlando Arias, Travis Meade, and Yier Jin. Microarchitectural minefields: 4k-aliasing covert channel and multi-tenant detection in iaas clouds. In *Network and Distributed Systems Security (NDSS) Symposium*. The Internet Society, 2018.

[50] Andrei Tatar, Radhesh Krishnan, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer attacks over the network and defenses. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, Boston, MA, 2018. USENIX Association.

[51] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data with papi-c. In *Tools for High Performance Computing 2009*, pages 157–173. Springer, 2010.

[52] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium. USENIX Association*, 2018.

[53] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Sgx-step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2Nd Workshop on System Software for Trusted Execution*, SysTEX'17, pages 4:1–4:6, New York, NY, USA, 2017. ACM.

[54] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 178–195. ACM, 2018.

[55] Victor Van Der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1675–1689. ACM, 2016.

[56] Pepe Vila, Boris Köpf, and José Francisco Morales. Theory and practice of finding eviction sets. *arXiv preprint arXiv:1810.01497*, 2018.

[57] WikiChip. Ivy Bridge - Microarchitectures - Intel. https://en.wikichip.org/wiki/intel/microarchitectures/ivy_bridge_(client). Accessed: 2019-02-05.

[58] WikiChip. Kaby Lake - Microarchitectures - Intel. https://en.wikichip.org/wiki/intel/microarchitectures/kaby_lake. Accessed: 2019-02-05.

[59] WikiChip. Skylake (client) - Microarchitectures - Intel. https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client). Accessed: 2019-02-05.

[60] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation. In *USENIX Security Symposium*, pages 19–35, 2016.

[61] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: a timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017.

[62] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93. IEEE, 2009.

[63] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. Cloudradar: A real-time side-channel attack detection system in clouds. In *Research in Attacks, Intrusions, and Defenses*, pages 118–140. Springer, 2016.

10 Appendix

10.1 Tested Hardware Performance Counters

Counters	*Correlation*
UNHALTED_CORE_CYCLES	0.3077
UNHALTED_REFERENCE_CYCLES	0.1527
INSTRUCTION_RETIRED	0.2718
INSTRUCTIONS_RETIRED	0.2827
BRANCH_INSTRUCTIONS_RETIRED	0.3143
MISPREDICTED_BRANCH_RETIRED	0.0872
CYCLE_ACTIVITY:CYCLES_L2_PENDING	-0.0234
CYCLE_ACTIVITY:STALLS_LDM_PENDING	0.9819
CYCLE_ACTIVITY:CYCLES_NO_EXECUTE	0.2317
RESOURCE_STALLS:ROB	0
RESOURCE_STALLS:SB	-0.0506
RESOURCE_STALLS:RS	-0.0044
LD_BLOCKS_PARTIAL:ADDRESS_ALIAS	-0.9511
IDQ_UOPS_NOT_DELIVERED	-0.1455
IDQ:ALL_DSB_CYCLES_ANY_UOPS	0.0332
ILD_STALL:IQ_FULL	0.1021
ITLB_MISSES:MISS_CAUSES_A_WALK	0
TLB_FLUSH:STLB_THREAD	0
ICACHE:MISSES	0
ICACHE:IFETCH_STALL	0
L1D:REPLACEMENT	0.3801
L2_DEMAND_RQSTS:WB_HIT	0.2436
LONGEST_LAT_CACHE:MISS	0.0633
CYCLE_ACTIVITY:CYCLES_L1D_PENDING	-0.0080
LOCK_CYCLES:CACHE_LOCK_DURATION	0
LOAD_HIT_PRE:SW_PF	0
LOAD_HIT_PRE:HW_PF	0
MACHINE_CLEARS:CYCLES	0
OFFCORE_REQUESTS_BUFFER:SQ_FULL	0
OFFCORE_REQUESTS:DEMAND_DATA_RD	0.1765

Table 5: Counters profiled for correlation test

10.2 Row conflict Side Channel

The row conflict side channel retrieves the timing information of the CPU while doing direct accesses (using `clflush`) from the DRAM. A higher timing indicates that the two addresses are mapped to the same bank in the DRAM because reading an address from the same bank forces the row buffer to copy the previous contents back to the original row and then load the newly accessed data into the row buffer. Whereas, a low timings indicates that two addresses are not in the same bank (not sharing the same row buffer) and are loaded into separate row buffers. Figure 15 shows a wide gap (around 100 cycles) between row hits and row conflicts.

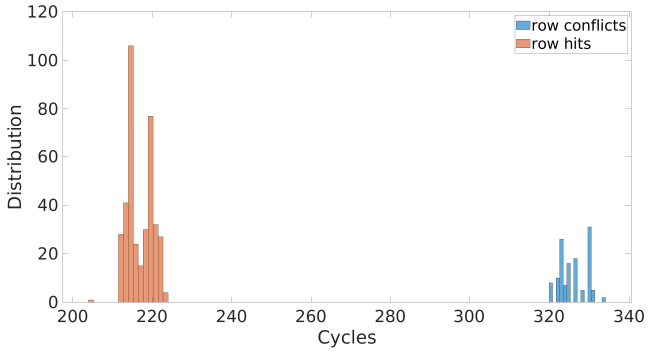


Figure 15: Timings for accessing the aliased virtual addresses (random addresses where 20 LSB of the physical address match). Row hits (orange/low timings) are clearly distinguishable from row conflicts (blue/high timings).

10.3 Memory Utilization and Contiguity

The probability of obtaining contiguous memory depends on memory utilization of the system. We conduct an experiment to examine the effect of memory utilization on availability of contiguous memory. In this experiment, 1 GB memory is allocated. During the experiment, the memory utilization of the system is increased gradually from 20% to 90%. We measure the probability of getting the contiguous memory with two methods. The first one is checking the physical frame numbers from `pagemap` file to look for 520 kB of contiguous memory. The second method is using SPOILER to find the 520 kB of contiguous memory. This 520 kB is required to get three consecutive rows within a bank for a DRAM configuration having 256 kB row offset and 8 kB row size.

Figure 16 and Figure 17 show that when the memory has been fragmented after intense memory usage, it gets more difficult to allocate a contiguous chunk of memory. Even decreasing the memory usage does not help to get a contiguous block of memory. Figure 17 depicts that after the memory utilization has been decreased from 70% to 60% and so on, there is not enough contiguous memory to mount a successful double-sided Rowhammer attack. Until the machine is restarted, the memory remains fragmented which makes a double-sided Rowhammer attack difficult, especially on targets like high-end servers where restarting is impractical.

The observed behavior can be explained by the *binary buddy allocator* which is responsible for the physical address allocation in the Linux OS [15]. This type of allocator is known to fragment memory significantly under certain circumstances [37]. The Linux OS uses a *SLAB/SLOB allocator* in order to circumvent the fragmentation problems. However, the allocator only serves the kernel directly. User space memory therefore still suffers from the fragmentation that the buddy allocator introduces. This also means that getting the contiguous memory required for a double-sided Rowhammer attack becomes more difficult if the system under attack has been active for a while.

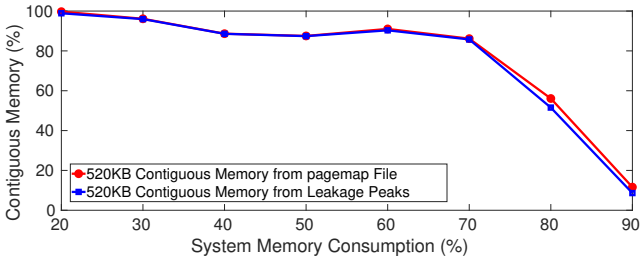


Figure 16: Finding contiguous memory of 520 kB with increasing memory utilization. The overlap between the red and blue plot indicates the high accuracy of the contiguous memory detection capability of SPOILER as verified by the `pagemap` file.

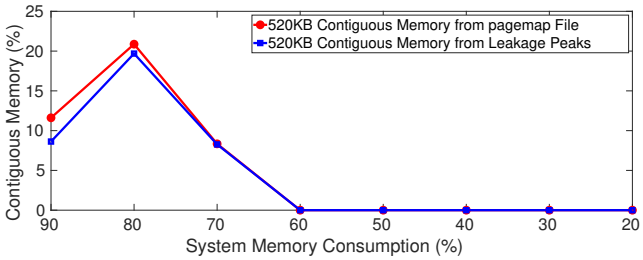


Figure 17: Finding contiguous memory of 520 kB with decreasing memory utilization.

Robust Website Fingerprinting Through the Cache Occupancy Channel

Anatoly Shusterman
Ben-Gurion University of the Negev
shustera@post.bgu.ac.il

Lachlan Kang
University of Adelaide
lachlan.kang@adelaide.edu.au

Yarden Haskal
Ben-Gurion Univ. of the Negev
yardenha@post.bgu.ac.il

Yosef Meltser
Ben-Gurion Univ. of the Negev
yosefmel@post.bgu.ac.il

Prateek Mittal
Princeton University
pmittal@princeton.edu

Yossi Oren
Ben-Gurion Univ. of the Negev
yos@bgu.ac.il

Yuval Yarom
University of Adelaide and Data61
yval@cs.adelaide.edu.au

Abstract

Website fingerprinting attacks, which use statistical analysis on network traffic to compromise user privacy, have been shown to be effective even if the traffic is sent over anonymity-preserving networks such as Tor. The classical attack model used to evaluate website fingerprinting attacks assumes an *on-path adversary*, who can observe all traffic traveling between the user's computer and the secure network.

In this work we investigate these attacks under a different attack model, in which the adversary is capable of sending a small amount of malicious JavaScript code to the target user's computer. The malicious code mounts a cache side-channel attack, which exploits the effects of contention on the CPU's cache, to identify *other* websites being browsed. The effectiveness of this attack scenario has never been systematically analyzed, especially in the open-world model which assumes that the user is visiting a mix of both sensitive and non-sensitive sites.

We show that cache website fingerprinting attacks in JavaScript are highly feasible. Specifically, we use machine learning techniques to classify traces of cache activity. Unlike prior works, which try to identify cache conflicts, our work measures the overall occupancy of the last-level cache. We show that our approach achieves high classification accuracy in both the open-world and the closed-world models. We further show that our attack is more resistant than network-based fingerprinting to the effects of response caching, and that our techniques are resilient both to network-based defenses and to side-channel countermeasures introduced to modern browsers as a response to the Spectre attack. To protect against cache-based website fingerprinting, new defense mechanisms must be introduced to privacy-sensitive browsers and websites. We investigate one such mechanism, and show that generating artificial cache activity reduces the effectiveness of the attack and completely eliminates it when used in the Tor Browser.

1 Introduction

Over the last decades the World Wide Web has grown from an academic exercise to a communication tool that encompasses all aspects of modern life. Users use the web to acquire information, manage their finances, conduct their social life, and more. This shift to the so called virtual life has resulted in new challenges to users' privacy. Monitoring the online behavior of users may reveal personal or sensitive information about the users, including information such as sexual orientation or political beliefs and affiliations.

Several tools have been developed to protect the online privacy of users and hide information about the websites they visit [18, 20, 71]. Prime amongst these is the Tor network [20], an overlay network of collaborating servers, called *relays*, that anonymously forward Internet traffic between users and web servers. Tor encrypts the network traffic of all of the users, and transmits it between relays in a way that prevents external observers from identifying the traffic of specific users. In addition to the network itself, the Tor Project also provides the *Tor Browser* [82], a modified version of the Mozilla Firefox web browser, that further protects users by disabling features that may allow web sites to track the users.

Past research has demonstrated that encrypting traffic is not sufficient for protecting the privacy of the users [10, 29, 35, 36, 37, 45, 46, 53, 60, 66, 67, 73, 88, 89, 93]. Observable patterns in the metadata of encrypted traffic, specifically, the size of the transmitted data, its direction, and its timing, may reveal the web page that the user is visiting. Applying such *website fingerprinting* techniques to Tor traffic results in a

success rate of over 90% in identifying the websites that a user visits over Tor [73].[1]

In this paper, we focus on an alternative attack model of exploiting micro-architectural side-channels, a less explored option for website fingerprinting. The attack model assumes a victim that visits a web site under the attacker's control. The web site monitors the state of the victim computer's cache, and uses that information to infer the victim's web activity in other tabs of the same browser, or even in other browsers.

Because the attack observes the internal state of the target PC, rather than the network traffic. It offers the potential of overcoming traffic shaping, often proposed as a defense for website fingerprinting [11, 12, 15, 63, 90]. Similarly, the attack may be applicable in scenarios where network-based fingerprinting is known to be less effective, such as when the browser caches the contents of the website [36].

We note that the malicious web site does not need to be fully under the control of the attacker. The attacker only needs to be able to inject JavaScript code via the web site to the victim's browser. This can be done, for example, through a malicious advertisement or pop-up window. Alternatively, documents released by former NSA contractor Edward Snowden indicate that some nation-state agencies have the operational capability to exploit this vector on a wide scale. In March 2013 the German magazine Der Spiegel reported on the existence of a tool called QUANTUMINSERT, which the GCHQ and the NSA could use to inject malicious code to any website [78]. The Der Spiegel claims that the GCHQ successfully used this tool to attack the computers of employees at the partly-government-held Belgian telecommunications company Belgacom, and that the NSA used the same technology to target high-ranking members of the Organization of the Petroleum Exporting Countries (OPEC) at the organization's Vienna headquarters. Finally, malicious advertisements are a viable option for injecting cache side-channel attacks to browsers [28].

For a small number of websites, under the closed-world model, Oren et al. [64] show the possibility of fingerprinting via malicious JavaScript code. However, beyond showing the ability to distinguish between a handful of websites, their work does not provide an analysis of the effectiveness of the technique. Furthermore, following the disclosure of the Spectre and the Meltdown attacks, which can also be potentially delivered via malicious JavaScript injection [48, 57], major vendors deployed defenses against browser-borne side-channel attacks. In particular, all modern browsers have reduced the resolution of the JavaScript time function, `performance.now()`, by several orders of magnitude [69, 87], making it difficult to tell apart cache hits and cache misses. Traditionally, cache attacks require high-resolution timers, and while mechanisms to generate such timers in web browsers have been published [31, 49, 76], it is not clear that these can be used for website fingerprinting.

Thus, in this paper we ask: *Are cache-based attacks a viable option for website fingerprinting?*

Our Contribution

We answer this question in the affirmative. We design and implement a cache-based website fingerprinting attack, and evaluate it in both the closed-world and the open-world models. We show that in both models our JavaScript-based attacker achieves high fingerprinting accuracy even when executed on modern mainstream browsers that include all recently introduced countermeasures for side-channel (Spectre) attacks. Even when taking these countermeasures to the extreme, as is done in the Tor Browser, our attack remains effective, although with a drop in accuracy.

Our attack consists of collecting traces of cache *occupancy* while the browser downloads and renders web sites. Adapting the techniques of Rimmer et al. [73], we use deep neural networks to analyze and to classify the collected traces. By focusing on cache occupancy rather than on activity within specific cache sets, our attack avoids the need for high resolution timers required by prior cache-based attacks. Furthermore, because our technique does not depend on the layout of the cache, it can overcome proposed countermeasures that randomize the cache layout [58, 70, 91].

We investigate the source of the information in the cache occupancy traces and show that they contain information from both the networking activity and the rendering activity of the browser. Using information from the rendering activity allows our attack to remain effective even in scenarios that thwart network-based fingerprinting, such as when the browser retrieves data from its response cache and not from the network, or when the network traffic is shaped.

Finally, we investigate a potential countermeasure that introduces a high level of activity into the last level cache. We show that the countermeasure reduces the success rate of the attack. In particular, the noise completely masks the activity of the Tor Browser, reducing the attack accuracy to that of a random guess. This countermeasure results in a mean slowdown of 5% for CPU benchmarks, which we consider reasonable when visiting privacy-sensitive web sites.

More specifically, we make the following contributions:

- We design and implement the cache occupancy side-channel attack, a cache-based side channel attack technique which can operate with the low timer resolution supported in modern JavaScript engines. Our attacks only require a sampling rate six orders of magnitude *lower* than required for the prior attacks of Oren et al. [64] (Section 4).

[1] *Website fingerprinting* is a misnomer. Fingerprinting identifies individual web pages rather than sites. Following this misnomer, in this work we use the term *website* to refer to specific pages, typically the homepage of the site.

- We evaluate the use of two machine learning techniques, CNN and LSTM, for fingerprinting websites based on the cache activity traces collected while loaded by the browsers (Section 5).
- We show that cache-based fingerprinting has high accuracy in both the closed- and the open-world models, under a variety of operating systems and browsers (Section 6).
- We evaluate both fingerprinting methods without deleting the browser response cache, and show that while the accuracy of network-based fingerprinting drops significantly, the accuracy of cache-based fingerprinting is not affected (Section 7.3).
- We show that cache-based fingerprints contain information both from the network activity and from the rendering activity of the target device. Therefore, cache-based fingerprinting maintains a high accuracy even in the presence of traffic molding countermeasures which force a constant bit rate on network traffic (Section 7.4).
- We design and evaluate a countermeasure that introduces noise in the cache. The countermeasure is applicable from both native code and from JavaScript, completely blocks the attack on the Tor Browser, and only causes a small performance degradation on CPU-bound workloads (Section 9).

2 Background

2.1 Tor

Tor [20], is a collection of collaborating servers called *relays*, designed to provide privacy for network communication. Tor aims to protect users from *on-path* adversaries that can observe the network traffic. In this scenario, a user uses a PC to browse the web, and an adversary positioned between the user's PC and the destination web server captures the information that the user exchanges with the web server.

A common protection for such an attack model is to use encryption, e.g., using protocols such as TLS [19] which underlies the security of the HTTPS scheme [72]. However, this solution only protects the contents of the communication, leaving the identity of the communicating parties exposed to the adversary. Knowing that users merely connected to a certain sensitive website may be enough to incriminate them, even if the actual data exchanged over the secure connection is not known. This risk became a reality in 2016, as tens of thousands of individuals were persecuted by the Turkish government for accessing the domain `bylock.net` [50].

The main aim of Tor is thus to protect the identity of the communicating parties. Tor achieves this protection by forwarding the users' communication through a *circuit* consisting of a few (typically three) Tor relays. The user encrypts the network traffic with multiple layers of encryption, and each relay in the circuit decrypts a successive layer to find out where to forward the traffic. See Dingledine et al. [20] for further information.

2.2 Website Fingerprinting Attacks and Defences

In the conventional attack model of a network-level attacker, much previous work has demonstrated the ability of an adversary to make probabilistic inferences about users' communications via statistical analysis, even if these communications are in their encrypted form. These works have investigated both the selection of features (such as packet sizes, packet timings, direction of communication), as well as the design of classifiers (such as support vector machines, random forests, Naive Bayes) to make accurate predictions [10, 29, 35, 36, 37, 45, 46, 53, 60, 66, 67, 73, 88, 89, 93]. In response, several defense mechanisms have been proposed in the literature [11, 12, 15, 63, 90]. The common idea behind these defenses is to inject random delays and spurious cover traffic to perturb the traffic features and therefore obfuscate users' communications. A common point of all of these defenses is a typical trade-off between latency/bandwidth and privacy, and thus they face deployment hurdles. Rimmer et al. [73] have recently proposed a family of classifiers based on deep learning algorithms such as SDAE, CNN and LSTM, which operate on the raw network traces and are therefore less sensitive to ad-hoc defenses against particular traffic features.

2.3 Cache Side-Channel Attacks

When programs execute on a processor, they share the use of micro-architectural components such as the cache. This sharing may result in unintended communication channels, often called *side channels*, between programs [27, 39], which may be used to leak secret information. In particular, cache-based attacks, which exploit contention on one of the processor's caches, can leak secrets such as cryptographic keys [4, 26, 65, 68, 83], keystrokes [32], address layout [23, 31, 33], etc.

Cache Operation. Caches bridge the speed gap between the faster processor and the slower memory. The cache is a small bank of memory, which stores the contents of recently accessed memory locations. Most caches in modern processors are *set associative*. The cache is divided into partitions called *sets*. Each memory location maps to a single set and can only be cached in the set it maps to. When the processor needs to access a specific memory location, it successively searches in a hierarchy of caches. In a *cache hit*, when the contents of the required address is found in the cache, access is performed on the cached contents. Otherwise, in a *cache*

miss, the process repeats on the next cache level. A miss on the last-level cache (LLC) results in a time-consuming access to the RAM.

The Prime+Probe Technique. Past cache-based attacks from web browsers [28, 64] employ the *Prime+Probe* technique [65, 68], which exploits the set-associative structure. Each round of attack consists of three steps. In the first step, the cache is *primed*, i.e., the attacker completely fills some of the cache sets with its own data. The attacker then waits some time to allow the victim to execute. Finally, the attacker *probes* the cache by measuring the time it takes to access the previously-cached data in each of the sets. If the victim accesses memory locations that map to a monitored cache set, the victim's memory contents will replace the attacker contents in the cache. Hence, the attacker will need to retrieve the data from lower levels in the hierarchy, increasing the access time to its data. Prime+Probe has been used for attacks on data [65, 68] and instruction [3, 4] caches, as well as for attacks on the LLC [43, 59]. It has been shown practical in multiple settings, including across different virtual machines in cloud environments [40] and from mobile code [28, 64].

Countermeasures in JavaScript. The time difference between the latencies of a memory access and cache access is on the order of 0.1 μs. To distinguish between cache hits and misses, cache attacks typically require a high resolution timer. Following the publication of the first demonstration of a cache attack in JavaScript [64], some browsers started reducing the resolution of the timers they provide as a countermeasure for cache side channel attacks. This approach had become wide-spread after the disclosure of the Spectre attack [48], and now all mainstream browsers incorporate this countermeasure. Furthermore, while non-traditional timers in browsers have been identified [25, 49, 76], browsers and extensions have since disabled many of the features that allow sub-microsecond resolution [61, 69, 77]. An extreme case of this behavior can be found in the Tor Browser, which restricts the timer resolution to 100 ms, or 10 Hz.

Several of the previously discovered timers rely on browser features that are accessible from JavaScript. These are not accessible in environments such as Cloudflare Workers [7], which rely on the absence of high-resolution timers to protect against timing attacks [85].

2.4 Related Work

Several past works have looked at the possibility of performing website fingerprinting based on local side-channel information. In all of these works, which we survey in Table 1, the adversary observes some property of the system while the victim browser is rendering a webpage. The adversary then applies a machine learning classifier to the observed side-channel trace to identify the rendered website.[2] Some of these works assume that the adversary has malicious control over a hardware component or peripheral [16, 56, 94]. Others assume that the adversary can execute arbitrary native code on the target hardware [34, 44, 51, 80]. Yet others make the much more modest assumption that the adversary can induce the victim to render a webpage containing malicious JavaScript code [8, 47, 64, 86]. We mainly investigate the last model.

Kim et al. [47] abuse a data leak in the Chrome implementation of the Quota Management API, which has been since fixed. Our attack, in contrast, is based on a fundamental property of the CPU running the browser application, which is far less trivial to fix. (See Section 9.) Moreover, the mitigations put in place as part of the response to the Spectre and Meltdown disclosures make the high sampling rates exploited thus far [64, 86] unattainable in modern secure browsers. Our attack, in contrast, achieves high accuracy at drastically lower sampling rates and is capable of classifying a significant number of websites at sampling rates as low as 10 Hz. To the best of our knowledge, no cache attack that uses such low clock resolutions has been demonstrated.

In addition, Oren et al. [64] only recorded a small number of traces from a few popular websites, and did not investigate the effectiveness of cache-based fingerprinting in open-world contexts, or in scenarios where various anti-fingerprinting measures are in place. We address all of these shortcomings in this work. Furthermore, while Oren et al. [64] do target the Tor Browser, the attack code executes in a different mainstream browser. Unlike our work, they do not demonstrate an attack from JavaScript code running within the Tor Browser.

Booth [8] is able to classify a moderate amount of websites using a non-cache-based method with a millisecond clock. Their attack, however, saturates all of the victim's CPU cores with math-intensive worker threads, making it highly noticeable and easy to detect by the victim.

Cock et al. [17] implement a covert channel using an L1 cache occupancy channel. Ristenpart et al. [74] show that a cache occupancy channel can detect keystroke timing and network load in co-located virtual machines on cloud servers. Both use the technique with high resolution (sub nanosecond) timers. We are not aware of any prior use of the cache occupancy channel to overcome low resolution timers.

3 The Website Fingerprinting Attack Model

The classical attack model used to evaluate website fingerprinting attacks is presented in Figure 1. In this model, a targeted user uses a web browser to display a sensitive website. To protect their privacy, the user does not connect to the website directly, but instead uses a secure network, such as the Tor network, for the connection. The attacker is typ-

[2] A different but closely related class of attacks are "history sniffing" attacks, such as [54, 92], in which the attacker wishes to learn which websites the victim has visited in the **past**.

Table 1: Related work on website fingerprinting based on local side channels.

Work	Target	Side Channel	Attack Model	Sampling rate [Hz]
Clark et al., 2013 [16]	Chrome (Mac, Win, Linux)	Power consumption	Hardware	250000
Yang et al., 2017 [94]	Multiple smartphones	Power consumption	Hardware	200000
Lifshits et al., 2018 [56]	Android Browser, Chrome Android	Power consumption	Hardware	1000
Jana and Shmatikov, 2012 [44]	Chrome Linux, Firefox Linux, Android Browser (VM)	App memory footprint	Native code	100000
Lee et al., 2014 [51]	Chromium Linux, Firefox Linux	GPU memory leaks	Native code	N/A
Spreitzer et al., 2016 [80]	Chrome Android, Android Browser, Tor Android	Data-Usage Statistics	Native code	20–50
Gülmezoglu et al., 2017 [34]	Chrome Linux (Intel and ARM), Tor Linux	Performance counters	Native code	10000
Oren et al, 2015 [64]	Safari MacOS, Tor MacOS	Last-level cache	JavaScript	10^8
Booth, 2015 [8]	Chrome (Mac, Win, Linux), Firefox Linux	CPU activity	JavaScript	1000
Kim et al., 2016 [47]	Chromium Linux, Chrome (Win, Android)	Quota Management API	JavaScript	N/A
Vila and Köpf, 2017 [86]	Chromium Linux, Chrome Mac	Shared event loop	JavaScript	40000
This work	**Chrome (Win, Linux), Firefox (Win, Linux), Safari MacOS, Tor Linux**	**Last-level cache**	**JavaScript**	**10–500**

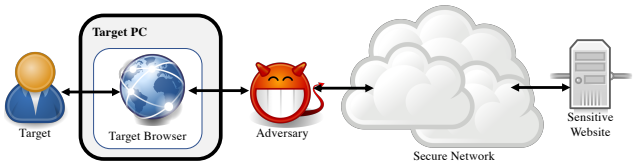


Figure 1: The classical website fingerprinting attack model. The (passive) adversary monitors the traffic between the target user and the secure network.

ically modeled as an *on-path adversary*, who is capable of observing all traffic entering and leaving the Tor network in the direction of the target user. The adversary cannot understand the contents of the network traffic since it is encrypted when it enters the Tor network. The adversary is furthermore unable to directly determine the ultimate destination of the communications after it exits the Tor network, thanks to Tor's routing protocol. Finally, due to the encryption and the validation of the Tor network, the attacker is unable to modify the traffic without terminating the connection. An important thread of research on the security of Tor has investigated the ability of such an adversary to perform statistical traffic analysis of encrypted traffic, and then to make probabilistic inferences about users' communications [10, 35, 36, 37, 45, 46, 53, 60, 66, 67, 73, 88, 89, 93].

Gong et al. [29] suggest a variation on this scheme, in which the attacker remotely probes routers to estimate the load of the network traffic they process and performs the statistical analysis based on this estimated traffic. Jansen et al. [45] suggest another variation in which the attacker monitors the traffic inside the Tor network, rather then monitoring traffic at the network's edge.

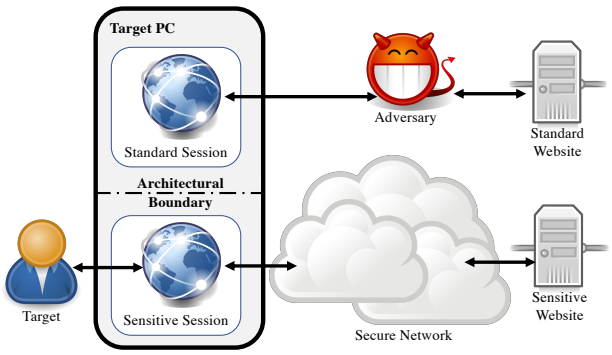


Figure 2: Remote cache-based website fingerprinting attack model. The remote attacker injects malicious JavaScript code into a browser running on the target machine.

In this work we discuss a different attack model, presented in Figure 2. In this model, the target user has two concurrent browsing sessions. In one session, the user browses to

an adversary-controlled site, which contains some malicious JavaScript code. In the other session, the user browses to some sensitive web site. Due to architectural boundaries, such as sandboxing or process isolation, the malicious code cannot directly observe the internal state of the sensitive session. Hence, the adversary cannot directly determine the ultimate destination of *any* communication issued from the sensitive session, even when the sensitive session is using a direct unencrypted connection to the remote server. The malicious code can, however, observe the micro-architectural state of the processor, and use this information to spy on the sensitive session.

Our attack can therefore be considered in the following scenarios:

- A *cross-tab scenario*, where a user is made to visit an attacker-controlled website containing malicious JavaScript, and this website tries to learn what other sensitive sites the user is visiting at the same time. These attacker-controlled and sensitive browsing sessions can be carried out on the same browser, on two different browsers belonging to the same user, or even on two browsers residing in two completely isolated virtual machines which share the same underlying hardware [75].
  One possible way of causing the user to browse to such an attacker-controlled site is through a phishing attack, where the attacker sends fraudulent messages, purporting to be from a benign source, that induces the victim to click on a link to a malicious web site. Alternatively, the attacker may pay an advertisement service to display a (malicious) advertisement when the user visits a third-party website [28].
- A *cross-network scenario*, where the attacker is an active on-path adversary capable of injecting JavaScript into any non-encrypted page. The attacker would like to leverage that access to try to learn about the user's sensitive activity, even though the attacker cannot manipulate or access this traffic directly. For example, the user may simultaneously run one browsing session over an unsecured connection for mundane tasks, and another browsing session over a second, secured connection for sensitive tasks. An attacker capable of modifying traffic on the standard link can learn about activity carried out over the secured link, whether this secure connection made through a VPN, through the Tor network, or even through a separate network adapter which the attacker cannot see.

The main challenge of the our attack model is the extremely restricted JavaScript runtime, which requires the attacker code to be written in a particular way, as we describe further in Section 4.

Regardless of the delivery vector, cache-based fingerprinting has a strong potential advantage over network-based fingerprinting, since it can indirectly observe both the computer's network activity and the browser's rendering process. As we demonstrate in Section 7.4, both of these elements contribute to the accuracy of our classifier.

4 Data Collection

4.1 Creating memorygrams

The raw data trace for network-based attacks takes the form of a *network trace*, commonly in the `pcap` file format, which contains a timestamped sequence of all traffic observed on a certain network link. The corresponding data trace in the case of cache attacks is the *memorygram* [64]—a trace of the cache access latency measured at a constant sampling rate over a given time period. The memorygrams of Oren et al. [64] describe the latency of multiple individual sets or groups of sets at each point in time, resulting in a two-dimensional array. In contrast, in this work we use a simplified, one-dimensional memorygram form. The contents of each entry in our memorygrams is a proxy for the occupancy of the cache at the specific time period. We collect memorygrams while the browser loads and displays websites, and use the data as fingerprints for website classification.

The Cache Occupancy Channel. Unlike prior works [28, 64], which use the Prime+Probe side-channel attack from JavaScript, we use a cache occupancy channel. The main difference is that the Prime+Probe attack measures contentions in specific cache sets, whereas our attack measures contention over the whole cache. Specifically, our JavaScript attack allocates an LLC-sized buffer and measures the time to access the entire buffer. The victim's access to memory evicts the contents of our buffer from the cache, introducing delays for our access. Thus, the time to access our buffer is roughly proportional to the number of cache lines that the victim uses. Cache occupancy has previously been implemented in native code and used for covert channels and for measuring co-resident activity [17, 74]. Both of these implementations rely on high resolution timers. To our knowledge, we are the first to use the cache occupancy channel with a low resolution timer.

Overcoming Hardware Prefetchers. Ideally, we would like to collect information across the whole cache. Intel processors, however, try to optimize memory accesses by prefetching memory locations that the processor predicts will be accessed in the future. Because prefetching changes the cache state, we need to fool the prefetchers. To fool the spatial prefetcher [42], we use the technique of Yarom and Benger [96] and do not probe adjacent cache sets. To fool the streaming prefetcher, which tries to identify sequences of cache accesses, we use a common approach of masking access patterns by randomizing the order of the memory accesses we perform [59, 65].

Spatial Information. Compared with the Prime+Probe attack, the cache occupancy channel does not provide any spatial information. That is, the adversary does not learn any information on the addresses that the victim accesses. While this is a clear disadvantage of the cache occupancy channel, our attack does not require spatial information. The main reason is that modern browsers have complex memory allocation patterns. Consequently, the location that data is allocated changes each time a page is downloaded, and the location carries little information on the downloaded page. In practice, not having spatial information is also an advantage. Without it, there is no need to build eviction sets for cache sets, a process that can take significant time [28].

Website Memorygrams. We capture memorygrams when the browser navigates to websites and displays them. We use a JavaScript-based memorygrammer to probe the cache at a fixed rate of one sample every 2 ms. We continue the probe for 30 seconds, resulting in a vector of length 15,000. When a probe takes longer than 2 ms, we miss the slot of the next probe. We use a special value to indicate this case. We use this collection method for all mainstream browsers other than the Tor Browser,

When the attack code is launched from within the Tor Browser, where the timer resolution is limited to 100 ms, we do not measure how long a sweep over the cache takes, but instead count how many sweeps over the entire cache fit into a single 100 ms timeslot. In addition, we do not probe for 30 seconds in this setting, but rather for 50 seconds, to account for the slower response time over the Tor network. Hence, Tor memorygrams contain 500 measurements over the entire 50 second measurement time period.

The native code memorygrammer used for the evaluations in Section 7 does not suffer from a reduced timing resolution when measuring the Tor Browser. Therefore, on mainstream browsers it runs for 30 seconds and produces 15,000 entries, and on the Tor Browser it runs for 50 seconds and produces 25,000 entries.

Sanity Check. Before proceeding, we want to verify that memorygrams can be used for fingerprinting. Indeed, Figure 3 shows graphical representations of memorygrams of three sites: Wikipedia (`https://www.wikipedia.com`), Github (`https://www.github.com`), and Oracle (`https://www.oracle.com`), collected through the native code memorygrammer. Each memorygram is displayed as a colored strip, where time goes from left to right and the shade corresponds to cache activity at each time. (Lighter shades correspond to fewer evictions.) We see that the three memorygrams of each site, while not identical, are similar to each other. The memorygrams of different websites are, however, very different from each other. This indicates that memorygrams may be used for identifying websites.

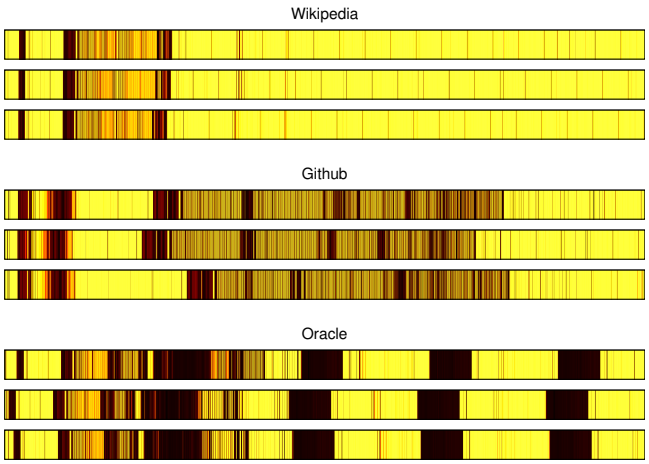


Figure 3: Examples of memorygrams. Time progresses from left to right, shade indicates the number of evictions. (Darker shades correspond to more eviction.)

4.2 Datasets

Closed World Datasets. We evaluate our cache-based fingerprinting on six different combinations of browsers and operating systems, summarized in Table 2. Many early works on website fingerprinting operated under a *closed world assumption*, where the attacker's aim is to distinguish among accesses to a relatively small list of websites. Our closed world datasets follow this line of work. These datasets consist of 100 traces each for a set of 100 websites, to a total of 10,000 memorygrams. We use the same list of 100 websites that Rimmer et al. [73] selected from the top Alexa sites. (See Appendix B for a complete list of websites included.) Similar to previous works, no traffic molding is applied and only one tab is opened at a time. The browser's response cache, however, is not cleared before accessing each website, an aspect of the experiment we analyze in more detail in Section 7.

Open World Datasets. One common criticism of the closed world assumption is that it requires the attacker to know the complete set of websites the victim is planning to visit, allowing the attacker to prepare and train classifiers for each of these websites. This assumption was challenged by many authors, for example Juárez et al. [46]. To address this criticism, website fingerprinting methods are often evaluated in an open-world setting. In this setting, the attacker wishes to monitor access to a set of sensitive websites, and is expected to classify them with high accuracy. Additionally, there is a large set of non-sensitive web pages, all of which the attacker is expected to generally label as "non-sensitive".

To evaluate our fingerprinting method in the open-world settings, we augment the closed-world datasets with additional 5,000 traces, each collected for a single unique web-

site, again using the list of websites provided by Rimmer et al. [73]. The base rate for this setting is 33.3%, since a trivial classifier can simply decide that all pages are non-sensitive.

5 Machine Learning

5.1 Problem Formulation

Website fingerprinting is generally formulated as a supervised learning problem, consisting of a template building step and an attack step. In the template building step, the adversary visits each target website multiple times and collects a set of labeled traces (either network traces or memorygrams), each corresponding to a visit to a certain website. Next, the adversary trains a classifier algorithm on these labeled traces, using either classical machine learning methods or deep learning methods.

In the attack step, the adversary is presented with a set of unlabeled traces, each one corresponding to a visit to an unknown website. The adversary then applies the previously trained classifier to each of these traces and outputs a guess for each trace. The accuracy of the classifier is finally calculated as the percentage of the correctly assigned labels.

5.2 Deep Learning Models

Early works on website fingerprinting, starting from Cheng and Avnur [14], used classical machine learning methods such as Naive Bayes, Support Vector Machine (SVM) and k-Nearest Neighbors (k-NN). As a prerequisite step to running these classical machine learning methods, the adversary needs to apply an additional feature extraction step which transforms the raw trace into a more succinct representation. Since these features were chosen through human insight into the nature of network traffic, there was no immediate way of directly applying them to memorygram analysis.

Abe and Goto [2] and later Rimmer et al. [73] suggest using deep learning for website fingerprinting. Deep learning performs automatic feature learning from the raw data, reducing the reliance on human insight at the cost of a larger required training set. Rimmer et al. [73] show that, given a large enough training set, deep-learning website-fingerprinting approaches are as effective as earlier methods which require manual feature selection. An advantage of this approach is that it allows us to compare network-based and cache-based fingerprinting based on the merit of the raw data, rather than on the specific choice of features.

Deep Neural Network Configuration. A deep neural network (DNN) is typically configured as a sequence of non-linear layers which transform the raw data, first extracting salient features and then selecting the appropriate ones [30]. Every layer in a DNN consists of a set of artificial neurons, each connected to a set of outputs from the previous layers. At the forward propagation stage, the activation function is applied to the product of the each neuron's input and its weight value, and then forwarded to the next layer.

For the last layer in the DNNs we evaluate we use a softmax layer, which outputs a vector containing a-posteriori probabilities for each one of the classes.

The process of training the neural network uses back-propagation to update the weights of each neuron to achieve a minimum loss at the output. First, the model calculates the cost between the true classification of the measurement and the predicted value using a loss function. Next, the model updates the weights of the each neuron based on the calculated loss. Every round of forward propagation and back-propagation is called an epoch. A neural network model runs multiple epochs to learn the weights for accurate classification.

We evaluate deep learning using two classifier models, Convolutional Neural Networks (CNN) and Long Short-Term Memory (LSTM) networks [38]. A CNN uses a sequence of feature mapping layers alternating between convolutions and max-pooling. Each of the layers sub-samples the previous layer, iteratively reducing the size of the input to a more succinct representation, while preserving the information they encode. Each convolutional layer is a neural network specialised for detecting complex patterns in its input. The convolution layer applies several filters to the input vector, each of which is designed to identify an abstract pattern in a sequence of input elements it is provided with. The max-pooling layers reduce the dimensionality of the data by subsampling the filters, choosing the maximum value from adjacent groups of neurons applied by the filters. This alternating sequence of layers extracts complicated features from the input and produces vectors short enough for the classifiers. The feature mapping layers are followed by a *dense* layer, in which every neuron is connected to every output of the feature extraction phase. The LSTM-based network has an initial feature selection step similar to the CNN, but then adds an additional layer in which each neuron has a memory cell, with the output of this neuron determined both by its inputs and by the value of this memory cell. This allows the classifier to identify patterns in time-based data.

Hyperparameter Selection. *Hyperparameters* describe the overall structure of the DNN and of each layer. The choice of hyperparameters depends on the specific classification problem. For network-based fingerprinting, we replicated the parameters specified in the dataset provided by Rimmer et al. [73]. For cache-based fingerprinting, we manually evaluated several choices for each hyperparameter.

To prevent overfitting, we use 10-fold cross validation. We split each dataset consisting of traces into 10 folds of equal size, and select one fold, consisting of 10% of the traces, as a *test set*. The remaining 90% of the traces are used for training the classifier, with 81% serving as the *training set* and 9% as the *validation set*. The model trains on the training set and the evaluation is done on the test set. The number of

epochs is regulated with an Early-Stop function which stops the epochs when the accuracy of the validation set no longer increases over successive iterations. The selected hyperparameters are summarized in Appendix A.

For the CNN classifier we use three pairs of convolution and max pooling layers. For the LSTM classifier we use two. As discussed above, the traces captured by the code running within the Tor Browser contain only 500 measurements, due to the reduced timer resolution. For these shorter traces, we modified the architecture of our LSTM-based classifier. The feature selection of this classifier contains only one convolution layer. We therefore used a pool-size of three for the max-pooling layer to limit the feature reduction before the LSTM layer. In addition, because of the small amount of features, we could increase the number of LSTM units to 128 and learn more complex patterns from the features.

6 Results

All of the results in this section were obtained by using keras version 2.1.4, with TensorFlow version 1.7 as the back end, running on two Ubuntu Linux 16.04 servers, one with two Xeon E5-2660 v4 processors and 128 GB of RAM, and one with two Xeon E5-2620 v3 processors and 128 GB of RAM. Our machine learning instances took approximately 40 minutes to run in this configuration.

Table 2 presents the fingerprinting accuracy we obtain. Recall that in this scenario the JavaScript interpreter of the targeted browser executes the memorygrammer. Considering that all modern browsers reduced their timer resolution and some added jitter as a countermeasure for the Spectre attack [69, 87], the first question we need to address is whether it is even possible to implement cache-based fingerprinting attacks in such an environment.

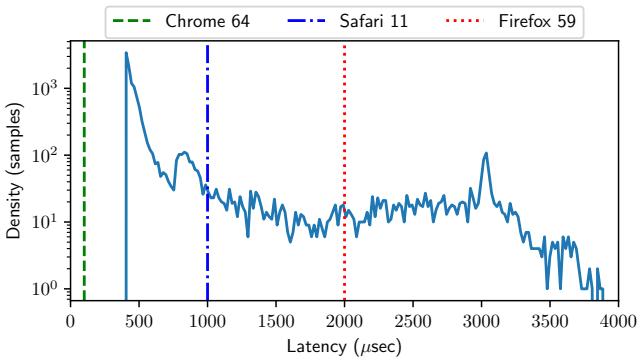


Figure 4: Cache probe latencies compared to modern browser timing resolutions.

To answer this question, we measured the latencies of the cache occupancy channel using a high-resolution timer while the browser was downloading a web page. Figure 4 shows the distribution of these latencies. The figure also uses vertical lines to indicate the timer resolutions of the various browsers. (See Table 2.) As we can see, even at the 2 ms resolution of the Firefox 59 timer, it is possible to distinguish between 80% of the probes which take less than 2 ms and the remaining 20%. This is a welcome side-effect of the use of a large buffer which is accessed at every probing step. None of the cache probes we measured, however, took longer than the 100 ms clock period of the Tor Browser. Hence, when running within the Tor Browser, we count the number of probes we can perform within each clock tick. (See Section 4.)

The next question is whether the information we collect with this low resolution is sufficient for fingerprinting. Indeed, Table 2 shows that in all of the environments we test our classifier is significantly better than a random guess. Remarkably, as our results show, even the highly restricted Tor Browser can be used for mounting cache attacks, albeit with a significantly lower accuracy than that of general-purpose browsers.

6.1 Closed World Results

We first look at the typical closed-world scenario investigated by past works. In mainstream browsers, our JavaScript attack code is consistently able to provide classification accuracies of 70–90%, well over the base rate of 1%. The Tor Browser attack, however, achieves a lower accuracy of 47%. If we, however, look not only at the top result output by the classifier, but also check whether the correct website is one of the top 5 detected websites, the accuracy of the Tor Browser attack climbs to 72%, with a base rate of 5%. This method of looking at the few most probable outputs of a classifier was previously used in similar classification problems [13, 62]. With some a-priori information an attacker can deduce which of the top 5 pages the victim has accessed.

We can compare the accuracy of our cache-based fingerprinting to the one obtained by state-of-the-art network-based methods, as reported by Rimmer et al. [73]. We see that while there are differences between the classification accuracy achieved in each case, the overall accuracy is comparable, assuming both attacks capture the same amount of traces per website. As in the network-based setting, we believe that capturing more than 100 traces per website is likely to increase the accuracy and the stability of our classifier.

6.2 Open World Results

We next turn to the more challenging open-world scenario, in which the 100 sensitive webpages must be distinguished from an additional set of 5,000 non-sensitive pages. As seen in Table 2 the JavaScript-based website fingerprinting code performs well under this scenario as well, again achieving classification accuracy of 70–90%. We note that in most cases the results are slightly better than the closed-world results. The reason is the larger size of the "non-sensitive"

Table 2: Accuracy obtained by in-browser memorygrammer— Mean (percents) and standard deviation.

Operating System	CPU	LLC Size	Browser	Timer Resolution	Closed World CNN	Closed World LSTM	Open World CNN	Open World LSTM
Linux	i5-2500	6 MB	Firefox 59	2.0 ms	78.5±1.7	80.0±0.6	86.8±0.9	87.4±1.2
Linux	i5-2500	6 MB	Chrome 64	0.1 ms	84.9±0.7	91.4±1.2	84.3±0.7	86.4±0.3
Windows	i5-3470	6 MB	Firefox 59	2.0 ms	86.8±0.7	87.7±0.8	84.3±0.6	87.7±0.3
Windows	i5-3470	6 MB	Chrome 64	0.1 ms	78.2±1.0	80.0±1.6	86.1±0.8	80.6±0.2
Mac OS	i7-6700	8 MB	Safari 11.1	1.0 ms	72.5±0.7	72.6±1.3	80.5±1.0	72.9±0.9
Linux	i5-2500	6 MB	Tor Browser 7.5	100.0 ms	45.4±2.7	46.7±4.1	60.5±2.2	62.9±3.3
Linux	i5-2500	6 MB	Tor Browser 7.5 (top 5)	100.0 ms	71.9±2.1	70.0±1.7	80.4±1.7	82.7±1.8

class. As discussed earlier, this also significantly increases the base rate for open-world scenarios to 33.3%.

As in the case of the closed-world setting, we can evaluate the accuracy of the Tor Browser under a top-5 assumption, i.e. when checking for the correct website in the top five outputs of the classifier. Under this relaxation the Tor Browser attack achieves a high accuracy rate of 83%, with a base rate of 37.3%.

The classification to sensitive vs. non-sensitive site is a binary classification problem, We can, therefore, apply standard analysis techniques to this aspect of the results. We achieved a near perfect classification in all of the open world settings we evaluated, achieving an area under curve (AUC) of more than 99% in all cases.

7 Robustness Tests

Having demonstrated the effectiveness of our website fingerprinting technique, we now turn our attention to its robustness and test its resilience to issues known to affect network-based fingerprinting.

7.1 Evaluation Setup

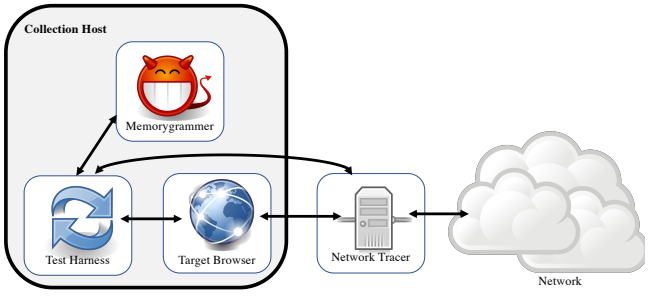


Figure 5: Data Collection Setup for the Robustness Tests.

To compare the results of network fingerprinting with cache-based fingerprinting, we need to modify our data collection setup. The setup, illustrated in Figure 5, consists of two data collection hosts. The *memorygram collection host*, which simulates the victim's machine, runs both the target browser and the memorygrammer software. The *network tracer* sits on-path between the memorygram collection hosts and the Internet, and collects a record of the network traffic. A test harness written in Perl and Python invokes the memorygrammer, the network tracer and the target browser at the same time, then saves a correlated data record consisting of the memorygram, the network trace in `pcap` format, and a screenshot of the target web page for monitoring purposes. For data collection, we use HP Elite 8300 desktop computers featuring Intel Core i5-2500 CPUs at 3.30 GHz, with a 6 MB last-level cache, running CentOS 7.2.1511 and either Firefox 59 or Tor Browser 7.5.

For the robustness tests we use a native-code memorygrammer, which is based on the Prime+Probe implementation of Mastik, a side-channel toolkit released under the GNU Public License [95]. We apply two modifications to the Mastik code. First, we change the Prime+Probe code to measure cache occupancy rather than activity in specific cache sets. Secondly, we use the processor's performance counters [41] to count the number of cache evictions rather than use the high resolution timer to identify evictions. The use of performance counters for attack purposes has already been proposed and investigated in the past [6, 9, 52, 84].

7.2 Baseline Scenario

Our baseline scenario replicates the results of our closed world JavaScript memorygrammer, as well as some of the results of Rimmer et al. [73]. As we can see in Table 3, the native-code memorygrammer gives a slightly better accuracy than the JavaScript memorygrammer on Firefox. When attacking the Tor Browser, the native code memorygrammer achieves much better results than the in-browser JavaScript code. We believe that the cause of the improvement is the higher probing accuracy afforded by the native-code memorygrammer. In both browsers, the results of the native-code memorygrammer are similar to those achievable with network-based fingerprinting.

Table 3: Accuracy obtained in robustness tests — Mean (percents) and Standard deviation.

	Firefox Network		**Firefox Cache**		Tor Network		**Tor Cache**	
Test	CNN	LSTM	**CNN**	**LSTM**	CNN	LSTM	**CNN**	**LSTM**
Baseline	86.4±1.0	93.2±0.5	**94.9±0.5**	**94.8±0.5**	77.6±1.6	90.9±0.7	**72.7±0.7**	**80.4±0.5**
Response cache enabled	56.1±1.5	70.6±1.5	**92.2±0.8**	**92.2±0.5**	55.5±1.7	65.9±1.0	**86.1±0.5**	**86.3±0.6**
Render only	–	–	–	–	1.0±0.0	1.0±0.0	**63.3±1.1**	**63.9±1.5**
Network only	–	–	–	–	77.6±1.6	90.9±0.7	**19.9±1.8**	**51.9±2.7**
Temporal drift	–	–	–	–	64.5±2.2	81.0±0.6	**68.3±0.5**	**75.6±0.7**

7.3 Enabling the Response Cache

Network-based fingerprinting methods, by definition, must rely on network traffic to perform classification. Typically, due to caching, many web pages are loaded with partial or no network traffic. As specified in RFC 7234 [24], the performance of web browsers is typically improved by the use of response caches. When a web browser client requests a remote resource from a web server, the server can specify that a particular response is cacheable, and the web browser can then store this response locally, either on disk or in memory. When the page is next requested, the web browser can ask the server to send the response only if it has been modified since the last time it was accessed by the client. In the case of a response cache hit, the server only returns a short header instead of the complete remote resource, resulting in a very short network traffic sequence. In some cases, the client can even reuse the cached response without querying the server for a remote copy, resulting in no network traffic at all. Herrmann et al. [36] demonstrate a significant decrease in the accuracy of web fingerprinting when the browser uses the response cache. Indeed, deleting or disabling the browser cache prior to fingerprinting attacks is a common practice [66, 88].

We enable caching of page contents by the browser, and measure the effect on fingerprinting accuracy. In the Firefox browser we simply refrain from clearing the response cache between sessions. For privacy reasons, the response cache in the Tor Browser does not persist across session restarts. Hence, when collecting data on the Tor Browser we "prime" the cache before every recording by opening the web page in another tab, allowing it to load for 15 seconds, then closing the tab.

When we keep the browser's response cache, the advantage of cache-based website fingerprinting starts to emerge. As Table 3 shows, the accuracy of the standard network-based methods degrades when the response caching is enabled. We can see a degradation in accuracy of over 20% in the fingerprinting accuracy.

In contrast, the cache-based methods are largely unaffected by the reduction in network traffic, achieving high accuracy rates. This result supports the conclusion that the cache-based detection methods are not simply detecting the CPU activity related to the handling of network traffic, making them essentially a special case of network-based classifiers, but are rather detecting rendering activities of the browser process.

7.4 Net-only and Render-only Results

Oren et al. [64] show that cache activity is correlated with network activity, raising the possibility that cache-based fingerprinting basically identifies the level of network activity. To rule out this possibility and show that website rendering also contributes to fingerprinting, we separate rendering (or more precisely, data processing) activity from handling of network data.

Render-Only Fingerprinting. To capture the data processing activity, we neutralize the network activity by guaranteeing constant traffic levels. More specifically, we apply molding to the network traffic, ensuring that data flow between the collection host and the network at a fixed bandwidth of 10 KB every 250 ms. To achieve that, we queue data transmitted at a higher rate, or send dummy packets when the transmitted data does not fill the desired bandwidth. These dummy packets are silently dropped by the receiver. The approach is, basically, BuFLO [22], with $\tau = \infty$, i.e., when the data stream continues indefinitely. This approach has a high bandwidth overhead compared to WTF-PAD and WT, however, it is designed to ensure that the network traffic is constant irrespective of the contents of the website. As expected, the raw network captures in this scenario all have the exact same size, which happens to be twice as large as the largest network capture recorded without traffic molding.

Because all the traces are identical, the network-based classifier assigns the same class to all of the traces, and its accuracy is the same as a random guess. The results of cache-based fingerprinting show a drop in accuracy compared with unmolded traffic. However, the accuracy is still significantly better than a random guess. This experiment demonstrates the resilience of cache-based website fingerprinting to mitigation techniques aimed at network-based fingerprinting,

and suggests that this privacy threat should be countered using a different class of mitigation techniques, as we explore further in Section 9.

Network-Only Fingerprinting. In a complementing experiment, we aim to capture only the network traffic. To collect this dataset, we first capture actual traffic data from a real browsing session. We then use a mock setup, that does not involve a browser at all. Instead, we use two `tcpreplay` [1] instances, one at the collection host, and the other at a server, to emulate the network traffic, by replaying the data from the `pcap` file.

The results for this experiment show that the cache-based classifier is capable of classifying many pages even when no rendering activity is taking place. However, the accuracy is significantly lower than in the case that rendering activity does take place. In particular, our CNN classifier only detects the correct website in about 20% of the cases, significantly lower than the 73% we get for the matching closed-world scenario. (But still much better than the 1% expected for a random guess.) The accuracy of the network-based classifier is the same as for the baseline, simply because the network traffic is replicated.

Combining these two experiments we therefore conclude that cache-based fingerprinting identifies features both in the network traffic patterns and in the actual *contents* of the displayed web pages.

7.5 Dealing with Temporal Drift

The accuracy of network-based website fingerprinting decays over time, when the contents of the website changes [73]. Many websites use content management systems (CMS), in which the page layout is based on a fixed template design, and only the resources loaded into this template vary over time. Since, as we have shown, the cache-based fingerprints capture rendering activities as well as network activities, it would seem that the rendering-related traces recorded by the cache-based method would have a longer lifetime, and be more resistant to drift, than the network-related traces captured by the traditional method.

To test this hypothesis, we repeat the data collection of the baseline experiment after a delay of 36 days (start to start). We then measure the ability of both cache-based and network-based classifiers to accurately classify the new traces, after being trained on the old traces. In this setting, we see a drop of 5–10% in the accuracy of both classifiers. We believe that further experiments are required for accurately assessing how cache-based and network-based fingerprinting handle temporal drifts.

8 Detecting Unknown Hardware Configurations

In contrast to network-based fingerprinting, which is largely target agnostic, cache-based fingerprinting needs to be tailored to the precise hardware configuration of the victim machine, specifically the set count and associativity of its last-level cache. Using a too large or a too small buffer reduces the effectiveness of the technique, and eventually the accuracy of the classifier. There are, however, not that many popular configurations. For example, four cache configurations (4096 or 8192 sets, 12 or 16 ways) cover most of the Intel Core processor models.

If the target hardware configuration is known beforehand (assuming, for example, that a particular user is singled out for attack) the attacker can customize the parameters of the JavaScript attack code to match the target PC's parameters. It would be interesting, however, to see how well an attacker can remotely determine an unknown target's cache configuration using JavaScript. To investigate this, we created a JavaScript program that allocates a 20MB array in memory and iterates over it in several patterns which should fit in well into different configurations of cache set-counts and associativities. We then recorded the minimum, maximum and mean access time per element, plus the standard deviation, for each of these configurations. We collected 1,350 such measurements from multiple systems with cache sizes of 3 MB, 4 MB, 6 MB, and 8 MB. We then used MATLAB's classification learner tool to apply a variety of machine learning classifiers to the measured data. Using both KNN and SVM classifiers, we were able to correctly classify the configuration of the target's last-level cache with over 99.8% classification accuracy under 5-fold cross validation. Interestingly, even a simple tree-based classifier which compared the minimum iteration time of three different configurations to a predefined threshold was 99.6% accurate. We ported this simple tree-based classifier to JavaScript, creating an LLC cache size detector which we tested and found capable of accurately detecting the cache sizes of 15 different machines with diverse browser, hardware and operating system configurations, taking less than 300 ms to run in all cases. Thus generic attacks that adapt to the specific hardware configuration seem feasible.

9 Countermeasures

We now discuss potential countermeasures to our fingerprinting attack. We first describe a cache masking technique we experimented with. We then follow with a review of other cache attack countermeasures suggested in the literature.

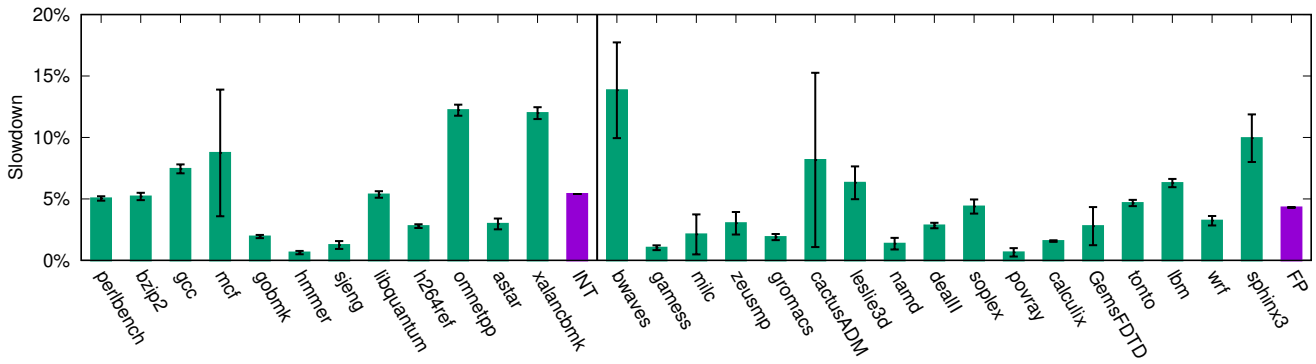


Figure 6: Performance slowdown of our countermeasure on the SPEC benchmark. Error bars indicate one standard deviation. INT and FP show the geometric mean of the SPEC integer and floating point benchmarks, respectively.

9.1 Cache Activity Masking

One well-studied mitigation method from the domain of network-based cache fingerprinting involves creating spurious network activity to mask the actual website traffic [22]. It is possible to adapt such a masking technique to our domain and mask the actual website rendering activity by creating spurious activity in the cache. Our initial experiments show that this is a promising mitigation, but further research is needed to assess its effectiveness and its effect on performance and on power consumption.

Masking implementation. Our countermeasure repeatedly evicts the entire last-level. More specifically, we allocate a cache-sized buffer and access every cache line in the buffer in a loop. Such masking could be applied in the browser, in the operating system, as a browser plugin, and even incorporated into a security-conscious website in the form of JavaScript delivered to the client. For our initial proof of concept implementation we chose to implement the countermeasure as a standalone native code application, based on a modification of the Mastik side-channel toolkit [95]. This setting allows us to investigate the effectiveness of our countermeasure while leaving deployment complexities for future work.

Evaluation. We evaluated this countermeasure on a desktop computer featuring an Intel Core i5-2500, running Centos Linux version 7.6.1810. We enabled the countermeasure, then collected website traces both for Firefox (Linux) and for the Tor Browser, using the same mix of traces described in Section 4.2—10,000 traces for the closed-world scenario, consisting of 100 traces for each of the Alexa top 100 websites, and 5,000 additional traces for the open-world scenario, each collected for a single unique website. We split the data set into training, testing and validation sets and applied 10-fold cross validation, as described in more detail in Section 5.2.

Our experiments show that the countermeasure completely thwarts the attack when training is done on an unprotected system—the accuracy of our classifier was at or below the base rate of 1% for the closed-world scenario and 33% for the open-world scenario. We also evaluated a scenario in which the adversary is allowed to train on traces with the countermeasure applied. In this more challenging scenario, the countermeasure completely thwarts the attack when the attack code is running from the Tor Browser. On Firefox, however, we only noticed a moderate reduction in the effectiveness of the attack. In the closed world scenario, the attack achieves 73% success and in the open world the success rate is 77%. (Down from 79% and 86%, respectively.)

Performance Impact. To understand the effect that our countermeasure has system performance, we used the industry-standard SPEC CPU benchmark [79], the de-facto standard benchmark for measuring the performance of the CPU and the memory subsystems. Figure 6 shows the results of the SPEC CPU 2006 benchmarks with our countermeasure, relative to no countermeasure. The countermeasure causes a slowdown of around 5% (geometric mean across the benchmarks) with a worst case slowdown of 14% for the bwaves benchmark. These results are from the average of ten executions of the benchmarks for each case. With Tor network performance being as it is, we believe that the performance hit on CPU benchmarks is acceptable for this scenario.

9.2 Other Countermeasures

Most of the past research into cache attacks has been done in the context of side-channel cryptanalysis. Due to the different scenario, many of the countermeasures typically suggested for cache-based attack are no longer effective. Techniques such as constant-time programming [5] are only applicable to regular code, typically found in implementations of cryptographic primitives. It is hard to see how such techniques can be applied to web browsers. Similarly, as

this work demonstrates, timer-based defenses that reduce the timer frequency or add jitter are not effective.

Cache randomization techniques [58, 70, 91] dissociate victim and adversary cache sets, and prevent the adversary from monitoring victim access to specific addresses. However, our attack measures the overall cache activity rather than looking at specific victim accesses. As such, such techniques are unlikely to be effective against our attack.

Cache partitioning, either using dedicated hardware [21, 91] or via page coloring [55], is a promising approach for mitigating cache attacks. In a nutshell, the approach partitions the cache between security domains, preventing cross-domain contention. Web pages are often rendered within the same browser process. A page-coloring countermeasure will, therefore, need to adapt to the browser scenario. Alternatively, the current shift to strict site isolation [81] as part of the mitigations for Spectre [48], may assist in applying page coloring to protect against our attack. A further limitation of page coloring is that caches support only a handful of colors. Hence, colors need to be shared, particularly when a large number of tabs are open. To provide protection, page coloring will have to be augmented with a solution that prevents concurrent use of the same color by multiple sites.

CACHEBAR [97] limits the contention caused by each process as a protection for the Prime+Probe attack. Like cache partitioning, this approach works at a process resolution and may require adaptions to work in the web browser scenario. Furthermore, unlike past cryptographic attacks that aim to identify specific memory accesses, our technique measures the overall memory use of the victim. Consequently, unless CACHEBAR is configured to partition the cache, some cross-process contention will remain, allowing our attack to work.

10 Limitations and Future Work

While the work demonstrates the feasibility of cache-based website fingerprinting and provides an analysis of the attack, it does leave some areas for further study. Being the first analysis of its kind, the scope of the work does not match the scope of similar works on network-based website fingerprinting. In particular, our datasets are significantly smaller than those of Rimmer et al. [73], for example. Providing larger datasets would allow better analysis of the effectiveness of the technique and would be a beneficial service for the research community as a whole.

In this work we collected the memorygrams on the same hardware configuration used by the victim PC. While we show that we can adapt the data collection to the specific victim hardware (Section 8), at this stage it is not clear how much a classifier trained on data collected with one hardware configuration would be effective for classifying memorygrams collected on a different configuration.

In the network-based website fingerprinting scenario, little to no traffic travels through the network unless the user is actively fetching a webpage. In the cache-based scenario, however, the cache is always active to a degree, even before the browser starts to receive and render the webpage. Recognizing the start of a trace may therefore be more difficult in the cache-based setting than in the network-based setting, especially in the case of a real attack. Our framework implicitly synchronizes the trace with the start of the download. Due to varying network conditions, we see differences of up to six seconds between trace start and render start. As such, we believe that our technique can identify web sites even without the synchronization. Further experimentation is required, however, to verify this fact. We also note that if the machine is otherwise idle, cache activity can serve as a (slightly noisy) indicator of the start of the trace.

The work further shares many of the limitations of network-based fingerprinting [46]. In particular, websites tend to change over time or based on the identity of the user or the specifications of the computer used for displaying them. Furthermore, our work, like most previous works, assumes that only one website is displayed at each time. Both Rimmer et al. [73] and our work briefly discuss temporal aspects of website fingerprinting, and we also looked a bit into the issue (Section 7.5). However, further work is required to assess the impact of this and other variables on the efficacy of cache-based fingerprinting.

11 Conclusions

In this work we investigate the use of cache side channels for website fingerprinting. We implement two memorygrammers, which capture the cache activity of the browser, and show how to use deep learning to identify websites based on the cache activity that displaying them induces.

We show that cache-based website fingerprinting achieves results comparable with the state-of-the-art network-based fingerprinting. We further show that cache-based fingerprinting outperforms network-based fingerprinting under a common operating scenario, where the browser maintains cached objects. Finally, we demonstrate that cache-based fingerprinting is resilient to both traffic molding and to reduced timer resolution. The former being the standard defense for network-based website fingerprinting and the latter the currently implemented countermeasure for mobile-code-based microarchitectural attacks. To the best of our knowledge, this is the first cache-based side channel attack that works with the 100 ms clock rate of the Tor Browser.

Acknowledgements

We would like to thank Vera Rimmer for her helpful comments and insights. We would also like to thank Roger Dingledine and our shepherd Rob Jansen for reviewing and commenting on the final version of this paper.

This research was supported by the ARC Centre of Excellence for Mathematical & Statistical Frontiers, Intel Corporation, Israel Science Foundation grants 702/16 and 703/16, NSF CNS-1409415, and NSF CNS-1704105.

References

[1] Tcpreplay. https://tcpreplay.appneta.com/.

[2] Kota Abe and Shigeki Goto. Fingerprinting attack on Tor anonymity using deep learning. In *Proceedings of the APAN – Research Workshop 2016*, 2016.

[3] Onur Acıiçmez. Yet another microarchitectural attack: : exploiting I-Cache. In *CSAW*, pages 11–18, 2007.

[4] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *CHES*, pages 110–124, 2010.

[5] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *LATINCRYPT*, pages 159–176, 2012.

[6] Sarani Bhattacharya and Debdeep Mukhopadhyay. Who watches the watchmen?: Utilizing performance monitors for compromising keys of RSA on Intel platforms. In *CHES*, pages 248–266, 2015.

[7] Zack Bloom. Cloud computing without containers. https://blog.cloudflare.com/cloud-computing-without-containers/, 2018.

[8] Jo M. Booth. Not so incognito: Exploiting resource-based side channels in JavaScript engines. Bachelor thesis, Harvard, April 2015.

[9] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *WOOT*, 2017.

[10] Xiang Cai, Xin Cheng Zhang, Brijesh Joshi, and Rob Johnson. Touching from a distance: website fingerprinting attacks and defenses. In *ACM CCS*, pages 605–616, 2012.

[11] Xiang Cai, Rishab Nithyanand, and Rob Johnson. Cs-buflo: A congestion sensitive website fingerprinting defense. In *WPES*, pages 121–130, 2014.

[12] Xiang Cai, Rishab Nithyanand, Tao Wang, Rob Johnson, and Ian Goldberg. A systematic approach to developing and evaluating website fingerprinting defenses. In *ACM CCS*, pages 227–238, 2014.

[13] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. De-anonymizing programmers via code stylometry. In *USENIX Sec*, pages 255–270, 2015.

[14] Heyning Cheng and Ron Avnur. Traffic analysis of SSL encrypted web browsing. Project paper, University of Berkeley, 1998.

[15] Giovanni Cherubin, Jamie Hayes, and Marc Juárez. Website fingerprinting defenses at the application layer. *PoPETs*, 2017(2):186–203, 2017.

[16] Shane S. Clark, Hossen A. Mustafa, Benjamin Ransford, Jacob Sorber, Kevin Fu, and Wenyuan Xu. Current events: Identifying webpages by tapping the electrical outlet. In *ESORICS*, pages 700–717, 2013.

[17] David Cock, Qian Ge, Toby C. Murray, and Gernot Heiser. The last mile: An empirical study of timing channels on seL4. In *ACM CCS*, pages 570–581, 2014.

[18] Wei Dai. PipeNet description. Post to the `cypherpunks` mailing list. Copy available at https://www.freehaven.net/anonbib/cache/pipenet10.html, 1998.

[19] T. Dierks and E. Rescola. The transport layer security (TLS) protocol version 1.2. RFC 5246, RFC Editor, 2008. https://tools.ietf.org/html/rfc5246.

[20] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The second-generation onion router. In *USENIX Security*, pages 303–320, 2004.

[21] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael B. Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *TACO*, 8(4):35:1–35:21, 2012.

[22] Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. Peek-a-Boo, I still see you: Why efficient traffic analysis countermeasures fail. In *IEEE SP*, pages 332–346, 2012.

[23] Dmitry Evtyushkin, Dmitry V. Ponomarev, and Nael B. Abu-Ghazaleh. Jump over ASLR: attacking branch predictors to bypass ASLR. In *MICRO*, pages 40:1–40:13, 2016.

[24] R. Fielding, M. Nottingham, and J. Reschke. Hypertext transfer protocol (HTTP/1.1): Caching. RFC 7234, RFC Editor, June 2014. http://www.rfc-editor.org/rfc/rfc7234.txt.

[25] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand pwning unit: Accelerating microarchitectural attacks with the GPU. In *IEEE SP*, pages 195–210, 2018.

[26] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. "Make sure DSA signing exponentiations really are constant-time". In *ACM CCS*, pages 1639–1650, 2016.

[27] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptographic Engineering*, 8(1):1–27, 2018.

[28] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by key-extraction cache attacks from portable code. In *ACNS*, 2018.

[29] Xun Gong, Nikita Borisov, Negar Kiyavash, and Nabil Schear. Website detection using remote traffic analysis. In *Privacy Enhancing Technologies*, pages 58–78, 2012.

[30] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning (Adaptive Computation and Machine Learning series)*. The MIT Press, 2016. ISBN 0262035618.

[31] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *NDSS*, 2017.

[32] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security*, pages 897–912, 2015.

[33] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *ACM CCS*, pages 368–379, 2016.

[34] Berk Gülmezoglu, Andreas Zankl, Thomas Eisenbarth, and Berk Sunar. PerfWeb: How to violate web privacy with hardware performance events. In *ESORICS (2)*, pages 80–97, 2017.

[35] Jamie Hayes and George Danezis. k-fingerprinting: A robust scalable website fingerprinting technique. In *USENIX Security*, pages 1187–1203, 2016.

[36] Dominik Herrmann, Rolf Wendolsky, and Hannes Federrath. Website fingerprinting: attacking popular privacy enhancing technologies with the multinomial naïve-bayes classifier. In *CCSW*, pages 31–42, 2009.

[37] Andrew Hintz. Fingerprinting websites using traffic analysis. In *Privacy Enhancing Technologies*, pages 171–178, 2002.

[38] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[39] Wei-Ming Hu. Lattice scheduling and covert channels. In *IEEE SP*, pages 52–61, 1992.

[40] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery on the cloud. In *CHES*, pages 368–388, 2016.

[41] Intel Corp. Intel 64 and IA-32 architectures software developer's manual volume 3B, September 2016. URL https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf.

[42] Intel Corp. Intel 64 and IA-32 architectures optimization reference manual, June 2016. URL https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html.

[43] Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. S$A: A shared cache attack that works across cores and defies VM sandboxing - and its application to AES. In *IEEE SP*, pages 591–604, 2015.

[44] Suman Jana and Vitaly Shmatikov. Memento: Learning secrets from process footprints. In *IEEE SP*, pages 143–157, 2012.

[45] Rob Jansen, Marc Juárez, Rafa Galvez, Tariq Elahi, and Claudia Díaz. Inside job: Applying traffic analysis to measure Tor from within. In *NDSS*, 2018.

[46] Marc Juárez, Sadia Afroz, Gunes Acar, Claudia Díaz, and Rachel Greenstadt. A critical evaluation of website fingerprinting attacks. In *ACM CCS*, pages 263–274, 2014.

[47] Hyungsub Kim, Sangho Lee, and Jong Kim. Inferring browser activity and status through remote monitoring of storage usage. In *ACSAC*, pages 410–421, 2016.

[48] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Haburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwartz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE SP*, pages 19–37, May 2019.

[49] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *USENIX Sec*, pages 463–480, 2016.

[50] Nil Köskal. 'terrifying': How a single line of computer code put thousands of innocent Turks in jail. http://www.cbc.ca/news/world/terrifying-how-a-single-line-of-computer-code-put-thousands-of-innocent-turks-in-jail-1.4495021, January 2018.

[51] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. Stealing webpages rendered on your browser by exploiting GPU vulnerabilities. In *IEEE SP*, pages 19–33, 2014.

[52] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security*, pages 557–574, 2017.

[53] Shuai Li, Huajun Guo, and Nicholas Hopper. Measuring information leakage in website fingerprinting attacks and defenses. In *ACM CCS*, pages 1977–1992, 2018.

[54] Bin Liang, Wei You, Liangkun Liu, Wenchang Shi, and Mario Heiderich. Scriptless timing attacks on web browser privacy. In *DSN*, pages 112–123, 2014.

[55] Jochen Liedtke, Hermann Härtig, and Michael Hohmuth. OS-controlled cache predictability for real-time systems. In *IEEE RTAS*, pages 213–224, 1997.

[56] Pavel Lifshits, Roni Forte, Yedid Hoshen, Matt Halpern, Manuel Philipose, Mohit Tiwari, and Mark Silberstein. Power to peep-all: Inference attacks by malicious batteries on mobile devices. *PoPETs*, 2018 (4):1–1, 2018.

[57] Moritz Lipp, Michael Schwartz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security*, August 2018.

[58] Fangfei Liu and Ruby B. Lee. Random fill cache architecture. In *MICRO*, pages 203–215, 2014.

[59] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE SP*, pages 605–622, 2015.

[60] Liming Lu, Ee-Chien Chang, and Mun Choon Chan. Website fingerprinting and identification using ordered feature sequences. In *ESORICS*, pages 199–214, 2010.

[61] Mozilla Foundation. Security advisory 2018-01. https://www.mozilla.org/en-US/security/advisories/mfsa2018-01/, 2018.

[62] Arvind Narayanan, Hristo Paskov, Neil Zhenqiang Gong, John Bethencourt, Emil Stefanov, Eui Chul Richard Shin, and Dawn Song. On the feasibility of internet-scale author identification. In *IEEE SP*, pages 300–314, 2012.

[63] Rishab Nithyanand, Xiang Cai, and Rob Johnson. Glove: A bespoke website fingerprinting defense. In *WPES*, pages 131–134, 2014.

[64] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In *ACM CCS*, pages 1406–1418, 2015.

[65] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA*, pages 1–20, 2006.

[66] Andriy Panchenko, Lukas Niessen, Andreas Zinnen, and Thomas Engel. Website fingerprinting in onion routing based anonymization networks. In *WPES*, pages 103–114, 2011.

[67] Andriy Panchenko, Fabian Lanze, Jan Pennekamp, Thomas Engel, Andreas Zinnen, Martin Henze, and Klaus Wehrle. Website fingerprinting at internet scale. In *NDSS*, 2016.

[68] Colin Percival. Cache missing for fun and profit. Presented at BSDCan. http://www.daemonology.net/hyperthreading-considered-harmful, 2005.

[69] Filip Pizlo. What Spectre and Meltdown mean for WebKit. https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/, January 2018.

[70] Moinuddin K. Qureshi. CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *MICRO*, 2018.

[71] Michael K. Reiter and Aviel D. Rubin. Crowds: Anonymity for web transactions. *ACM Trans. Inf. Syst. Secur.*, 1(1):66–92, 1998.

[72] E. Rescola. HTTP over TLS. RFC 2818, RFC Editor, 2000. https://tools.ietf.org/html/rfc2818.

[73] Vera Rimmer, Davy Preuveneers, Marc Juarez, Tom Van Goethem, and Wouter Joosen. Automated website fingerprinting through deep learning. In *NDSS*, 2018.

[74] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *ACM CCS*, pages 199–212, 2009.

[75] Joanna Rutkowska and Rafal Wojtczuk. *Qubes OS Architecture*, February 2010. URL https://www.qubes-os.org/attachment/wiki/QubesArchitecture/arch-spec-0.3.pdf.

[76] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript. In *Financial Cryptography*, pages 247–267, 2017.

[77] Michael Schwarz, Moritz Lipp, and Daniel Gruss. JavaScript zero: Real JavaScript and zero side-channel attacks. In *NDSS*, 2018.

[78] Spiegel Online. Documents reveal top NSA hacking unit. http://www.spiegel.de/international/world/the-nsa-uses-powerful-toolbox-in-effort-to-spy-on-global-networks-a-940969-2.html, December 2013.

[79] Cloyce D. Spradling. SPEC CPU2006 benchmark tools. *SIGARCH Computer Architecture News*, 35(1):130–134, 2007. doi: 10.1145/1241601.1241625. URL https://doi.org/10.1145/1241601.1241625.

[80] Raphael Spreitzer, Simone Griesmayr, Thomas Korak, and Stefan Mangard. Exploiting data-usage statistics for website fingerprinting attacks on Android. In *WISEC*, pages 49–60, 2016.

[81] The Chromium Project. Site isolation. https://www.chromium.org/Home/chromium-security/site-isolation.

[82] The Tor Project, Inc. The Tor Browser. https://www.torproject.org/projects/torbrowser.html.en.

[83] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of DES implemented on computers with cache. In *CHES*, pages 62–76, 2003.

[84] Leif Uhsadel, Andy Georges, and Ingrid Verbauwhede. Exploiting hardware performance counters. In *FDTC*, pages 59–67, 2008.

[85] Kenton Varda. https://news.ycombinator.com/item?id=18280156, 2018.

[86] Pepe Vila and Boris Köpf. Loophole: Timing attacks on shared event loops in Chrome. In *USENIX Security*, pages 849–864, 2017.

[87] Luke Wagner. Mitigations landing for new class of timing attack. https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/, January 2018.

[88] Tao Wang and Ian Goldberg. Improved website fingerprinting on Tor. In *WPES*, pages 201–212, 2013.

[89] Tao Wang and Ian Goldberg. On realistically attacking Tor with website fingerprinting. *PoPETs*, 2016(4):21–36, 2016.

[90] Tao Wang and Ian Goldberg. Walkie-Talkie: An efficient defense against passive website fingerprinting attacks. In *USENIX Security*, pages 1375–1390, 2017.

[91] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *ISCA*, pages 494–505, 2007.

[92] Zachary Weinberg, Eric Yawei Chen, Pavithra Ramesh Jayaraman, and Collin Jackson. I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks. In *IEEE SP*, pages 147–161, 2011.

[93] Junhua Yan and Jasleen Kaur. Feature selection for website fingerprinting. *PoPETs*, 2018(4):200–219, 2018.

[94] Qing Yang, Paolo Gasti, Gang Zhou, Aydin Farajidavar, and Kiran S. Balagani. On inferring browsing activity on smartphones via USB power analysis side-channel. *IEEE Trans. Information Forensics and Security*, 12(5):1056–1066, 2017.

[95] Yuval Yarom. Mastik: A micro-architectural side-channel toolkit. http://cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf, September 2016.

[96] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack. Cryptology ePrint Archive, Report 2014/140, 2014. URL http://eprint.iacr.org/2014/140.

[97] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. A software approach to defeating side channels in last-level caches. In *ACM CCS*, pages 871–882, 2016.

A Selected Hyperparameters

Tables 4, 5, and 6 summarize the hyperparameters for the classifiers used in this work.

Table 4: Hyperparameters for the CNN classifier

Hyperparameter	Value	Space
Optimizer	Adam	Adamax, Adam, SGD, RMSprop
Learning rate	0.001	0.001–0.002
Batch size	100	40–100
Training epoch	20–30	Early stop by accuracy
Convolution layers	3	3–4
Input units (FF)	15000	15000–25000
Input units (Tor)	25000	15000–25000
CNN activation	relu	relu, tanh
Kernels	256	2–512
Kernel size	16,8,4	2–31
Pool size	4	2–8

Table 5: Hyperparameters for the LSTM classifier

Hyperparameter	Value	Space
Optimizer	Adam	Adamax, Adam, SGD, RMSprop
Learning rate	0.001	0.001–0.002
Batch size	100	40–100
Training epoch	20–30	Early stop by accuracy
Convolution layers	2	1–3
Input units (FF)	15000	15000–25000
Input units (Tor)	25000	15000–25000
CNN activation	relu	relu, tanh
LSTM activation	tanh	relu,tanh
Kernels	256	2–512
Kernel size	16,8	2–32
Pool size	4	2–8
Dropout	0.2	0.1–0.2
LSTM units	32	8,32

Table 6: Hyperparameters for the LSTM classifier for the Tor attack

Hyperparameter	Value	Space
Optimizer	Adam	Adamax, Adam, SGD, RMSprop
Learning rate	0.001	0.001–0.002
Batch size	100	40–100
Training epoch	20–30	Early stop by accuracy
Convolution layers	1	1–3
Input units	500	500
CNN activation	relu	relu, tanh
LSTM activation	tanh	relu,tanh
Kernels	256	2–512
Kernel size	32	2–32
Pool size	3	2–8
Dropout	0.4	0.1–0.4
LSTM units	128	8,32,128

B Websites Included in Closed-World Datasets

9gag.com	abs-cbn.com
adf.ly	adobe.com
aliexpress.com	allegro.pl
amazon.com	amazonaws.com
aol.com	apple.com
archive.org	askcom.me
battle.net	blastingnews.com
booking.com	breitbart.com
bukalapak.com	businessinsider.com
conservativetribune.com	dailymail.co.uk
dailymotion.com	detik.com
deviantart.com	dictionary.com
digikala.com	doubleclick.net
doublepimp.com	ebay.com
espncricinfo.com	exoclick.com
extratorrent.cc	facebook.com
feedly.com	gamepedia.com
github.com	go.com
godaddy.com	goodreads.com
google.com	hclips.com
hola.com	hotmovs.com
imdb.com	instructure.com
intuit.com	kompas.com
leboncoin.fr	liputan6.com
livejasmin.com	livejournal.com
ltn.com.tw	microsoftonline.com
mozilla.org	msn.com
naver.com	netflix.com
nicovideo.jp	nih.gov
ntd.tv	office.com
onedio.com	openload.co
oracle.com	ouo.io
outbrain.com	pinterest.com
popads.net	quora.com
researchgate.net	roblox.com
rt.com	rutracker.org
scribd.com	skype.com
soundcloud.com	sourceforge.net
spotify.com	spotscenered.info
stackexchange.com	stackoverflow.com
steamcommunity.com	steampowered.com
t.co	theguardian.com
thesaurus.com	tistory.com
tokopedia.com	torrentz2.eu
tribunnews.com	tumblr.com
twitter.com	weather.com
wikia.com	wikipedia.org
wittyfeed.com	xhamster.com
xvideos.com	yandex.ru
yelp.com	zippyshare.com

Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation

Shuai Wang*[1], Yuyan Bao[2], Xiao Liu[2], Pei Wang*[3], Danfeng Zhang[2], and Dinghao Wu[2]

[1]The Hong Kong University of Science and Technology
[2]The Pennsylvania State University
[3]Baidu X-Lab
shuaiw@cse.ust.hk, {yxb88, xvl5190}@ist.psu.edu, wangpei10@baidu.com, zhang@cse.psu.edu, dwu@ist.psu.edu

Abstract

Cache-based side channels enable a dedicated attacker to reveal program secrets by measuring the cache access patterns. Practical attacks have been shown against real-world crypto algorithm implementations such as RSA, AES, and ElGamal. By far, identifying information leaks due to cache-based side channels, either in a static or dynamic manner, remains a challenge: the existing approaches fail to offer high precision, full coverage, and good scalability simultaneously, thus impeding their practical use in real-world scenarios.

In this paper, we propose a novel static analysis method on binaries to detect cache-based side channels. We use abstract interpretation to reason on program states with respect to abstract values at each program point. To make such abstract interpretation scalable to real-world cryptosystems while offering high precision and full coverage, we propose a novel abstract domain called the Secret-Augmented Symbolic domain (**SAS**). **SAS** tracks program secrets and dependencies on them for precision, while it tracks only coarse-grained public information for scalability.

We have implemented the proposed technique into a practical tool named CacheS and evaluated it on the implementations of widely-used cryptographic algorithms in real-world crypto libraries, including Libgcrypt, OpenSSL, and mbedTLS. CacheS successfully confirmed a total of 154 information leaks reported by previous research and 54 leaks that were previously unknown. We have reported our findings to the developers. And they confirmed that many of those unknown information leaks do lead to potential side channels.

1 Introduction

Cache-based timing channels enable attackers to reveal secret program information, such as private keys, by measuring the runtime cache behavior of the victim program. Practical attacks have been executed with different attack scenarios, such as time-based [16, 44], access-based [37, 60, 62], and trace-based [5], each of which exploits a victim program through either coarse-grained or fine-grained monitoring of cache behavior. Additionally, previous research has successfully launched attacks on commonly used cryptographic algorithm implementations, for example, AES [37, 60, 74, 16], RSA [23, 44, 7, 62, 86], and ElGamal [90].

Pinpointing cache-based side channels from production cryptosystems remains a challenge. Existing research employs either static or dynamic methods to detect underlying issues [77, 32, 33, 41, 82, 22, 81]. However, the methods are limited to low detection coverage, low precision, and poor scalability, which impede their usage in analyzing real-world cryptosystems in the wild.

Abstract interpretation is a well-established framework that can be tuned to balance precision and scalability for static analysis. It models program execution within one or several carefully-designed *abstract domains*, which abstract program concrete semantics by tracking certain program states of interest in a concise representation. Usually, the elements in an abstract domain form a complete lattice of finite height, and the operations of the program concrete semantics are mapped to the abstract transfer functions over the abstract domain. A well-designed abstract interpretation framework can correctly approximate program execution and usually yields a terminating analysis within a finite step of computations. Nevertheless, the art is to carefully design an abstraction domain that fits the problem under consideration, while over-approximating others to bound the analysis to a controllable size; this enables the analysis of non-trivial cases.

We propose a novel abstract domain named the Secret-Augmented Symbolic domain (**SAS**), which is specifically designed to perform abstract interpretation on *large-scale* secret-aware software, such as real-world cryptosystems. **SAS** is designed to perform fine-grained tracking of program secrets (e.g., private keys) and dependencies on them, while coarsely approximating non-secret information to speed up the convergence of the analysis.

We implement the proposed technique as a practical tool named CacheS, which models program execution within the **SAS** and pinpoints cache-based side channels with constraint solving techniques. Like many bug finding tech-

*Most of this work is done while Shuai Wang and Pei Wang were working at PSU.

niques [55, 84, 54], CacheS is soundy [53]; the implementation is unsound for speeding up analysis and optimizing memory usage, due to its lightweight but unsound treatment of memory. However, in contrast to previous studies that analyze only small-size programs, single procedure or single execution trace [32, 33, 77, 22, 81], CacheS is scalable enough to deliver whole program static analysis of real-world cryptosystems without sacrificing much accuracy. We have evaluated CacheS on multiple popular crypto libraries. Although most libraries have been checked by many previous tools, CacheS is able to detect 54 unknown information leakage sites from the implementations of RSA/ElGamal algorithms in three real-world cryptosystems: Libgcrypt (ver. 1.6.3), OpenSSL (ver. 1.0.2k and 1.0.2f), and mbedTLS (ver. 2.5.1). We show that CacheS has good scalability as it largely outperforms previous research regarding coverage; it is able to complete context-sensitive interprocedural analysis of over 295 K lines of instructions within 0.5 CPU hour. In summary, we make the following contributions:

- We propose a novel abstract interpretation-based analysis to pinpoint information leakage sites that may lead to cache-based side channels. We propose a novel abstract domain named **SAS**, which performs fine-grained tracking of program secrets and dependencies, while over-approximating non-secret values to enable precise reasoning in a scalable way.
- Enabled by the "symbolic" representation of abstract values in **SAS**, we facilitate information leak checking in this research with constraint solving techniques. Compared with previous abstract interpretation-based methods, which only reason on the information leakage upper-bound, our technique adequately simplifies the process of debugging and fixing side channels.
- We implement the proposed technique into a practical tool named CacheS and apply it to detect cache-based side channels in real-world cryptosystems. From five popular crypto library implementations, CacheS successfully identified 208 information leakage sites (with only one false positive), among which 54 are unknown to previous research, to the best of our knowledge.

2 Background

Abstract Interpretation. Abstract interpretation is a well-established framework to perform sound approximation of program semantics [28]. Considering that program concrete semantics forms a value domain **C**, abstract interpretation maps **C** to an abstract (and usually more concise) representation, namely, an abstract domain **A**. The design of the abstraction is usually based on certain program properties of interest, and (possibly infinite) sets of concrete program states are usually represented by one abstract state in **A**. To ensure termination, abstract states could form a lattice with a finite height, and computations of program concrete semantics are mapped into operators over the abstract elements in **A**.

The abstract function (α) and concretization function (γ) need to be defined jointly with an abstract domain **A**. Function α lifts the elements in **C** to their corresponding abstract elements in **A**, while γ casts an abstract value to a set of values in **C**. To establish the correctness of an abstract interpretation, the abstract domain and the concrete domain need to form a Galois connection, and operators defined upon elements in an abstract domain are required to form the local and global soundness notions [28].

Cache Structure and Cache-Based Timing Channels. A cache is a fast on-CPU data storage unit with a very limited capacity compared to the main memory. Caches are usually organized to be set-associative, meaning that the storage is partitioned into several disjoint sets while each set exclusively stores data of a particular part of the memory space. Each cache set can be further divided into smaller storage units of equal size, namely cache lines. Given the size of each cache line as 2^L bytes, usually the upper $N-L$ bits of a N-bit memory address uniquely locate a cache line where the data from that address will be temporally held.

When the requested data is not found in the cache, the CPU will have to fetch them from the main memory. This is called a cache miss and causes a significant delay in execution, compared with fetching data directly from the cache. Therefore, an attacker may utilize the timing difference to reveal the cache access pattern and further infer any information on which this pattern may depend.

Threat Model. As mentioned above, some bits of a memory address can be directly mapped to cache lines being visited, which potentially enables information leakage via *secret-dependent memory traffic*. In this research, attackers are assumed to share the same hardware platform with the victim program, and therefore are able to "probe" the shared cache state and infer cache lines being accessed by the victim. As illustrated in Fig. 2, our threat model assumes that the attacker can observe the address of every memory access, expect for the low bits of addresses that distinguish locations in the same cache line. Overall, by tracking the secret-dependent cache access of the victim, several bits of program secrets (w.r.t. entropy) could be leaked to the attacker.

We note that this threat model indeed captures most infamous and practical side channel attacks [39], including prime-and-probe [60], flush-and-reload [86], and prime-and-abort [31], which are designed to infer the cache line access by measuring the latency of the victim program or attacker's program at different scales and for different attack scenarios. Additionally, while this threat model is aligned with many existing side channel detection works [77, 33, 41, 82, 22], novel techniques proposed in this work enable us to perform scalable static analysis and reveal much more information leaks of real-world cryptosystems.[1] In addition, while this

[1] Consistent with this line of research, CacheS pinpoints information leaks in cryptosystems where cache access depends on secrets. Cryptosystem developers can fix the code with information provided by CacheS. Contrarily, the exploitability of the leaks (e.g., reconstruct the entire key by recovering half bits of the RSA private key [19]) is beyond the scope of this work.

```
1  foo:
2    mov   eax, ebx
3    add   eax, 0x1
4    load  ecx, esi
5    add   ecx, 0x12
6    mov   edx, edi
7    add   eax, ecx
```

(a) Sample Code.

```
1  {ebx = {k1}}
2  {ebx = {k1}, eax = {k1}}
3  {ebx = {k1}, eax = {k1+1}}
4  {ebx = {k1}, eax = {k1+1}, ecx = {m1}}
5  {ebx = {k1}, eax = {k1+1}, ecx = {m1+12}}
6  {ebx = {k1}, eax = {k1+1}, ecx = {m1+12}, edx = {edi0}}
7  {ebx = {k1}, eax = {k1+m1+13}, ecx = {m1+12}, edx = {edi0}}
```

(b) Modeling program states with logic formulas $l \in L$.

```
1  {ebx = {s1}}
2  {ebx = {s1}, eax = {s1}}
3  {ebx = {s1}, eax = {s1+1}}
4  {ebx = {s1}, eax = {s1+1}, ecx = {p}}
5  {ebx = {s1}, eax = {s1+1}, ecx = {p}}
6  {ebx = {s1}, eax = {s1+1}, ecx = {p}, edx = {p}}
7  {ebx = {s1}, eax = {⊤}, ecx = {p}, edx = {p}}
```

(c) Modeling program states with **SAS**.

Figure 1: Execute assembly code with different program representations. Program secrets and all the affected registers are marked as red in Fig. 1a. Program states at line 1 of Fig. 1b and Fig. 1c represent the initial state. Here $\mathtt{k}_1$ is a symbol exhibiting one piece of program secrets (e.g., the first element in a key array), and $\mathtt{m}_1$ is a free symbol representing non-secret content of unknown memory cells. $\mathtt{edi}_0$ is a symbol representing the initial value of register `edi`. Symbol s_1, p, and $\top$ defined in **SAS** stand for one piece of secret, entire non-secret information and all the program information, respectively (see Sec. 4).

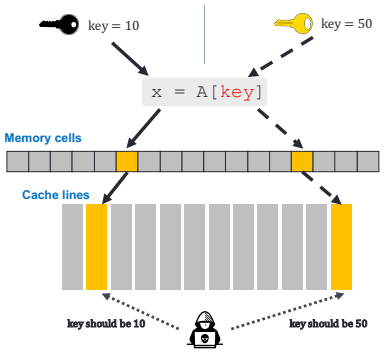


Figure 2: The threat model. Different secrets lead to the access of different cache lines at one particular program point, which may leak secret information to the attackers by indirectly observing cache line access variants. At least one bit information (w.r.t. entropy) could be leaked in this example.

model is relatively stronger than those based on cache status [32], cache status at any point can be determined by analyzing the accessed cache units in execution.

3 Motivation

In general, capturing cache-based side channels requires modeling program secret-dependent semantics (we will discuss the connection between program semantics and cache access in Sec. 5). In this section we begin by discussing two baseline approaches to modeling program semantics; the limitations of both approaches naturally motivate the design of our novel abstract domain.

Modeling Program Semantics with Logic Formulas. An intuitive way is to represent program concrete semantics with logic formulas (as in a typical symbolic execution approach [77]), and perform whole-program static reasoning until a fixed point is reached. The overall workflow exhibits a typical dataflow analysis procedure, and upon termination, each program point maintains a program state that maps variables (i.e., registers, CPU flags, and memory cells) to sets of formulas representing the possible values each variable may hold regarding any execution paths and inputs. For ease of presentation, we name the value domain formed by logic formulas l as logic domain L.

An example is given in Fig. 1, where we model the execution of instructions with logic formulas (Fig. 1b). While the overall approach will precisely model program semantics, some tentative studies indicate its low scalability. Indeed, we implement this approach and evaluate it with two real-world cases: the AES and RSA implementations of OpenSSL. We report that both tests are unable to terminate (evaluation results are given in Sec. 8). In summary, the analysis is impeded for the following reasons:

- Typically, more and more memory cells would be modeled throughout the analysis, and for each variable, its value set (i.e., set of formulas) would also continue to increase. Therefore, the memory usage could become significant to even unrealistic for real-world cases.
- Program states could be continuously updated within loop iterations. In addition, "loops" on the call graph (e.g., recursive calls) could exist in cryptosystems as well and complicate the analysis.

We implement algorithms to detect loop induction variables [11] considering both registers and stack memories. Identified induction variables are lifted into a linear function of symbolic loop iterators; operations on induction variables are "merged" into the linear function, thereby leading to a stable stage. While the simpler AES case terminated when we re-ran the test, the RSA case still yielded a "timeout" due to the practical challenges mentioned above (see results in Sec. 8.1).

Modeling Program Semantics with Free Symbols. Another "baseline" approach is to model program semantics in a permissive way. That is, we introduce two free symbols: one for any public information and the other for secrets. Any secret-related computation outputs the same secret symbol, while others preserve the same public symbol. Note that this is comparable to static taint tracking, where each value is either "tainted" or "untainted". Despite its simplicity, our tentative study reveals new hurdles as follows:

- Memory tracking becomes pointless. Every memory address becomes (syntactically) identical because it

holds the same public or secret symbol. Therefore, a memory store could overturn the entire memory space.
- Even if memory addresses are tracked in a more precise way, representing any secret value and their dependencies coarsely as one free secret symbol yields many false positives (since secret-dependent memory accesses do not necessarily lead to vulnerable cache accesses; see our cache modeling in Sec. 5). Tentative tests of the AES case report a false positive rate of 20% (8 out of 40) due to such modeling. In contrast, our novel program modeling yields no false positive when testing this case (see Sec. 8).

Motivation of Our Approach. This paper presents a novel abstract domain that enables abstract interpretation of large-scale cryptosystems in the wild. Our observation is that imprecise tracking of secrets impedes the accurate modeling of cache behaviors (cache access modeling is discussed in Sec. 5). Nevertheless, tracking too much information, such as modeling whole-program semantics with logic formulas, could face scalability issues when analyzing real-world cryptosystems due to various practical challenges.

Our study of real-world cryptosystems actually reveals an interesting and intuitive finding. That is, program secrets and their dependencies usually exhibit at a very *small portion* of program points, and even in such secret-carrying points, most variables maintain *only public information*. It should be noted that in common scenarios non-secret information is not critical for modeling cache-based timing channels. Hence, based on our observation, we promote a novel abstract domain that is particularly designed to *model the secret-dependent semantics of real-world crypto systems*. Our abstract domain delivers fine-grained tracking of program secrets and their dependencies with different identifiers for each piece of secret information, while performing coarse-grained tracking of other public values to effectively enhance scalability.

4 Secret-Augmented Symbolic Domain

This section presents the definition of our abstract domain **SAS**. We formally define each component following convention, including the concrete semantics, the abstract domain, and the abstract transfer functions. We also prove that the computations specified in **SAS** correctly over-approximate concrete semantics. Due to space limitations, we highlight only certain necessary components to make the paper self-contained. We refer readers to the extended version of this paper for more details [76] .

4.1 Abstract Values

We start by defining abstract values $f \in \mathbf{AV}$ (soon we will show that **SAS** is defined as the powerset of **AV**). Comparable to "symbolic formulas" in symbolic execution, f combines symbols and constants via operators. Elementary symbols in each abstract value are defined as follows:

- p: a unique symbol representing all the program public information.

Literal	n	$\in \mathbb{Z}$
OP_1	$\oplus$	$::= + \mid -$
OP_2	$\otimes$	$::= \times \mid \div \mid \% \mid$ AND $\mid$ OR $\mid$ XOR $\mid$ SHIFT
Atom	t	$::= \top \mid p \mid s_i \mid n$
Expression	exp	$::= t \mid t \oplus exp \mid t \otimes exp$
Formula	f	$::= e \mid exp \mid e \oplus exp$

Figure 3: Syntax of abstract value.

- s_i: a symbol representing a piece of program secrets; for instance, the i-th element of a secret array.
- e: a unique symbol representing the initial value of the x86 stack register `esp`.

While only one free symbol p is used to represent any and all unknown non-secret information (e.g., initial value $\texttt{edi}_0$ of register `edi` in Fig. 1b), we retain finer-grained information about program secrets. Multiple s_i are generated, and are mapped to different pieces of program secrets (e.g., a symbol s_1 representing $\texttt{k}_1$ in Fig. 1c). Therefore, different s_i symbols are *semantically different*, meaning each of them stands for different secrets.

Syntax. The syntax of a core of abstract values $f \in \mathbf{AV}$ is defined in Fig. 3. Literal specifies that concrete data is preserved in **AV**. OP_1 and OP_2 explain typical operators in **AV**. Atom includes symbols and literals, among which $\top$ (top) is the abstraction of any concrete value. Expression and Formula additionally define expressions and formulas. Note that stack memory expands linearly in the process address space, and stack register `esp` at any program point shall hold a value which adds or subtracts an offset from the initial value of `esp` (i.e., e). In the syntax definition, stack memory offsets could be a constant or an exp.

Since the symbol $\{s_i\}$ represents the secrets, which our analysis intends to keep track of, the formulas that contain these symbols usually need to be specially treated. We denote this infinite set of special formulas by $\mathbf{AV}_s$, where $\mathbf{AV}_s = \{f \in \mathbf{AV} \mid \exists s \in \{s_i\} \text{ s.t. } s \text{ occurs in } f\}$.

Reduction of Abstract Formulas. We now define the operator semantics of abstract value $f \in \mathbf{AV}$. For any operator $\odot \in \{\oplus\} \cup \{\otimes\}$, we define a reduction rule $T_\odot : \mathbf{AV} \times \mathbf{AV} \to \mathbf{AV}$ such that $[\![a_1 \odot a_2]\!] = T_\odot([\![a_1]\!], [\![a_2]\!])$ for any $a_1, a_2 \in \mathbf{AV}$, where $[\![\cdot]\!]$ denotes the semantics. We then define $T_\odot(a_1, a_2)$ as follows:

$$T_\odot(a_1,a_2) = \begin{cases} \top & \text{if } a_1 = \top \text{ or } a_2 = \top \\ \top & \text{else if } a_1 = p \wedge a_2 \in \mathbf{AV}_s \text{ or} \\ & a_2 = p \wedge a_1 \in \mathbf{AV}_s \\ p & \text{else if } a_1 = p \wedge a_2 \notin \mathbf{AV}_s \text{ or} \\ & a_2 = p \wedge a_1 \notin \mathbf{AV}_s \\ a_1 \odot a_2 & \text{otherwise} \end{cases}$$

Essentially, the first three cases perform reasonable over-approximation on $f \in \mathbf{AV}$ with different degrees of abstraction. The last case would apply if no other case can be matched; indeed similar to symbolic execution, most operations on $f \in \mathbf{AV}$ "concatenates" abstract values via abstract

operators following this rule. For the implementation, we also implement "constant folding" rules for operands of concrete data; such rules help the reduction of stack increment and decrement operations.

Since abstract interpretation typically needs to process sets of facts, we extend $T_\odot$ so that it can be applied to pairs of subsets of abstract values $f \in \mathbf{AV}$, where

$$\forall X, Y \in \mathscr{P}(\mathbf{AV}), \forall \odot \in \{\oplus\} \cup \{\otimes\}, \\ T_\odot(X,Y) = \{T_\odot(a,b) \mid a \in X, b \in Y\}$$

4.2 Abstract Domain

Naturally, each element in **SAS** represents the possible values that a program variable may hold; therefore each element in **SAS** forms a set of abstract values. That is,

Definition 1. *Let* **AV** *be the set of abstract values. Then*

$$\mathbf{SAS} = \mathscr{P}(\mathbf{AV})$$

forms a domain whose elements are subsets of all valid abstract values.

Claim 1. ***SAS*** *forms a lattice, with the top element* $\top_{\mathbf{SAS}}$*, bottom element* $\bot_{\mathbf{SAS}}$ *and a join operator* $\sqcup$ *defined over* ***SAS****.*

We specify the $\top$, $\bot$, and join operator $\sqcup$ in Appendix A. We bound the size of each element in **SAS** with a maximal number N (therefore the lattice has a finite height) and give corresponding evaluations in Appendix B. For further discussion, see the extended version [76].

Example. Fig. 1 explains typical computations within **SAS**. We present a set of abstract values for each register in Fig. 1c. While the computations over secret symbol s_i are precisely tracked (line 3 in Fig. 1c), the computations over p preserve this symbol (line 5 in Fig. 1c), and the computations between abstract value $a \in \mathbf{AV}_s$ and p lead to $\top$ (line 7 in Fig. 1c).

5 Pinpointing Information Leakage Sites

Upon the termination of static analysis, we check abstract memory addresses of each memory load and store instruction. When a secret-dependent address $a \in \mathbf{AV}_s$ is identified, its corresponding memory access instruction is considered to be "secret-dependent." We then translate each secret-dependent address a into an SMT formula f for constraint checking (this translation is discussed in Sec. 6.4).

In this research, we adopt a cache model proposed by the existing work to check each secret-dependent memory access [77]. Given an SMT formula f translated from $a \in \mathbf{AV}_s$ that represents a memory address, CacheS checks potential cache line access variants by solving the satisfiability of the following predicate:

$$f \gg L \neq f[s_i'/s_i] \gg L \tag{1}$$

As discussed in Sec. 2, assuming the cache has the line size of 2^L bytes, for a memory address of N bits, the upper $N-L$ bits map a memory access to its corresponding cache line access. In other words, the upper $N-L$ bits decide which cache line the upcoming memory access would visit. Therefore, for an SMT formula f derived from $a \in \mathbf{AV}_s$, we right shift f by L bits, and the result $f \gg L$ indicates the cache line being accessed. Furthermore, by replacing each s_i with a fresh secret symbol s_i', we obtain $f[s_i'/s_i] \gg L$. As a standard setting, the cache line size is assumed to be 64 (2^6) in this work; therefore, we set L as 6.

The constructed constraint checks whether different secrets (s_i and s_i') can lead to the access of different cache lines at this memory access. Recall the threat model shown in Fig. 2, the existence of at least one satisfiable solution reveals potential side channels at this point. From an attacker's perspective, by (indirectly) observing the access of different cache lines, a certain number of secrets could be leaked to adversaries. In addition, while this constraint assumes that accesses to different offsets within cache lines are indistinguishable, Constraint 1 can be extended to detect related issues. For example, information leaks which enable cache bank attacks can be detected by changing L from 6 to 2 [87].

6 Design of CacheS

We now present CacheS, a tool that uses precise and scalable static analysis to detect cache-based timing channels in real-world cryptosystems. Fig. 4 presents the workflow of CacheS. Given a binary as the input, CacheS first leverages a reverse engineering tool to recover the assembly code and the control flow structures from the input. The assembly instructions are further lifted into *platform-independent* representations before analysis. Technical details on reverse engineering are discussed in Sec. 7.

Given all the recovered program information, we initialize the abstract program state at each program point. In particular, we update the initial state of certain program points with one or several "secret" symbols to represent program secrets (e.g., a sequence of memory cells) when the analysis starts. We then perform abstract interpretation on the whole program until the fixed point in **SAS** is reached.

Abstract interpretation reasons the program execution within **SAS** (Sec. 6.1), and as mentioned, the proposed abstract domain performs fine-grained tracking of program secret-related semantics while maintaining only coarse-grained public information for scalability. The entire analysis framework forms a standard worklist algorithm, where each program point maintains its own program state mapping variables to sets of abstract values (Sec. 6.1.2).

We define information flow rules to propagate secret information (Sec. 6.2) in our context-sensitive and interprocedural analysis (Sec. 6.3). Upon the termination of analyzing one function, we identify secret-dependent memory accesses and translate corresponding memory addressing formulas into SMT formulas (Sec. 6.4) and check for side channels (Sec. 5).

Application Scope. In this research we design our abstract domain **SAS** to analyze assembly code: program memory access can be accurately uncovered by analyzing assembly

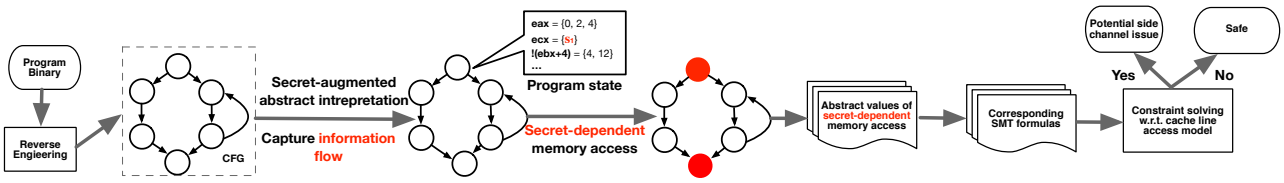


Figure 4: The overall workflow of CacheS.

code, thus supporting a "down-to-earth" modeling of cache behavior (see Sec. 5).

To assist the analysis of off-the-shelf cryptosystems and capture information leaks in the wild, we designed CacheS to directly process binary executables, including stripped executables with no debug or relocation information. We rely on reverse engineering tools to recover program control structures from the input binary, and further build our analysis framework on top of that (see Sec. 7).

6.1 Abstract Interpretation

In this section, we discuss how the proposed abstract domain **SAS** is adopted in our tool, and elaborate on several key points to deliver a practical and scalable analysis.

6.1.1 Initialization

Before the analysis, we first initialize certain program points with $\{s_i\}$ to represent the initial secret program information; for the rest their corresponding initial states are naturally defined as $\{\}$, or $\{e\}$ for the stack register `esp`.

Program secrets are maintained in registers or memory cells (e.g., on the stack) during execution. Since CacheS is designed to directly analyze binary code, we must first recognize the location of program secrets. We reverse-engineer the input binary and mark the location of secrets manually. Once the locations of secrets are flagged, we update the initial value set of corresponding variables (i.e., registers or memory cells) with a secret symbol s_i. Additionally, while "manual reverse engineering" is sufficient for studies in this research, it is always feasible to leverage automatic techniques [26] to search for secrets directly from executables or secret-aware compilers to track secret locations when source code is available. We leave this to future work.

In addition, since program secrets may be stored in a region of sequential memory cells (e.g., in an array), we create another identifier named u to represent the base address of the secret memory region. While u itself is treated as *public information*, we specify that memory loading from u will obtain program secrets; that is, we introduce one s_i for each memory loading via u.

6.1.2 Program State

At each program point, CacheS maintains a lookup table that maps variables to value sets; each value set $S \in$ **SAS** consists of abstract values $f \in$ **AV** representing possible values of a variable at the current program point. While the "lookup table" is an essential piece of any non-trivial analysis framework, our study has shown that naively-designed program state representations in CacheS could consume significant amounts of computing resources and impede the analysis of non-trivial programs. Thus, at this step we seek to design a *concise* and *practical* representation of program states. For the rest of this section, we first explain a "baseline" implementation of the lookup table, and further discuss two refinements.

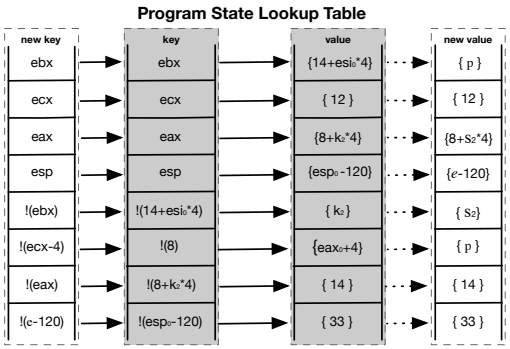


Figure 5: A sample program state lookup table. $\texttt{esp}_0$, $\texttt{eax}_0$ and $\texttt{esi}_0$ in the "key" and "value" entries are symbols representing the register initial values. Symbol ! means pointer dereference, for example `!(eax)` means memory loading from address stored in `eax`. Lookup tables at each program point are the major factor for memory usage, and we optimize the design by replacing "key" and "value" columns with "new key" and "new value" columns, respectively (see Sec. 6.1.2). Hence, shaded boxes are eliminated in CacheS.

The "Baseline" Approach. A sample lookup table is shown in Fig. 5 (the "key" and "value" columns), where each table maps registers and memory addressing formulas to their corresponding sets for logic formulas $l \in$ **L**. When it encounters a memory access instruction, CacheS computes the memory addressing formula and searches for its existence in the lookup table. (This requires some "equivalence checking"; the details will be explained in Sec. 6.1.4). If the search identifies an entry in the table, CacheS extracts or updates the content of that entry accordingly. Consider the example in Listing 1, where we first store concrete data 14 into memory via address stored in `eax`, and further load it out into `ebx`.

Listing 1: Sample instructions.

```
store eax, 14
load  ebx, eax
```

Knowing that value set of `eax` is 8+k_2*4 (third entry in Fig. 5), the first instruction creates an entry from address 8+k_2*4 to 14 (Fig. 5 shows program states after executing the first instruction). Further memory loading would acquire the value set in `eax`, and then reset the entry of `ebx` with 14 in the state lookup table of the second instruction.

Reading from unknown registers and memory locations would introduce symbols of different credentials regarding our information flow policy (see Sec. 6.2 for details).

Optimization of Table Values. While the precisely tracked logic formulas $l \in \mathbf{L}$ result in notable computing resource usage (Sec. 3), the proposed abstract domain **SAS** (Sec. 4) enables succinct representation of abstract values. As shown in Fig. 5, the "value" column of the lookup table is now replaced by the "new value" column. Consequently, memory consumption is considerably reduced (details are reported in our evaluation section).

Optimization of Table Keys. Since only abstract values are traced in **SAS**, the "key" column can be updated into a compact representation as well. However, using symbols such as p as the key will result in an imprecise modeling of memory addresses.

CacheS optimizes the "key" column in the following way. For most memory related entries, instead of using abstract memory addressing formulas, memory access expressions (expressions of registers and constant offsets) are used as keys. For example, the first instruction in Listing 1 uses memory access expression (i.e., "eax") instead of its abstract value $8 + s_2 * 4$ for memory lookup. Hence, when analyzing the store instruction, CacheS creates (or updates) an entry in the lookup table, which uses !(eax) as the key (here symbol "!" means pointer dereference). Likewise, for memory load, !(eax) will be used to look up the program state table. To safely preserve lookup entries via expressions, whenever the value set of a register is reset in the analysis, entries in the table are deleted if their keys are memory access expressions via the newly-updated register.

Nevertheless, since stack register `esp` is frequently manipulated to access stack memory, we preserve abstract addressing formulas via e to keep track of stack memory access precisely (e.g., the last entry in the "new key" column of Fig. 5).

6.1.3 Order of Program State

When multiple program states are possible for a program point, it is important to define the "merge" operation in abstract interpretation. Such an operation can be defined based on the least upper bound operation $\sqcup$ of **SAS** (recall that **SAS** forms a lattice (Sec. 4.2)).

Given lookup tables T_1 and T_2 representing two program states, $T_1 \sqcup T_2$ is defined as the following table, say T_3:

- T_3's key set is the union of the key sets of T_1 and T_2;
- For each key k in T_3, $T_3[k] = T_1[k] \sqcup T_2[k]$ (assuming $T_1[k]$ or $T_2[k]$ is an empty set if k is not in the table).

Moreover, the least upper bound of program states entails the partial order of any two program states: $T_1 \sqcup T_2 = T_1 \leftrightarrow T_2 \subseteq T_1$.

6.1.4 Memory Model

When encountering a memory load and store operation, we must decide which memory cell is accessed by tracing the memory address. However, considering CacheS models program semantics with abstract values, a memory address can usually contain one or several symbols instead of only concrete data. Therefore, policies (i.e., a "memory model") are usually required to determine the location of an accessed memory cell given a symbolic pointer.

When defining the abstract semantics within **SAS** (see our technical report [76]), we assume the assistance of a sound points-to analysis module as pre-knowledge. Nevertheless, finding such a convenient tool for assembly code of large-scale cryptosystems is quite difficult in practice. We have tried several popular "end-to-end" binary analysis platforms that take an executable as the input and perform various reverse engineering campaigns including points-to analysis; nevertheless, so far we cannot find a practical and robust solution to our scenario.

Therefore, we aim to implement a rigorous memory model by solving the *equality constraints* of two abstract formulas. However, tentative tests show that such a memory model may lose considerable precision in terms of reasoning symbolic pointers and may also not be scalable enough. On the other hand, since keys in the memory lookup table are formulas of e (for stack pointers; recall that e represents the initial value of `esp`) or memory access expressions (for other pointers), the current implementation of CacheS rigorously reasons on the equality constraints if abstract values are composed of e and concrete offsets, which is indeed often the case in analyzing assembly code. For the rest (e.g., e and symbolic offsets), we reason on the syntactical equivalence of memory access expressions. This design tradeoff may incorrectly deem equivalent symbolic pointers inequivalent (due to the symbolic "alias" issue) but not vice versa. Experiments show that this memory model is efficient enough to handle real-world cryptosystems while being promisingly accurate.

6.2 Information Flow

Considering that information leaks detected in this research are derived from secret-dependent memory accesses, CacheS keeps track of the secret program information flow throughout the analysis. In this section we elaborate on cases where the secret information can be propagated.

Variable-Level Information Flow. The explicit information flow is modeled in a straightforward way. Since variables (i.e., registers, memory cells, and CPU flags) are modeled as abstract formulas, high credential information (exhibited as abstract value $f \in \mathbf{AV}_s$) would naturally "flow" among variables during the computations. Moreover, reading from unknown variables (those with empty value sets) generates a symbol p as a proper over-approximation.

Information Flow via Memory Loading. By knowing the underlying memory layout, it could be feasible to infer table lookup indexes by observing the memory load outputs,

hence leaking table indexes of secrets to attackers. It should be noted that such cases are not rare in real-world cryptosystems, where many precomputed data structures are deployed in the memory to speed up computations. Thus, we define policies to capture information flow through memory loading. To do so, for a load operation, whenever the value sets of its base address or memory offset include formula $f \in \mathbf{AV}_s$, CacheS assigns the memory content to a fresh s_i, indicating secret information could have potentially propagated to the value being read. In contrast, when loading from unknown memory cells (memory cells of empty value sets) via non-secret addresses, we create a p to update the memory reader.

While most memory addressing formulas refer to specific locations in the memory, symbols p and $\top$ represent any program (public) information. To safely approximate memory read access via p and $\top$, CacheS assigns $\top$ to the memory reader. In case a memory storing is via symbol p or $\top$, we terminate the analysis since this would rewrite the whole memory space. Additionally, we note that memory loading and storing via $\top$ are considered to be information leaks as well since $\top$ implies that a variable has certain residual secrets (see Sec. 6.4).

6.3 Interprocedural Analysis

Our interprocedural analysis is context-sensitive. We build a classic function summary-based interprocedural analysis framework, where a summary $(\langle f', i\rangle, o)$ of a function call towards f maps the calling context $\langle f', i\rangle$ (f' is the caller name and i is the input) to the function call output o. CacheS maintains a set of summaries for each function f, and for an upcoming call of f, its calling context is first checked regarding the existing summaries of f. In case the context is a subset of any recorded entries (the partial order of calling context is derived from the order of program states defined in Sec. 6.1.2), the analysis will be skipped and we directly return the corresponding output.

To recover the function inputs, we inquire the employed reverse engineering platform (details are given in Sec. 7) to obtain the number of parameters the approaching function has. According to the calling convention of 32-bit x86 platforms, a memory stack is used to store function parameters; thus, we construct stack memory addresses of function parameters and acquire the value set of each parameter from the program state lookup table at the call site. If some memory cells of function parameters are absent, symbol p is used as an over-approximation. To compute the output information of a function, we join program states at every return instruction when the analysis of the target function terminates, which over-approximates the function return states.

6.4 Translating Abstract Values into SMT Formulas

As noted earlier (Sec. 5), cache-access side channels are summarized into SMT constraints. Upon the termination of analyzing each function, we identify secret-dependent memory addresses $a \in \mathbf{AV}_s$ and build the side channel constraints. SMT solvers are used to solve the constraint and check whether different secrets can lead to cache line access variants. Nevertheless, while many works to date leverage symbolic execution to construct SMT formulas, here we reason on program states within **SAS**. Therefore, before constraint checking, we first translate abstract formulas into SMT formulas.

Each abstract formula is maintained as a symbolic "tree" in CacheS, where tree leaves are symbols and concrete data while other nodes are operators. At this step, we translate each leaf on the tree into a bit vector implemented by a widely-used SMT solver—Z3 [30]; a bit vector would be instantialized with a numeric value if it was derived from a constant. In addition, we translate abstract operators on the tree into bit vector operations in Z3. Hence, an abstract formula tree would be reduced bottom-up into an SMT formula.

Translate Secret Symbols into Unique Bit Vectors. As noted earlier, s_i symbols are semantically different, each of which represents different pieces of secrets. For the implementation, we assign a unique id for each newly-created s_i symbol, which further leads to the creation of unique bit vectors at this step. In contrast, p (and e) symbols are transformed into identical bit vectors.

Memory Access via $\top$. It is easy to see that $\top$ implies that a variable has some residual secrets along with possibly public information. Hence, in addition to checking the constructed SMT constraints with Z3, memory accesses are flagged as vulnerable whenever their corresponding addressing formulas are $\top$.

7 Implementation

CacheS is mainly written in Scala (in 6,764 LOC; counted by CLOC [29]). The tentative implementation (in 7,163 LOC), which models program semantics with logic formulas (Sec. 3), is maintained as a separate "branch" of the code base.

Starting from an input binary code, the first step is to recover the assembly program as well as control flow and call graphs. Here we employ a popular reverse engineering tool, IDA-Pro (version 6.9) for the reverse engineering task [1]. We use the default configurations of IDA-Pro to recover assembly code and program control structures from the input executables.

Assembly Lifting. Many existing binary analysis infrastructures have provided facilities to lift x86 assembly code into a high-level intermediate representation. Without reinventing the wheel, here we employ a well-developed binary analysis platform BINNAVI [34] to transform x86 assembly code into a *platform-independent* intermediate language, REIL [72]. Our analysis procedures are built on top of the recovered representations. In addition, for a formal definition of program concrete semantics in terms of the REIL language, please refer to our technical report [76].

The current implementation of CacheS analyzes ELF binaries on the x86 platform. Nevertheless, since REIL language is designed as *platform-independent*, there is no fundamental limitation for CacheS to analyze binaries of other

Table 1: Cryptosystems analyzed by CacheS.

Implementation	Versions	Analysis Starting Function	Implement Which Algorithm
Libgcrypt [48]	1.6.1, 1.7.3	_gcry_mpi_powm	RSA/ElGamal
OpenSSL [59]	1.0.2f, 1.0.2k	BN_mod_exp_mont_consttime	
mbedTLS [57]	2.5.1	mbedtls_mpi_exp_mod	RSA
OpenSSL [59]	1.0.2f, 1.0.2k	_x86_AES_decrypt_compact	AES
mbedTLS [57]	2.5.1	mbedtls_internal_aes_decrypt	

formats or from other platforms (e.g., PE binaries on Windows) as long as the assembly instructions can be translated into REIL statements. As aforementioned, our current prototype focuses on 32-bit ELF binaries since the state-of-the-art REIL lifter (BinNavi [34]) does not have an official support for 64-bit binaries. However, the proposed technique shall be applicable to 64-bit binaries with no additional technical hurdles.

Recover x86 Memory Access Instructions from REIL Statements. As noted in Sec. 6.1.2, we use memory access expressions instead of address formulas as the key to simplify the memory lookup. While the memory access expressions can be acquired by checking assembly instructions, note that our analysis is launched on REIL IR; one memory access instruction is extended into multiple IR statements. Hence, we perform def-use analysis to "collapse" IR statements belonging to the same instruction and recover the corresponding memory access expression.

Critical Functions. CacheS is designed to perform both inter and intra-procedural analysis on any binary code component. For the evaluations in this research, instead of starting from the program entry point, analyses were launched on critical functions of cryptosystems that have become the target for many previous attacks. Such critical functions are the starting points of our interprocedural analysis, and we recursively discover all the reachable functions on the call graph. As reported in our evaluation (see Table 2), these recursively collected functions usually form a non-trivial subgraph on the program call graph. In addition, taking these critical functions as the starting points of CacheS makes it easier to compare our findings with existing work.

8 Evaluation

In contrast to many previous studies in which cache-based side channels are detected from only simple cases, CacheS is evaluated on several real-world cryptosystems. As reported in Table 1, three cryptosystems are evaluated in this research. OpenSSL and Libgcrypt are widely used cryptosystems on multi-purpose computers, while mbedTLS is commonly adopted by embedded devices. Eight critical functions are selected as the starting point of our analysis, which covers major security-sensitive components in three crypto algorithm implementations: RSA, AES, and ElGamal.

To prepare CacheS inputs, we compile test programs shipped in each cryptosystem and link with the corresponding libraries. All the crypto libraries are written in C. We build each library and test program into a 32-bit ELF binary on Ubuntu 12.04 with gcc compiler (version 4.6.3).

8.1 Evaluation Result Overview

Table 2 presents the evaluation result overview. In summary, 208 information leak points are reported from the real world cryptosystems evaluated in this research. We interpret the results as promising; most of the evaluated cryptosystems contain information leaks due to cache-based side channels, and CacheS helps to pinpoint these leaks with program-wide static analysis.

It is commonly acknowledged that the table lookup implementation of the AES decryption routine is vulnerable to various real-world cache attacks. CacheS identifies 32 information leaks from the AES implementations of OpenSSL (versions 1.0.2f and 1.0.2k), and 64 leaks from mbedTLS. Indeed, all of these issues are lookup table queries via direct usages of secrets, which is consistent with findings in existing research [25, 77].

Existing research has pinpointed multiple information leaks in the modular exponentiation implementation of OpenSSL and Libgcrypt [77, 52]; vulnerable functions are adopted by both RSA and ElGamal for decryption. CacheS confirmed these findings (see Sec. 8.4 for one false positive in OpenSSL). Furthermore, CacheS successfully revealed a much larger information leakage surface than existing trace and static analysis based techniques, because of its scalable modeling of program semantics. Table 2 shows that CacheS identifies more information leaks from Libgcrypt and OpenSSL in addition to confirming all issues reported by CacheD [77]. Moreover, CacheS identifies multiple information leakage sites from the modular exponentiation implementation of mbedTLS, which, to the best of our knowledge, is unknown to the research community.

While 40 information leakage sites are reported in Libgcrypt (version 1.6.1), our study shows that they have been fixed in version 1.7.3. Without secret-dependent memory accesses, the RSA/ElGamal implementation of Libgcrypt 1.7.3 is generally accepted as safe regarding our threat model. Our evaluation reports consistent findings that no leak is detected regarding our threat model on secret-dependent cache-line accesses (but we do find secret-dependent control flows, see Sec. 8.5).

Computing Resource. Our evaluation is launched on a machine with 2.90 GHz Intel Xeon(R) E5-2690 CPU and 128 GB memory. For each context-sensitive analysis campaign, Table 2 presents the covered functions, contexts, and processed IR instructions. We report that CacheS takes less than 1700 CPU seconds to process all the test cases, and on average the peak memory usage to evaluate one case is less than 5 GB. Overall, CacheS finished all the analysis campaigns with reasonable amount of computing resources, and we interpret that the promising results demonstrate the high scalability of CacheS in analyzing real-world cryptosystems.

Modeling Program Semantics with Logic Formulas. As noted in Sec. 3, we tentatively implement the idea of modeling program concrete semantics with logic formulas. Note that in addition to the semantics modeling, all the design and evaluation settings are unchanged.

Table 2: Evaluation result overview. We compare the identified information leakage sites by CacheS with a recent research (CacheD [77]), and we report CacheS can identify all the leakage sites reported by CacheD. A summary of all leaks can be found at the extended version of this paper [76].

Algorithm	Implementation	Information Leakage Sites (known/unknown)	# of Analyzed Procedures	# of Analyzed Contexts	Processing Time (CPU Seconds)	# of Processed REIL Instructions	Peak Memory Usage (MB)	Information Leakage Units	Results Reported in CacheD [77]		
									Leakage Sites	Processing Time	Leakage Units
RSA/ElGamal	Libgcrypt 1.6.1	22/18	60	81	228.8	50,436	7,749	11	22	14293.6	5
RSA/ElGamal	Libgcrypt 1.7.3	0/0	59	59	182.2	33,386	5,823	0	0	11626.0	0
RSA/ElGamal	OpenSSL 1.0.2k	2/3	71	81	179.2	83,183	6,134	2	N/A	N/A	N/A
RSA/ElGamal	OpenSSL 1.0.2f	2/4	68	72	169.5	80,096	6,113	3	2	165.6	2
RSA	mbedTLS 2.5.1	0/29	29	36	775.9	35,963	9,654	2	N/A	N/A	N/A
AES	OpenSSL 1.0.2k	32/0	1	1	33.2	3,748	620	1	N/A	N/A	N/A
AES	OpenSSL 1.0.2f	32/0	1	1	35.8	3,748	578	1	32	48.5	1
AES	mbedTLS 2.5.1	64/0	1	1	32.8	4,803	619	1	N/A	N/A	N/A
Total		154/54	290	332	1,637.4	295,363	37,290	21	56	26,133.7	8

Table 3: Model program semantics in the logic formulas $l \in \mathbf{L}$ and **SAS** and test OpenSSL 1.0.2k. The second and third rows report the modeling results with logic formulas, while the last row reports results in **SAS**. The comparison of these two program modelings is given in Sec. 3.

Algorithm	Execution Time (CPU Second)	# of Processed Function	# of Processed Context	Peak Memory Usage (MB)	Detected Leaks
RSA/ElGamal	timeout (> 5 CPU hours)	15	28	7,283	N/A
AES	timeout (> 5 CPU hours)	1	1	47,798	N/A
RSA/ElGamal	timeout (> 5 CPU hours)	28	85	53,054	N/A
AES	115.8	1	1	621	32
RSA/ElGamal	179.2	71	81	6,134	5
AES	33.2	1	1	620	32

The first two rows of Table 3 give the evaluation results for the AES and RSA/ElGamal implementations in OpenSSL 1.0.2k, both of which report a "timeout" after 5 CPU hours. As explained in Sec. 3, we extend the prototype with loop induction variable detection, and the third row reports the results of the re-launched tests. Still, the RSA/ElGamal case throws a timeout (a reflection on this tentative evaluation is given in Sec. 3). In summary, we interpret that the **SAS** proposed in this research has largely improved the analysis scalability, which serves as an indispensable component to pinpoint cache-based timing channels in real-world cryptosystems.

Comparison with CacheAudit.[2] Besides CacheD [77], we also compare our results with CacheAudit [32]. CacheAudit failed on all of our test cases for two reasons. First, two of our cases contain some x86 instructions that are not handled by CacheAudit. Second, CacheAudit refuses to analyze indirect function calls when constructing the control flow graph. In addition, we also describe the key differences between CacheS and CacheAudit in Sec. 10.

Identifying Information Leakage Units. Considering some occurrences of information leaks are on adjacent lines of a code component (a summary of all leaks can be found at the extended version of this paper [76]), once a leak is flagged by CacheS, presumably any competent programmer shall spot and remove all the related defects. Therefore, we group the flagged information leaks to assess the utility of CacheS and also estimate the bug fixing effort. Though it can be slightly subjective, we propose a metric according to the source code locations of defects: information leaks will be grouped together as a "leakage unit" if they are within the same or adjacent C statements (e.g., within the same loop or adjacent if branches). Also, if a macro is expanded at different program points (e.g., the macro `MPN_COPY` which contains information leaks in Libgcrypt 1.6.1), we count it only once.

As reported in Table 2, CacheS identified 21 units of information leaks. We also grouped the findings of CacheD with the same metric. We have confirmed that CacheS covered all leakage units reported in CacheD, and further revealed new leakage units within statements or functions not covered by CacheD (e.g., 6 new leakage units in Libgcrypt 1.6.1). Overall, we interpret the evaluation results as promising; trace-based analysis, like CacheD, is incapable of modeling the program collecting semantics, and therefore underestimates the attack surface.

Confirmation with Library Authors. As shown in Table 2, we found unknown information leaks from OpenSSL (versions 1.0.2f and 1.0.2k) and mbedTLS (version 2.5.1). Our findings were reported and promptly confirmed by the OpenSSL developers [4]; the latest OpenSSL has been patched to eliminate these leaks (the leaks are discussed shortly in Sec. 8.4). At the time of writing, we are waiting for responses from the mbedTLS developers.

8.2 Exploring the Leaks in mbedTLS

Although mbedTLS developers have not confirmed our findings, we conduct further study of the 29 flagged information leakage sites from this library to check whether they can lead to cache-based side channels.

As mentioned above, the constraint solver provides at least one pair of satisfiable solutions (a pair of secrets k and k') to each leakage site (Sec. 5). To verify one leak, we instrument the program source code and modify secrets with k and k'. We then compile the instrumented programs into two binaries and monitor the execution of each binary executable via a widely-used hardware simulator (`gem5` [18]). The compiled code is fed with test cases shipped with the cryptosystems, and we use the full-system simulation mode of `gem5` to monitor the execution of the instrumented program. The full-system simulation mode uses 64-bit Ubuntu 12.04 (this mode only supports 64-bit OS) to host the application code. We compile the instrumented source code into 64-bit binaries since executing 32-bit binaries on the 64-bit OS throws some TLB translation exceptions (this issue is also reported in [77]). The configuration of `gem5` is reported in Table 4. At

[2] `https://github.com/cacheaudit/cacheaudit`

Table 4: gem5 configurations.

ISA	x86
Processor type	single core, out-of-order
L1 Cache	4-way, 32KB, 2-cycle latency
L2 Cache	8-way, 1MB, 50-cycle latency
Cache line size	64 Bytes
Cache replacement policy	LRU

Table 5: Hardware simulation results.

# of CacheS Detected Leakage Sites	# of Executed Leakage Sites	Cache Line Access Variants	Cache Status Variants
29	14	14	6

the leakage point, we intercept the cache access from CPU to L1 Data Cache; the accessed cache line and corresponding cache status (hit vs. miss) are recorded.

As shown in Table 5, among 29 information leakage sites found in mbedTLS, 14 sites are covered during simulation. We observe that different cache lines are accessed at these leakage points, when instrumenting the program with secrets k and k'. In other words, by observing the access of different cache lines, attackers will be able to infer a certain amount of secret information. In addition, we report that cache status variants (in terms of cache hit vs. miss) are observed in several cases. In summary, we interpret the verification results as highly promising; we have confirmed that all the executed information leakages are *true positives* since cache line access variants are observed.

Although the employed program inputs cannot lead to the full coverage of every leakage site, we manually checked all the uncovered cases, and we found that these cases share the same pattern as the covered leaks. For instance, the covered and uncovered leaks are the same inline assembly sequences residing within different paths. Overall, we interpret it as convincing to conclude that all the detected information leaks in mbedTLS are true positives.

8.3 Case Study of Leaks in mbedTLS

This section presents a thorough case study of several information leaks identified by our tool. As presented in Table 2, we identified 29 information leakage points in mbedTLS 2.5.1. In particular, the first four leaks were found in the function `mpi_montmul` (source code is given in Fig. 6(a)), which is a major component of the modular exponentiation implementation in mbedTLS. The value of function parameter `B` is derived from a window size of the secret key (line 2). In `mpi_montmul`, `B` is used as a pointer to access elements in a C struct (line 6, line 10, line 11). We envision that different program secrets would derive into different values of `B`, which further lead to the access of different cache lines in secret-dependent memory accesses.

The evaluation shows consistent findings. As shown in Fig. 6(b), CacheS identifies four suspicious memory accesses in `mpi_montmul` (two pointer dereferences at line 6 of Fig. 6(a) are optimized into one memory load at line 2 of Fig. 6(b)). By checking the constraint solver, we find a pair of program secrets that affect the value of `B` and further lead to the access of different cache lines at the first memory access (the solution is given in Fig. 6(c)).

We then instrument the program private key with the solver provided solutions in Fig. 6(c) and observe the runtime cache access within `gem5`. This secret pair is generated by analyzing the first leakage memory access, but since variants of `B` may affect the following memory traffic as well, we report the cache status at all the suspicious memory accesses in `mpi_montmul`. We note that while CacheS analyzes 32-bit binaries, at this step we compile the instrumented source code into 64-bit binaries since the simulated OS throws some exceptions when running 32-bit code. After compilation, the five leakage points in the source code actually produce three memory load instructions in the 64-bit assembly code. Cache behaviors, including the accessed cache line and the corresponding cache status, are recorded at these points. Fig. 6(d) presents the simulation results. Due to the limited space, we provide only the first seven records (59568 records in total). Program counters 0x40770a, 0x407744 and 0x40775d represent the three identified memory loads of information leaks. It is easy to see that different cache lines are accessed at each point. Additionally, a timing window of one cache hit vs. miss is found (this memory access represents a table lookup in the first element of `B->p`).

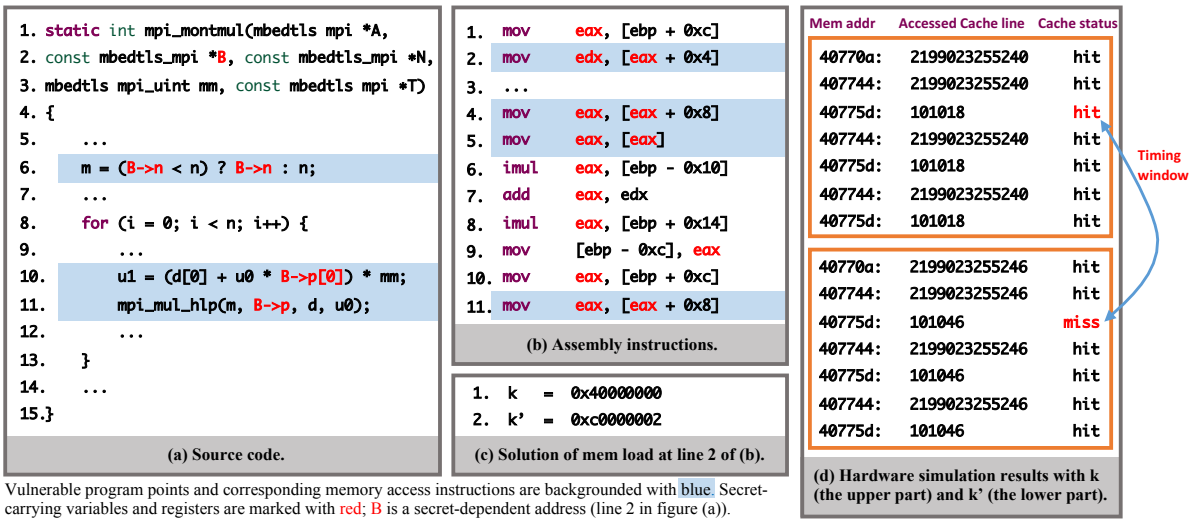
8.4 Information Leaks in the Modular Exponentiation Algorithm

Both RSA and ElGamal algorithms employ the modular exponentiation algorithm for decryption. Existing research has reported that such an algorithm is vulnerable to cache-based timing channel attacks [77, 52]. Here, we evaluate the corresponding implementations in OpenSSL, Libgcrypt, and mbedTLS. As reported in Table 2, CacheS successfully revealed a much larger leakage surface, including 80 (54 unknown and 26 known) information leaks, from our test cases.

Information Leaks in Libgcrypt. A large number of leakage points are reported from the sliding window-based modular exponentiation implementation in Libgcrypt 1.6.1. Existing research has pointed out the direct usages of (window-size) secret keys as *exploitable* [52], and CacheS pinpointed this issue. In addition to the 4 direct usages of secrets, we further uncovered 36 leaks due to the propagation of secret information flows, as CacheS keeps track of both variable-level and memory loading based information flows (Sec. 6.2).

While previous trace-based analysis also keeps track of information flow propagation (i.e., CacheD [77]), CacheS still outperforms CacheD because of its program-wide analysis. With the help of CacheD's authors, we confirmed that CacheS can detect all 22 leaks reported in CacheD [77], and further reveals 18 additional points.

Information Leaks in mbedTLS. CacheS has also identified leaks in another commonly used cryptosystem, mbedTLS. Appendix E presents several leaks found in the mbedTLS case. In general, function `mbedtls_mpi_exp_mod` implements a sliding window-



Vulnerable program points and corresponding memory access instructions are backgrounded with blue. Secret-carrying variables and registers are marked with red; B is a secret-dependent address (line 2 in figure (a)).

Figure 6: Case study of information leaks in mbedTLS. The constraint solver finds a pair of secrets (`k` and `k'`) which leads to the access of different cache lines at line 2 of (b).

based modular exponentiation, which leads to secret-dependent memory accesses (precomputed table lookup). The table lookup statement (line 10) does not generate a leak point since it only gets a pointer referring to an array element, however, further memory dereferences on the acquired pointer reveal 4 direct usages of secrets (discussed in Sec. 8.3). We also find 25 leaks due to the propagation of secret information flows (Sec. 6.2).

We note that mbedTLS uses RSA exponent blinding as a countermeasure [43], which practically introduces noise and mitigates cache side channels (but is still exploitable with enough collisions or if the attacker can derive the exponent from a single trace). We leave it as future work to model the feasibility of exploitations, given the identified leaks and also taking program randomness (e.g., exponent blinding) into consideration.

Information Leaks in OpenSSL. Information leaks in OpenSSL are within or derived from functions counting the length of a secret array. Fig. 7 presents a function that contains 4 memory accesses which engender information leaks: `BIGNUM` maintains an array of 32-bit elements, and `BN_num_bits` counts the number of bits within a given big number. Since the last element of the array may be less than 32 bits, a lookup table is used to determine the exact bits in the last element of the secret array (function call at line 7 in Fig. 7). By storing secrets within a big number structure, table queries in `BN_num_bits_word` could lead to secret-dependent memory accesses.

While CachedD [77] flags only one information leak (line 26 in Fig. 7) covered by its execution trace, CacheS detects more leaks. As shown in Fig. 7, four table queries are analyzed by CacheS, and all of them are flagged as leaks. In addition to these four direct usages of secrets, CacheS also finds one more leak in OpenSSL 1.0.2k, and two in OpenSSL 1.0.2f, both are due to the propagation of secret information flows.

False Positive. In addition to the issues in Appendix C that have been confirmed and fixed by the OpenSSL developers, we also find one *false positive* when analyzing OpenSSL 1.0.2f. To defeat side channel attacks against the precomputed table lookup, OpenSSL forces the cache access at the table lookup point in a constant order [33]. This constant order table lookup is demonstrated with a sample C code in Appendix D. The base address of the lookup table is aligned to zero the least-significant bits, and scatter and gather methods are employed to mimic the Fortran-style memory allocation to access the table in a constant order and remove timing channels.

Ideally, with the base address being aligned, the table access should not produce an information leak regarding the cache line access model (but scatter-gather implementation can also be exploited with cache bank attacks [87]. As discussed in Sec. 5, our cache access constraint can be further extended to capture cache bank side channels). However, since public information (e.g., base address of the table) is abstracted as a symbol p in CacheS, the alignment is not modeled. Therefore, CacheS incorrectly flags the table lookup as a leak point, which leads to a false positive.

8.5 Flag Secret-Dependent Control Flow

To reduce false negatives and also show the versatility of CacheS, we extend CacheS to search for secret-dependent branch conditions. Similar to the detection of secret-dependent memory accesses (Sec. 5), we check each conditional jump and flag secret-dependent jump conditions. The conditional jump in REIL IR is `jcc`, and the value of its first operand specifies whether the jump is taken or not. We trans-

late each secret-dependent condition c into an SMT formula f and solve the following constraint:

$$f \neq f[s_i'/s_i] \tag{2}$$

where a satisfiable solution indicates that different secrets lead to the execution of different branches. In addition, since REIL IR creates *additional* `jcc` statements to model certain x86 instructions (e.g., the shift arithmetic right `sar` and bit scan forward `bsf`), we rule out `jcc` statements if their corresponding x86 instructions are *not* conditional jumps. In this research we do not take `jcc` into consideration if it does not represent an x86 conditional jump, since in general the silicon implementations of x86 instructions on mainstream CPUs have *fixed latency* [2].

Table 6 presents the evaluation results, including the information leakage units produced by the same metric used in Table 2. While secret-dependent control flow is absent in all the AES cases, CacheS pinpoints multiple instances in every RSA/ElGamal implementation. We further manually studied each of them, and we report that besides 4 false positives (explained later in this section), all the other cases represent secret-dependent branch conditions. In the example given in Appendix F, the value of `bits`, which is derived from the private key, is used to construct several conditions. Similar patterns are also found in other cases.

False Negative. Bernstein et al. exploited the secret-dependent control flows in Libgcrypt 1.7.6 [17], where the leading and trailing zeros of a window-size secret are used to compute a branch condition. While the corresponding vulnerable branches also exist in Libgcrypt 1.7.3, they are not detected by CacheS. In general, 32-bit x86 opcode `bsr` and `bsf` are used to count the leading and trailing zeros of a given operand, and both opcodes are lifted into a while loop implemented by a `jcc` statement (for the definition of their semantics, see the `bsr` and `bsf` sections of the x86 developer manual [3]). Consider a proof-of-concept pseudo-code below:

```
1  t = 0;
2  while (getBit(t, src) == 0) //src could be a secret
3  {
4    t += 1;
5  }
6  return t; // the number of trailing zeros in src
```

where the lifted while loop entails implicit information flow, which is not supported (see Sec. 6.2 for the information flow policy). In addition, although we disable the checking of `jcc` regarding Constraint 2 if its corresponding x86 instruction is *not* a conditional jump (like the `bsr` and `bsf` cases), we report that once enabling the checking of such `jcc` statements, secret-dependent control flows (e.g., line 3 of the pseudo-code) are detected for both cases.

False Positive. We find 4 false positives when analyzing Libgcrypt 1.6.1. This is due to the imprecise modeling of interprocedural call sites. Consider a sample pseudo-code below:

```
1  foo(k, p) { // k is {⊤} and p is {12}
2    if (...) {
3      r = bar(k); // r is {⊤}
4    } else {
5      r = bar(p); // r is {⊤} since ⟨foo,{12}⟩ ⊆ ⟨foo,{⊤}⟩
6      if (r)  // false positive
7        ...
8  }
9
10 bar(i){return i;}
```

where `foo` performs two function calls to `bar` with different parameters. The summary of the first call (line 3) is represented as $(\langle foo, \{\top\}\rangle, \{\top\})$, where $\langle foo, \{\top\}\rangle$ forms the calling context (as explained in Sec. 6.3, a calling context includes the caller name and the input), and the second $\{\top\}$ is the function call output. Then the following function call (line 5) with $\langle foo, \{12\}\rangle$ as the calling context will directly return $\{\top\}$ and cause a false positive (line 6) according to the recorded summary, since $\langle foo, \{12\}\rangle \subseteq \langle foo, \{\top\}\rangle$. Our study shows that such sound albeit imprecise modeling caused 4 false positives when analyzing Libgcrypt 1.6.1.

Table 6: Secret-dependent control branches. We found no issue in the AES implementations. A summary of all leakage points can be found at the extended version of this paper [76].

Implementation	Algorithm	# of Secret-dependent conditions	False Positive	Information Leakage Unit
Libgcrypt 1.6.1	RSA/ElGamal	21	4	9
Libgcrypt 1.7.3	RSA/ElGamal	6	0	4
mbedTLS 2.5.1	RSA	8	0	4
OpenSSL 1.0.2f	RSA/ElGamal	12	0	5
OpenSSL 1.0.2k	RSA/ElGamal	12	0	5
Total		59	4	27

9 Discussion

Soundness. Our abstraction is sound (see our technical report for the proof [76]), but the CacheS implementation is soundy [53] as it roots the same assumption as previous techniques that aim to find bugs rather than performing rigorous verification [55, 84, 54].

CacheS adopts a lightweight but unsound memory model implementation; program state representations are optimized to reduce the memory usage and speed up the analysis. There is a line of research aiming to deliver a (nearly) sound memory model when analyzing x86 assembly [13, 64, 65, 21]. We leave it to future work to explore practical methods to improve CacheS with a sound model without undermining the strength of CacheS in terms of scalability and precision.

Reduce False Positives. Our abstract domain **SAS** models public program information with free public symbols. To further improve the analysis precision and eliminate false positives, such as in the case discussed in Sec. 8.4, one approach is to perform a finer-grained modeling of public program information. To this end, so-called "lazy abstraction" can be adopted to postpone abstraction until necessary [71]. In contrast to our current approach where analyses are performed directly over **SAS**, lazy abstraction provides a flexible abstraction strategy on demand, where different program points can exhibit distinct levels of precision. Well-selected program points for lazy abstraction are critical to achieve scalability. For example, abstraction can be performed at ev-

ery loop merge point or whenever abstract formulas become too large and exhaust the memory resource. We leave it to future work to explore practical strategies for lazy abstraction.

10 Related Work

Timing Attacks. Kocher's seminal paper [44] identifies timing attacks as a potential threat to crypto system. Later work finds that timing information reveals the victim program's usage of data/instruction cache, leading to efficient timing attacks against real world cryptography software, including AES [37, 60, 74, 16, 20, 6], DES [75], RSA [7, 62, 86], ElGamal [90], and ECDSA [15]. Recent work shows that such cache-based timing attacks are possible on emerging platforms, such as cloud computing, VM environments, trusted computing environments, and mobile platforms [66, 85, 83, 89, 52, 49, 56, 69, 24, 36].

Detect Cache-Based Timing Channels. CacheAudit leverages static analysis techniques (i.e., abstract interpretation) to reason information leakage due to cache side channels [32, 33]. CacheS outperforms CacheAudit due to our novel abstract domain. CacheAudit uses relational and numerical abstract domains to only infer the information leakage bound, while our abstract domain models semantics with symbolic formulas, pinpoints information leaks with constraint solving, and enables the generation of counter examples to promote debugging. In addition, we propose a principled way to improve the scalability by tracking secrets and public information with *different granularities*. This enables a context-sensitive interprocedural analysis of real-world cryptosystems for which CacheAudit is not capable of handling. Brotzman et al. [22] propose a static symbolic reasoning technique that also covers multiple program paths. However, their analysis lacks abstraction of public values, and can analyze only small-size programs.

In contract, dynamic analysis-based approaches, such as taint analysis or trace-based symbolic execution, are incapable of analyzing the whole program [77, 82, 41, 81, 38]. CacheD [77] performs symbolic execution towards a single trace to detect side channels. In contrast, abstract interpretation framework approximates the program *collecting semantics*, which formalizes program abstract semantics at arbitrary program points regarding any path and any input. This is fundamentally different and much more comprehensive comparing to a path-based tool, like CacheD. Wichelmann et al. [82] log execution traces and perform differential analysis of various granularities to detect side channels. Weiser et al. [81] detect address-based side-channels by executing test programs under input variants and further compare traces to detect leakages.

Countermeasure. Existing countermeasures against cache side-channel attacks can be categorized into hardware-based and software-based approaches. Hardware-based solutions focus on randomizing the cache accesses with new cache design [79, 80, 45, 78, 51, 50], or enforcing fine-grained isolation with respect to cache usage [70, 42]. Wang et al. propose locking the cache lines and hiding cache access patterns [79], which further obfuscates cache accesses by diversifying the cache mappings [80]. Tiwari et al. [73] devise a novel micro architecture for information-flow tracking by design, where noninterference is deployed as the baseline confidentiality property. Another direction at the hardware level is based on contracts between software and hardware [91, 47, 88], where contracts are enforced by formal methods (e.g., type systems) on the hardware side. Furthermore, some advanced hardware extensions, like hardware transactional memory, have also been leveraged to prevent side channels even inside Intel SGX [35].

Analyses are also conducted on the software level to mitigate side channel attacks [27, 12, 63, 67, 68]. Program transformation techniques are leveraged to remove control-flow timing leaks by equalizing branches of conditionals with secret guards [8], together with a binary static checker [58], and its practicality is evaluated [27]. Constant time code defeats timing attacks by ensuring the control flow, memory accesses, and execution time of individual instruction is secret independent [10, 40, 61, 14, 9, 46].

11 Conclusion

In this paper, we have presented CacheS for cache-based timing channel detection. Based on a novel abstract domain **SAS**, CacheS does fine-grained tracking of sensitive information and its dependencies, while performing scalable analysis with over-approximated public information. We evaluated CacheS on multiple real-world cryptosystems. CacheS confirmed over 154 information leaks reported by previous research and pinpointed 54 leaks not known previously.

12 Acknowledgments

We thank the Usenix Security anonymous reviewers and Gary T. Leavens for their valuable feedback. The work was supported in part by the National Science Foundation (NSF) under grant CNS-1652790, and the Office of Naval Research (ONR) under grants N00014-16-12912, N00014-16-1-2265, and N00014-17-1-2894.

References

[1] IDAPro. https://goo.gl/snmrk3.

[2] Intel® 64 and IA-32 architectures optimization reference manual.

[3] Intel® 64 and IA-32 architectures software developers manual.

[4] Patched OpenSSL vulnerabilities. `https://git.io/fj0iz`, 2018.

[5] ACIICMEZ, O., AND KOC, C. K. Trace-driven cache attacks on AES. In *ICICS* (2006).

[6] ACIICMEZ, O., SCHINDLER, W., AND KOC, C. K. Cache based remote timing attack on the AES. In *CT-RSA* (2006).

[7] ACIICMEZ, O., AND SEIFERT, J. Cheap hardware parallelism implies cheap security. In *FDTC* (2007).

[8] AGAT, J. Transforming out timing leaks. In *POPL* (2000).

[9] ALMEIDA, J. B., BARBOSA, M., BARTHE, G., DUPRESSOIR, F., AND EMMI, M. Verifying constant-time implementations. In *USENIX Sec.* (2016).

[10] ALMEIDA, J. B., BARBOSA, M., PINTO, J. S., AND VIEIRA, B. Formal verification of side-channel countermeasures using self-composition. *Science of Computer Programming* (2013).

[11] APPEL, A. W. *Modern Compiler Implementation in ML*. Cambridge University Press, 2004.

[12] AVIRAM, A., HU, S., FORD, B., AND GUMMADI, R. Determinating timing channels in compute clouds. In *CCSW* (2010).

[13] BALAKRISHNAN, G., AND REPS, T. Analyzing memory accesses in x86 executables. In *CC* (2004).

[14] BARTHE, G., REZK, T., AND WARNIER, M. Preventing timing leaks through transactional branching instructions. *Electronic Notes in Theoretical Computer Science* (2006).

[15] BENGER, N., VAN DE POL, J., SMART, N. P., AND YAROM, Y. "Ooh aah... just a little bit" : A small amount of side channel can go a long way. In *CHES* (2014).

[16] BERNSTEIN, D. J. Cache-timing attacks on AES, 2005.

[17] BERNSTEIN, D. J., BREITNER, J., GENKIN, D., BRUINDERINK, L. G., HENINGER, N., LANGE, T., VAN VREDENDAAL, C., AND YAROM, Y. Sliding right into disaster: Left-to-right sliding windows leak. In *CHES* (2017).

[18] BINKERT, N., BECKMANN, B., BLACK, G., REINHARDT, S. K., SAIDI, A., BASU, A., HESTNESS, J., HOWER, D. R., KRISHNA, T., SARDASHTI, S., SEN, R., SEWELL, K., SHOAIB, M., VAISH, N., HILL, M. D., AND WOOD, D. A. The Gem5 simulator. *ACM SIGARCH Computer Architecture News* (2011).

[19] BONEH, D., DURFEE, G., AND FRANKEL, Y. An attack on RSA given a small fraction of the private key bits. In *ASIACRYPT* (1998).

[20] BONNEAU, J., AND MIRONOV, I. Cache-collision timing attacks against AES. In *CHES* (2006).

[21] BRADLEY, A. R., MANNA, Z., AND SIPMA, H. B. What's decidable about arrays? In *VMCAI* (2006).

[22] BROTZMAN, R., LIU, S., ZHANG, D., TAN, G., AND KANDEMIR, M. CaSym: Cache aware symbolic execution for side channel detection and mitigation. In *IEEE SP* (2018).

[23] BRUMLEY, D., AND BONEH, D. Remote timing attacks are practical. *Computer Networks* (2005).

[24] BULCK, V., MINKIN, M., WEISSE, O., GENKIN, D., KASIKCI, B., PIESSENS, F., SILBERSTEIN, M., WENISCH, T. F., YAROM, Y., AND STRACKX, R. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Sec.* (2018).

[25] C, A., GIRI, R. P., AND MENEZES, B. Highly efficient algorithms for aes key retrieval in cache access attacks. In *EuroSP* (2016).

[26] CALVET, J., FERNANDEZ, J. M., AND MARION, J.-Y. Aligot: Cryptographic function identification in obfuscated binary programs. In *CCS* (2012).

[27] COPPENS, B., VERBAUWHEDE, I., BOSSCHERE, K. D., AND SUTTER, B. D. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *IEEE SP* (2009).

[28] COUSOT, P., AND COUSOT, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL* (1977).

[29] DANIAL, A. CLOC. `https://goo.gl/3KFACB`.

[30] DE MOURA, L., AND BJØRNER, N. Z3: An efficient SMT solver. In *TACAS* (2008).

[31] DISSELKOEN, C., KOHLBRENNER, D., PORTER, L., AND TULLSEN, D. Prime+Abort: A timer-free high-precision L3 cache attack using Intel TSX. In *USENIX Sec.* (2017).

[32] DOYCHEV, G., FELD, D., KOPF, B., MAUBORGNE, L., AND REINEKE, J. CacheAudit: A tool for the static analysis of cache side channels. In *USENIX Sec.* (2013).

[33] DOYCHEV, G., AND KÖPF, B. Rigorous analysis of software countermeasures against cache attacks. In *PLDI* (2017).

[34] GOOGLE. BinNavi. `https://github.com/google/binnavi`, 2017.

[35] GRUSS, D., LETTNER, J., SCHUSTER, F., OHRIMENKO, O., HALLER, I., AND COSTA, M. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX Sec.* (2017).

[36] GRUSS, D., MAURICE, C., FOGH, A., LIPP, M., AND MANGARD, S. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *CCS* (2016).

[37] GULLASCH, D., BANGERTER, E., AND KRENN, S. Cache games—bringing access-based cache attacks on AES to practice. In *IEEE SP* (2011).

[38] GUO, S., WU, M., AND WANG, C. Adversarial symbolic execution for detecting concurrency-related cache timing leaks. In *FSE* (2018).

[39] HE, Z., AND LEE, R. B. How secure is your cache against side-channel attacks? In *MICRO* (2017).

[40] HEDIN, D., AND SANDS, D. Timing aware information flow security for a JavaCard-like bytecode. *Electronic Notes in Theoretical Computer Science* (2005).

[41] IRAZOQUI, G., CONG, K., GUO, X., KHATTRI, H., KANUPARTHI, A. K., EISENBARTH, T., AND SUNAR, B. Did we learn from LLC side channel attacks? A cache leakage detection tool for crypto libraries. *CoRR* (2017).

[42] KIM, T., PEINADO, M., AND MAINAR-RUIZ, G. Stealthmem: System-level protection against cache-

based side channel attacks in the cloud. In *USENIX Sec.* (2012).

[43] KOCHER, P. C. Timing attacks on implementations of Diffie–Hellman, RSA, DSS, and other systems. In *CRYPTO* (1996).

[44] KOCHER, P. C. *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*. 1996.

[45] KONG, J., ACIICMEZ, O., SEIFERT, J. P., AND ZHOU, H. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *HPCA* (2009).

[46] KÖPF, B., AND RYBALCHENKO, A. Approximation and randomization for quantitative information-flow analysis. In *CSF* (2010).

[47] LI, X., KASHYAP, V., OBERG, J. K., TIWARI, M., RAJARATHINAM, V. R., KASTNER, R., SHERWOOD, T., HARDEKOPF, B., AND CHONG, F. T. Sapper: A language for hardware-level security policy enforcement. In *ASPLOS* (2014).

[48] Libgcrypt. `https://www.gnu.org/software/libgcrypt/`.

[49] LIPP, M., GRUSS, D., SPREITZER, R., MAURICE, C., AND MANGARD, S. Armageddon: Cache attacks on mobile devices. In *USENIX Sec.* (2016).

[50] LIU, F., GE, Q., YAROM, Y., MCKEEN, F., ROZAS, C., HEISER, G., AND LEE, R. B. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *HPCA* (2016).

[51] LIU, F., AND LEE, R. B. Random fill cache architecture. In *MICRO* (2014).

[52] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. Last-level cache side-channel attacks are practical. In *IEEE S&P* (2015).

[53] LIVSHITS, B., SRIDHARAN, M., SMARAGDAKIS, Y., LHOTÁK, O., AMARAL, J. N., CHANG, B.-Y. E., GUYER, S. Z., KHEDKER, U. P., MØLLER, A., AND VARDOULAKIS, D. In defense of soundiness: A manifesto. *Commun. ACM* (2015).

[54] LIVSHITS, V. B., AND LAM, M. S. Tracking pointers with path and context sensitivity for bug detection in c programs. *SIGSOFT Softw. Eng. Notes* (2003).

[55] MACHIRY, A., SPENSKY, C., CORINA, J., STEPHENS, N., KRUEGEL, C., AND VIGNA, G. DR. CHECKER: A soundy analysis for linux kernel drivers. In *USENIX* (2017).

[56] MAURICE, C., WEBER, M., SCHWARZ, M., GINER, L., GRUSS, D., BOANO, C. A., MANGARD, S., AND RÖMER, K. Hello from the other side: SSH over robust cache covert channels in the cloud. In *NDSS* (2017).

[57] mbedtls. `https://tls.mbed.org/`.

[58] MOLNAR, D., PIOTROWSKI, M., SCHULTZ, D., AND WAGNER, D. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *ICISC* (2005).

[59] Openssl. `https://www.openssl.org/`.

[60] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache attacks and countermeasures: the case of AES. *CT-RSA* (2006).

[61] PASAREANU, C., PHAN, Q.-S., AND MALACARIA, P. Multi-run side-channel analysis using symbolic execution and Max-SMT. In *CSF* (2016).

[62] PERCIVAL, C. Cache missing for fun and profit. In *BSDCan* (2005).

[63] RAJ, H., NATHUJI, R., SINGH, A., AND ENGLAND, P. Resource management for isolation enhanced cloud services. In *CCSW* (2009).

[64] REPS, T., AND BALAKRISHNAN, G. Improved memory-access analysis for x86 executables. In *CC* (2008).

[65] REYNOLDS, J. C. Reasoning about arrays. *Commun. ACM* (1979).

[66] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *CCS* (2009), ACM.

[67] SCHWARZ, M., LIPP, M., AND GRUSS, D. Javascript zero: Real javascript and zero side-channel attacks. In *NDSS* (2018).

[68] SCHWARZ, M., LIPP, M., GRUSS, D., WEISER, S., MAURICE, C., SPREITZER, R., AND MANGARD, S. Keydrown: Eliminating software-based keystroke timing side-channel attacks. In *NDSS* (2018).

[69] SCHWARZ, M., WEISER, S., GRUSS, D., MAURICE, C., AND MANGARD, S. Malware guard extension: Using SGX to conceal cache attacks. In *DIMVA* (2017).

[70] SHI, J., SONG, X., CHEN, H., AND ZANG, B. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *DSNW* (2011).

[71] THAKUR, A. V., ELDER, M., AND REPS, T. W. Bilateral algorithms for symbolic abstraction. In *SAS* (2012).

[72] THOMAS, D., AND PORST, S. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. In *CanSecWest* (2009).

[73] TIWARI, M., OBERG, J. K., LI, X., VALAMEHR, J., LEVIN, T., HARDEKOPF, B., KASTNER, R., CHONG, F. T., AND SHERWOOD, T. Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. In *ACM SIGARCH Computer Architecture News* (2011), ACM.

[74] TROMER, E., OSVIK, D., AND SHAMIR, A. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology 23*, 1 (2010), 37–71.

[75] TSUNOO, Y., SAITO, T., SUZAKI, T., SHIGERI, M., AND MIYAUCHI, H. Cryptanalysis of DES implemented on computers with cache. In *CHES* (2003).

[76] WANG, S., BAO, Y., LIU, X., WANG, P., ZHANG, D., AND WU, D. Identifying cache-based side chan-

nels through secret-agumented abstract interpretation. In *Arxiv* (2019).

[77] WANG, S., WANG, P., LIU, X., ZHANG, D., AND WU, D. CacheD: Identifying cache-based timing channels in production software. In *USENIX Sec.* (2017).

[78] WANG, Z., AND LEE, R. B. Covert and side channels due to processor architecture. In *ACSAC* (2006).

[79] WANG, Z., AND LEE, R. B. New cache designs for thwarting software cache-based side channel attacks. In *ISCA* (2007).

[80] WANG, Z., AND LEE, R. B. A novel cache architecture with enhanced performance and security. In *MICRO* (2008).

[81] WEISER, S., ZANKL, A., SPREITZER, R., MILLER, K., MANGARD, S., AND SIGL, G. DATA – differential address trace analysis: Finding address-based side-channels in binaries. In *USENIX Sec.* (2018).

[82] WICHELMANN, J., MOGHIMI, A., EISENBARTH, T., AND SUNAR, B. MicroWalk: A framework for finding side channels in binaries. In *ACSAC* (2018).

[83] WU, Z., XU, Z., AND WANG, H. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *USENIX Sec.* (2012).

[84] XIE, Y., AND AIKEN, A. Scalable error detection using boolean satisfiability. In *POPL* (2005).

[85] XU, Y., BAILEY, M., JAHANIAN, F., JOSHI, K., HILTUNEN, M., AND SCHLICHTING, R. An exploration of L2 cache covert channels in virtualized environments. In *CCSW* (2011).

[86] YAROM, Y., AND FALKNER, K. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Sec.* (2014).

[87] YAROM, Y., GENKIN, D., AND HENINGER, N. CacheBleed: A timing attack on OpenSSL constant time RSA. Tech. rep., Cryptology ePrint Archive, Report 2016/224, 2016.

[88] ZHANG, D., WANG, Y., SUH, G. E., AND MYERS, A. C. A hardware design language for timing-sensitive information-flow security. In *ASPLOS* (2015).

[89] ZHANG, Y., JUELS, A., OPREA, A., AND REITER, M. K. HomeAlone: Co-residency detection in the cloud via side-channel analysis. In *IEEE SP* (2011).

[90] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-VM side channels and their use to extract private keys. In *CCS* (2012).

[91] ZHANG, Y., AND REITER, M. K. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *CCS* (2013).

A SAS as a Lattice

To further make **SAS** a lattice, we will need to specify a top element $\top \in \mathbf{SAS}$, a bottom element $\bot \in \mathbf{SAS}$, and a join operator $\sqcup$ over **SAS**.

Set Collapse and Bound. Each element in **SAS** is a set of abstract values $f \in \mathbf{AV}$. Considering f with different degrees of abstractions may exist in one set, here we define reasonable rules to "collapse" elements in a set. The "collapse" function $\mathrm{COL} : \mathbf{SAS} \to \mathbf{SAS}$ is given by:

$$\mathrm{COL}(X) = \begin{cases} \{\top\} & \text{if } \top \in X \\ \{\top\} & \text{else if } p \in X \wedge \mathbf{AV}_s \cap X \neq \varnothing \\ \{p\} & \text{else if } p \in X \wedge \mathbf{AV}_s \cap X = \varnothing \\ X & \text{otherwise} \end{cases}$$

While the first three rules introduce single symbols as a safe and concise approximation, the last rule preserve a set in **SAS**.

In addition, each set in **SAS** is also bounded with a maximum size of N through function BOU as follows:

$$\mathrm{BOU}(X) = \begin{cases} \{\top\} & \text{if } |X| > N \wedge \mathbf{AV}_s \cap X \neq \varnothing \\ \{p\} & \text{else if } |X| > N \wedge \mathbf{AV}_s \cap X = \varnothing \\ X & \text{otherwise} \end{cases}$$

Hence, the abstract value set of any variable is bounded by N during computations within **SAS**, which practically speed ups the analysis convergence (N is set as 50 in this research, see Appendix B for a discussion of different configurations).

With COL and BOU defined, we can finally complete **SAS** as a lattice.

Claim 2. $\mathbf{SAS} = \mathscr{P}(\mathbf{AV})$ *forms a lattice with the top element*

$$\top_{\mathbf{SAS}} = \{\top\}$$

bottom element

$$\bot_{\mathbf{SAS}} = \{\}$$

and the join operator

$$\sqcup = \mathrm{BOU} \circ \mathrm{COL} \circ \cup$$

For further discussion of **SAS**, including the concrete and abstract semantics, soundness proof, etc., please refer to the extended version of this paper [76].

B Evaluating Different Configurations of the BOU Function

The definition of the BOU function includes a parameter N as the maximum size of each abstract value set. Table 7 reports the evaluation results of CacheS with respect to different N. As expected, with the increase of the allowed size, analyses took more time before reaching the fixed point. Also, when the allowed size is small (i.e., N is 1 or 10), the value set of certain registers is lifted into $\{p\}$ rapidly and terminates the analysis due to memory write accesses through p (see Sec. 6.2; we terminate the analysis for memory access of p since it rewrites the whole memory). The full evaluation data in terms of different configurations is available in the extended paper [76].

Table 7: Evaluating different configurations of BOU. When N is set as 1 and 10, several analyses terminated before reaching the fixed point due to memory write accesses through the public symbol p. The full evaluation data in terms of each configuration can be found at [76].

Value of N	True Positive	False Positive	Processing Time (CPU Seconds)
1	N/A	N/A	N/A
10	167	1	584.5
25	207	1	1,446.8
50 (the default config)	207	1	1,637.4
100	207	1	3,563.46

C Unknown Information Leaks in OpenSSL

```
int BN_num_bits(const BIGNUM *a) {
 int i = a->top - 1;
 bn_check_top(a);

 if (BN_is_zero(a))
   return 0;
 return ((i * BN_BITS2) + BN_num_bits_word(a->d[i]));
}

int BN_num_bits_word(BN_ULONG l) {
  static const char bits[256]={
    0,1,2,2,3,3,3,3,4,4,4,4,4,4,4,4,
    ...
    8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,
  };
  if (l & 0xffff0000L) {
    if (l & 0xff000000L)
      return bits[l >> 24] + 24;
    else
      return bits[l >> 16] + 16;
  }
  else {
    if (l & 0xff00L)
      return bits[l >> 8] + 8;
  else
      return bits[l];
  }
}
```

Figure 7: RSA information leaks found in OpenSSL (1.0.2f). Program secrets and their dependencies are marked as red and the leakage points are **boldfaced**.

D Scatter & Gather Methods in OpenSSL

```
char* align(char* buf) {
  uintptr_t addr = (uintptr_t) buf;
  return (char*)(addr - (addr&(BLOCK_SZ-1)) + BLOCK_SZ);
}

void scatter(char* buf, char p[][16], int k) {
  for (int i = 0; i < N; i++) {
    buf[k+i*spacing] = p[k][i];
  }
}

void gather(char* r,char* buf,int k) {
  for (int i = 0; i < N; i++) {
    r[i] = buf[k+i*spacing];
  }
}
```

Figure 8: Simple C program demonstrating the scatter & gather methods in OpenSSL to remove timing channels. This program should be secure regarding our threat model, but it would become insecure by skipping the alignment function.

E Unknown Information Leaks in mbedTLS

```
int mbedtls_mpi_exp_mod(mbedtls_mpi *X, mbedtls_mpi *A,
  mbedtls_mpi *E, mbedtls_mpi *N, mbedtls_mpi *_RR)
{
  ...
  while (1) {
    ei = (E->p[nblimbs] >> bufsize) & 1;
    ...
    wbits |= (ei << (wsize - nbits));
    ...
    mpi_montmul(X, &W[wbits], N, mm, &T);
  }
  ...
}

static int mpi_montmul(mbedtls_mpi *A, mbedtls_mpi *B,
  mbedtls_mpi *N, mbedtls_mpi_uint mm, mbedtls_mpi *T)
{
   ...
   m = (B->n < n) ?  B->n :  n;
   for(i = 0; i < n; i++)
   {
     u1 = (d[0] + u0 * B->p[0]) * mm;
     mpi_mul_hlp(m, B->p, d, u0);
   }
   ...
}
```

Figure 9: RSA information leaks found in mbedTLS (2.5.1). Program secrets and their dependencies are marked as red and the leakage points are **boldfaced**.

F Secret-Dependent Branch Conditions in OpenSSL

```
int BN_mod_exp_mont_consttime(BIGNUM *rr,
  const BIGNUM *a, const BIGNUM *p,
  const BIGNUM *m, BN_CTX *ctx,
  BN_MONT_CTX *in_mont) {
 ...
 bits = BN_num_bits(p);
 if (bits == 0)
   ...

 window = BN_window_bits_for_exponent_size(bits);
 for (wvalue = 0, i = bits%window; i>=0; i--,bits--)
 {
   ...
   while (bits >= 0){
     ...
   }
 }
  ...
}

#define BN_window_bits_for_exponent_size(b) \
        ((b) > 671 ? 6 : \
         (b) > 239 ? 5 : \
         (b) > 79 ? 4 : \
         (b) > 23 ? 3 : 1)
```

Figure 10: Several secret-dependent branch conditions found in OpenSSL (1.0.2f). Program secrets and their dependencies are marked as red and the information leakage conditions are **boldfaced**. Note that the output of BN_num_bits depends on the private key.

SCATTERCACHE: Thwarting Cache Attacks via Cache Set Randomization

Mario Werner, Thomas Unterluggauer, Lukas Giner,
Michael Schwarz, Daniel Gruss, Stefan Mangard
Graz University of Technology

Abstract

Cache side-channel attacks can be leveraged as a building block in attacks leaking secrets even in the absence of software bugs. Currently, there are no practical and generic mitigations with an acceptable performance overhead and strong security guarantees. The underlying problem is that caches are shared in a predictable way across security domains.

In this paper, we eliminate this problem. We present SCATTERCACHE, a novel cache design to prevent cache attacks. SCATTERCACHE eliminates fixed cache-set congruences and, thus, makes eviction-based cache attacks unpractical. For this purpose, SCATTERCACHE retrofits skewed associative caches with a keyed mapping function, yielding a security-domain-dependent cache mapping. Hence, it becomes virtually impossible to find fully overlapping cache sets, rendering current eviction-based attacks infeasible. Even theoretical statistical attacks become unrealistic, as the attacker cannot confine contention to chosen cache sets. Consequently, the attacker has to resort to eviction of the entire cache, making deductions over cache sets or lines impossible and fully preventing high-frequency attacks. Our security analysis reveals that even in the strongest possible attacker model (noise-free), the construction of a reliable eviction set for PRIME+PROBE in an 8-way SCATTERCACHE with 16384 lines requires observation of at least 33.5 million victim memory accesses as compared to fewer than 103 on commodity caches. SCATTERCACHE requires hardware and software changes, yet is minimally invasive on the software level and is fully backward compatible with legacy software while still improving the security level over state-of-the-art caches. Finally, our evaluations show that the runtime performance of software is not curtailed and our design even outperforms state-of-the-art caches for certain realistic workloads.

1 Introduction

Caches are core components of today's computing architectures. They bridge the performance gap between CPU cores and a computer's main memory. However, in the past two decades, caches have turned out to be the origin of a wide range of security threats [10, 15, 27, 38, 39, 43, 44, 51, 76]. In particular, the intrinsic timing behavior of caches that speeds up computing systems allows for cache side-channel attacks (cache attacks), which are able to recover secret information.

Historically, research on cache attacks focused on cryptographic algorithms [10, 44, 51, 76]. More recently, however, cache attacks like PRIME+PROBE [44, 48, 51, 54, 62] and FLUSH+RELOAD [27, 76] have also been used to attack address-space-layout randomization [23, 25, 36], keystroke processing and inter-keystroke timing [26, 27, 60], and general purpose computations [81]. For shared caches on modern multi-core processors, PRIME+PROBE and FLUSH+RELOAD even work across cores executing code from different security domains, e.g., processes or virtual machines.

The most simple cache attacks, however, are covert channels [46, 48, 72]. In contrast to a regular side-channel attack, in a covert channel, the "victim" is colluding and actively trying to transmit data to the attacker, e.g., running in a different security domain. For instance, Meltdown [43], Spectre [38], and Foreshadow [15] use cache covert channels to transfer secrets from the transient execution domain to an attacker. These recent examples highlight the importance of finding practical approaches to thwart cache attacks.

To cope with cache attacks, there has been much research on ways to identify information leaks in a software's memory access pattern, such as static code [19, 20, 41, 45] and dynamic program analysis [34, 71, 74, 77]. However, mitigating these leaks both generically and efficiently is difficult. While there are techniques to design software without address-based information leaks, such as unifying control flow [17] and bitsliced implementations of cryptography [37, 40, 58], their general application to arbitrary software remains difficult. Hence, protecting against cache attacks puts a significant burden on software developers aiming to protect secrets in the view of microarchitectural details that vary a lot across different Instruction-Set Architecture (ISA) implementations.

A different direction to counteract cache attacks is to design

more resilient cache architectures. Typically, these architectures modify the cache organization in order to minimize interference between different processes, either by breaking the trivial link between memory address and cache index [22, 55, 67, 69, 70] or by providing exclusive access to cache partitions for critical code [53, 57, 69]. While cache partitioning completely prevents cache interference, its rather static allocation suffers from scalability and performance issues. On the other hand, randomized cache (re-)placement [69, 70] makes mappings of memory addresses to cache indices random and unpredictable. Yet, managing these cache mappings in lookup tables inheres extensive changes to the cache architecture and cost. Finally, the introduction of a keyed function [55, 67] to pseudorandomly map the accessed memory location to the cache-set index can counteract PRIME+PROBE attacks. However, these solutions either suffer from a low number of cache sets, weakly chosen functions, or cache interference for shared memory and thus require to change the key frequently at the cost of performance.

Hence, there is a strong need for a practical and effective solution to thwart both cache attacks and cache covert channels. In particular, this solution should (1) make cache attacks sufficiently hard, (2) require as little software support as possible, (3) embed flexibly into existing cache architectures, (4) be efficiently implementable in hardware, and (5) retain or even enhance cache performance.

Contribution. In this paper, we present SCATTERCACHE, which achieves all these goals. SCATTERCACHE is a novel and highly flexible cache design that prevents cache attacks such as EVICT+RELOAD and PRIME+PROBE and severely limits cache covert channel capacities by increasing the number of cache sets beyond the number of physically available addresses with competitive performance and implementation cost. Hereby, SCATTERCACHE closes the gap between previous secure cache designs and today's cache architectures by introducing a minimal set of cache modifications to provide strong security guarantees.

Most prominently, SCATTERCACHE eliminates the fixed cache-set congruences that are the cornerstone of PRIME+PROBE attacks. For this purpose, SCATTERCACHE builds upon two ideas. First, SCATTERCACHE uses a keyed mapping function to translate memory addresses and the active security domain, e.g., process, to cache set indices. Second, similar to skewed associative caches [63], the mapping function in SCATTERCACHE computes a different index for each cache way. As a result, the number of different cache sets increases exponentially with the number of ways. While SCATTERCACHE makes finding fully identical cache sets statistically impossible on state-of-the-art architectures, the complexity for exploiting inevitable partial cache-set collisions also rises heavily. The reason is in part that the mapping of memory addresses to cache sets in SCATTERCACHE is different for each security domain. Hence, and as our security analysis shows, the construction of a reliable eviction set for PRIME+PROBE in an 8-way SCATTERCACHE with 16384 lines requires observation of at least 33.5 million victim memory accesses as compared to fewer than 103 on commodity caches, rendering these attacks impractical on real systems with noise.

Additionally, SCATTERCACHE effectively prevents FLUSH+RELOAD-based cache attacks, e.g., on shared libraries, as well. The inclusion of security domains in SCATTERCACHE and its mapping function preserves shared memory in RAM, but prevents any cache lines to be shared across security boundaries. Yet, SCATTERCACHE supports shared memory for inter-process communication via dedicated separate security domains. To achieve highest flexibility, managing the security domains of SCATTERCACHE is done by software, e.g., the operating system. However, SCATTERCACHE is fully backwards compatible and already increases the effort of cache attacks even without any software support. Nevertheless, the runtime performance of software on SCATTERCACHE is highly competitive and, on certain workloads, even outperforms cache designs implemented in commodity CPUs.

SCATTERCACHE constitutes a comparably simple extension to cache and processor architectures with minimal hardware cost: SCATTERCACHE essentially only adds additional index derivation logic, *i.e.*, a lightweight cryptographic primitive, and an index decoder for each scattered cache way. Moreover, to enable efficient lookups and writebacks, SCATTERCACHE stores the index bits from the physical address in addition to the tag bits, which adds $< 5\,\%$ storage overhead per cache line. Finally, SCATTERCACHE consumes one bit per page-table entry ($\approx 1.5\,\%$ storage overhead per page-table entry) for the kernel to communicate with the user space.

Outline. This paper is organized as follows. In Section 2, we provide background information on caches and cache attacks. In Section 3, we describe the design and concept of SCATTERCACHE. In Section 4, we analyze the security of SCATTERCACHE against cache attacks. In Section 5, we provide a performance evaluation. We conclude in Section 6.

2 Background

In this section, we provide background on caches, cache side-channel attacks, and resilient cache architectures.

2.1 Caches

Modern computers have a memory hierarchy consisting of many layers, each following the principle of locality, storing data that is expected to be used in the future, e.g., based on what has been accessed in the past. Modern processors have a hierarchy of caches that keep instructions and data likely to be used in the future near the execution core to avoid the latency of accesses to the slow (DRAM) main memory. This cache hierarchy typically consists of 2 to 4 layers, where the

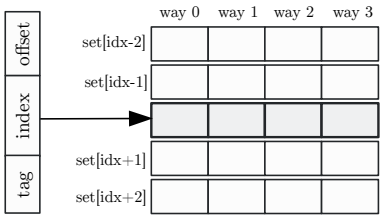


Figure 1: Indexing cache sets in a 4-way set-associative cache.

lowest layer is the smallest and fastest, typically only a few kilobytes. The last-level cache is the largest cache, typically in the range of several megabytes. On most processors, the last-level cache is shared among all cores. The last-level cache is often inclusive, *i.e.*, any cache line in a lower level cache must also be present in the last-level cache.

Caches are typically organized into *cache sets* that are composed of multiple *cache lines* or *cache ways*. The cache set is determined by computing the cache index from address bits. Figure 1 illustrates the indexing of a 4-way set-associative cache. As the cache is small and the memory large, many memory locations map to the same cache set (*i.e.*, the addresses are *congruent*). The replacement policy (e.g., pseudo-LRU, random) decides which way is replaced by a newly requested cache line. Any process can observe whether data is cached or not by observing the memory access latency which is the basis for cache side-channel attacks.

2.2 Cache Side-Channel Attacks

Cache side-channel attacks have been studied for over the past two decades, initially with a focus on cryptographic algorithms [10, 39, 51, 52, 54, 68]. Today, a set of powerful attack techniques enable attacks in realistic cross-core scenarios. Based on the access latency, an attacker can deduce whether or not a cache line is in the cache, leaking two opposite kinds of information. (1) By continuously removing (*i.e.*, evicting or flushing) a cache line from the cache and measuring the access latency, an attacker can determine whether this cache line has been accessed by another process. (2) By continuously filling a part of the cache with attacker-accessible data, the attacker can measure the contention of the corresponding part, by checking whether the attacker-accessible data remained in the cache. Contention-based attacks work on different layers:

The Entire Cache or Cache Slices. An attacker can measure contention of the entire cache or a cache slice. Maurice et al. [46] proposed a covert channel where the sender evicts the entire cache to leak information across cores and the victim observes the cache contention. A similar attack could be mounted on a cache slice if the cache slice function is known [47]. The granularity is extremely coarse, but with statistical attacks can leak meaningful information [61].

Cache Sets. An attacker can also measure the contention of a cache set. For this, additional knowledge may be required, such as the mapping from virtual addresses to physical addresses, as well as the functions mapping physical addresses to cache slices and cache sets. The attacker continuously fills a cache set with a set of congruent memory locations. Filling a cache set is also called cache-set eviction, as it evicts any previously contained cache lines. Only if some other process accessed a congruent memory location, memory locations are evicted from a cache set. The attacker can measure this for instance by measuring runtime variations in a so-called EVICT+TIME attack [51]. The EVICT+TIME technique has mostly been applied in attacks on cryptographic implementations [31, 42, 51, 65]. Instead of the runtime, the attacker can also directly check how many of the memory locations are still cached. This attack is called PRIME+PROBE [51]. Many PRIME+PROBE attacks on private L1 caches have been demonstrated [3, 14, 51, 54, 80]. More recently, PRIME+PROBE attacks on last-level caches have also been demonstrated in various generic use cases [4, 44, 48, 50, 59, 79].

Cache Lines. At a cache line granularity, the attacker can measure whether a memory location is cached or not. As already indicated above, here the logic is inverted. Now the attacker continuously evicts (or flushes) a cache line from the cache. Later on, the attacker can measure the latency and deduce whether another process has loaded the cache line into the cache. This technique is called FLUSH+RELOAD [28, 76]. FLUSH+RELOAD has been studied in a long list of different attacks [4–6, 27, 32, 35, 42, 76, 78, 81]. Variations of FLUSH+RELOAD are FLUSH+FLUSH [26] and EVICT+RELOAD [27, 42].

Cache Covert Channels

Cache covert channels are one of the simplest forms of cache attacks. Instead of an attacker process attacking a victim process, both processes collude to covertly communicate using the cache as transmission channel. Thus, in this scenario, the colluding processes are referred to as sender and receiver, as the communication is mostly unidirectional. A cache covert channel allows bypassing all architectural restrictions regarding data exchange between processes.

Cache covert channels have been shown using various cache attacks, such as PRIME+PROBE [44, 48, 73, 75] and FLUSH+RELOAD [26]. They achieve transmission rates of up to 496 kB/s [26]. Besides native attacks, covert channels have also been shown to work within virtualized environments, across virtual machines [44, 48, 75]. Even in these restricted environments, cache-based covert channels achieve transmission rates of up to 45 kB/s [48].

2.3 Resilient Cache Architectures

The threat of cache-based attacks sparked several novel cache architectures designed to be resilient against these attacks. While fixed cache partitions [53] lack flexibility, randomized

cache allocation appears to be more promising. The following briefly discusses previous designs for a randomized cache.

RPCache [69] and NewCache [70] completely disrupt the meaningful observability of interference by performing random (re-)placement of lines in the cache. However, managing the cache mappings efficiently either requires full associativity or content addressable memory. While optimized addressing logic can lead to efficient implementations, these designs differ significantly from conventional architectures.

Time-Secure Caches [67] is based on standard set-associative caches that are indexed with a keyed function that takes cache line address and Process ID (PID) as an input. While this design destroys the obvious cache congruences between processes to minimize cache interference, a comparably weak indexing function is used. Eventually, re-keying needs to be done quite frequently, which amounts to flushing the cache and thus reduces practical performance. SCATTERCACHE can be seen as a generalization of this approach with higher entropy in the indexing of cache lines.

CEASER [55] as well uses standard set-associative caches with keyed indexing, which, however, does not include the PID. Hence, inter-process cache interference is predictable based on in-process cache collisions. As a result, CEASER strongly relies on continuous re-keying of its index derivation to limit the time available for conducting an attack. For efficient implementation, CEASER uses its own lightweight cryptographic primitive designed for that specific application.

3 ScatterCache

As Section 2 showed, caches are a serious security concern in contemporary computing systems. In this section, we hence present SCATTERCACHE—a novel cache architecture that counteracts cache-based side-channel attacks by skewed pseudorandom cache indexing. After discussing the main idea behind SCATTERCACHE, we discuss its building blocks and system integration in more detail. SCATTERCACHE's security implications are, subsequently, analyzed in Section 4.

3.1 Targeted Properties

Even though contemporary transparent cache architectures are certainly flawed from the security point of view, they still feature desirable properties. In particular, for regular computations, basically no software support is required for cache maintenance. Also, even in the case of multitasking and -processing, no dedicated cache resource allocation and scheduling is needed. Finally, by selecting the cache size and the number of associative ways, chip vendors can trade hardware complexity and costs against performance as desired.

SCATTERCACHE's design strives to preserve these features while adding the following three security properties:

1. Between software defined security domains (e.g., different processes or users on the same machine, different VMs, ...), even for exactly the same physical addresses, cache lines should only be shared if cross-context coherency is required (*i.e.*, writable shared memory).
2. Finding and exploiting addresses that are congruent in the cache should be as hard as possible (*i.e.*, we want to "break" the direct link between the accessed physical address and the resulting cache set index for adversaries).
3. Controlling and measuring complete cache sets should be hard in order to prevent eviction-based attacks.

Finally, to ease the adoption and to utilize the vast knowledge on building efficient caches, the SCATTERCACHE hardware should be as similar to current cache architectures as possible.

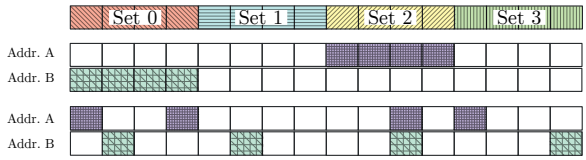


Figure 2: Flattened visualization of mapping addresses to cache sets in a 4-way set-associative cache with 16 cache lines. *Top:* Standard cache where index bits select the cache set. *Middle:* Pseudorandom mapping from addresses to cache sets. The mapping from cache lines to sets is still static. *Bottom:* Pseudorandom mapping from addresses to a set of cache lines that dynamically form the cache set in SCATTERCACHE.

3.2 Idea

Two main ideas influenced the design of SCATTERCACHE to reach the desired security properties. First, addresses should be translated to cache sets using a keyed, security-domain aware mapping. Second, which exact n_{ways} cache lines form a cache set in a n_{ways}-way associative cache should not be fixed, but depend on the currently used key and security domain too. SCATTERCACHE combines both mappings in a single operation that associates each address, depending on the key and security domain, with a set of up to n_{ways} cache lines. In other words, in a generic SCATTERCACHE, any possible combination of up to n_{ways} cache lines can form a cache set.

Figure 2 visualizes the idea and shows how it differs from related work. Traditional caches as well as alternative designs which pseudorandomly map addresses to cache sets statically allocate cache lines to cache sets. Hence, as soon as a cache set is selected based on (possibly encrypted) index bits, always the same n_{ways} cache lines are used. This means that all addresses mapping to the same cache set are congruent and enables PRIME+PROBE-style attacks.

In SCATTERCACHE, on the other hand, the cache set for a particular access is a pseudorandom selection of arbitrary n_{ways} cache lines from all available lines. As a result, there is a much higher number of different cache sets and finding addresses with identical cache sets becomes highly unlikely.

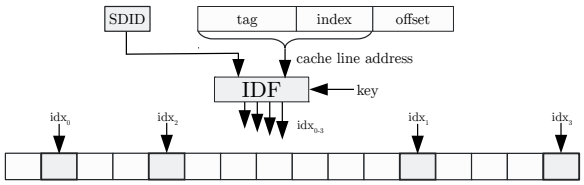


Figure 3: Idea: For an n_{ways} associative cache, n_{ways} indices into the cache memory are derived using a cryptographic IDF. This IDF effectively randomizes the mapping from addresses to cache sets as well as the composition of the cache set itself.

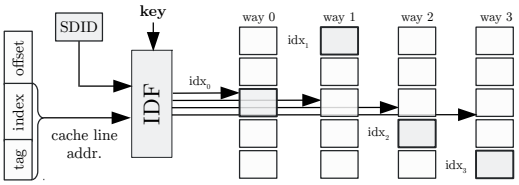


Figure 4: 4-way set-associative SCATTERCACHE where each index addresses exclusively one cache way.

Instead, as shown at the bottom of Figure 2, at best, partially overlapping cache sets can be found (cf. Section 4.3), which makes exploitation tremendously hard in practice.

A straightforward concept for SCATTERCACHE is shown in Figure 3. Here, the Index Derivation Function (IDF) combines the mapping operations in a single cryptographic primitive. In a set-associative SCATTERCACHE with set size n_{ways}, for each input address, the IDF outputs n_{ways} indices to form the cache set for the respective access. How exactly the mapping is performed in SCATTERCACHE is solely determined by the used key, the Security Domain Identifier (SDID), and the IDF. Note that, as will be discussed in Section 3.3.1, hash-based as well as permutation-based IDFs can be used in this context.

Theoretically, a key alone is sufficient to implement the overall idea. However, separating concerns via the SDID leads to a more robust and harder-to-misuse concept. The key is managed entirely in hardware, is typically longer, and gets switched less often than the SDID. On the other hand, the SDID is managed solely by the software and, depending on the implemented policy, has to be updated quite frequently. Importantly, as we show in Section 4, SCATTERCACHE alone already provides significantly improved security in PRIME+PROBE-style attack settings even without software support (*i.e.*, SDID is not used).

3.3 SCATTERCACHE Design

In the actual design we propose for SCATTERCACHE, the indices (*i.e.*, IDF output) do not address into one huge joint cache array. Instead, as shown in Figure 4, each index addresses a separate memory, *i.e.*, an independent cache way.

On the one hand, this change is counter-intuitive as it decreases the number of possible cache sets from $\binom{n_{ways} \cdot 2^{b_{indices}} + n_{ways} - 1}{n_{ways}}$ to $2^{b_{indices} \cdot n_{ways}}$. However, this reduction in possibilities is acceptable. For cache configurations with up to 4 cache ways, the gap between both approaches is only a few bits. For higher associativity, the exponential growth ensures that sufficiently many cache sets exist.

On the other hand, the advantages gained from switching to this design far outweigh the costs. Namely, for the original idea, no restrictions on the generated indices exist. Therefore, a massive n_{ways}-fold multi-port memory would be required to be able to lookup a n_{ways}-way cache-set in parallel. The design shown in Figure 4 does not suffer from this problem and permits to instantiate SCATTERCACHE using n_{ways} instances of simpler/smaller memory. Furthermore, this design guarantees that even in case the single index outputs of the IDF collide, the generated cache always consists of exactly n_{ways} many cache lines. This effectively precludes the introduction of systematic biases for potentially "weak" address-key-SDID combinations that map to fewer than n_{ways} cache lines.

In terms of cache-replacement policy, SCATTERCACHE uses simple random replacement to ensure that no systematic bias is introduced when writing to the cache and to simplify the security analysis. Furthermore, and as we will show in Section 5, the performance of SCATTERCACHE with random replacement is competitive to regular set associative caches with the same replacement policy. Therefore, evaluation of alternative replacement policies has been postponed. Independent of the replacement policy, it has to be noted that, for some IDFs, additional tag bits have to be stored in SCATTERCACHE. In particular, in case of a non invertible IDF, the original index bits need to be stored to facilitate write back of dirty cache lines and to ensure correct cache lookups. However, compared to the amount of data that is already stored for each cache line, the overhead of adding these few bits should not be problematic ($< 5\,\%$ overhead).

In summary, the overall hardware design of SCATTERCACHE closely resembles a traditional set-associative architecture. The only differences to contemporary fixed-set designs is the more complex IDF and the amount of required logic which permits to address each way individually. However, both changes are well understood. As we detail in the following section, lightweight (*i.e.*, low area and latency) cryptographic primitives are suitable building blocks for the IDF. Similarly, duplication of addressing logic is already common practice in current processors. Modern Intel architectures, for example, already partition their Last-Level Cache (LLC) into multiple smaller cache slices with individual addressing logic.

3.3.1 Suitable Index Derivation Functions

Choosing a suitable IDF is essential for both security and performance. In terms of security, the IDF has to be an unpredictable (but still deterministic) mapping from physical addresses to indices. Following Kerckhoffs's principle, even

for attackers which know every detail except the key, three properties are expected from the IDF: (1) Given perfect control over the public inputs of the function (*i.e.*, the physical address and SDID) constructing colliding outputs (*i.e.*, the indices) should be hard. (2) Given colliding outputs, determining the inputs or constructing further collisions should be hard. (3) Recovering the key should be infeasible given input and output for the function.

Existing Building Blocks: Cryptographic primitives like (tweakable) block ciphers, Message Authentication Codes (MACs), and hash functions are designed to provide these kind of security properties (e.g., indistinguishability of encryptions, existential unforgeability, pre-image and collision resistance). Furthermore, design and implementation of cryptographic primitives with tight performance constraints is already a well-established field of research which we want to take advantage of. For example, with PRINCE [13], a low-latency block cipher, and QARMA [8], a family of low-latency tweakable block ciphers, exist and can be used as building blocks for the IDF. Such tweakable block ciphers are a flexible extension to ordinary block ciphers, which, in addition to a secret key, also use a public, application-specific tweak to en-/decrypt messages. Similarly, sponge-based MAC, hash and cipher designs are a suitable basis for IDFs. These sponge modes of operation are built entirely upon permutations, e.g., Keccak-p, which can often be implemented with low latency [7, 11]. Using such cryptographic primitives, we define the following two variants of building IDFs:

Hashing Variant (SCv1): The idea of SCv1 is to combine all IDF inputs using a single cryptographic primitive with pseudo random output. MACs (e.g., hash-based) are examples for such functions and permit to determine the output indices by simply selecting the appropriate number of disjunct bits from the calculated tag. However, also other cryptographic primitives can be used for instantiating this IDF variant.

It is, for example possible to slice the indices from the ciphertext of a regular block cipher encryption which uses the concatenation of cache line address and the SDID as the plaintext. Similarly, tweakable block ciphers allow to use the SDID as a tweak instead of connecting it to the plaintext. Interestingly, finding cryptographic primitives for SCv1 IDFs is comparably simple given that the block sizes do not have to match perfectly and the output can be truncated as needed.

However, there are also disadvantages when selecting the indices pseudo randomly, like in the case of SCv1. In particular, when many accesses with high spatial locality are performed, index collisions get more likely. This is due to the fact that collisions in SCv1 output have birthday-bound complexity. Subsequently, performance can degrade when executing many different accesses with high spatial locality. Fortunately, this effect weakens with increasing way numbers, *i.e.*, an increase in associativity decreases the probability that all index outputs of the IDF collide.

In summary, SCv1 translates the address without distinguishing between index and tag bits. Given a fixed key and SDID, the indices are simply pseudo random numbers that are derived using a single cryptographic primitive.

Permutation Variant (SCv2): The idea behind the permutation variant of the IDF is to distinguish the index from the tag bits in the cache line address during calculation of the indices. Specifically, instead of generating pseudo random indices from the cache line address, tag dependent permutations of the input index are calculated.

The reason for preferring a permutation over pseudo random index generation is to counteract the effect of birthday-bound index collisions, as present in SCv1. Using a tag dependent permutation of the input index mitigates this problem by design since permutations are bijections that, for a specific tag, cannot yield colliding mappings.

Like in the hashing variant, a tweakable block cipher can be used to compute the permutation. Here, the concatenation of the tag bits, the SDID and the way index constitutes the tweak while the address' index bits are used as the plaintext. The resulting ciphertext corresponds to the output index for the respective way. Note that the block size of the cipher has to be equal to the size of the index. Additionally, in order to generate all indices in parallel, one instance of the tweakable block cipher is needed per cache way. However, as the block size is comparably small, each cipher instance is also smaller than an implementation of the hashing IDF (SCv1).

Independently of the selected IDF variant, we leave the decision on the actually used primitive to the discretion of the hardware designers that implement SCATTERCACHE. They are the only ones who can make a profound decision given that they know the exact instantiation parameters (e.g., SDID/key/index/tag bit widths, number of cache ways) as well as the allocatable area, performance, and power budget in their respective product. However, we are certain that, even with the already existing and well-studied cryptographic primitives, SCATTERCACHE implementations are feasible for common computing platforms, ranging from Internet of Things (IoT) devices to desktop computers and servers.

Note further that we expect that, due to the limited observability of the IDF output, weakened (*i.e.*, round reduced) variants of general purpose primitives are sufficient to achieve the desired security level. This is because adversaries can only learn very little information about the function output by observing cache collisions (*i.e.*, no actual values). Subsequently, many more traces have to be observed for mounting an attack. Cryptographers can take advantage of this increase in data complexity to either design fully custom primitives [55] or to decrease the overhead of existing designs.

3.3.2 Key Management and Re-Keying

The key in our SCATTERCACHE design plays a central role in the security of the entire approach. Even when the SDIDs are known, it prevents attackers from systematically constructing

eviction sets for specific physical addresses and thwarts the calculation of addresses from collision information. Keeping the key confidential is therefore of highest importance.

We ensure this confidentiality in our design by mandating that the key of is fully managed by hardware. There must not be any way to configure or retrieve this key in software. This approach prevents various kinds of software-based attacks and is only possible due to the separation of key and SDID.

The hardware for key management is comparably simple as well. Each time the system is powered up, a new random key is generated and used by the IDF. The simplicity of changing the key during operation strongly depends on the configuration of the cache. For example, in a write-through cache, changing the key is possible at any time without causing data inconsistency. In such a scenario, a timer or performance-counter-based rekeying scheme is easily implementable. Note, however, that the interval between key changes should not be too small as each key change corresponds to a full cache flush.

On the other hand, in a cache with write-back policy, the key has to be kept constant as long as dirty cache lines reside in the cache. Therefore, before the key can be changed in this scenario without data loss, all modified cache lines have to be written back to memory first. The x86 Instruction-Set Architecture (ISA), for example, features the `WBINVD` instruction that can be used for that purpose.

If desired, also more complex rekeying schemes, like way-wise or cache-wide dynamic remapping [55], can be implemented. However, it is unclear if adding the additional hardware complexity is worthwhile. Even without changing the key, mounting cache attacks against SCATTERCACHE is much harder than on traditional caches (see Section 4). Subsequently, performing an occasional cache flush to update the key can be the better choice.

3.3.3 Integration into Existing Cache Architectures

SCATTERCACHE is a generic approach for building processor caches that are hard to exploit in cache-based side channel attacks. When hardening a system against cache attacks, independent of SCATTERCACHE, we recommend to restrict flush instructions to privileged software. These instruction are only rarely used in benign userspace code and restricting them prevents the applicability of the whole class of flush-based attacks from userspace. Fortunately, recent ARM architectures already support this restriction.

Next, SCATTERCACHES can be deployed into the system to protect against eviction based attacks. While not inherently limited to, SCATTERCACHES are most likely to be deployed as LLCs in modern processor architectures. Due to their large size and the fact that they are typically shared across multiple processor cores, LLCs are simply the most prominent cache attack target and require the most protection. Compared to that, lower cache levels that typically are only accessible by a single processor core, hold far less data and are much harder to attack on current architectures. Still, usage of (unkeyed) skewed [63] lower level caches is an interesting option that has to be considered in this context.

Another promising aspect of employing a SCATTERCACHE as LLC is that this permits to hide large parts of the IDF latency. For example, using a fully unrolled and pipelined IDF implementation, calculation of the required SCATTERCACHE indices can already be started, or even performed entirely, in parallel to the lower level cache lookups. While unneeded results can easily be discarded, this ensures that the required indices for the LLC lookup are available as soon as possible.

Low latency primitives like QARMA, which is also used in recent ARM processors for pointer authentication, are promising building blocks in this regard. The minimal latency Avanzi [8] reported for one of the QARMA-64 variants is only 2.2 ns. Considering that this number is even lower than the time it takes to check the L1 and L2 caches on recent processors (e.g., 3 ns on a 4 GHz Intel Kabylake [2], 9 ns on an ARM Cortex-A57 in an AMD Opteron A1170 [1]), implementing IDFs without notable latency seems feasible.

3.4 Processor Interaction and Software

Even without dedicated software support, SCATTERCACHE increases the complexity of cache-based attacks. However, to make full use of SCATTERCACHE, software assistance and some processor extensions are required.

Security Domains. The SCATTERCACHE hardware permits to isolate different security domains from each other via the SDID input to the IDF. Unfortunately, depending on the use case, the definition on what is a security domain can largely differ. For example, a security domain can be a chunk of the address space (e.g., SGX enclaves), a whole process (e.g., TrustZone application), a group of processes in a common container (e.g., Docker, LXC), or even a full virtual machine (e.g., cloud scenario). Considering that it is next to impossible to define a generic policy in hardware that can capture all these possibilities, we delegate the distinction to software that knows about the desired isolation properties, e.g., the Operating System (OS).

SCATTERCACHE Interface. Depending on the targeted processor architecture, different design spaces can be explored before deciding how the current SDID gets defined and what channels are used to communicate the identifier to the SCATTERCACHE. However, at least for modern Intel and ARM processors, binding the currently used SDID to the virtual memory management via user defined bits in each Page Table Entry (PTE) is a promising approach. In more detail, one or more bits can be embedded into each PTE that select from a list, via one level of indirection, which SDID should be used when accessing the respective page.

Both ARM and Intel processors already support a similar mechanism to describe memory attributes of a memory mapping. The x86 architecture defines so-called Page Attribute Ta-

bles (PATs) to define how a memory mapping can be cached. Similarly, the ARM architecture defines Memory Attribute Indirection Registers (MAIRs) for the same purpose. Both PAT and MAIR define a list of 8 memory attributes which are applied by the Memory Management Unit (MMU). The MMU interprets a combination of 3 bits defined in the PTE as index into the appropriate list, and applies the corresponding memory attribute. Adding the SDID to these attribute lists permits to use up to 8 different security domains within a single process. The absolute number of security domains, on the other hand, is only limited by the used IDF and them number of bits that represent the SDID.

Such indirection has a huge advantage over encoding data directly in a PTE. The OS can change a single entry within the list to affect all memory mappings using the corresponding entry. Thus, such a mechanism is beneficial for SCATTERCACHE, where the OS wants to change the SDID for all mappings of a specific process.

Backwards Compatibility. Ensuring backwards compatibility is a key factor for gradual deployment of SCATTERCACHE. By encoding the SDID via a separate list indexed by PTE bits, all processes, as well as the OS, use the same SDID, *i.e.*, the SDID stored as first element of the list (assuming all corresponding PTE bits are '0' by default). Thus, if the OS is not aware of the SCATTERCACHE, all processes—including the OS—use the same SDID. From a software perspective, functionally, SCATTERCACHE behaves the same as currently deployed caches. Only if the OS specifies SDIDs in the list, and sets the corresponding PTE bits to use a certain index, SCATTERCACHE provides its strong security properties.

Implementation Example. In terms of capabilities, having a single bit in each PTE, for example, is already sufficient to implement security domains with process granularity and to maintain a dedicated domain for the OS. In this case, $SDID_0$ can always be used for the OS ID while $SDID_1$ has to be updated as part of the context switch and is always used for the scheduled user space process. Furthermore, by reusing the SDID of the OS, also shared memory between user space processes can easily be implemented without security impact.

Interestingly, SCATTERCACHE fully preserves the capability of the OS to share read-only pages (*i.e.*, libraries) also across security domains as no cache lines will be shared. In contrast, real shared memory has to always be accessed via the same SDID in all processes to ensure data consistency. In general, with SCATTERCACHE, as long as the respective cache lines have not been flushed to RAM, data always needs to be accessed with the same SDID the data has been written with to ensure correctness. This is also true for the OS, which has to ensure that no dirty cache lines reside in the cache, e.g., when a page gets assigned to a new security domain.

A case which has to be explicitly considered by the OS is copying data from user space to kernel space and vice versa. The OS can access the user space via the direct-physical map or via the page tables of the process. Thus, the OS has to select the correct SDID for the PTE used when copying data. Similarly, if the OS sets up page tables, it has to use the same SDID as the MMU uses for resolving page tables.

4 Security Evaluation

SCATTERCACHE is a novel cache design to efficiently thwart cache-based side-channel attacks. In the following, we investigate the security of SCATTERCACHE in terms of state-of-the-art side-channel attacks using both theoretical analysis and simulation-based results. In particular, we elaborate on the complexity of building the eviction sets and explore the necessary changes to the standard PRIME+PROBE technique to make it viable on the SCATTERCACHE architecture.

4.1 Applicability of Cache Attacks

While certain types of cache attacks, such as FLUSH+FLUSH, FLUSH+RELOAD and EVICT+RELOAD, require a particular cache line to be shared, attacks such as PRIME+PROBE have less stringent constraints and only rely on the cache being a shared resource. As sharing a cache line is the result of shared memory, we analyze the applicability of cache attacks on SCATTERCACHE with regard to whether the underlying memory is shared between attacker and victim or not.

Shared, read-only memory. Read-only memory is frequently shared among different processes, e.g., in case of shared code libraries. SCATTERCACHE prevents cache attacks involving shared read-only memory by introducing security domains. In particular, SCATTERCACHE maintains a separate copy of shared read-only memory in cache for each security domain, *i.e.*, the cache lines belonging to the same shared memory region are not being shared in cache across security domains anymore. As a result, reloading data into or flushing data out of the cache does not provide any information on another security domain's accesses to the respective shared memory region. Note, however, that the cache itself is shared, leaving attacks such as PRIME+PROBE still feasible.

Shared, writable memory. Exchanging data between processes requires shared, writable memory. To ensure cache coherency, writing shared memory regions must always use the same cache line and hence the same security domain for that particular memory region—even for different processes. While attacks on these shared memory regions involving `flush` instructions can easily be mitigated by making these instructions privileged, EVICT+RELOAD remains feasible. Still, SCATTERCACHE significantly hampers the construction of targeted eviction sets by skewing, *i.e.*, individually addressing, the cache ways. Moreover, its susceptibility to EVICT+RELOAD attacks is constrained to the processes sharing the respective memory region. Nevertheless, SCATTERCACHE requires writable shared memory to be used only as an interface for data transfer rather than sensitive computations. In addition, PRIME+PROBE attacks are still possible.

Unshared memory. Unshared memory regions never share the same cache line, hence making attacks such as FLUSH+FLUSH, FLUSH+RELOAD and EVICT+RELOAD infeasible. However, as the cache component itself is shared, cache attacks such as PRIME+PROBE remain possible.

As our analysis shows, SCATTERCACHE prevents a wide range of cache attacks that exploit the sharing of cache lines across security boundaries. While PRIME+PROBE attacks cannot be entirely prevented as long as the cache itself is shared, SCATTERCACHE vastly increases their complexity in all aspects. The pseudorandom cache-set composition in SCATTERCACHE prevents attackers from learning concrete cache sets from memory addresses and vice versa. Even if attackers are able to profile information about the mapping of memory addresses to cache-sets in their own security domain, it does not allow them infer the mapping of cache-sets to memory addresses in other security domains. To gain information about memory being accessed in another security domain, an attacker needs to profile the mapping of the attacker's address space to cache lines that are being used by the victim when accessing the memory locations of interest. The effectiveness of PRIME+PROBE attacks thus heavily relies on the complexity of such a profiling phase. We elaborate on the complexity of building eviction sets in Section 4.3.

4.2 Other Microarchitectural Attacks

Many other microarchitectural attacks are not fully mitigated but hindered by SCATTERCACHE. For instance, Meltdown [43] and Spectre [38] attacks cannot use the cache efficiently anymore but must resort to other covert channels. Also, DRAM row buffer attacks and Rowhammer attacks are negatively affected as they require to bypass the cache and reach DRAM. While these attacks are already becoming more difficult due to closed row policies in modern processors [24], we propose to make flush instructions privileged, removing the most widely used cache bypass. Cache eviction gets much more difficult with SCATTERCACHE and additionally, spurious cache misses will open DRAM rows during eviction. These spurious DRAM row accesses make the row hit side channel impractical and introduce a significant amount of noise on the row conflict side channel. Hence, while these attacks are not directly in the scope of this paper, SCATTERCACHE arguably has a negative effect on them.

4.3 Complexity of Building Eviction Sets

Cache skewing significantly increases the number of different cache sets available in cache. However, many of these cache sets will overlap partially, *i.e.*, in $1 \leq i < n_{ways}$ ways. The complexity of building eviction sets for EVICT+RELOAD and PRIME+PROBE in SCATTERCACHE thus depends on the overlap of cache sets.

4.3.1 Full Cache-Set Collisions

The pseudorandom assembly of cache sets in SCATTERCACHE results in $2^{b_{indices} \cdot n_{ways}}$ different compositions. For a given target address, this results in a probability of $2^{-b_{indices} \cdot n_{ways}}$ of finding another address that maps exactly to the same cache lines in its assigned cache set. While dealing with this complexity alone can be considered impractical in a real-world scenario, note that it will commonly even exceed the number of physical addresses available in current systems, rendering full cache-set collisions completely infeasible. A 4-way cache, for example, with $b_{indices} = 12$ index bits yields 2^{48} different cache sets, which already exceeds the address space of state-of-the-art systems.

4.3.2 Partial Cache-Set Collisions

While full cache-set collisions are impractical, partial collisions of cache sets frequently occur in skewed caches such as SCATTERCACHE. If the cache sets of two addresses overlap, two cache sets will most likely have a single cache line in common. For this reason, we analyze the complexity of eviction for single-way collisions in more detail.

Randomized Single-Set Eviction. Without knowledge of the concrete mapping from memory addresses to cache sets, the trivial approach of eviction is to access arbitrary memory locations, which will result in accesses to pseudorandom cache sets in SCATTERCACHE. To elaborate on the performance of this approach, we consider a cache with $n_{lines} = 2^{b_{indices}}$ cache lines per way and investigate the eviction probability for a single cache way, which contains a specific cache line to be evicted. Given that SCATTERCACHE uses a random (re-)placement policy, the probabilities of each cache way are independent, meaning that each way has the same probability of being chosen. Subsequently, the attack complexity on the full SCATTERCACHE increases linearly with the number of cache ways, *i.e.*, the attack gets harder.

The probability of an arbitrary memory accesses to a certain cache way hitting a specific cache line is $p = n_{lines}^{-1}$. Performing $n_{accesses}$ independent accesses to this cache way increases the odds of eviction to a certain confidence level α.

$$\alpha = 1 - (1 - n_{lines}^{-1})^{n_{accesses}}$$

Equivalently, to reach a certain confidence α in evicting the specific cache line, attackers have to perform

$$\mathbb{E}(n_{accesses}) = \frac{\log(1-\alpha)}{\log(1-n_{lines}^{-1})}$$

independent accesses to this cache way, which amounts to their attack complexity. Hence, to evict a certain cache set from an 8-way SCATTERCACHE with 2^{11} lines per way with $\alpha = 99\,\%$ confidence, the estimated attack complexity using this approach is $n_{accesses} \cdot n_{ways} \approx 2^{16}$ independent accesses.

Randomized Multi-Set Eviction. Interestingly, eviction of multiple cache sets using arbitrary memory accesses has

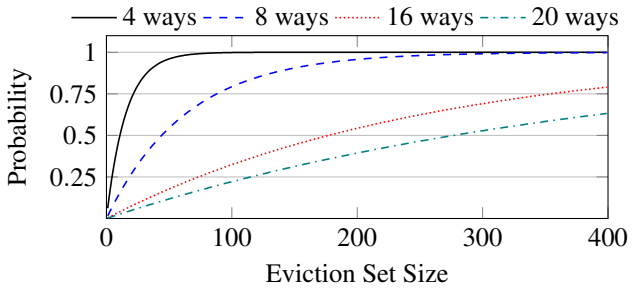


Figure 5: Eviction probability depending on the size of the eviction set and the number of ways.

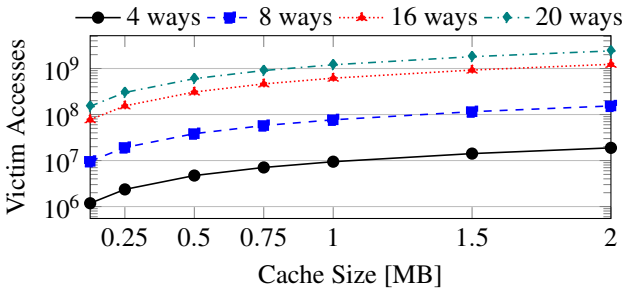


Figure 6: Number of required accesses to the target address to construct a set large enough to achieve 99 % eviction rate when no shared memory is available (cache line size: 32 bytes).

similar complexity. In this regard, the *coupon collector's problem* gives us a tool to estimate the number of accesses an attacker has to perform to a specific cache way to evict a certain percentage of cache lines in the respective way. In more detail, the coupon collector's problem provides the expected number of accesses $n_{accesses}$ required to a specific cache way such that n_{hit} out of all n_{lines} cache lines in the respective way are hit.

$$\mathbb{E}(n_{accesses}) = n_{lines} \cdot (H_{n_{lines}} - H_{n_{lines}-h_{hit}})$$

Hereby, H_n denotes the n-th Harmonic number, which can be approximated using the natural logarithm. This approximation allows to determine the number of cache lines n_{hit} that are expected to be hit in a certain cache way when $n_{accesses}$ random accesses to the specific way are performed.

$$\mathbb{E}(n_{hit}) = n_{lines} \cdot (1 - e^{-\frac{n_{accesses}}{n_{lines}}}) \quad (1)$$

Using n_{hit}, we can estimate the number of independent accesses to be performed to a specific cache way such that a portion β of the respective cache way is evicted.

$$\mathbb{E}(n_{accesses}) = -n_{lines} \cdot \ln(1-\beta)$$

For the same 8-way ScatterCache with 2^{11} lines per way as before, we therefore require roughly 2^{16} independent accesses to evict $\beta = 99\,\%$ of the cache.

Profiled Eviction for Prime+Probe. As shown, relying on random eviction to perform cache-based attacks involves significant effort and yields an overapproximation of the eviction set. Moreover, while random eviction is suitable for attacks such as Evict+Reload, in Prime+Probe settings random eviction fails to provide information related to the concrete memory location that is being used by a victim. To overcome these issues, attackers may profile a system to construct eviction sets for specific memory addresses of the victim, *i.e.*, they try to find a set of addresses that map to cache sets that partially overlap with the cache set corresponding to the victim address. Eventually, such sets could be used to speed up eviction and to detect accesses to specific memory locations. In the following, we analyze the complexity of finding these eviction sets. In more detail, we perform analysis w.r.t. eviction addresses whose cache sets overlap with the cache set of a victim address in a single cache way only.

To construct a suitable eviction set for Prime+Probe, the attacker needs to provoke the victim process to perform the access of interest. In particular, the attacker tests a candidate address for cache-set collisions by accessing it (prime), waiting for the victim to access the memory location of interest, and then measuring the time when accessing the candidate address again (probe). In such a profiling procedure, after the first attempt, we have to assume that the cache line belonging to the victim access already resides in the cache. As a result, attackers need to evict a victim's cache line in their prime step. Hereby, hitting the right cache way and index have probability ${n_{ways}}^{-1}$ and $2^{-b_{indices}}$, respectively. To be able to detect a collision during the probe step, the victim access must then fall into the same cache way as the candidate address, which has a chance of ${n_{ways}}^{-1}$. In total, the expected number of memory accesses required to construct an eviction set of t colliding addresses hence is

$$\mathbb{E}(n_{accesses}) = {n_{ways}}^2 \cdot 2^{b_{indices}} \cdot t.$$

The number of memory addresses t needs to be chosen according to the desired eviction probability for the victim address with the given set. When the eviction set consists of addresses that collide in the cache with the victim in exactly one way each, the probability of evicting the victim with an eviction set of size t is

$$p(\text{Eviction}) = 1 - \left(1 - \frac{1}{n_{ways}}\right)^{\frac{t}{n_{ways}}}.$$

Figure 5 depicts this probability for the size of the eviction set and different numbers of cache ways. For an 8-way ScatterCache with 2^{11} cache lines per way, roughly 275 addresses with single-way cache collisions are needed to evict the respective cache set with 99 % probability. Constructing this eviction set, in the best case, requires profiling of approximately $8^2 \cdot 2^{11} \cdot 275 \approx 2^{25}$ (33.5 million) victim accesses. Figure 6 shows the respective number of Prime+Probe experiments needed to generate sets with 99 % eviction probability for different cache configurations. We were able to empirically confirm these numbers within a noise-free standalone simulation of ScatterCache.

For comparison, to generate an eviction set on a commodity cache, e.g., recent Intel processors, for a specific victim memory access, an attacker needs fewer than 103 observations of that access in a completely noise-free attacker-controlled scenario. Hence, our cache increases the complexity for the attacker by factor 325 000. In a real-world scenario the complexity is even higher.

Profiled Eviction for EVICT+RELOAD. For shared memory, such as in EVICT+RELOAD, the construction of eviction sets, however, becomes easier, as shared memory allows the attacker to simply access the victim address. Hence, to build a suitable eviction set, the attacker first primes the victim address, then accesses a candidate address, and finally probes the victim address. In case a specific candidate address collides with the victim address in the cache way the victim access falls into , the attacker can observe this collision with probability $p = {n_{ways}}^{-1}$. As a result, the expected number of memory accesses required to build an eviction set of t colliding addresses for EVICT+RELOAD is

$$\mathbb{E}(n_{accesses}) = n_{ways} \cdot 2^{b_{indices}} \cdot t.$$

For an 8-way SCATTERCACHE with 2^{11} lines per way, constructing an EVICT+RELOAD eviction set of 275 addresses (*i.e.*, 99 % eviction probability) requires profiling with roughly $8 \cdot 2^{11} \cdot 275 = 2^{22}$ memory addresses. Note, however, that EVICT+RELOAD only applies to writable shared memory as used for Inter Process Communication (IPC), whereas SCATTERCACHE effectively prevents EVICT+RELOAD on shared read-only memory by using different cache-set compositions in each security domain. Moreover, eviction sets for both PRIME+PROBE and EVICT+RELOAD must be freshly created whenever the key or the SDID changes.

4.4 Complexity of PRIME+PROBE

As demonstrated, SCATTERCACHE strongly increases the complexity of building the necessary sets of addresses for PRIME+PROBE. However, the actual attacks utilizing these sets are also made more complex by SCATTERCACHE.

In this section, we make the strong assumption that an attacker has successfully profiled the victim process such that they have found addresses which collide with the victim's target addresses in exactly 1 way each, have no collisions with each other outside of these and are sorted into subsets corresponding to the cache line they collide in.

Where in normal PRIME+PROBE an attacker can infer victim accesses (or a lack thereof) with near certainty after only 1 sequence of priming and probing, SCATTERCACHE degrades this into a probabilistic process. At best, one PRIME+PROBE operation on a target address can detect an access with a probability of ${n_{ways}}^{-1}$. This is complicated further by the fact that any one set of addresses is essentially single-use, as the addresses will be cached in a non-colliding cache line with a probability of $1 - {n_{ways}}^{-1}$ after only 1 access, where they cannot be used to detect victim accesses anymore until they themselves are evicted again.

Given the profiled address sets, we can construct general probabilistic variants of the PRIME+PROBE attack. While other methods are possible, we believe the 2 described in the following represent lower bounds for either victim accesses or memory requirement.

Variant 1: Single collision with eviction. We partition our set of addresses, such that one PRIME+PROBE set consists of n_{ways} addresses, where each collides with a different way of the target address. To detect an access to the target, we prime with one set, cause a target access, measure the primed set and then evict the target address. We repeat this process until the desired detection probability is reached. This probability is given by $p(n_{accesses}) = 1 - (1 - {n_{ways}}^{-1})^{n_{accesses}}$. The eviction of the target address can be achieved by either evicting the entire cache or using preconstructed eviction sets (see Section 4.3.2). After the use of an eviction set, a different priming set is necessary, as the eviction sets only target the victim address. After a full cache flush, all sets can be reused. The amount of colliding addresses we need to find during profiling depends on how often a full cache flush is performed. This method requires the least amount of accesses to the target, at the cost of either execution time (full cache flushes) or memory and profiling time (constructing many eviction sets).

Variant 2: Single collision without eviction. Using the same method but without the eviction step, the detection probability can be recursively calculated as

$$p(n_{acc.}) = p(n_{acc.} - 1) + (1 - p(n_{acc.} - 1))(\frac{2 \cdot n_{ways} - 1}{{n_{ways}}^3})$$

with $p(1) = {n_{ways}}^{-1}$. This variant provides decreasing benefits for additional accesses. The reason for this is that the probability that the last step evicted the target address influences the probability to detect an access in the current step. While this approach requires many more target accesses, it has the advantage of a shorter profiling phase.

These two methods require different amounts of memory, profiling time and accesses to the target, but they can also be combined to tailor the attack to the target. Which is most useful depends on the attack scenario, but it is clear that both come with considerable drawbacks when compared to PRIME+PROBE in current caches. For example, achieving a 99 % detection probability in a 2 MB Cache with 8 ways requires 35 target accesses and 9870 profiled addresses in 308 MB of memory for variant 1 if we use an eviction set for every probe step. Variant 2 would require 152 target accesses and 1216 addresses in 38 MB of memory. In contrast, regular PRIME+PROBE requires 1 target access and 8 addresses while providing 100 % accuracy (in this ideal scenario). Detecting non-repeating events is made essentially impossible; to measure any access with confidence requires either the knowledge that the victim process repeats the same access pattern for long periods of time or control of the victim in a way that

allows for repeated measurements. In addition to the large memory requirements, variant 1 also heavily degrades the temporal resolution of a classical PRIME+PROBE attack because of the necessary eviction steps. This makes trace-based attacks like attacks on square-and-multiply in RSA [76] much less practical. Variant 2 does not suffer from this drawback, but requires one PRIME+PROBE set for each time step, for as many high-resolution samples as one trace needs to contain. This can quickly lead to an explosion in required memory when thousands of samples are needed.

4.5 Challenges with Real-World Attacks

We failed at mounting a real-world attack (*i.e.*, with even the slightest amounts of noise) on SCATTERCACHE. Generally, for a PRIME+PROBE attack we need to (1) generate an eviction set (cf. Section 4.3), and (2) use the eviction set to monitor a victim memory access. If we assume step 1 to be solved, we can mount a cache attack (*i.e.*, step 2) with a complexity increases by a factor of 152 (cf. Section 4.4). For some real-world attacks this would not be a problem, in particular if a small fast algorithm is attacked, e.g., AES with T-tables. Gülmezoglu et al. [29] recovered the full AES key from an AES T-tables implementation with only 30 000 encryptions in a fully synchronized setting (that can be implemented with PRIME+PROBE as well [26]), taking 15 seconds, *i.e.*, 500 µs per encryption. The same attack on SCATTERCACHE takes $4.56 \cdot 10^6$ encryptions, *i.e.*, 38 minutes assuming the same execution times, which is clearly viable.

However, the real challenge is solving step 1, which we did not manage for any real-world example. In particular, even if AES would only perform a single attacker-chosen memory access (instead of 160 to the T-tables alone, plus additional code and data accesses), which would be ideal for the attacker in the profiling during step 1, we would need to observe 33.5 million encryptions. In addition to the runtime reported by Gülmezoglu et al. [29] we also need a full cache flush after each attack round (*i.e.*, each encryption). For a 2 MB cache, we need to iterate over a 6 MB array to have a high probability of covering all cache lines. The time for an L3-cache access is e.g., for Kaby Lake 9.5 ns [2]. The absolute minimum number of cache misses here is 65536 (=4 MB), but in practice it will be much higher. A cache miss takes around 50 ns, hence, the full cache eviction will take at least 3.6 ms. Consequently, with 33.5 million tests required to generate the eviction set and a runtime of 4.1 ms per test, the total runtime to generate the eviction set is 38 hours.

This number still only considers the theoretical setting of a completely noise-free and idle system. The process doing AES computations must not be restarted during these 38 hours. The operating system must not replace any physical pages and, most importantly, our hypothetical AES implementation only performs a single memory access. In any realistic setting with only the slightest amount of activity (noise) on the system, this easily explodes to multiple weeks or months. With a second memory access, these two memory accesses can already not be distinguished anymore with the generated eviction set, because the eviction set is generated for an invocation of the entire victim computation, not for an address.

4.6 Noise Sampling

The previous analysis considered a completely noise-free scenario, where the attacker performs PRIME+PROBE on a single memory access executed by the victim. However, in a real system, an attacker will typically not be able to perform an attack on single memory accesses, but face different kinds of noise. Namely, on real systems cache attacks will suffer from both systematic and random noise, which reduces the effectiveness of profiling and the actual attack.

Systematic noise is introduced, for example, by the victim as it executes longer code sequences in between the attacker's prime and probe steps. The victim's code execution intrinsically performs additional memory accesses to fetch code and data that the attacker will observe in the cache deterministically. In SCATTERCACHE, the mappings of memory addresses to cache lines is unknown. Hence, without additional knowledge, the attacker is unable to distinguish the cache collision belonging to the target memory access from collisions due to systematic noise. Instead, the attacker can only observe and learn both simultaneously. As a result, larger eviction sets need to be constructed to yield the same confidence level for eviction. Specifically, the size of an eviction set must increase proportionally to the number of systematic noise accesses to achieve the same properties. While this significantly increases an attackers profiling effort, they may be able to use clustering techniques to prune the eviction set prior to performing an actual attack.

Random noise, on the other hand, stems from arbitrary processes accessing the cache simultaneously or as they are scheduled in between. Random noise hence causes random cache collisions to be detected by an attacker during both profiling and an actual attack, *i.e.*, produces false positives. While attackers cannot distinguish between such random noise and systematic accesses in a single observation, these random noise accesses can be filtered out statistically be repeating the same experiment multiple times. Yet, it increases an attackers effort significantly. For instance, when building eviction sets an attacker can try to observe the same cache collision multiple times for a specific candidate address to be certain about its cache collision with the victim.

Random noise distributes in SCATTERCACHE according to Equation 1 and hence quickly occupies large parts of the cache. As a result, there is a high chance of sampling random noise when checking a candidate address during the construction of eviction sets. Also when probing addresses of an eviction set in an actual attack, random noise is likely to be sampled as attacks on SCATTERCACHE demand for large

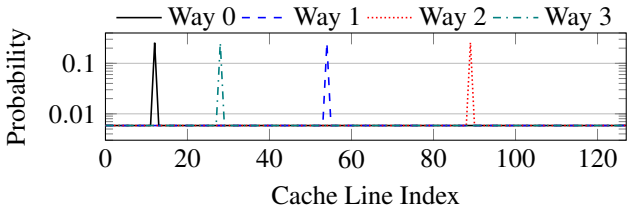


Figure 7: Example distribution of cache indices of addresses in profiled eviction sets ($n_{ways} = 4$, $b_{indices} = 7$).

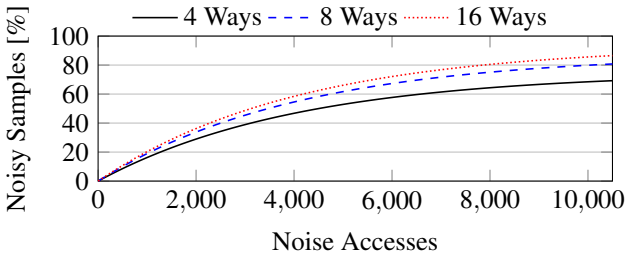


Figure 8: Expected percentage of noisy samples in an eviction set for a cache consisting of 2^{12} cache lines.

eviction sets. As our analysis shows, for a single cache way the distribution of cache line indices corresponding to the memory accesses of profiled eviction sets (cf. Section 4.3) adheres to Figure 7. Clearly, due to profiling there is a high chance of roughly $1/n_{ways}$ to access the index that collides with the victim address. However, with $p = (n_{ways} - 1)/n_{ways}$ the index adheres to an uniformly random selection from all possible indices and hence provides a large surface for sampling random noise. Consequently, for a cache with $n_{lines} = 2^{b_{indices}}$ lines per way and n_{noise} lines being occupied by noise in each way, the probability of sampling random noise when probing an eviction set address is

$$p(\text{Noise}) \approx \frac{n_{ways} - 1}{n_{ways}} \frac{n_{noise}}{n_{lines}}.$$

Figure 8 visualizes this effect and in particular the percentage of noisy samples encountered in an eviction set for different cache configurations and noise levels. While higher random noise clearly increases an attackers effort, the actual noise level strongly depends on the system configuration and load.

4.7 Further Remarks

In the previous analysis, the SDIDs of both attacker and victim were assumed to be constant throughout all experiments for statistical analysis to be applicable. Additionally, systematic and random noise introduced during both profiling and attack further increase the complexity of actual attacks, rendering attacks on most real-world systems impractical.

Also note that the security analysis in this section focuses on SCv1. In a noise-free scenario, SCv2 may allow to construct eviction sets slightly more efficiently since its IDF is a permutation. This means that, once a collision in a certain cache way is found, there will not be any other colliding address for that cache way in the same index range, *i.e.*, for the same address tag. Considering the expected time to find the single collision in a given index range, this could give an attacker a benefit of up to a factor of two in constructing eviction sets. However, in practice multiple cache ways are profiled simultaneously, which results in a high chance of finding a collision in any of the cache ways independent of the address index bits, *i.e.*, the n_{ways} indices for a certain memory address will very likely be scattered over the whole index range. Independent of that, the presence of noise significantly hampers taking advantage of the permuting property of SCv2.

5 Performance Evaluation

SCATTERCACHE significantly increases the effort of attackers to perform cache-based attacks. However, a countermeasure must not degrade performance to be practical as well. This section hence analyzes the performance of SCATTERCACHE using the gem5 full system simulator and GAP [9], MiBench [30], lmbench [49], and the C version of scimark2 [1] as micro benchmarks. Additionally, to closer investigate the impact of SCATTERCACHE on larger workloads, a custom cache simulator is used for SPEC CPU 2017 benchmarks. Our evaluations indicate that, in terms of performance, SCATTERCACHE behaves basically identical to traditional set-associative caches with the same random replacement policy.

5.1 gem5 Setup

We performed our cache evaluation using the gem5 full system simulator [12] in 32-bit ARM mode. In particular, we used the CPU model TimingSimpleCPU together with a cache architecture such as commonly used in ARM Cortex-A9 CPUs: the cache line size was chosen to be 32 bytes, the 4-way L1 data and instruction caches are each sized 32 kB, and the 8-way L2 cache is 512 kB large. We adapted the gem5 simulator such as to support SCATTERCACHE for the L2 cache. This allows to evaluate the impact of six different cache organizations. Besides SCATTERCACHE in both variants (1) SCv1 and (2) SCv2 and standard set-associative caches with (3) LRU, (4) BIP, and (5) random replacement, we also evaluated (6) skewed associative caches [63] with random replacement as we expect them to have similar performance characteristics as SCv1 and SCv2.

On the software side, we used the Poky Linux distribution from Yocto 2.5 (Sumo) with kernel version 4.14.67 after applying patches to run within gem5. We then evaluated the performance of our micro benchmarks running on top of Linux. In particular, we analyzed the cache statistics provided by

[1] https://math.nist.gov/scimark2/

gem5 after booting Linux and running the respective benchmark. Using this approach, we reliably measure the cache performance and execution time for each single application, *i.e.*, without concurrent processes. Since only the L2-cache architecture (*i.e.*, replacement policy, skewed vs. fixed sets) changed between the individual simulation runs, execution performance is simply direct proportional to the resulting cache hit rate. To enable easier comparison between the individual benchmarks as well as with related work we therefore mainly report L2-cache hit results.

ScatterCache IDF Instantiations. Both ScatterCache variants have been instantiated using the low-latency tweakable block cipher QARMA-64 [8]. In particular, in the SCv1 variant, the index bits for the individual cache ways have been sliced from the ciphertext of encrypting the cache line address under the secret key and SDID. On the other hand, due to the lack of an off-the-shelf tweakable block cipher with the correct block size, a stream cipher construction was used in the SCv2 variant. Namely, the index is computed as the XOR between the original index bits and the ciphertext of the original tag encrypted using QARMA-64. Note, however, that, although this construction for SCv2 is a proper permutation and entirely sufficient for evaluating the performance of SCv2, we do not recommend the construction as pads are being reused for addresses having the same tag bits.

While the majority of the following results are latency agnostic LLC hit rates, all following results are reported for the zero cycle latency case. For QARMA-64 with 5 rounds, ASIC implementation results with as little as 2.2 ns latency have been reported [8]. We are therefore confident that, if desired, hiding the latency of the IDF by computing it in parallel to the lower level cache lookup is feasible.

However, we still also conducted simulations with latency overheads between 1 and 5 cycles by increasing the `tag_latency` of the cache in gem5. The acquired results show that, even for IDFs which introduce 5 cycles of latency, less than 2 % performance penalty are encountered on the GAP benchmark suite. These numbers are also in line with Qureshi's results reported for CEASER [55].

5.2 Hardware Overhead Discussion

ScatterCache is designed to be as similar to modern cache architectures as possible in terms of hardware. Still, area and power overheads have to be expected due to the introduction of the IDF and the additional addressing logic. Unfortunately, while probably easy for large processor and SoC vendors, determining reliable overhead numbers for these two metrics is a difficult task for academia that requires an actual ASIC implementation of the cache. To the best of our knowledge, even in the quite active RISC-V community, no open and properly working LLC designs are available that can be used as foundation. Furthermore, for merely simulating such a design with a reasonably large cache, commercial EDA tools, access to state-of-the-art technology libraries, and large memory macros with power models are required. As the result, secure cache designs typically fail to deliver hardware implementation results (see Table 6 in [18]).

Because of these problems, similar to related work, we can also not provide concrete numbers for the area and power overhead. However, due to the way we designed ScatterCache and the use of lightweight cryptographic primitives, we can assert that the hardware overhead is reasonable. For example, the 8-way SCv1 ScatterCache with 512 kB that is simulated in the following section, uses two parallel instances of QARMA-64 with 5 rounds as IDF. One fully unrolled instance has a size of 22.6 kGE [8] resulting in an IDF size of less then 50 kGE even in case additional pipeline registers are added. The added latency of such an IDF is the same as the latency of the used primitive which has been reported as 2.2 ns. However, this latency can (partially or fully) be hidden by computing the IDF in parallel to the lower level cache lookup. Interestingly, with similar size, also a sponge-based SCv1 IDF (e.g., 12 rounds of Keccak[200] [11]) can be instantiated. Finally, there is always the option to develop custom IDF primitives [55] that demand even less resources.

For comparison, in the BROOM chip [16], the SRAM macros in the 1 MB L2 cache already consume roughly 50 % of the 4.86 mm^2 chip area. Assuming an utilization of 75 % and a raw gate density of merely 3 $MGate/mm^2$ [21] for the used 28 nm TSMC process, these 2.43 mm^2 already correspond to 5.5 MGE. Subsequently, even strong IDFs are orders of magnitude smaller than the size of a modern LLC.

In terms of overhead for the individual addressing of the cache ways, information is more sparse. Spjuth et al. [64] observed a 17 % energy consumption overhead for a 2-way skewed cache. They also report that skewed caches can be built with lower associativity and still reach similar performance as traditional fixed set-associative caches. Furthermore, modern Intel architectures already feature multiple addressing circuits in their LLC as they partition it into multiple smaller caches (*i.e.*, cache slices).

5.3 gem5 Results and Discussion

Figure 9 visualizes the cache hit rate of our L2 cache when executing programs from the GAP benchmark suite. To ease visualization, the results are plotted in percentage points (pp), *i.e.*, the differences between percentage numbers, using the fixed set-associative cache with random replacement policy as baseline. All six algorithms (*i.e.*, `bc`, `bfs`, `cc`, `pr`, `sssp`, `tc`) have been evaluated. Moreover, as trace sets, both synthetically generated `kron` (`-g16 -k16`) and `urand` (`-u16 -k16`) sets have been used. As can be seen in the graph, the BIP and LRU replacement policies outperform random replacement on average by 4.6 pp and 4 pp respectively. Interestingly, however, all random replacement based schemes, including the skewed variants, perform basically identical.

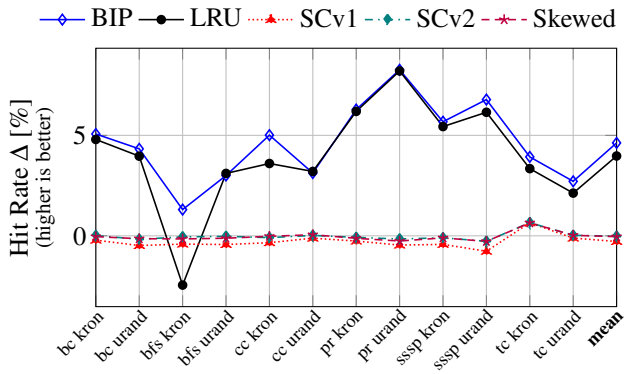


Figure 9: Cache hit rate, simulated with gem5, for the synthetic workloads in the GAP benchmark suite with random replacement policy as baseline.

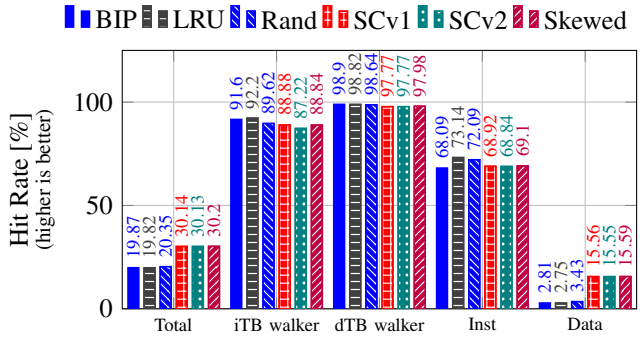


Figure 10: Cache hit rate, simulated with gem5, for scimark2.

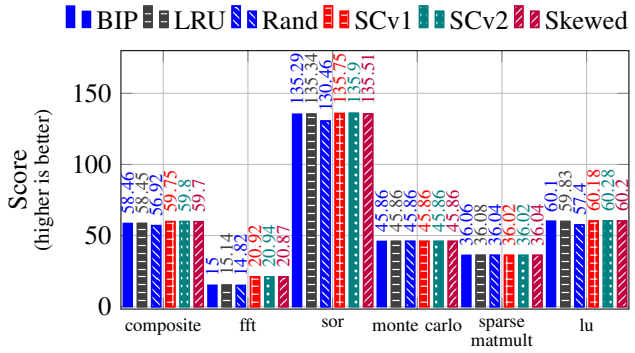


Figure 11: Scimark2 score simulated with gem5.

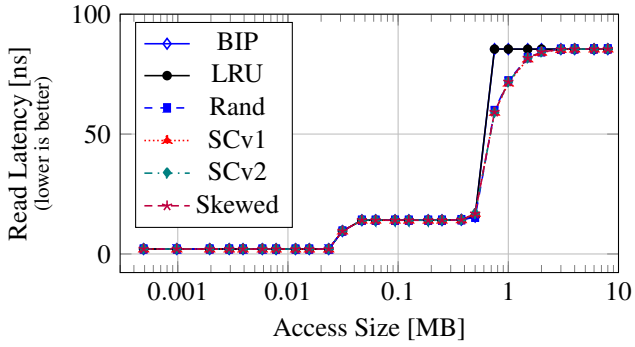


Figure 12: Memory read latency, simulated with gem5, with 32 byte stride (*i.e.*, one access per cache line).

The next benchmark, we visualized in Figure 10, is scimark2 (`-large 0.5`). This benchmark shows an interesting advantage of the skewed cache architectures over the fixed-set architectures, independent of the replacement policy, of approximately 10 pp for the total hit rate. This difference is mainly caused by the 5x difference in hit rate for data accesses. Comparing the achieved benchmark scores in Figure 11 further reveals that the `fft` test within scimark2 is the reason for the observed discrepancy in cache performance.

To investigate this effect in more detail, we measured the memory read latency using using `lat_mem_rd 8M 32` from lmbench in all cache configurations. The respective results in Figure 12 feature two general steps in the read latency at 32 kB (L1-cache size) and at 512 kB (L2-cache size). Notably, configurations with random replacement policy feature a smoother transition at the second step, *i.e.*, when accesses start to hit main memory instead of the L2 cache.

Even more intersting results, as shown in Figure 13, have been acquired by increasing the stride size to four times the cache line size. Skewed caches like ScatterCache break the strong alignment of addresses and cache set indices. As a consequence, a sparse, but strongly aligned memory access pattern such as in `lat_mem_rd`, which in a standard set-associative caches only uses every 4th cache index, gives high cache hit rates and low read latencies for larger memory ranges due to less cache conflicts. This effect becomes visible in Figure 13 as shift of the second step from 512 kB to 2 MB for the skewed cache variants.

Finally, as last benchmark, MiBench has been evaluated in small and large configuration. The individual results are visualized in Figure 14 and Figure 15 respectively. On average, the achieved performance results in MiBench are very similar to the results from the GAP benchmark suite. Again, caches with BIP and LRU replacement policy outperform the configurations with random replacement policy by a few percent. However, in some individual benchmarks (e.g., qsort in small, `jpeg` in large), skewed cache architectures like ScatterCache outmatch the fixed set appraoches.

In summary, our evaluations with gem5 in full system simulation mode show that the performance of ScatterCache, in terms of hit rate, is basically identical to contemporary fixed set-associative caches with random replacement policy. Considering that we employ the same replacement strategy, this is an absolutely satifying result by itself. Moreover, no tests indicated any notable performance degradation and in some tests ScatterCache even outperformed BIP and LRU replacement policies.

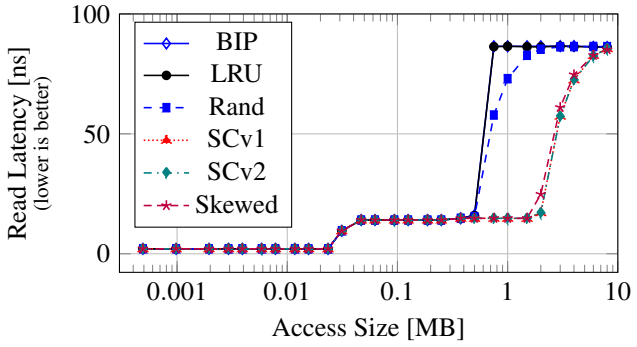


Figure 13: Memory read latency, simulated with gem5, with 128 byte stride (*i.e.*, one access in every fourth cache line).

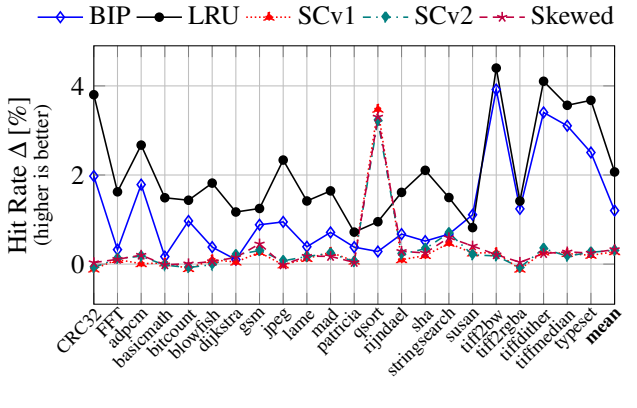


Figure 14: Cache hit rate, simulated with gem5, for MiBench in small configuration compared to random replacement.

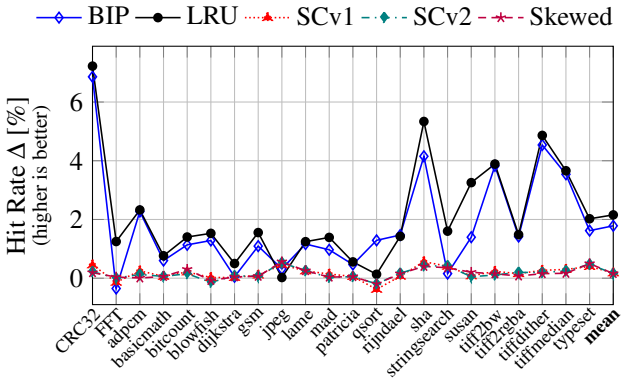


Figure 15: Cache hit rate, simulated with gem5, for MiBench in large configuration compared to random replacement.

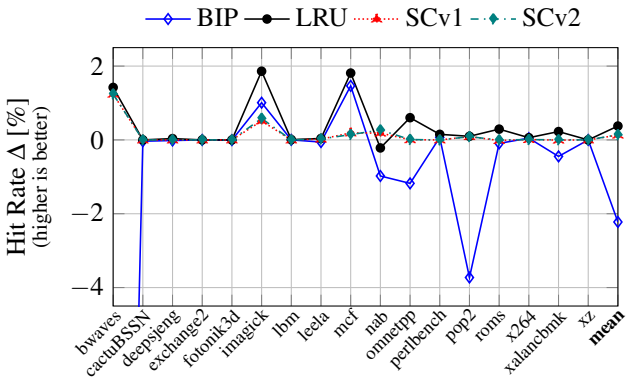


Figure 16: Average cache hit rate for SPEC CPU 2017 benchmarks compared to random replacement over 10 runs.

5.4 Cache Simulation and SPEC Results

Lastly, we evaluated the performance of SCATTERCACHE using the SPEC CPU 2017 [66] benchmark with both the "SPECspeed 2017 Integer" and "SPECspeed 2017 Floating Point" suites. We performed all benchmarks in these suites with the exception of `gcc`, `wrf` and `cam4`, as these failed to compile on our system. Because these benchmarks are too large to be run in full system simulation, we created a software cache simulator, capable of simulating different cache models and replacement policies. Even so, the benchmarks proved to be too large to run in full, so we opted to run segments of 250 million instructions from each, following the methodology of Qureshi et al. [56]. We made an effort to select parts of the benchmarks that are representative of their respective core workloads. To be able to run the benchmarks with our simulator, we recorded a trace of all instruction addresses and memory accesses with the Intel PIN Tool [33]. We then replayed this access stream for different cache configurations. The simulator implements the set-associative replacement policies Pseudo-LRU (Tree-PLRU), LRU (ideal), BIP as described in [56], and random replacement, as well as the two SCATTERCACHE variants. The number of ways per set, total cache size, number of slices, and cache line size are fully configurable. Additionally, the simulator supports multiple levels of inclusive caches, as well as a cache that is split for data and instructions. All simulations were run on an inclusive two level cache, where the L1 was separated into instruction and data caches, both of which use LRU replacement. Figure 16 shows results for the cache configuration, as described in Section 5.1, as the difference in percentage points for last-level hit rates when compared to random replacement. While we can see large differences in individual tests, the mean shows that both versions of SCATTERCACHE perform at least as well as random replacement and very similar to LRU. Using the same cache configuration but with 64 B cache lines, we actually observe a mean advantage of 0.23 ± 0.76 pp of SCATTERCACHE over random replacement, where LRU sees a marginally worse result of -0.21 ± 1.02 pp. On a larger configuration with 64 B cache lines, 32 kB 8-way L1 and 2 MB 16-way LLC, the results show a slim improvement of 0.035 ± 0.10 pp for SCATTERCACHE and 0.37 ± 1.14 pp for LRU over random replacement.

6 Conclusion

In this paper, we presented SCATTERCACHE, a novel cache design to eliminate cache attacks that eliminates fixed cache-set congruences and, thus, makes eviction-based cache attacks unpractical. We showed how skewed associative caches when retrofitted with a keyed mapping function increase the attack complexity so far that it exceeds practical scenarios Furthermore, high-frequency attacks become infeasible. Our evaluations show that the runtime performance of software is not curtailed and SCATTERCACHE can even outperform state-of-the-art caches for certain realistic workloads.

Acknowledgments

We want to thank the anonymous reviewers and especially our shepherd, Yossi Oren, for their comments and suggestions that substantially helped in improving the paper. This project has received funding from the European Research Council (ERC) under Horizon 2020 grant agreement No 681402. Additional funding was provided by a generous gift from Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] 7-cpu. ARM Cortex-A57. www.7-cpu.com/cpu/Cortex-A57.html.

[2] 7-cpu. Intel Skylake. www.7-cpu.com/cpu/Skylake.html.

[3] O. Aciiçmez, B. B. Brumley, and P. Grabher. New Results on Instruction Cache Attacks. In *CHES*, 2010.

[4] G. I. Apecechea, T. Eisenbarth, and B. Sunar. S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing - and Its Application to AES. In *S&P*, 2015.

[5] G. I. Apecechea, M. S. Inci, T. Eisenbarth, and B. Sunar. Wait a Minute! A fast, Cross-VM Attack on AES. In *RAID*, 2014.

[6] G. I. Apecechea, M. S. Inci, T. Eisenbarth, and B. Sunar. Lucky 13 Strikes Back. In *CCS*, 2015.

[7] V. Arribas, B. Bilgin, G. Petrides, S. Nikova, and V. Rijmen. Rhythmic Keccak: SCA Security and Low Latency in HW. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018.

[8] R. Avanzi. The QARMA Block Cipher Family. Almost MDS Matrices Over Rings With Zero Divisors, Nearly Symmetric Even-Mansour Constructions With Non-Involutory Central Rounds, and Search Heuristics for Low-Latency S-Boxes. *IACR Trans. Symmetric Cryptol.*, 2017.

[9] S. Beamer, K. Asanovic, and D. A. Patterson. The GAP Benchmark Suite. *arXiv abs/1508.03619*, 2015.

[10] D. J. Bernstein. Cache-Timing Attacks on AES. Technical report, University of Illinois at Chicago, 2005.

[11] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche, and R. V. Keer. Keccak implementation overview, 2012.

[12] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. S. B. Altaf, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comp. Arch. News*, 39, 2011.

[13] J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, P. Rombouts, S. S. Thomsen, and T. Yalçin. PRINCE - A Low-Latency Block Cipher for Pervasive Computing Applications - Extended Abstract. In *ASIACRYPT*, 2012.

[14] B. B. Brumley and R. M. Hakala. Cache-Timing Template Attacks. In *ASIACRYPT*, 2009.

[15] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*, 2018.

[16] C. Celio, P. Chiu, K. Asanovic, B. Nikolic, and D. A. Patterson. BROOM: An Open-Source Out-of-Order Processor With Resilient Low-Voltage Operation in 28-nm CMOS. *MICRO*, 39, 2019.

[17] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. In *S&P*, 2009.

[18] S. Deng, W. Xiong, and J. Szefer. Analysis of Secure Caches and Timing-Based Side-Channel Attacks. *ePrint 2019/167*.

[19] G. Doychev, D. Feld, B. Köpf, L. Mauborgne, and J. Reineke. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *USENIX Security*, 2013.

[20] G. Doychev and B. Köpf. Rigorous analysis of software countermeasures against cache attacks. In *PLDI*, 2017.

[21] Europractice. TSMC Standard cell libraries. http://www.europractice-ic.com/libraries_TSMC.php.

[22] M. Gallagher, L. Biernacki, S. Chen, Z. B. Aweke, S. F. Yitbarek, M. T. Aga, A. Harris, Z. Xu, B. Kasikci, V. Bertacco, S. Malik, M. Tiwari, and T. M. Austin. Morpheus: A Vulnerability-Tolerant Secure Architecture Based on Ensembles of Moving Target Defenses with Churn. In *ASPLOS*, 2019.

[23] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS*, 2017.

[24] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom. Another Flip in the Wall of Rowhammer Defenses. In *S&P*, 2018.

[25] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *CCS*, 2016.

[26] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*, 2016.

[27] D. Gruss, R. Spreitzer, and S. Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security*, 2015.

[28] D. Gullasch, E. Bangerter, and S. Krenn. Cache Games - Bringing Access-Based Cache Attacks on AES to Practice. In *S&P*, 2011.

[29] B. Gülmezoglu, M. S. Inci, G. I. Apecechea, T. Eisenbarth, and B. Sunar. A Faster and More Realistic Flush+Reload Attack on AES. In *COSADE*, 2015.

[30] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *WWC*, 2001.

[31] R. Hund, C. Willems, and T. Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *S&P*, 2013.

[32] M. S. Inci, B. Gülmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar. Cache Attacks Enable Bulk Key Recovery on the Cloud. In *CHES*, 2016.

[33] Intel Corporation. Pin - A Dynamic Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[34] G. Irazoqui, K. Cong, X. Guo, H. Khattri, A. K. Kanuparthi, T. Eisenbarth, and B. Sunar. Did we learn from LLC Side Channel Attacks? A Cache Leakage Detection Tool for Crypto Libraries. *arXiv abs/1709.01552*, 2017.

[35] G. Irazoqui, T. Eisenbarth, and B. Sunar. Cross Processor Cache Attacks. In *CCS*, 2016.

[36] Y. Jang, S. Lee, and T. Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *CCS*, 2016.

[37] E. Käsper and P. Schwabe. Faster and Timing-Attack Resistant AES-GCM. In *CHES*, 2009.

[38] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P*, 2019.

[39] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*, 1996.

[40] R. Könighofer. A Fast and Cache-Timing Resistant Implementation of the AES. In *CT-RSA*, 2008.

[41] B. Köpf, L. Mauborgne, and M. Ochoa. Automatic Quantification of Cache Side-Channels. In *CAV*, 2012.

[42] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security*, 2016.

[43] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security*, 2018.

[44] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *S&P*, 2015.

[45] H. Mantel, A. Weber, and B. Köpf. A Systematic Study of Cache Side Channels Across AES Implementations. In *ESSoS*, 2017.

[46] C. Maurice, C. Neumann, O. Heen, and A. Francillon. C5: Cross-Cores Cache Covert Channel. In *DIMVA*, 2015.

[47] C. Maurice, N. L. Scouarnec, C. Neumann, O. Heen, and A. Francillon. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *RAID*, 2015.

[48] C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. A. Boano, S. Mangard, and K. Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *NDSS*, 2017.

[49] L. W. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, 1996.

[50] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *CCS*, 2015.

[51] D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: The Case of AES. In *CT-RSA*, 2006.

[52] D. Page. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. *ePrint 2002/169*.

[53] D. Page. Partitioned Cache Architecture as a Side-Channel Defence Mechanism. *ePrint 2005/280*.

[54] C. Percival. Cache missing for fun and profit. In *BSDCan*, 2005.

[55] M. K. Qureshi. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *MICRO*, 2018.

[56] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. S. Jr., and J. S. Emer. Adaptive insertion policies for high performance caching. In *ISCA*, 2007.

[57] H. Raj, R. Nathuji, A. Singh, and P. England. Resource management for isolation enhanced cloud services. In *CCSW*, 2009.

[58] C. Rebeiro, A. D. Selvakumar, and A. S. L. Devi. Bitslice Implementation of AES. In *CANS*, 2006.

[59] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS*, 2009.

[60] M. Schwarz, M. Lipp, D. Gruss, S. Weiser, C. Maurice, R. Spreitzer, and S. Mangard. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In *NDSS*, 2018.

[61] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss. NetSpectre: Read Arbitrary Memory over Network. *arXiv abs/1807.10535*, 2018.

[62] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*, 2017.

[63] A. Seznec. A Case for Two-Way Skewed-Associative Caches. In *ISCA*, 1993.

[64] M. Spjuth, M. Karlsson, and E. Hagersten. Skewed caches from a low-power perspective. In *Computing Frontiers – CF*, 2005.

[65] R. Spreitzer and T. Plos. Cache-Access Pattern Attack on Disaligned AES T-Tables. In *COSADE*, 2013.

[66] Standard Performance Evaluation Corporation. SPEC CPU 2017. https://www.spec.org/cpu2017/.

[67] D. Trilla, C. Hernández, J. Abella, and F. J. Cazorla. Cache side-channel attacks and time-predictability in high-performance critical real-time systems. In *DAC*, 2018.

[68] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi. Cryptanalysis of DES Implemented on Computers with Cache. In *CHES*, 2003.

[69] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *ISCA*, 2007.

[70] Z. Wang and R. B. Lee. A novel cache architecture with enhanced performance and security. In *MICRO*, 2008.

[71] S. Weiser, A. Zankl, R. Spreitzer, K. Miller, S. Mangard, and G. Sigl. DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries. In *USENIX Security*, 2018.

[72] Z. Wu, Z. Xu, and H. Wang. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *USENIX Security*, 2012.

[73] Z. Wu, Z. Xu, and H. Wang. Whispers in the Hyper-Space: High-Bandwidth and Reliable Covert Channel Attacks Inside the Cloud. *IEEE/ACM Trans. Netw.*, 23, 2015.

[74] Y. Xiao, M. Li, S. Chen, and Y. Zhang. STACCO: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves. In *CCS*, 2017.

[75] Y. Xu, M. Bailey, F. Jahanian, K. R. Joshi, M. A. Hiltunen, and R. D. Schlichting. An exploration of L2 cache covert channels in virtualized environments. In *CCSW*, 2011.

[76] Y. Yarom and K. Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*, 2014.

[77] A. Zankl, J. Heyszl, and G. Sigl. Automated Detection of Instruction Cache Leaks in Modular Exponentiation Software. In *CARDIS*, 2016.

[78] X. Zhang, Y. Xiao, and Y. Zhang. Return-Oriented Flush-Reload Side Channels on ARM and Their Implications for Android Devices. In *CCS*, 2016.

[79] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In *S&P*, 2011.

[80] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In *CCS*, 2012.

[81] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *CCS*, 2014.

Pythia: Remote Oracles for the Masses

Shin-Yeh Tsai
Purdue University

Mathias Payer
EPFL

Yiying Zhang
Purdue University

Abstract

Remote Direct Memory Access (RDMA) is a technology that allows direct access from the network to a machine's main memory without involving its CPU. RDMA offers low-latency, high-bandwidth performance and low CPU utilization. While RDMA provides massive performance boosts and has thus been adopted by several major cloud providers, security concerns have so far been neglected.

The need for RDMA NICs to bypass CPU and directly access memory results in them storing various metadata like page table entries in their on-board SRAM. When the SRAM is full, RNICs swap metadata to main memory across the PCIe bus. We exploit the resulting timing difference to establish side channels and demonstrate that these side channels can leak access patterns of victim nodes to other nodes.

We design Pythia, a set of RDMA-based remote side-channel attacks that allow an attacker on one client machine to learn how victims on other client machines access data a server exports as an in-memory data service. We reverse engineer the memory architecture of the most widely used RDMA NIC and use this knowledge to improve the efficiency of Pythia. We further extend Pythia to build side-channel attacks on Crail, a real RDMA-based key-value store application. We evaluated Pythia on four different RDMA NICs both in a laboratory and in a public cloud setting. Pythia is fast (57 μs), accurate (97% accuracy), and can hide all its traces from the victim or the server.

1 Introduction

Direct Memory Access (DMA) allows a machine's peripherals like storage and network devices to access its main memory directly without involving CPU, vastly increasing I/O performance and reducing CPU utilization. Inspired by DMA, Remote Direct Memory Access, or *RDMA*, is a technology that allows remote hosts to directly access (exported) memory of a node without having to go through its CPU. RDMA enables high throughput and low latency data transfers and largely reduces CPU utilization in clusters.

In recent years, major cloud vendors like Microsoft Azure [73] and Alibaba Cloud [6] have adopted RDMA in their datacenters to speed up processing and to reduce cost of accessing large amounts of data. As the underlying protocols prosper [32], more and more servers leverage the RDMA protocol to speed up processing. The use of RDMA has revolutionized data sharing in cloud environments with implementations for efficient key-value stores [24, 25, 40, 56, 57], in-memory databases and transactional systems [19, 83, 88], and graph processing systems [68, 85].

There is a plethora of research work on RDMA, but the focus so far was all on performance, usability, and network protocols. Security has been largely overlooked in RDMA research and production[1]. With the rise of information leaks through memory-based side-channels [17,26,39,41,42,86,87], we have set out to evaluate the side-channel resistance of existing RDMA implementations.

In a scenario where multiple nodes connect to a server that provides remote access to its local memory through RDMA (*e.g.*, for a key-value store), a malicious node may want to learn what data was accessed by benign nodes. We assume that the server is trusted and that the attacker tries to learn what data was accessed by the victim nodes through a *remote* side channel which leaks access patterns. Figure 1 illustrates this environment.

We discovered a new side channel that prevails across all RDMA hardware that we know of. NICs that support RDMA, or *RNIC*s, cache metadata such as page table entries in their on-board SRAM so that they can perform all operations needed to access main memory on their own without involving CPU. However, the on-board SRAM size is limited and the RNIC can only cache hot metadata while leaving the rest in main memory. When an RDMA request's metadata is not cached, the RNIC takes extra time to fetch the metadata from main memory to its SRAM. We observe and characterize different side channels that can lead to this timing difference

[1] We made an initial exploration of security issues and opportunities in one-sided communication in a recent workshop [77].

on three generations of RNIC devices.

Based on our findings, we designed *Pythia*, a set of side-channel attacks that can be launched completely from the network through RDMA. The basic idea is to issue RDMA network requests to the server to fill its RNIC SRAM, eventually evicting the metadata of the target data. Then the attacker reloads the target data with an RDMA request and based on the time it takes, predict if the victim has accessed the data.

Although the basic idea is similar to the EVICT+RELOAD CPU cache side-channel attack [30], designing Pythia presents many new challenges. The first challenge is the difficulty in achieving good eviction performance. Existing CPU-cache based side-channel attacks leverage cache associativity to reduce the eviction set size, thereby improving eviction performance. However, RNICs are vendor owned and are complete black boxes to public knowledge. To confront this challenge, we reverse engineered the memory architecture of the Mellanox ConnectX-4 RNIC [48], the type of RNIC that is used in all major datacenter RDMA deployment. We successfully discovered the internal architectural organization of RNIC SRAM and leverage this knowledge to achieve low-latency eviction.

The second challenge is in the reload and prediction process. Because of our environment of being in a shared datacenter network, the latency of an RDMA request can vary with different network state. The traditional approach of using a static threshold to differentiate cache hit from cache miss is not a good fit for our environment. We take an adaptive approach to dynamically train a hit/miss classifier based on RDMA access latency at the time of attack and use the trained classifier to statistically predict victim accesses [22, 28].

We evaluated Pythia in our lab environment and in a public cloud [65] with four different types of RNICs. Pythia completes one EVICT+RELOAD cycle (across the network) in as low as 57 μs with 97% accuracy[2] Moreover, Pythia effectively hides its traces from the server and victims because it performs all its attack using RDMA operations from a separate machine.

We further built three variations of Pythia to attack a real RDMA-based system, the Apache Crail key-value store system [7, 70]. On a real application like Crail, it is more challenging to establish a strong side-channel attack because of limited application interface and noise coming from application performance overhead. After improving Pythia to accommodate these difficulties, we successfully launched a side-channel attack solely from a separate client machine using the unmodified Crail client interface. This attack is efficient and can accurately learn a victim's key-value pair access patterns.

The contributions of this paper are:

1. Discovery of new side channels in RDMA-based systems that leak client RDMA access patterns;

[2]The definition of accuracy throughout the paper is the percentage of successful guesses over total guesses.

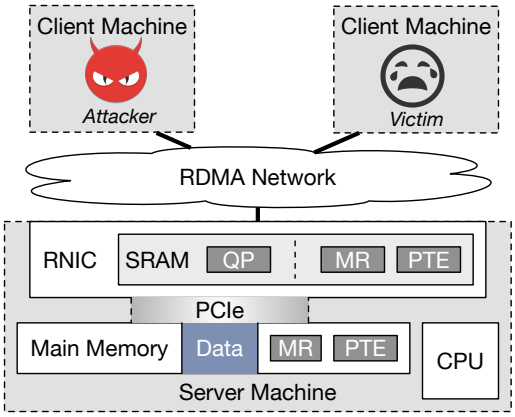


Figure 1: **Attack Environment and RNIC Architecture.** *The attacker and the victim are both clients that can access data in the server machine's memory throuth RDMA.*

2. Reverse engineering of the most widely used RNIC hardware architecture, which can be leveraged in designing efficient side channels;
3. Design, implementation, and evaluation of a set of Pythia side-channel attacks, which are fast, accurate, and can be launched solely from a separate machine across the network;
4. A case study of Pythia in a real-world setting;
5. Discussion of possible mitigations, most of which are uniquely applicable to RDMA systems.

Pythia is the first work that explores side-channel vulnerabilities in RDMA and exploits the vulnerabilities to launch attacks on RDMA-based datacenter systems. With today's datacenters all having robust defenses against direct sniffing or hijacking of network traffic, side channels are more feasible attack mechanisms and we believe that our work raises serious security concerns in a young but already widely-adopted network technology.

We have responsibly disclosed the weaknesses to Mellanox and Crail. Our implementation of Pythia is publicly available at https://github.com/Wuklab/Pythia.

2 Background on RDMA

2.1 RDMA Basics

Remote Direct Memory Access, or RDMA, is a network technology designed to offer remote low-latency, low-CPU-utilization access to exported memory regions. RDMA supports both *one-sided* and *two-sided* communication. One-sided RDMA operations directly access memory at a remote node without involving the remote node's CPU, similar to DMA on a single machine. Two-sided RDMA operations involve both sender and receiver processing, similar to `send`/`recv` in traditional network messaging.

RDMA improves performance along several dimensions. First, one-sided RDMA requests bypass the CPU of the receiver. Second, applications issue RDMA requests directly from user space, bypassing kernel and avoiding kernel trap cost. Third, RDMA avoids memory copying (a technique called *zero-copy*). As a result, RDMA achieves low-latency, high-throughput performance.

There are three implementations of RDMA: InfiniBand (IB) [10, 11], RoCE [8, 9], and iWARP [63]. All implementations follow the standard RDMA protocol [64]. Among them, RoCE, or *RDMA over Converged Ethernet*, implements the RDMA protocol over standard Ethernet (RoCEv1) and UDP (RoCEv2), and is the preferred technology in existing datacenters [32].

One-sided RDMA is the key area where significant performance and CPU utilization improvements over other network technologies happen. Thus, we focus on one-sided RDMA. To perform a one-sided RDMA operation, an application process at a receiver node needs to first allocate a consecutive virtual memory space and then use the virtual memory address range to register a *memory region*, or *MR*, with the RNIC. An application can register multiple MRs over the same or different memory spaces. The RNIC will assign a pair of local and remote protection keys (called *lkey* and *rkey*) to each MR. This application then conveys the virtual address of the MR and its rkey to processes running on other nodes. After building connections between these other nodes (senders) and the node that the MR-registering application runs on (receiver), these processes can use 1) a virtual memory address that falls in the MR's virtual memory address range, 2) a size, and 3) the rkey of the MR to perform one-sided RDMA `read` and `write`. In RDMA's term, a connection is called a *Queue Pair*, or *QP*.

2.2 RDMA NICs

RDMA NICs, or *RNIC*s, are where most RDMA functionalities are implemented. They usually contain complex hardware logic that implements the RDMA protocol and some SRAM to store metadata, and they are often connected to the host's PCIe bus (allowing the card access to main memory through DMA). Because of the need to bypass the kernel and receiver's CPU, most RDMA functionalities and data structures have to be offloaded to the RNIC hardware.

An RNIC's on-board SRAM stores three types of metadata. First, it stores metadata for each QP in its memory. Second, it stores lkeys, rkeys, and virtual memory addresses for all registered MRs. Third, it caches page table entries (PTEs) for MRs to obtain the DMA address of an RDMA request from its virtual memory address. RNICs have a limited amount of on-board SRAM which can only hold metadata for hot data. When the SRAM is full, an RNIC will evict its cached metadata to the main memory on the host machine, and on a future access, fetch the evicted metadata from the host main memory back through the PCIe bus. The timing difference between an RDMA access whose metadata is in RNIC SRAM and one that is not is what we exploit in our side-channel attack. The SRAM architecture is vendor-specific and not disclosed or specified in the RDMA standard. We reverse engineer the SRAM architecture of the state-of-the-art RNIC in Section 4.4.

2.3 RDMA-Based Applications

RDMA was originally designed for high-performance computing environments, and it has been a popular choice of network system in these environments for the past two decades [34, 44, 54]. In recent years, major datacenters and public clouds adopted RDMA for its low CPU utilization and superior performance. For example, Microsoft Azure [73] and Alibaba [6] have deployed RDMA with RoCE at large, production scale.

Many datacenter systems and applications have been ported to or rebuilt with RDMA. These include in-memory key-value stores [7, 24, 25, 40, 56, 57, 70], in-memory databases and transactional systems [19, 83, 88], graph processing systems [68, 85], distributed machine learning systems [15], consensus implementations [60, 80], distributed non-volatile memory systems [45, 67, 93], and remote swap systems [5, 31]. Most of these applications use both one-sided and two-sided RDMA operations, with some being pure one-sided [14, 88]. Our work is applicable to all RDMA-based applications that use one-sided RDMA (but not necessary purely one-sided).

3 Threat Model

In our attack, there are three parties: the server which hosts data in its main memory for other client machines to access (*e.g.*, an in-memory database or an in-memory key-value store), the victim who accesses the server's in-memory data through RDMA, and the attacker who tries to infer the victim's accesses and access patterns. The attacker and the victim are both normal clients that can access the data store service the server provides, and they run on separate machines. Following the threat models of related work that introduces and evaluates side channels, we assume that the attacker does not have direct control over the victim. As victim and attacker execute on different machines and communication to the server happens through the network, we assume that the attacker *cannot* observe the victim's network packets (as otherwise, the attacker could directly infer the accessed addresses and values as RDMA is currently not encrypted). This assumption is reasonable as sniffing victim's packets would require an attacker to have `root` access on either the victim's machine or the server's machine [52, 72, 74] or to launch man-in-the-middle attack to the network, both of which are well defended in cloud datacenters. We also assume that the server cannot *directly* observe memory accesses of either the victim or the attacker as both victim and attacker interact with the server through one-sided RDMA operations, not involving the server's CPU.

4 Side-Channel Attacks on RDMA

RDMA exposes node-internal memory to external hosts. Due to best practices of optimizing accesses and caching, current RDMA hardware is vulnerable to a variety of timing side channels. We present an overview of RDMA-based side channels, two basic attacks, refined attacks with our reverse engineered knowledge of RNIC internals, and evaluation results of the attacks. Unless otherwise stated, all our experiments use three machines, each equipped with a Mellanox ConnectX-4 100 Gbps network adapter [48], two Intel Xeon E5-2620 2.40GHz CPUs, and 128 GB main memory. They are connected with a Mellanox SB7700 100 Gbps InfiniBand switch [50]. One machine is used as the server that serves in-memory data through RDMA. The other two machines are the victim client machine and the attacker client machine, both of which can perform RDMA operations to access data on the server. In all the experiments in this section, the victim has a 50% chance of accessing the targeted data that the attacker tries to infer accesses on.

4.1 Attack Overview

The basic idea of our side-channel attacks is to exploit two weaknesses in RNIC: 1) the RNIC caches metadata in its SRAM, RDMA accesses whose metadata is not in SRAM must wait until that data is fetched from main memory, and 2) all RDMA accesses from all applications share the RNIC SRAM. As explained in Section 2.2, RNICs store three types of metadata in their SRAM: QP information, MR information, and PTEs. An RDMA access involves all three types of metadata: upon receiving a network request, an RNIC needs to locate which QP the request belongs to, which MR it falls into, and which page it is accessing. If any of these metadata is not in the RNIC SRAM, the RNIC will fetch it from the host memory, stalling the request until the required data arrives. By exploiting this timing difference, we can launch side-channel attacks to know which QP, which MR, and which PTE the victim has accessed.

Traditional CPU-cache-based side-channel attacks take three major forms: *prime*-based (*e.g.*, PRIME+PROBE [1–4, 42, 59, 75, 90]), *flush*-based (*e.g.*, FLUSH+RELOAD [86]), and *evict*-based (*e.g.*, EVICT+RELOAD [30]). Because RNICs do not provide any interface to flush their SRAM, all flush-based side-channel attacks such as FLUSH+RELOAD [86] and FLUSH+FLUSH [29] are incompatible with RDMA. All the operations that are needed in prime-based and evict-based attacks can be implemented through RDMA network requests that an attacker performs over the network. Attackers can hide their traces during the attacks, since one-sided RDMA reads are oblivious to the server or other clients. Hardware performance counters [35] may help servers track DMA traffic, but it is challenging to associate traffic with RDMA accesses or attacks. Even if the server suspects that an attack is happening, it is still hard, in practice, to attribute an attack based on traffic counters.

For the rest of the paper, we focus on RDMA-based EVICT+RELOAD attacks. PRIME+PROBE attacks are also possible on RDMA and we briefly discuss them in Section 7.

4.2 Unique Advantages and Challenges

There are three unique advantages for attackers in RDMA systems. First, RDMA's one-sided communication pattern allows the attacker to hide her traces, since the receiving node is unaware of any one-sided accesses. Second, the RDMA network is much faster both in latency and in bandwidth than traditional datacenter networks. The latest generation of RDMA switches and RNICs can sustain 200 Gbps bandwidth and under 0.6 μs latency [53]. RDMA's superior performance enables fine-grained, high-throughput, timing-based side-channel attacks over the network. Finally, RDMA's one-sided communication bypasses the sender's OS and does not involve CPU at the receiver, both of which help reduce disturbance to timing-based attacks.

At the same time, attacking the RNIC presents several novel challenges that no CPU-cache-based side-channel attack experiences. First, it is hard to discover efficient side channels in RNIC hardware. Unlike CPU caches, there is no public knowledge of how RNICs organize or use their SRAM. RNICs store different types of information in SRAM compared to a linear layout of CPU caches. Second, we set a strict threat model where attacks are launched from a separate machine that is different from the victim's machine and the server machine. This goal means that attacks have to be performed using RDMA network requests only. Finally, since our side channels are established over the network, noise in the network could potentially increase difficulties for timing-based attacks.

4.3 Basic Attack

Before presenting our side-channel attacks, we first discuss the type of victim information we choose as our attack target and the type of metadata we use to perform the eviction phase. Notice that these two dimensions are orthogonal and both have three options: QP, MR, and PTE.

Among these three types of information, knowing which QP the victim accesses leaks little information about the victim and usually is not useful in real attacks. MRs and PTEs can both leak more information. Using PTE as the attack target unit will reveal memory page (in virtual memory) accesses. All OSes use 4 KB as the default page size. The MR size is decided by the application that creates and registers it. For performance reasons [55], most RDMA-based applications choose to use large MRs. Thus, we choose PTE as the target of our attack. However, most of our ideas and techniques can be used to perform attacks that target MRs.

After choosing the attack target, we must decide what metadata to use to evict RNIC SRAM. To answer this question, we tried to evict SRAM using the three types of metadata and

Algorithm 1: MR-based eviction

```
Input  : a target victim virtual memory address
Output :

if No access to sufficient amount of MRs then
    start process at server to create MR_set;
else
    foreach MR that the attacker has access to do
        if MR ≠ victim's MR then
            insert MR into MR_set;
        end
    end
end

foreach MR in MR_set do
    perform 8-byte RDMA-read to MR;
end
```

Algorithm 2: Naive PTE-based eviction

```
Input  : a target victim virtual memory address
Output :

VictimVPN ← victim_address >> 12;
generate eviction_set using VictimVPN;

foreach VPN in eviction_set do
    perform 8-byte RDMA-read to address VPN << 12;
end
```

Algorithm 3: Reload and predict

```
Input  : a target victim virtual memory address
Output : prediction of if the victim has accessed the target
         address

determine Threshold according to network status;

start timer;
RDMA-read victim_address;
end timer;
time ← elapsed_time;

if time < Threshold then
    output accessed;
else
    output not_accessed;
end
```

reload our targeted information (*i.e.*, PTE). We can successfully evict a PTE with PTEs, no matter whether or not the PTEs we use to evict belong to the same MR as the target PTE. We can also evict an MR with other MRs. We further find that when an MR is evicted, PTEs of all the pages belonging to this MR will also be evicted. But we can only use QPs to evict QP. This behavior implies that RNICs isolates the SRAM used for QPs and for MRs and PTEs. We present the evaluation results in Section 4.5. From this initial test, we discovered that we can evict a PTE by either evicting the MR it belongs to (using a large number of MRs) or by evicting the PTE directly using a large number of other PTEs.

4.3.1 Eviction by MRs

We now present our attack that evicts SRAM with MRs, *PythiaMR*. Algorithm 1 presents the pseudocode of *PythiaMR*.

Because the MR-based attack requires the attacker to use many MRs to evict the server's RNIC SRAM, the attacker requires access to a sufficient amount of MRs. If the number of MRs is restricted, the attacker may resort to a (hypothetical) MR gadget that allows her to register multiple MRs. One approach is to launch a process on the server that allows her to register multiple MRs (see Section 5.1 for details). Since RDMA provides the functionality of registering multiple MRs with the same memory space, the attacker process at the server only needs to allocate a small memory space (of arbitrary size) and register it multiple times. This process then needs to send the rkeys corresponding to these registered MRs to the attacker running on a client machine.

In the eviction phase, the attacker performs one-sided RDMA reads from a client machine to the MRs it has access to at the server (except for the MR that the victim PTE is in). Since the server's RNIC needs to fetch and store metadata for each MR when the MR is accessed, its SRAM will eventually be filled with MR metadata that the attacker accessed.

4.3.2 Eviction by PTEs

Alternatively, we can use PTEs to establish a side channel. Compared to MR-based attacks, PTE-based attacks can often be performed entirely from a client machine. Algorithm 2 presents the pseudocode of our PTE-based attack.

To perform PTE-based eviction, the attacker issues one-sided RDMA reads to a sufficiently big memory space (1 GB to 4 GB for the RNICs we study). Most RDMA-based applications are services that provide in-memory data storage and use a large amount of memory, thus meeting the requirements of PTE-based attacks.

Accessing different memory pages will cause the RNIC to fetch PTEs to its SRAM. Because all accesses to the same memory page will hit the same PTE, we only need to perform one RDMA read (with the smallest RDMA operation size of 8 bytes) in every 4 KB virtual memory address range. To avoid loading the PTE that the victim accesses, we skip the memory addresses that are close to the victim address.

4.3.3 Reload and Predict

After the eviction phase (either by MRs or by PTEs), the attacker reloads the targeted victim data. If the reload time is smaller than a threshold, the attacker determines that the data has been accessed by the victim. Algorithm 3 presents the pseudocode of the reload and prediction process.

The threshold used at the reload time directly affects the result of an attack. Different from traditional CPU-cache side-

channel attacks, our attacks are in a distributed environment where network status can vary with other workloads in the datacenter. Thus, instead of a fixed threshold, we adapt the threshold dynamically according to the network status. To adjust the threshold, the attacker periodically measures the latency of an RDMA operation which hits the RNIC SRAM and the latency of one that misses the SRAM. The threshold can be set as a value in the middle of these two latencies.

In addition to using an average threshold, other more advanced methods can also be used to determine the reload result. In fact, we design a statistical method to determine the reload result for our real-application attacks, as will be presented in Section 5.

4.4 Finding PTE Eviction Sets

We call the attack that uses the eviction phase presented in Section 4.3.2 the basic PTE-based attack, or $PythiaPTE_{Basic}$. In order to reduce the time to perform eviction and improve the efficiency of $PythiaPTE_{Basic}$, we search for a smaller eviction set that achieves similar accuracy as $PythiaPTE_{Basic}$. Specifically, we perform a set of experiments to systematically reverse engineer the internal organization of RNIC SRAM and use our learned knowledge to construct a minimal PTE eviction set.

Reverse engineering RNIC SRAM organization is significantly harder than reverse engineering traditional CPU caches because there is no public knowledge of the internal organization of any RNIC. All we know is that the RNIC caches three types of metadata (PTEs, MRs, and QPs) in its SRAM. Moreover, reverse engineering the RNIC involves network operations which add noise compared to a well-isolated CPU cache environment.

First attempt in finding index bits. We initially guess that RNICs organize their in-SRAM PTE caches as set-associative caches, similar to how CPUs organize their caches. To validate this guess, we assume that the PTE cache is organized as a fixed number of sets (*e.g.*, 2, 4, 8) and different number of bits (*e.g.*, 1, 2, 3) are used to calculate the index into these sets. Since PTEs are identified by virtual page number (*VPN*), we can ignore the lowest 12 bits (with page size being 4 KB). We then use the lowest K bits of *VPN*s to calculate the index into one of the 2^K cache sets. These K bits correspond to the 12th to the $(11+K)$th bits of the full virtual memory address (we call the lowest bit the 0th bit and count upwards to higher bits).

We call the *VPN* of a victim memory address *VictimVPN*. The eviction set of a *VictimVPN* is formed by setting the same K bits as the *VictimVPN* and varying other bits in *VPN*s. To put it another way, for every 2^K pages, we pick one *VPN* to add to the eviction set. We keep adding distinct *VPN*s in this way until the number of *VPN*s in the set reaches an *eviction set size*. Algorithm 4 presents the pseudocode of how we form an eviction set. This is our straw-man approach and we call the PTE-based attack that uses this approach of forming eviction sets $PythiaPTE_{Straw}$.

Algorithm 4: Forming Eviction Set - Strawman

Input : *VictimVPN*, eviction set size, num of index bits
Output: an eviction set targeting *VictimVPN*

```
eviction_set ← {};
mask = VictimVPN & (1 << num_index_bits − 1);
for i = 0 to evict_set_size do
    VPN ← i << num_index_bits + mask;
    if VPN ≠ VictimVPN then
        insert VPN into eviction_set;
    end
end
output eviction_set;
```

Figure 2 plots the attack accuracy when we vary K from 0 to 15. We use two groups of attacks on *VictimVPN* 0. In the first group (the solid line), each eviction set has $2^7 = 128$ operations. The accuracy keeps improving until K is 13 and then flattens out. This result hints at the possibility of the RNIC using a set-associative cache with $2^{13} = 8192$ sets. It is because when K is smaller than 13, part of the operations in the eviction set will fall into a different cache set as the *VictimVPN*'s set, making the eviction set too small to evict the whole victim's cache set.

To verify this guess, we perform a second group of attacks (the dashed line). In this group, we double the eviction set size every time when we decrease K by 1. For example, we set the eviction set size to 128 when K is 13 and to 256 when K is 12. If the RNIC uses 8192 cache sets, then when K is 13, all of the eviction set will fall into the *VictimVPN*'s set, and when K is 12, half of the eviction set will fall into the *VictimVPN*'s set. These two attacks will then have the same effect in evicting the *VictimVPN*. Our results confirm this assumption. When K is less than 13, the attack accuracy is similar to when K is 13. However, when K is larger than 14, the accuracy drops. This is because we only use 64 and 32 eviction set size when K is 14 and 15, and these eviction sets are not large enough to evict a whole cache set.

From this set of experiments, we suspect that virtual memory address bits 12 to 24 are used in calculating the PTE cache set index and that each PTE cache set has 128 entries (*i.e.*, a 128-way cache).

Discovering prefetching behavior. Our guess above uses 13 bits as the index into the PTE cache and assumes that the PTE cache has 8192 sets. If this guess is correct, then an eviction set whose (*VPN*%8192) is different from (*VictimVPN*%8192) should fall completely into a different cache set from the victim's. To verify this assumption, we perform another set of attacks to the *VictimVPN* 0. In the nth attack, we construct its eviction set using *VPN*s where (*VPN*%8192 $= n$), and we change n from 0 to 8191.

Figure 3 plots the accuracy of the first 64 attacks (the rest of

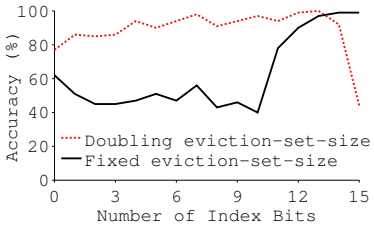


Figure 2: **Effect of Number of Index Bits.**

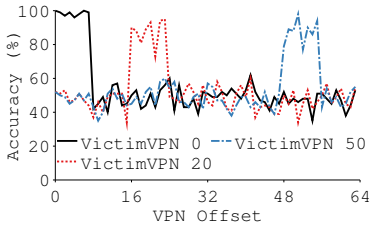


Figure 3: **Effect of Eviction Set Offset.** *X axis represents the first VPN in an eviction set (i.e., the "offset" of an eviction set").*

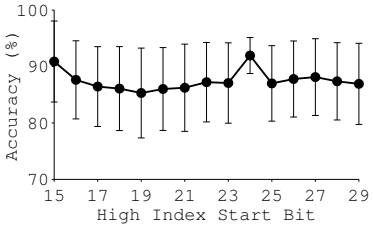


Figure 4: **Effect of Secondary Index.** *Error bars show the standard deviation across 1000 VictimVPNs.*

the attacks have the same pattern and we omit them from the figure). The black solid line shows the result of *VictimVPN* 0. Surprisingly, not only the 0th attack ($VPN\%8192 = 0$) has high accuracy, but also the 1st to the 7th attacks. To further understand this effect, we perform the same set of attacks on another two *VictimVPN*s, 20 and 50. With these *VictimVPN*s, the accuracy is high from 16th to 23rd attacks and 48th to 55th attacks respectively. These results imply that evicting any *VPN* within an eight-*VPN* range of ($VictimVPN - VictimVPN\%8$) to ($VictimVPN - VictimVPN\%8 + 7$) has the same effect.

We suspect that the RNIC prefetches eight *VPN*s at a time. This finding implies that instead of using 13 bits as cache index bits as in $PythiaPTE_{Straw}$, only the higher 10 bits (bits [15:24]) are used for index and the lower 3 bits (bits [12:14]) are used for prefetching. The RNIC cache thus only has $2^{10} = 1024$ sets.

Discovering secondary index. Our attack strategies so far work well ($> 90\%$ accuracy) with the *VictimVPN*s we tested (*e.g.*, 0, 20, 50). However, when we test the same attack strategy on some other *VictimVPN*s (*e.g.*, 6195, 30950), the accuracy can sometimes drop to around 75%. A dropped accuracy means that our eviction set cannot evict the *VictimVPN*, *i.e.*, the *VictimVPN* is in another cache set whose index is not the same as the index calculated using bits [15:24].

We thus suspect that there exists another 10 bits that are used to calculate where out of the 1024 sets a *VictimVPN* can fall into. To answer this question, we use a moving window of 10 bits, from bits [15:24] to bits [29:48], in the victim's virtual memory address. We randomly pick 1000 *VictimVPN*s and attack each of them by forming an eviction set with an index calculated with the moving window of 10 bits and an index calculated with bits [15:24] (64 operations under each index). We use two indices in this experiment because our alternative experiment of using just the index calculated by the moving window does not yield good accuracy. Figure 4 plots the average attack accuracy across 1000 *VictimVPN*s and their standard deviation (in error bars). Bits [24:33] yields the best average accuracy and smallest deviation. Thus, we believe that these bits are used as a second index into the PTE cache. Note that when the moving window is bits [15:24], the accuracy deviation is high, indicating that using bits [15:24] alone is not good enough.

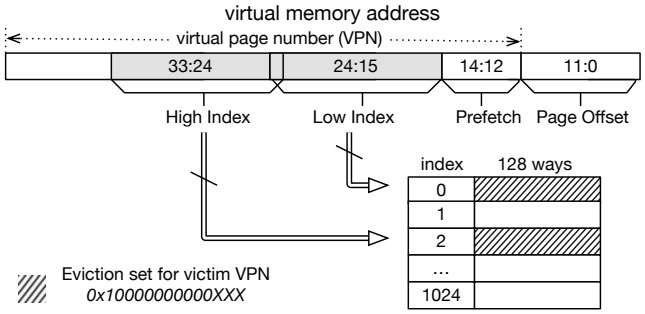


Figure 5: **Reverse-Engineered PTE Cache Organization.**

Complete algorithm. Figure 5 presents the final PTE cache architecture we speculate RNICs use based on our reverse engineering results. The PTE cache has 1024 sets and 128 ways. A PTE can be cached at one of the two cache sets. Two groups of bits are used to calculate the index of these two cache sets. The first group is bits [15:24], and the second group is bits [24:33]. We call them *low index bits* and *high index bits*. From our observation, both the high and the low index bits can decide which cache set will be used to cache a PTE. A PTE will be cached in either the cache set calculated by the high index bits or the cache set indicated by the low index bits. Every time when a PTE is accessed, its neighboring PTEs will also be fetched to the same set and bits [12:14] determine the 8 PTEs that will be prefetched.

Based on this reverse-engineered architecture, we present the final PTE-based EVICT+RELOAD attack, $PythiaPTE_{Full}$. We form half of the eviction set of a *VictimVPN* with *VPN*s that have the same low index bits as the *VictimVPN* and another half with *VPN*s that have the same high index bits as the *VictimVPN*. Algorithm 5 presents the complete algorithm.

4.5 Evaluation Results

We now present our evaluation results with the attacks described above.

4.5.1 Isolated Environment

We performed a set of experiments with three machines as described in the beginning of this section in our lab environment

Algorithm 5: Forming Eviction Set - Full

Input : *VictimVPN*, eviction set size
Output : an eviction set targeting the victim virtual memory address

eviction_set ← {}; *prefetch_bits* ← 3; *index_bits* ← 10; *high_index_start* ← 12;
low_index ← (*VictimVPN* >> *prefetch_bits*)&(1 << *index_bits* − 1); *mask_low* ← *low_index* << *prefetch_bits*;
high_index ← (*VictimVPN* >> *high_index_start*)&(1 << *index_bits* − 1); *mask_high* ← *high_index* << *prefetch_bits*;

for *i* = 0 *to evict_set_size*/2 **do**
    VPN ← *i* << (*index_bits* + *prefetch_bits*) + *mask_low*;
    if *VPN* ≠ *VictimVPN* **then**
        insert *VPN* into *eviction_set*;
    end
end

for *i* = 0 *to evict_set_size*/2 **do**
    VPN ← *i* << (*index_bits* + *high_index_start*) + *mask_high*;
    if *VPN* ≠ *VictimVPN* **then**
        insert *VPN* into *eviction_set*;
    end
end

output *eviction_set*;

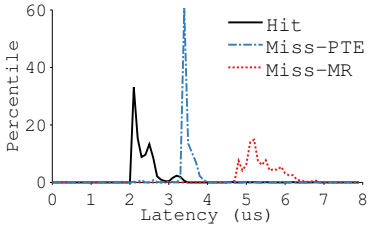


Figure 6: **Timing Differences.** *Each line presents the timing differences of each case over 1000 trials.*

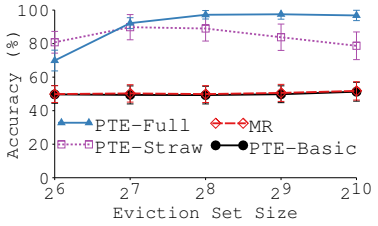


Figure 7: **Accuracy of Attacks.** *Error bars show the standard deviation across 1000 VictimVPNs.*

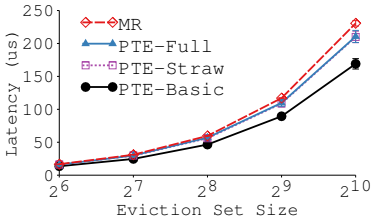


Figure 8: **Latency of Attacks.**

without any other network traffic.

Timing differences. Our side channels are based on timing differences between consecutive loads of the same memory address. To measure miss latency, we evict the RNIC SRAM using either MRs or PTEs and then issue an RDMA operation. To measure hit latency, we simply repeatedly issue the same RDMA operation. The measurements in Figure 6 show a clear timing difference between hit and miss latency. When we use MRs to evict SRAM, both the victim's MR and all the PTEs under this MR will be evicted. An RDMA operation afterwards will need to fetch both the PTE and the metadata for the MR containing the page from host main memory through the PCIe bus. On the other hand, using PTEs to evict will only evict the victim's PTE and reloading will only fetch the PTE. This explains why the miss latency of MR-based eviction is higher than that of PTE-based eviction.

Attack accuracy and latency. Figures 7 and 8 plot the accuracy and latency of four attack strategies: *PythiaMR*, $PythiaPTE_{Basic}$, $PythiaPTE_{Straw}$, and $PythiaPTE_{Full}$, as we change the eviction set size. As expected, with the same eviction set size, the time to perform these four attacks is similar, since they all use the same amount of RDMA operations. With bigger eviction sets, all attacks become slower.

$PythiaPTE_{Full}$'s accuracy is the highest: it can achieve 97% accuracy with only 57 μs per attack (when the eviction set size is 256). *PythiaMR* and $PythiaPTE_{Basic}$ have low accuracy, although we do observe *PythiaMR*'s accuracy improves significantly as the eviction set size increases (*PythiaMR*'s accuracy reaches 90% with 2^{15} eviction set size). This result demonstrates the benefit of using our reverse engineering findings.

Another observation is that the accuracy of $PythiaPTE_{Full}$ peaks when the eviction set size is 256 and remains the same when increasing the size further. This implies that the PTE cache has 128 ways, since we construct two cache sets with 256 entries in total.

Evaluation with different RNICs. All our experiments so far are performed with the Mellanox ConnectX-4 RNIC (most RDMA deployments in real datacenters use ConnectX-4 [32, 47]). We further validate our attacks on Mellanox ConnectX-5 [49] and ConnectX-3 [46] RNICs. ConnectX-5 is the latest generation of RNICs from Mellanox and ConnectX-

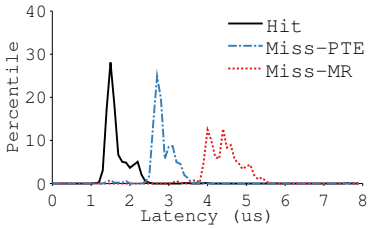


Figure 9: **Timing Differences in ConnectX-5.** *Each line presents the timing differences of each case over 1000 trials.*

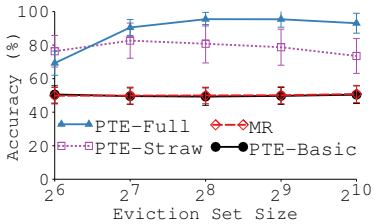


Figure 10: **Accuracy of Attacks in ConnectX-5.** *Error bars show the standard dev of 1000 VictimVPNs.*

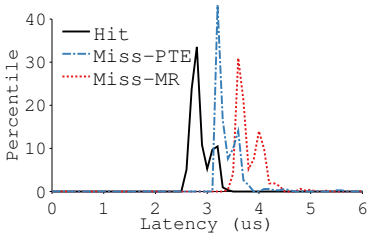


Figure 11: **Timing Differences in ConnectX-3.** *Each line presents the timing differences of each case over 1000 trials.*

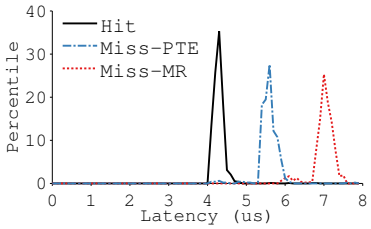


Figure 12: **Timing Differences in CloudLab.** *Each line presents the timing differences of each case over 1000 trials.*

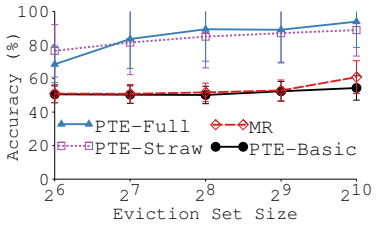


Figure 13: **Accuracy of Attacks in CloudLab.** *Error bars show the standard deviation across 1000 VictimVPNs.*

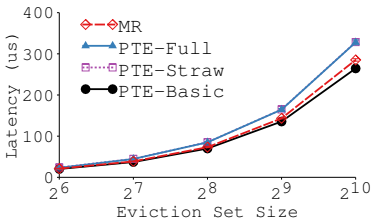


Figure 14: **Latency of Attacks in CloudLab.**

3 is the previous generation of ConnectX-4.

Figure 9 plots the timing results of SRAM hits and misses (due to eviction by MRs and by PTEs) on ConnectX-5. ConnectX-5's performance is better than ConnectX-4 on all cases. A clear timing difference between misses and hits remains, and misses caused by MR-based eviction are slower than by PTE-based eviction. Figure 10 plots the accuracy of the four types of attacks. The accuracy results are similar to ConnectX-4. Attack latency is also similar to ConnectX-4 and we omit the latency figure. Thus, we can confirm that ConnectX-5 uses a similar SRAM architecture as ConnectX-4, and it has the same side channels as ConnectX-4. We can launch the same attacks on ConnectX-5 with high accuracy and low latency.

We then perform the same set of experiments on ConnectX-3, see Figure 11. The hit latency with ConnectX-3 is longer than ConnectX-4. As hardware evolves, its internal performance often improves, which can explain why hit latency improves over generations of Mellanox RNICs. Surprisingly, the miss latency due to MR-based eviction is shorter on ConnectX-3 than on ConnectX-4 and ConnectX-5. Misses in RNIC SRAM involve the RNIC fetching metadata from the host main memory. We suspect the reason why miss performance drops in newer generations is because RNICs add more metadata for each data entry in newer generations, requiring longer time to fetch more metadata. As a result, the timing difference between miss and hit for ConnectX-3 is small.

Comparing ConnectX-3, ConnectX-4, and ConnectX-5, the three generations of RNICs from Mellanox, we found that as hardware RNICs evolve, their performance improves quickly, while the PCIe bus and host memory speed improve very slowly. As a result, the discrepancy between hit performance and miss performance becomes larger and we believe that this trend will continue in the future.

4.5.2 Public Cloud Environment

CloudLab [65] is a public cloud that has close to 15,000 cores distributed across three sites in the United States. We evaluated our attacks on a cluster that is connected with RoCE switches. Each machine in this cluster equips two Mellanox ConnectX-4 25 Gbps adapters. These RNICs are of the same product generation as our lab's ConnectX-4 RNICs, with the difference that our RNICs are 100 Gbps adapters. Both types of adapters can be configured for Ethernet (RoCE) and for InfiniBand. We configure ours for InfiniBand, and CloudLab's are configured for RoCE. Apart from RNIC differences, CloudLab is used concurrently by many different users; it has a more complex, hierarchical network topology; and it uses a RoCE network instead of InfiniBand. At the time of our test, 129 out of 199 physical machines in the cluster were in use.

We repeat the same set of experiments as Section 4.5.1. Similar to our lab's experiments, we use three machines, a server, a victim client, and an attacker client. Figure 12 plots the timing difference of RNIC SRAM hit and misses (due to MR-based eviction and PTE-based eviction). Similar to our isolated environment results, misses caused by MR-based eviction are slower than misses caused by PTE-based eviction, and both types of misses are slower than hits. In CloudLab's shared network and shared machine environments, the latencies of all accesses are longer than in our lab environment,

but the timing differences are still clear.

Figures 13 and 14 plot the accuracy and latency of the four types of attacks in CloudLab. Similar to the results in Figures 7 and 8, With the same eviction set size (and thus similar latency), *PythiaPTE*$_{Full}$ and *PythiaPTE*$_{Straw}$ have higher accuracy than *PythiaMR* and *PythiaPTE*$_{Basic}$. However, these attacks have larger variation in accuracy compared to attacks in our lab's environment because of the more dynamic environment in CloudLab.

5 Attacking Real RDMA-Based Systems

To demonstrate the feasibility of launching side-channel attacks on real RDMA-based applications, we design and perform a set of attacks on *Crail* [7, 70], an open-source RDMA-based key-value store written in Java. A Crail system consists of several roles: a server which stores key-value pairs, a namenode which stores metadata and manages the control path, and clients which issue key-value pair *get*s and *set*s to the server via a Crail-provided API. We install each component on a separate machine and connect all of them with RDMA. This section presents our design and evaluation of attacks on Crail.

5.1 Attacks

Based on the attack primitives described in Section 4, we designed three attacks on Crail. All these attacks have the same goal: knowing whether or not the victim Crail client accesses a specific key-value pair.

MR-based attack (*PythiaCrail*$_{MR}$). Our first attack uses MR-based eviction as described in Section 4.3.1. This attack requires three attacker processes. The first is a Crail client process (P_c). The second and the third processes run our attack code, with the second one running on the Crail server machine (P_s) and the third one running on any other machine (P_a) (it can be the same machine as the one where P_c runs). In the preparation phase, P_s registers a large number of MRs. In the eviction phase, P_a issues one-sided RDMA reads to these MRs. Finally, P_c performs a Crail *get* operation to reload the victim key-value pair. *PythiaCrail*$_{MR}$ requires P_s and P_a because MR-based attacks need to access many MRs but by default Crail only registers a small set of MRs.

PTE-based attack (*PythiaCrail*$_{PTE}$). The second attack uses PTE-based eviction. In this attack, we require three processes as in the MR-based attack: P_c, P_s, and P_a. In the preparation phase, we first use P_s to allocate a big chunk of memory and register it with an MR. In the eviction phase, P_a performs one-sided RDMA reads to different VPNs in the allocated memory space. Afterwards, P_c issues a Crail *get* request to the victim key-value pair.

Our reverse engineering results in Section 4.4 can be leveraged to reduce the eviction set size in *PythiaCrail*$_{PTE}$. However, to form the eviction set, we need to know the index of the SRAM cache set(s), which is calculated by the virtual memory address. Without modifying the source code of Crail which is written in Java, it is difficult to directly know the virtual memory address of a target key-value pair. Instead, we use a "learning" phase before launching the actual attack to determine the eviction set to use for a target key-value pair. Specifically, we let P_c access the target key-value pair and let P_a try all 1024 different cache sets for eviction. After 1024 trials, we pick the cache set that yields the best accuracy as our attack eviction set.

Client-only attack (*PythiaCrail*$_{Client}$). Our last attack on Crail is launched exclusively from a regular Crail client process and requires no other privileges or resources. The attacker (as a normal Crail client) issues Crail *get* requests to different key-value pairs during the eviction phase. After the eviction phase, it performs a Crail *get* operation to the victim key-value pair.

Our initial design of *PythiaCrail*$_{Client}$ randomly picks key-value pairs to access during the eviction phase. However, we soon discovered two issues with this naive approach. First, it needs a large number of key-value pairs to effectively evict the target key-value pair. Doing so not only makes the attack slow but also requires the Crail system under attack to already be storing many key-value pairs. Second, we found that the Crail system becomes slower and unstable as the server processes more client requests. We suspect this to be caused by Crail's own (memory) management overhead. Unstable access latency makes our timing-based attack harder and prohibits an accurate prediction during the reload phase. We improve our initial design with the following optimization. We selectively choose a small set of key-value pairs as the eviction set. We make the assumption that key-value pairs are sequentially allocated in chunks of memory and pick the pairs that are likely to be in the same RNIC SRAM cache set as the victim key-value pair. After reducing the eviction set size, our attack runs very fast. Instead of continuously launching the attack in loops, we add some sleep time between eviction and reloading so that we do not issue too many Crail requests to make Crail unstable.

Probabilistic prediction. Under real workloads and noisy network environments, we found that a simple threshold as used in Section 4.3 cannot accurately determine if the victim has accessed the target data. Thus, we use a more dynamic and adaptive approach to predict the outcome of the attack. Similar to the approach used in TLBleed [28], we perform a learning phase to train a classifier of operation latency with KNN [22] before the attack. We use the trained model to predict the probability of a reload latency implying a victim access (*i.e.*, a hit).

5.2 Results

We evaluated *PythiaCrail*$_{MR}$, *PythiaCrail*$_{PTE}$, and *PythiaCrail*$_{Client}$ using both controlled tests and work-

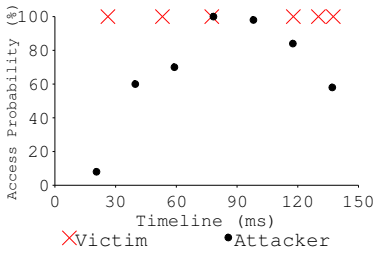


Figure 15: *PythiaCrail*$_{MR}$

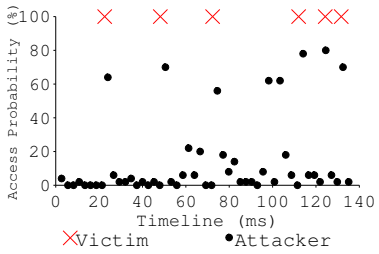


Figure 16: *PythiaCrail*$_{PTE}$

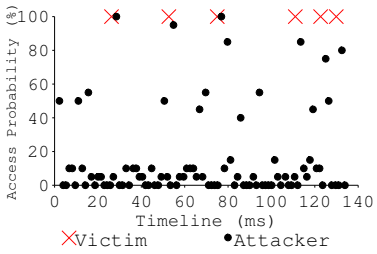


Figure 17: *PythiaCrail*$_{Client}$

loads that model real datacenter key-value stores.

5.2.1 Controlled Test

We first compare the latency of a Crail client key-value pair *get* operation that hits RNIC SRAM, a client *get* that misses RNIC SRAM after the eviction phase in *PythiaCrail*$_{MR}$, in *PythiaCrail*$_{PTE}$, and in *PythiaCrail*$_{Client}$. In these controlled tests, the victim client has a 50% chance of accessing the targeted key-value pair that the attacker tries to infer accesses on. Figure 18 plots these four types of latencies, each performing 1000 trials. All the three types of misses take longer than hits, with the timing difference of *PythiaCrail*$_{MR}$ the biggest and *PythiaCrail*$_{Client}$ the smallest. The timing difference implies that it is easiest to separate hits and misses with *PythiaCrail*$_{MR}$.

We launch the *PythiaCrail*$_{MR}$, and *PythiaCrail*$_{PTE}$, and *PythiaCrail*$_{Client}$ attacks by first performing their respective eviction phases. Next, we let victim access or not access the target key-value pair. Finally, we measure the time to reload the key-value pair and compare it with a threshold we determined from the timing difference testing phase. As expected, *PythiaCrail*$_{MR}$ gives the best accuracy. The accuracies of *PythiaCrail*$_{MR}$, *PythiaCrail*$_{PTE}$, and *PythiaCrail*$_{Client}$ are 96%, 85%, and 79% respectively, and the time to perform these attacks are 19*ms*, 0.1*ms*, and 0.3*ms*.

5.2.2 Macro-benchmark Results

Workloads. To evaluate how our attacks perform with real datacenter key-value store workloads, we construct a macro-benchmark with the Yahoo! Cloud Serving Benchmark (YCSB) [21] and statistics reported by Facebook in their production key-value store [12]. YCSB provides key-value *get/set* access pattern but no inter-arrival time between requests. Facebook provides the inter-arrival time of requests received at a server in its cluster, which includes requests from all the clients to this server. We set each key-value pair size to be 1 KB, the average key-value pair size reported by Facebook.

Attack environment setup. In this experiment, the victim (on a Crail client machine) executes our macro-benchmark to access key-value pairs on a Crail server machine using the Crail APIs. Since Facebook only provides aggregated request inter-arrival time across clients and does not reveal how many clients there are, we use one client machine to model the aggregated effect of all clients with the provided inter-arrival time. We run an attacker process as a normal Crail client on another machine. A fourth machine serves as the Crail namenode. While the victim process executes the macro-benchmark, we repeatedly perform *PythiaCrail*$_{Client}$, *PythiaCrail*$_{MR}$, or *PythiaCrail*$_{PTE}$ to detect if the victim accesses a target key-value pair.

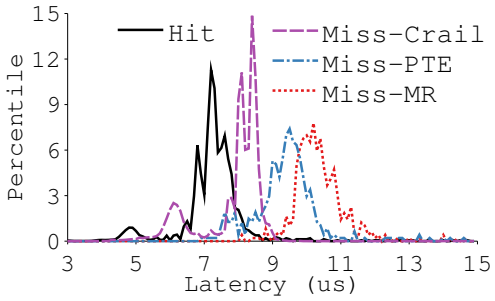


Figure 18: **Timing Difference in Crail.** *Each line presents the timing differences of each case over 1000 trials.*

Results. Figures 15, 16, and 17 present the timeline of the victim accessing the target key-value pair (red crosses) and the attacker's prediction (black dots with values as access probability). All three attacks can capture most if not all victim accesses. Among them, *PythiaCrail*$_{MR}$ is the worst in attack accuracy. This is because each attack in *PythiaCrail*$_{MR}$ takes 19*ms*, which is much longer than the Facebook inter-arrival time. As a result, *PythiaCrail*$_{MR}$ misses victim accesses that happen more frequent than its attack length. Both *PythiaCrail*$_{PTE}$ and *PythiaCrail*$_{Client}$ run very fast and capture all victim accesses. In fact, these two attacks run so fast that we add a sleep time of 1*ms* between evict and reload to avoid issuing too many Crail requests and making Crail's performance unstable. Comparing *PythiaCrail*$_{PTE}$ and *PythiaCrail*$_{Client}$, *PythiaCrail*$_{PTE}$'s predictions are of low access probabilities and *PythiaCrail*$_{Client}$ has more predictions of around 50% access probabilities. The attacker can set a threshold accordingly to determine the final set of victim accesses (*e.g.*, those with probabilities $>$ 60% for *PythiaCrail*$_{Client}$).

Overall, we believe *PythiaCrail*$_{Client}$ to be the most effec-

tive attack, since it predicts victim accesses with high confidence and it requires the least amount of attacker resources: *PythiaCrail*$_{Client}$ can be launched exclusively from a separate client machine through the unmodified Crail client interface. If attackers can run modified Crail clients, they can launch more efficient side-channel attacks by forming the eviction set with known virtual addresses.

6 Mitigation Techniques

Defending against RDMA-based side-channel attacks is possible and feasible. We discuss both mitigations for current hardware as well as those for future hardware.

Huge virtual memory page or no virtual memory. PTE-based attacks are only possible when RNICs cache PTEs and when the attacker can form an effective eviction set. One way to prevent PTE-based attacks is to force all RDMA registrations and operations to directly use physical memory addresses. When physical memory addresses are used, RDMA does not need to access or cache PTEs, thereby preventing PTE-based attacks. Registering physical memory addresses is a privileged operation that RNICs allow the kernel [76] and privileged users to perform [51]. However, using physical memory addresses loses all the benefits of virtual memory and introduces new security concerns.

Another method to defend against PTE-based attacks is to use huge memory pages [24]. Using huge pages (*e.g.*, 1 GB pages) introduces two types of difficulties for attackers. First, the attacker can only guess victim accesses at coarse granularity (*e.g.*, 1 GB). Second, the attacker will need to have access to a huge memory space to form an eviction set with enough PTEs.

Isolate server's resource. Our experience with Crail demonstrates that attacking Crail is difficult when the attacker can only use Crail's interface without the access to a large number of PTEs or MRs and without knowing Crail's data layout in the virtual memory address space. Our experiments show that for *PythiaCrail*$_{MR}$ and *PythiaCrail*$_{PTE}$ to work, an attacker needs to run a process, P_s, on the server machine. Otherwise, the attacker would not be able to launch those attacks (although *PythiaCrail*$_{Client}$ still works). Thus, a server that hosts RDMA service can prohibit normal users from running any processes to help defend against side-channel attacks. Various address randomization techniques can also complicate attacks.

Separate protection domains. When we disclosed the attacks in this paper to Mellanox, the Mellanox engineers stated that separating *Protection Domain*s (*PD*s) between different clients and connections can potentially mitigate the attacks. We evaluated this mitigation by moving the attacker to a different PD and found that doing so mitigates Pythia attacks. Unfortunately, all existing RDMA applications that we are aware of [7, 24, 83] use only one PD for higher performance. Using multiple PDs results in low throughput and high latency overhead (15% throughput reduction and 21% latency overhead with 256 PDs in our experiments). We plan to further investigate both attack and defense mechanisms when separating PDs across clients.

Introduce noise. Our side channels are established on timing differences at the microsecond or sub-microsecond level. Attacking Crail running real workloads is more difficult than attacking raw RDMA accesses mainly because of Crail's non-deterministic performance overhead. Therefore, an effective countermeasure is to introduce random latency overhead at an RDMA-based application or in the datacenter RDMA network, which, however, could impact application performance.

Detect and throttle attacker's network traffic. Our attackers can hide their attacks because one-sided RDMA operations are completely hidden from the receiver CPU (the server in our case). To detect these attacks, the server can deploy traffic sniffing tools to sniff all incoming RDMA network requests. If the sniffer detects heavy network activity from a client, it can raise a flag that this client may be malicious. If it further detects an access pattern that matches eviction sets described in Section 4.4, this client is more likely to be an attacker. This defense comes with the same drawbacks of other heuristic-based defenses that an attacker may stay under the detection threshold.

A further countermeasure is to throttle the maximum bandwidth allowed at every client. If an attacker cannot issue enough operations to evict RNIC SRAM, its attack accuracy will drop significantly. However, throttling client bandwidth can hurt normal clients' performance.

Better hardware design. All existing RNICs share their SRAM across all users and across all connections. Because of this, an attacker can evict a victim's PTE and MR even when the attacker and the victim have different connections to the server. If RNICs can partition their SRAM to different isolated domains for different connections, then attackers can never evict victim's PTEs or MRs. However, isolation resources at hardware level will inevitably hurt performance and increase hardware complexity, which gives little incentive for RNIC vendors to change their hardware design.

7 Discussion

We now briefly discuss the implications, impact, and limitations of Pythia, and some other attacks on RDMA that can be designed based on Pythia.

7.1 RDMA Vulnerabilities

We discovered new vulnerabilities in RDMA systems that are fundamental to the design of RDMA and not specific to just one RDMA device. RNICs cache metadata as a result

of RDMA's design philosophy of *one-sided* network communication. Because one-sided operations cannot involve host CPUs, RNICs have to handle and serve RDMA requests on their own, which involves accessing various types of metadata. With limited on-board SRAM, RNICs cannot store all the metadata and have to move metadata between their SRAM and the host machine's main memory through the PCIe bus. As a result, there exists timing difference between RDMA operations that hit or miss SRAM, and this timing difference keeps increasing as RNICs evolve over generations.

We demonstrated the feasibility of exploiting the above vulnerability to launch side-channel attacks on RDMA-based systems. Pythia attacks are fast and accurate, and they can be performed completely from the network. Moreover, attackers can hide their traces because the attack uses one-sided network requests.

Both the RNIC side channels we discovered and our attacks' unique advantages are fundamental to one-sided network communication. One-sided communication offers many performance and cost benefits that are attractive for datacenter systems. However, it also raises new security concerns [77], as we demonstrate in Pythia. Our work can inspire future security researchers in discovering and defending more vulnerabilities in RDMA.

7.2 Attacking Real Applications

We demonstrated that it is feasible to launch Pythia attacks on Crail, a real RDMA-based system developed by the Crail team. $PythiaCrail_{Client}$, the attack that is launched by performing Crail-provided client APIs only, can successfully infer victim's access patterns under real workloads.

We believe that Pythia can similarly attack other RDMA-based applications as well. Pythia only requires two features from an RDMA-based application: the application uses one-sided RDMA operations and allocates regular paged memory. Many applications meet these requirements, such as the NAM-DB RDMA-based in-memory database [88], the Pilaf RDMA-based key-value store [56], and the Wukong RDMA-based graph system [68]. Unfortunately, most of these systems are not available publicly.

7.3 Attack Limitations

Although our side-channel attacks are fast, accurate, and can be launched entirely from the network, they do have several limitations. First, the granularity of Pythia attacks (and therefore information leakage of accesses) is a memory page. Pythia currently cannot differentiate between two victim accesses that access the same target page. Second, Pythia can only predict if a data entity at the server has been accessed, but not which client machine(s) accessed it. Third, our MR-based attacks require access to a large number of MRs, and PTE-based attacks require access to large memory spaces. Finally, our attacks consume network bandwidth and can be detected by sniffing the network.

7.4 Other RDMA-Based Attacks

Pythia serves as a starting point for designing other types of RDMA-based attacks. For example, similar to Pythia EVICT+RELOAD attacks, it is possible to launch PRIME+PROBE attacks from the network by exploiting the MR or the PTE side channels.

The MR and the PTE side channels we established can also be used as covert channels. We implemented a naive covert-channel attack of using one EVICT+RELOAD cycle to transmit one bit and it could reach a sending rate of 20 Kbps with $PythiaPTE_{Full}$.

8 Related Work

Single-node side-channel attacks. In recent years, a host of attacks that exploit various hardware features to establish side channels have been proposed. CPU-cache-based side-channel attacks such as PRIME+PROBE [1–4, 42, 59, 75, 78, 90], EVICT+RELOAD [30], FLUSH+RELOAD [86], and FLUSH+FLUSH [30, 78] can leak victim's memory access patterns at fine granularity. CPU-cache-based side channels are also the key enabling factors in attacks like Meltdown [41], Spectre [39], and Foreshadow [17]. Other than CPU caches, TLB [28] or port contention [13] also expose hardware-based side channels. Side-channel attacks brought key concerns in cloud environments where one tenant can steal information of other tenants when they share the same physical resource [79, 89–91] or the same service [33]. But all these single-node side-channel attacks require the attacker to run on the same machine as the victim. Pythia is a remote side-channel attack that can be launched completely from a separate remote machine.

Remote side-channel attacks. Several remote side-channel attacks exploit TCP sequence numbers to hijack connections [18, 61, 62]. Another line of work relies on traffic analysis to exploit sensitive information [27, 37]. Brumley *et al.* perform a timing-based attack on OpenSSL's ladder implementation to obtain the private key of a TLS server [16]. Cock *et al.* present an empirical study of remote timing channels on microkernel [20]. NetSpectre [66] presents an access-driven remote EVICT+RELOAD cache attack. Weinberg *et al.* combined CSS and JavaScript to remotely sniff victims' browsing patterns [84]. Different from all previous remote side-channel attacks, Pythia targets the RDMA network. Moreover, Pythia exploits RNIC hardware features to establish timing-based side channels, while previous remote side channels exploit network protocols or software features. As far as we know, Throwhammer [71] is the only other attack related to RDMA. However, it simply uses RDMA network requests to launch a Rowhammer attack and does not explore or exploit vulnerabilities in the RDMA

technology itself.

Mitigations to side-channel attacks. Various defense mechanisms have been proposed to combat CPU cache side-channel attacks in both hardware [23, 36, 58, 81, 82] and software [38, 43, 69, 92, 94]. Unfortunately, none of the existing defense mechanisms can be directly applied to RDMA-based side-channel attacks. We propose a set of new mitigations that target RDMA-based side-channel attacks.

9 Conclusion

This paper presents Pythia, the first set of side-channel attacks on RDMA-based systems. We reverse engineer the internal data structures of current RDMA systems and leverage this information to improve our attack. Pythia can be launched completely from a normal client machine to steal access patterns of victims on other machines. We evaluate Pythia in laboratory settings to showcase the capabilities of the attack, on real software such as Crail, and on real data centers to show real-world impact. Pythia is fast, accurate, and can hide its trace from victims and the server.

Acknowledgments

We would like to thank the anonymous reviewers for their tremendous feedback and comments, which have substantially improved the content and presentation of this paper. We are also thankful to Ahmad Atamlh, Noam Bloch, Brandon Hathaway, Yuval Itkin, Ariel Levanon, Alex Polak, Randy Splinter, and Patrick Stuedi for their feedback during our responsible disclosure to Mellanox and the Crail team.

This material is based upon work supported by the National Science Foundation under the following grant: NSF 1719215. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

[1] Onur Aciiçmez. Yet another microarchitectural attack: Exploiting i-cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture (CSAW '07)*, Fairfax, VA, USA, November 2007.

[2] Onur Aciiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *Proceedings of the 12th International Conference on Cryptographic Hardware and Embedded Systems (CHES '10)*, Santa Barbara, CA, USA, August 2010.

[3] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *Proceedings of the 2Nd ACM Symposium on Information, Computer and Communications Security (ASIACCS '07)*, Singapore, March 2007.

[4] Onur Aciiçmez and Werner Schindler. A vulnerability in rsa implementations due to instruction cache analysis and its demonstration on openssl. In *Proceedings of the 2008 The Cryptopgraphers' Track at the RSA Conference on Topics in Cryptology (CT-RSA '08)*, San Francisco, CA, USA, April 2008.

[5] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: A simple abstraction for remote memory. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (ATC '18)*, Boston, MA, USA, July 2018.

[6] Alibaba Cloud. Super computing cluster. https://www.alibabacloud.com/product/scc, 2018.

[7] Apache. Crail: High-performance distributed data store. https://crail.incubator.apache.org/, 2018.

[8] InfiniBand Trade Association. InfiniBand Architecture Annex A 16: RoCE. https://cw.infinibandta.org/document/dl/7148, April 2010.

[9] InfiniBand Trade Association. InfiniBand Architecture Annex A 16: RoCEv2. https://cw.infinibandta.org/document/dl/7148, September 2014.

[10] InfiniBand Trade Association. InfiniBand Architecture Volume 1 – Architecture Specification, Release 1.3. https://cw.infinibandta.org/document/dl/7859, March 2015.

[11] InfiniBand Trade Association. InfiniBand Architecture Volume 2 – Architecture Specification, Release 1.3.1. https://cw.infinibandta.org/document/dl/8125, November 2016.

[12] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*, London, United Kingdom, June 2012.

[13] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: exploiting speculative execution through port contention. https://arxiv.org/abs/1903.01843, 2018.

[14] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The End of Slow Networks: It's Time for a Redesign. *Proceedings of the VLDB Endowment*, 9(7):528–539, 2016.

[15] Rajarshi Biswas, Xiaoyi Lu, and Dhabaleswar Panda. Accelerating tensorflow with adaptive rdma-based grpc. In *25th IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC '18)*, Bengaluru, India, December 2018.

[16] Billy Bob Brumley and Nicola Tuveri. Remote timing

attacks are still practical. In *Proceedings of the 16th European Conference on Research in Computer Security (ESORICS '11)*, Leuven, Belgium, September 2011.

[17] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Conference on Security Symposium (SEC '18)*, Baltimore, MD, USA, August 2018.

[18] Yue Cao, Zhiyun Qian, Zhongjie Wang, Tuan Dao, Srikanth V. Krishnamurthy, and Lisa M. Marvel. Off-path tcp exploits: Global rate limit considered dangerous. In *Proceedings of the 25th USENIX Conference on Security Symposium (SEC '16)*, Austin, TX, USA, August 2016.

[19] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems (EUROSYS '16)*, London, UK, April 2016.

[20] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. The last mile: An empirical study of timing channels on sel4. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*, Scottsdale, Arizona, USA, November 2014.

[21] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*, New York, New York, June 2010.

[22] Thomas. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, September 1967.

[23] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization*, 8(4):35:1–35:21, January 2012.

[24] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI '14)*, Seattle, WA, USA, April 2014.

[25] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*, Monterey, CA, USA, October 2015.

[26] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by key-extraction cache attacks from portable code. In *Applied Cryptography and Network Security - 16th International Conference (ACNS '18)*, Leuven, Belgium, July 2018.

[27] Xun Gong, Nikita Borisov, Negar Kiyavash, and Nabil Schear. Website detection using remote traffic analysis. In *Privacy Enhancing Technologies - 12th International Symposium (PETS '12)*, Vigo, Spain, July 2012.

[28] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *Proceedings of the 27th USENIX Conference on Security Symposium (SEC '18)*, Baltimore, MD, USA, August 2018.

[29] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '16)*, San Sebastián, Spain, July 2016.

[30] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC '15)*, Washington, D.C., USA, August 2015.

[31] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with infiniswap. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI '17)*, Boston, MA, USA, March 2017.

[32] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*, Florianopolis, Brazil, August 2016.

[33] Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Side channels in cloud services: Deduplication in cloud storage. *IEEE Security and Privacy*, 8(6):40–47, November 2010.

[34] Wei Huang, Gopalakrishnan Santhanaraman, Hyun-Wook Jin, Qi Gao, and Dhabaleswar K. Panda. Design of High Performance MVAPICH2: MPI2 over InfiniBand. In *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID '06)*, Rio de Janeiro, Brazil, May 2006.

[35] Intel. Intel Performance Counter Monitor, 2012. http://www.intel.com/software/pcm.

[36] Intel. Improving Real-Time Performance by Utilizing Cache Allocation Technology , 2015. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf.

[37] Rob Jansen, Marc Juarez, Rafa Galvez, Tariq Elahi, and Claudia Diaz. Inside job: Applying traffic analysis to measure tor from within. In *25th Annual Network and*

Distributed System Security Symposium (NDSS '18), San Diego, CA, USA, February 2018.

[38] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX Conference on Security Symposium (SEC '12)*, Bellevue, WA, USA, August 2012.

[39] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP '19)*, San Francisco, CA, USA, May 2019.

[40] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, October 2017.

[41] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Conference on Security Symposium (SEC '18)*, Baltimore, MD, USA, August 2018.

[42] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*, San Jose, CA, USA, May 2015.

[43] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA '16)*, Barcelona, Spain, March 2016.

[44] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. High Performance RDMA-based MPI Implementation over infiniBand. *International Journal of Parallel Programming*, 32(3):167–198, 2004.

[45] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an rdma-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (ATC '17)*, Santa Clara, CA, USA, July 2017.

[46] Mellanox. Mellanox ConnectX-3 VPI Card, 2017. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-3_Pro_Card_VPI.pdf.

[47] Mellanox. Mellanox Network Adapters for 25G RoCE Ethernet Cloud Deployed in Alibaba, 2017. http://www.mellanox.com/page/press_release_item?id=1964.

[48] Mellanox. Mellanox ConnectX-4 VPI Card, 2018. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-4_VPI_Card.pdf.

[49] Mellanox. Mellanox ConnectX-5 VPI Card, 2018. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-5_VPI_Card.pdf.

[50] Mellanox. Mellanox InfiniBand EDR 100Gb/s Switch, 2018. http://www.mellanox.com/related-docs/prod_ib_switch_systems/pb_sb7700.pdf.

[51] Mellanox. Physical Address Memory Region, 2018. https://community.mellanox.com/s/article/physical-address-memory-region.

[52] Mellanox. RDMA/RoCE Solutions. https://community.mellanox.com/s/article/rdma-roce-solutions, 2018.

[53] Mellanox. Mellanox ConnectX-6 VPI Card, 2019. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-6_VPI_Card.pdf.

[54] Mellanox Technologies. InfiniBand Now Connecting More than 50 Percent of the TOP500 Supercomputing List. https://tinyurl.com/yy2ualhg, 2015.

[55] Frank Mietke, Robert Rex, Robert Baumgartl, Torsten Mehlan, Torsten Hoefler, and Wolfgang Rehm. Analysis of the memory registration process in the mellanox infiniband software stack. In *Proceedings of the 12th International Conference on Parallel Processing (Euro-Par '06)*, Dresden, Germany, September 2006.

[56] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (ATC '13)*, San Jose, CA, USA, June 2013.

[57] Christopher Mitchell, Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. Balancing cpu and network in the cell distributed b-tree store. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (ATC '16)*, Denver, CO, USA, June 2016.

[58] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (ATC '18)*, Boston, MA, USA, July 2018.

[59] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA '06)*, San Jose, CA, USA, February 2006.

[60] Marius Poke and Torsten Hoefler. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '15)*, Portland, OR, USA, June 2015.

[61] Zhiyun Qian and Z. Morley Mao. Off-path tcp sequence number inference attack - how firewall middleboxes reduce security. In *Proceedings of the 2012 IEEE Sym-*

posium on Security and Privacy (SP '12), San Francisco, CA, USA, May 2012.

[62] Zhiyun Qian, Z. Morley Mao, and Yinglian Xie. Collaborative tcp sequence number inference attack: How to crack sequence number under a second. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*, Raleigh, NC, USA, October 2012.

[63] RDMA Consortium. iWARP, Protocol of RDMA over IP Networks, 2009. http://www.rdmaconsortium.org/.

[64] Recio, R., Metzler, B., Culley, P., Hilland, J., and D. Garcia. A Remote Direct Memory Access Protocol Specification, 2007. https://tools.ietf.org/html/rfc5040.

[65] Robert Ricci, Eric Eide, and the CloudLab Team. Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications. *The USENIX Magazine*, 39(6):36–38, December 2014.

[66] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. Netspectre: Read arbitrary memory over network, 2018. http://arxiv.org/abs/1807.10535.

[67] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 8th Annual Symposium on Cloud Computing (SOCC '17)*, Santa Clara, CA, USA, September 2017.

[68] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and concurrent rdf queries with rdma-based distributed graph exploration. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, USA, November 2016.

[69] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSNW '11)*, Hong Kong, China, June 2011.

[70] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Nikolas Ioannou, and Ioannis Koltsidas. Crail: A high-performance i/o architecture for distributed data processing. *IEEE Bulletin of the Technical Committee on Data Engineering*, 40:40–52, March 2017. Special Issue on Distributed Data Management with RDMA.

[71] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer attacks over the network and defenses. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC '18)*, Boston, MA, USA, July 2018.

[72] Mellanox Technologies. Mellanox OFED for Linux User Manual. http://www.mellanox.com/related-docs/prod_software/Mellanox_OFED_Linux_User_Manual_v3.1-1.0.0.pdf.

[73] Tejas Karmarkar. Availability of linux rdma on microsoft azure. https://azure.microsoft.com/en-us/blog/azure-linux-rdma-hpc-available, 2015.

[74] The Tcpdump Group. tcpdump - Dump Traffic on A Network. https://www.tcpdump.org/manpages/tcpdump.1.html, 2018.

[75] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 23(1):37–71, January 2010.

[76] Shin-Yeh Tsai and Yiying Zhang. LITE Kernel RDMA Support for Datacenter Applications. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, October 2017.

[77] Shin-Yeh Tsai and Yiying Zhang. A double-edged sword: Security threats and opportunities in one-sided network communication. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '19)*, Renton, WA, USA, July 2019.

[78] Stephan van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious management unit: Why stopping cache attacks in software is harder than you think. In *Proceedings of the 27th USENIX Conference on Security Symposium (SEC '18)*, Baltimore, MD, USA, August 2018.

[79] Luis M. Vaquero, Luis Rodero-Merino, and Daniel Morán. Locking the sky: A survey on iaas cloud security. *Computing*, 91(1):93–118, 2011.

[80] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. Apus: Fast and scalable paxos on rdma. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*, Santa Clara, CA, USA, September 2017.

[81] Zhenghong Wang and Ruby B. Lee. Covert and side channels due to processor architecture. In *Proceedings of the 22Nd Annual Computer Security Applications Conference (ACSAC '06)*, Miami Beach, FL, USA, December 2006.

[82] Zhenghong Wang and Ruby B. Lee. A novel cache architecture with enhanced performance and security. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '41)*, Lake Como, ITALY, November 2008.

[83] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, Carlsbad, CA, USA, October 2018.

[84] Zachary Weinberg, Eric Y. Chen, Pavithra Ramesh Jayaraman, and Collin Jackson. I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (SP '11)*, Oakland, CA, USA, May 2011.

[85] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and

Lidong Zhou. Gram: Scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15)*, Kohala Coast, HI, USA, August 2015.

[86] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium (SEC '14)*, San Diego, CA, USA, August 2014.

[87] Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: A timing attack on openssl constant time RSA. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017.

[88] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The End of a Myth: Distributed Transactions Can Scale. *Proceedings of the VLDB Endowment*, 10(6):685–696, 2017.

[89] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (SP '11)*, Oakland, CA, USA, May 2011.

[90] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*, Raleigh, NC, USA, October 2012.

[91] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*, Scottsdale, Arizona, USA, November 2014.

[92] Yinqian Zhang and Michael K. Reiter. Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 2013 ACM SIGSAC conference on Computer and communications Security (CCS '13)*, Berlin, Germany, November 2013.

[93] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*, Istanbul, Turkey, March 2015.

[94] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. A software approach to defeating side channels in last-level caches. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*, Vienna, Austria, October 2016.

HideMyApp : Hiding the Presence of Sensitive Apps on Android

Anh Pham
ABB Corporate Research, Switzerland

Italo Dacosta
EPFL, Switzerland

Eleonora Losiouk
University of Padova, Italy

John Stephan
EPFL, Switzerland

Kévin Huguenin
University of Lausanne, Switzerland

Jean-Pierre Hubaux
EPFL, Switzerland

Abstract

Millions of users rely on mobile health (mHealth) apps to manage their wellness and medical conditions. Although the popularity of such apps continues to grow, several privacy and security challenges can hinder their potential. In particular, the simple fact that an mHealth app is installed on a user's phone can reveal sensitive information about the user's health. Due to Android's open design, any app, even without permissions, can easily check for the presence of a specific app or collect the entire list of installed apps on the phone. Our analysis shows that Android apps expose a significant amount of metadata, which facilitates fingerprinting them. Many third parties are interested in such information: Our survey of 2917 popular apps in the Google Play Store shows that around 57% of these apps explicitly query for the list of installed apps. Therefore, we designed and implemented `HideMyApp` (`HMA`), an effective and practical solution for hiding the presence of sensitive apps from other apps. `HMA` does not require any changes to the Android operating system or to apps yet still supports their key functionalities. By using a diverse dataset of both free and paid mHealth apps, our experimental evaluation shows that `HMA` supports the main functionalities in most apps and introduces acceptable overheads at runtime (*i.e.,* several milliseconds); these findings were validated by our user-study ($N = 30$). In short, we show that the practice of collecting information about installed apps is widespread and that our solution, `HMA`, provides a robust protection against such a threat.

1 Introduction

Mobile health (mHealth), the use of technologies such as smartphones and wearable sensors for wellness and medical purposes, promises to improve the quality of and reduce the costs of medical care and research. An increasing number of people rely on mHealth apps to manage their wellness and to prevent and manage diseases.[1] For instance, more than a third of physicians in the US recommend mHealth apps to their patients [23], and there are around 325,000 mHealth apps available in major mobile app stores.[2]

Given the sensitivity of medical data, the threats of privacy leakage are one of the main hindrances to the success of mHealth technologies [37]. In this area, a serious and often overlooked threat is that an adversary can infer sensitive information simply from the presence of an app on a user's phone. Previous studies have shown that private information, such as age, gender, race, and religion, can be inferred from the list of installed apps [22, 29, 47]. With the increasing popularity of mHealth apps, an adversary can now infer even more sensitive information. For example, learning that a user has a diabetes app reveals that the user probably suffers from this disease; such information could be misused to profile, discriminate, or blackmail the user. When inquired about this threat, 87% of the participants in our user-study expressed concern about it (Section 10.6).

Due to Android's open design, a zero-permission app can easily infer the presence of specific apps, or even collect the full list of installed apps on the phone [55]. Our analysis shows that Android exposes a considerable amount of static and runtime metadata about installed apps (Section 4); this information can be misused by a nosy app to accurately fingerprint these apps. In 2014, Twitter was criticized for collecting the list of installed apps in order to offer targeted ads.[3] But Twitter is not the only app interested in such information. Our static and dynamic analysis of 2917 popular apps in the US Google Play Store shows that approximately 57% of these apps include calls to API methods that explicitly collect the list of installed apps (Section 5). Our analysis, corroborating the findings of previous studies [29, 32], also shows that free apps are more likely to query for such information and that third-party libraries (libs) are the main requesters of the list of installed apps. As users have on average 80 apps installed on their phones,[4] most of them being free, there is a high chance of untrusted third-parties obtaining the list of installed apps.

Since 2015, Android has classified as potentially harmful apps (PHA)[5] the apps that collect information about other apps without user consent [1]. To avoid this classification,

developers simply need to provide a privacy policy that describes how the app collects, uses, and shares user data.[6] We find it interesting that only 7.7% of the evaluated apps clearly declared that they collect the list of installed apps in their privacy policies, and some even claim that such a list is non-personal information (Section 5.4). Also, few users read privacy policies [41], as our user study also confirmed (Section 10.6).

Android does not provide mechanisms to hide the use of sensitive apps on a phone; a few third-party tools, designed for other purposes, can provide only partial protection to some users (Section 6). Android announced that their security services will display warnings on apps that collect without consent users' personal information, including the list of installed apps.[7] This is a welcomed step, but the effectiveness of security warnings is known to be limited [30, 49] and it is unclear how queries by third-party libraries will be handled. It is also unclear if such an approach will be able to prevent more subtle attacks, where a nosy app checks for the existence of a specific app or a small set of sensitive apps by using more advanced fingerprinting techniques (Section 4).

We propose `HideMyApp` (`HMA`), the first system that enables organizations and developers to distribute sensitive apps to their users while considerably reducing the risk of such apps being detected by nosy apps on the same phone. Apps protected by `HMA` expose significantly less identifying metadata, therefore, it is more difficult for nosy apps to detect their presence, even when the nosy apps have all Android permissions and debugging privileges. With `HMA`, an organization such as a consortium of hospitals sets up an `HMA` app store where authorized developers collaborating with the hospitals can publish their mHealth and other sensitive apps. Users employ a client app called `HMA Manager` to anonymously (un)install, use, and to update the apps selected from the `HMA` app store; an the `HMA App Store` does not learn about the set of apps that a user has installed from the store. `HMA` transparently works on stock Android devices, it does not require root access, and it preserves the app-isolation security model of the Android operating system (OS). Still, `HMA` preserves the key functionalities of mHealth apps, *e.g.,* connecting to external devices via Bluetooth, sending information over the Internet, and storing information in databases.

With `HMA`, users launch a sensitive app inside the context of a container app, without requiring the sensitive app to be installed. A container app is a dynamically generated wrapper around the Android application package (APK) of the sensitive app, and it is designed in such a way that the sensitive app cannot be fingerprinted yet still can support inter-process communication between the sensitive app and other installed apps. To launch the APK from the container app, `HMA` relies on techniques described in existing work: the dynamic loading of compiled source code and app resources from the APKs and user-level app-virtualization techniques, *e.g.,* [24, 25]. However, note that app virtualization alone is insufficient in providing robust protection against fingerprinting attacks, as many of the information leaks uncovered by our analysis are still possible when just app virtualization is used. Therefore, our main contribution is the design and evaluation of mechanisms built on top of app-virtualization in order to reduce the information leaks that could be exploited to fingerprint sensitive apps. `HMA` provides multiple tiers of protection: For baseline protection against current threats, `HMA` obfuscates static meta-data of sensitive apps (*e.g.,* their package names and components). To provide more advanced protection (*e.g.,* against side-channel attacks), HMA can add an additional layer of obfuscation for sensitive apps (*e.g.,* randomizing memory access). In some cases, app developers might need to be involved to make changes to the apps. Moreover, we are the first to identify the security and functional limitations of using app virtualization for the purpose of hiding apps.

Our evaluation of `HMA` on a diverse set of both free and paid mHealth apps on the Google Play Store shows that `HMA` is practical, and that it introduces reasonable operational delays to the users. For example, in 90% of the cases, the delay introduced by `HMA` to the cold start of an mHealth app by a non-optimized proof-of-concept implementation of `HMA` is less than one second. At runtime, the delay introduced is of only several milliseconds. Moreover, our user-study ($N = 30$) suggests that `HMA` is user-friendly and of interest to users.

Our main contributions in this work are as follows.

- *Systemized knowledge*: We are the first to investigate the techniques that an app can use to fingerprint another app.[8] Also, through our static and dynamic analysis on apps from the Google Play Store, we gain understanding about the prevalence of the problem of apps fingerprinting other installed apps.

- *Design and implementation of a solution for hiding sensitive apps*: We present `HMA`, a practical system that provides robust defense against fingerprinting attacks that target sensitive apps on Android. `HMA` works on stock Android, and no firmware modification or root privilege is required.

- *Thorough evaluation of `HMA`*: The evaluation of `HMA`'s prototype on apps from the Google Play Store suggests that `HMA` is practical. Also, our user study suggests that `HMA` is perceived as usable. `HMA`'s source code is available at `https://github.com/lca1/HideMyApp`.

2 Related Work

Researchers have actively investigated security and privacy problems in the Android platform. Existing works show that third-party libs often abuse their permissions to collect users' sensitive information [35, 48], and that apps have suspicious activities *e.g.,* collecting call logs, phone numbers, and browser bookmarks [29, 42]. Zhou *et al.* [55] show that

Android's open design has made publicly available a number of seemingly innocuous phone resources, including the list of installed apps; these resources could be used to infer sensitive information about their users, *e.g.,* users' gender and religion [40, 46]. Similarly, Chen *et al.* [27] show how to fingerprint Android apps based on their power consumption profiles. A significant research effort has been devoted to fingerprinting Android apps based on their (encrypted) network traffic patterns [28, 51, 54]. Researchers have also shown that re-identification attacks are possible using a small subset of installed apps [22, 33]. Demetriou *et al.* [29], in the same line as our work, used static analysis to quantify the prevalence of the collections of the list of installed apps and their metadata by third-party libs. We go beyond their work, however, by systematically investigating all possible information leaks that nosy apps can exploit to fingerprint other apps and by performing a dynamic analysis and privacy-policy analysis.

Existing mechanisms for preventing apps from learning about the presence of another app are not sufficient (Section 6). As we will show in Section 8, user-level virtualization techniques that enable an app (called *target app*) to be encapsulated within the context of another app (called *container app*) can be used as a building block for `HMA`. These techniques are used to sandbox untrusted target apps (*e.g.,* [24,25]) or to compartmentalize third-party libs from the host apps (*e.g.,* [34]). As they were designed for a different problem, however, they do not directly help hide the presence of a sensitive target app: They either require the target app to be first installed, thus exposing them to nosy apps through public APIs, or they run multiple target apps inside the same container app, thus violating the Android's app-isolation security model. They also do not provide any insight into the possible information leaks that can be exploited to fingerprint apps and how their techniques can be used for hiding the presence of apps.

3 Background on Android

Android Security Model. Android requires each app to have a unique package name defined by its developers and cannot be changed during its installation or execution. Upon installation, the Android OS automatically assigns a unique user ID (UID) to each app and creates a private directory where only this UID has read and write permissions. Additionally, each app is executed in its dedicated processes. Thus, apps are isolated, or sandboxed, both at the process and file levels.

Apps interact with the underlying system via methods defined by the Java API framework and the shell commands defined by the Linux-layer interface. Some API methods require users to grant apps certain permissions. Android defines three main protection levels for apps: normal, signature, and dangerous permissions.[9] Apps can have special permissions; users are required to grant these permissions to apps through the Settings app. Any app can execute shell commands; however, depending on its privilege, *i.e.,* default app privilege, debugging (adb)[10] or root, the outputs of the same shell commands are different.

Android Apps and APK Files. An Android app must contain a set of mandatory information: a unique package name, an icon, a label, a folder containing resources, and at least one of the following components: activity, service, broadcast receiver and content provider. An activity represents a screen, and a service performs long-running operations in the background. A broadcast receiver enables an app to subscribe and respond to specific system-wide events. A content provider manages the sharing of data between components in the same app or with other apps. Apps can optionally support other features such as implicit or explicit intents, permissions, and some customized app configurations. Apps are distributed in the form of APK files. An APK is a signed zip archive that contains the compiled code and resources of the app. Each APK also includes a manifest configuration file, called `AndroidManifest.xml`; this file contains a description of the app (*e.g.,* its package name and components).

4 Fingerprintability of Android Apps

Here, we demonstrate that an app, depending on its capabilities (its granted permissions and/or privileges), can retrieve information about other installed apps. This includes static information (*i.e.,* information available after apps are installed and that typically does not change during apps' lifetimes), and runtime information (*i.e.,* information generated or updated by apps at runtime). Our analysis was conducted on Android 8.0. Its findings are summarized in Table 1.

Without Permissions. An app can easily check if a specific app is installed on the phone. This can be done by invoking two methods `getInstalledApplications()` and `getInstalledPackages()` (hereafter abbreviated as `getIA()` and `getIP()`, respectively); they return the entire list of installed apps. An app can also register broadcast receivers (*e.g.,* `PACKAGE_INSTALLED`) to be notified when a new app is installed. It can also use various methods of the `PackageManager` class (*e.g.,* `getResourcesForApplication()`) as an oracle to check for the presence of a specific app. These methods take a package name as a parameter and return *null* if the package name does not exist on the phone.

If Android restricts access to package names of installed apps (*e.g.,* by requiring permissions), an app can still retrieve other static information about installed apps for fingerprinting attacks. This includes their mandatory information: the names of their components, their icons, labels, resources, developers' signatures and signing certificates. This also includes custom features used by installed apps: their permissions, apps configurations (themes, styles, and supported SDK). Such information can be obtained through a number of methods in

	Without Permissions	With Permissions	Default App Privilege	Debugging Privilege (adb)
Static Information	*Core attributes:* + Package name + Component's names + Resources + Icon, label + Developers' signatures *Customizations:* ⋆ Permissions ⋆ Themes ⋆ Phone configurations	(*) See note	+ Package names	+ Package names + APK path + APK file
Runtime Information	None	*Dangerous Permissions:* ⋆ Files in external storage ◇ Network traffic *Special Permissions:* ◇ Storage consumption + Running processes - Layouts and their content	◇ UI states† ◇ Power consumption† ◇ Memory footprints†	⋆ Files in external storage ⋆ System log ⋆ System diagnostic outputs + Running processes ◇ Network consumption - Screenshots

Table 1: Identifying information about installed apps that an app can learn, w.r.t. its permissions and privileges, through the Java API framework and the Linux-layer kernel. Analysis was conducted on Android 8.0. Superscript † means that the information can be learnt only in older versions of Android (*e.g.,* Android 8.0 requires the calling app to have adb privilege). *(*) Note:* Granting permissions to a zero-permission app does not enable it to obtain more static information about other installed apps. The notations $+, \star$ and $\diamond$ indicate the resources that our system (`HMA`, see Section 8) can protect by default, by collaborating with app developers or by randomizing runtime information of the container apps, respectively. Resources marked with the $-$ sign cannot be protected by `HMA`.

the `PackageManager` class, *e.g.,* `getPackageInfo()`. Note that this can be done even when apps are installed with the forward-lock option enabled (option `-l` in the `adb install` command). We tested this in Android 6.0; Android 8.0 threw an exception for this `-l` option. A nosy app cannot retrieve the list of intent filters declared by other apps. However, it can learn the names of the components of installed apps that can handle specific intent requests, by using methods such as `resolveActivity()`.

With Permissions. An app granted with the `READ_EXTERNAL_STORAGE` permission, a frequently requested dangerous-permission, can inspect for unique folders and files in a phone's external storage (a.k.a. SD card). Apps with VPN capabilities (permission `BIND_VPN_SERVICE`) can intercept network traffic of other apps; existing work shows that network traffic, even encrypted, can be used to fingerprint apps with good accuracy [50, 51, 54].

With special permissions, an app can obtain certain identifying information about other apps at their runtime. For instance, the `PACKAGE_USAGE_STATS` permission permits an app to obtain the list of running processes (method `getRunningAppProcesses()`), and statistics about network and storage consumption of all installed apps, including their package names, during a time interval (method `queryUsageStats()`). In addition, accessibility services[11] (with the `BIND_ACCESSIBILITY_SERVICE` permission) can have access to the layouts and the layouts' contents of other apps.

With Default App-Privilege. An app can retrieve the list of all package names on the phone. This can be done by obtaining the set of UIDs in the `/proc/uid_stat` folder and using the `getNameForUid()` API call to map a UID to a package name. An app can also infer the UI states (*e.g.,* knowing that another app is showing a login screen) [26], memory footprints (sequences of snapshots of the app's data resident size) [36] and power consumption [27] of other apps. Note that access to this information has been restricted in recent versions of Android (*e.g.,* Android 8.0 requires the app to have adb privilege).

With Debugging Privilege (adb). An app can retrieve the list of package names (command `pm list packages`) and learn the path to the APK file of a specific app (command `pm path [package name]`). Moreover, the adb privilege enables an app to retrieve the APK files of other apps (command `pull [APK path]`); the app can then use API methods such as `getPackageArchiveInfo()` to extract identifying information from the APK files. Also an app can learn about runtime behaviors of other apps by inspecting the system logs and diagnostic outputs (commands `logcat` and `dumpsys`). Moreover, with the adb privilege, apps can directly retrieve the list of running processes (command `ps`), take screenshots [38] or gain access to statistics about network usage of other apps (folder `/proc/uid_stat/[uid]`).

Our analysis shows that Android's open design exposes a significant amount of information that facilitates app-fingerprinting attacks. App developers themselves cannot obfuscate most of the aforementioned information for the purpose of hiding sensitive apps. For example, by design, the package name of an app is a global identifier in the Google Play Store. As a result, the obfuscation of apps' package

names has to be done per user, *i.e.,* for each user, the same app needs to be uploaded to the Google Play Store with a different package name. Similarly, the names of the app's components also need to be obfuscated per user, hence this approach is not practical. To mitigate app-fingerprinting attacks, Android could follow an approach similar to iOS, *i.e.,* to remove or restrict API methods and OS resources that leak identifying information of apps. However, such an approach would be difficult to implement in Android, as most of these methods and resources have valid use cases and are widely used by apps. For instance, methods `getIA()` and `getIP()`, are used by many popular apps with millions of users, *e.g.,* launcher, security/antivirus, and storage/memory manager apps. Removing or restricting such methods would break many apps and anger both developers and users. Such an approach would also negatively affect the competitive advantages of Android, *i.e.,* its customizability and rich set of features, over iOS. In addition, restricting API methods would not solve the problem completely, as more subtle fingerprinting attacks would still be possible. For example, in iOS, the `canOpenURL()` method can be used to check if a particular app is on the phone. Since iOS 9.0, in order to have an arbitrarily high number of calls to this method, an app has to declare beforehand the set of apps that it wants to check. Otherwise, it can only call this method at most 50 times.[12] This restriction reduces the risks of fingerprinting attacks, but negatively affects both developers and users, *e.g.,* apps need to be updated frequently to update the list of apps. More importantly, even with 50 queries, a nosy app can still check if a specific app or small set of apps are installed on the phone.

A possibly better approach is for Android to include a new "sensitive" flag that enables users to hide sensitive apps from other apps in the same phone, *i.e.,* other apps will not be able to use Android API methods to infer the existence of apps flagged as sensitive. Moreover, Android can include a new permission that users can grant to certain apps in order to enable these apps to detect apps flagged as sensitive. This approach, however, requires significant modifications and testing of Android's APIs. Therefore, our goal is to design a solution that does not require changes to Android or sensitive apps and that can be available to users immediately.

5 Apps Inquiring about Other Apps

We analyze apps from the Google Play Store to estimate how common it is for apps to inquiry about other installed apps. Our analysis focuses on API calls that directly retrieve the list of installed apps (hereafter called `LIA`): `getIA()` and `getIP()`, because these two methods clearly show the intent of developers to learn about other apps, whereas the other methods presented in Section 4 can be used in valid use cases. Therefore, the results presented in this section is a *lower-bound* on the number of apps that fingerprint other apps.

5.1 Data Collection

We gathered the following datasets for our analysis.

APK Dataset. We collected APK files of popular free apps in the Google Play Store (US site). For each app category in the store (55 total), we gathered the 60 most popular apps. After eliminating duplicate entries, default Android apps, and brand-specific apps, we were left with 2917 apps.

Privacy-Policy Dataset. We collected privacy policies that corresponded to the apps in our dataset. Out of 2917 apps, we gathered 2499 privacy policies by following the links included in the apps' Google Play Store pages.

5.2 Static Analysis

For our static analysis, by using Apktool,[13] we decompiled the APKs to obtain their smali code, a human-readable representation of the app's bytecode. We searched in the smali code for occurrences of two methods `getIA()` and `getIP()`.[14] API calls can be located in three parts of the decompiled code: in the code of Android/Google libs and SDKs, in the code of third-party libs and SDKs, or in the code of the app itself. To differentiate among these three origins, we applied the following heuristic. First, methods found in paths containing the "com/google", "com/Android" or "Android/support" substrings, are considered part of Android/Google libs and SDKs. Second, methods found in paths containing the name of the app are considered part of the code of the app. We believe this is a reasonable heuristics, because package names of Android apps follow the Java package-name conventions with the reversed internet domain of the companies, generally two words long. If the methods do not match the first two categories, then they are considered part of the code of a third-party lib or SDK. Note that this approach, also used in previous work [29], cannot precisely classify obfuscated code or code in paths with no meaningful names. Such cases, however, represent only a small fraction in our analysis (less than 5%).

Table 2 shows the proportions of apps that invoke `getIA()` and `getIP()` w.r.t. different call origins. Of the 2917 apps evaluated, 1663 apps (57.0%) include at least one invocation of these two methods in the code from third-party libs and the apps. These results show a significant increase in comparison with the results presented in 2016 by Demetriou *et al.* [29]. These results also show that most sensitive requests come from third-party libs or SDKs; app developers might not be aware of this activity, as has been the case for other sensitive data such as location.[15]

Static analysis has two main limitations. First, methods appearing in the code might never be executed by the app. Second, it is possible that the sensitive methods do not appear in the code included in the APK, rather in the code loaded dynamically at runtime. To address these issues, we also performed a dynamic analysis of the apps in our dataset.

Analysis method	Call origin	`getIA()` (%)	`getIP()` (%)	`getIA()` or `getIP()` (%)
Static	Third-party libs + Apps	36.4	43.6	57.0
Static	Apps only	8.1	8.4	13.9
Dynamic	Third-party libs + Apps	6.5	15.0	19.2

Table 2: Proportion of free apps that invoke `getIA()` and `getIP()`, to collect `LIAs` w.r.t. different call origins.

5.3 Dynamic Analysis

For our dynamic analysis, by using XPrivacy[16] on a phone with Android 6.0, we intercepted the API calls from apps. For the analysis to scale, for each app, we installed it and granted it all the permissions requested. Next, we launched all the runnable activities declared by the app for 10 minutes. Although this approach has limitations, as it only has a short period of time per app and it cannot emulate all the activities a user could do, it is sufficient to estimate a *lower-bound* on the number of apps that query for `LIAs` at runtime, as shown in our results.

Our results, shown in Table 2, show that 190 apps (6.54%) called `getIA()`, 436 apps (15.0%) called `getIP()`, and 19.2% of the apps called at least one of these two methods. Because XPrivacy does not provide information about the origin of the request, we performed some additional steps. For each app, we used the results of our static analysis and searched for occurrences of `getIA()` and `getIP()` in the code belonging to Google/Android libs. We found that most apps did not include calls to these sensitive methods in the code belonging to Google/Android libs: 181 out of 190 for `getIA()` and 412 out of 436 for `getIP()`. Hence, we conclude that these sensitive requests came mainly from third-party libs or from the code of the apps.

Interestingly, we found 49 apps that called at least one of the two sensitive methods in our dynamic analysis, but not in our static analysis. This could be because the decompiler tool produced incorrect smali code, or because these requests were dynamically loaded at runtime. Still, this represents only a small number of the apps found through our analysis.

Our static and dynamic analysis shows that a significant number of free apps actively queries for `LIAs`: between 19.2% (dynamic analysis) and 57% (static analysis) of the tested apps.[17] This shows that many third parties are interested in knowing about the installed apps on users' phones, and that, if Android blocked `getIA()` and `getIP()`, they would likely attempt to use other methods (see Section 4).

5.4 Analysis of Privacy Policies

Google's privacy-policy guidelines require apps that handle personal or sensitive user data to comprehensively disclose how they collect, use and share the collected data. An example of a common violation, shown in these guidelines, is "*An app that doesn't treat a user's inventory of installed apps as personal or sensitive user data*".[18] Next, we explain what developers understand about the guidelines.

As mentioned in Section 5.1, out of 2917 apps in our dataset, we found 2499 privacy policies. From the 1674 nosy apps found in the static and dynamic analysis, 1524 apps have privacy policies. We semi-automated the policy analysis as follows. We built a set of keywords consisting of nouns and verbs that might be used to construct a sentence to express the intention of collecting `LIAs`: retrieve, collect, fetch, acquire, gather, package, ID, installed, app, name, application, software, and list. For each privacy policy, we extracted the sentences that contain at least one of the keywords. From the extracted sentences, we manually searched for specific expressions such as "installed app", "app ID" and "installed software". Thereafter, we read the matched sentences and the corresponding privacy policy.

From the set of 2499 policies, we found 162 policies that explicitly mention the collection of `LIAs`. Among these, 129 belong to the set of 1674 nosy apps (7.7%). Some apps have exactly the same privacy policies, even though they are from different companies (*e.g.,* [20] and [6]). 33 apps mentioned the collection of `LIAs`, but we did not find these apps in both static and dynamic analyses. For these apps, we performed a more thorough dynamic analysis: we used them as a normal user would, while intercepting API calls. We did not capture, however, any calls to the two sensitive methods. This might be because developers copy the privacy policies from other apps, or because the apps will make these calls in the future.

Besides the generic declared purposes of the collections of `LIAs` by apps, *e.g.,* for improving the service (*e.g.,* [14, 21]), some apps explicitly state that they collect `LIAs` for targeted ads (*e.g.,* [3, 12]), and targeted ads by third-party ad networks (*e.g.,* [15]). Unexpectedly, we found that of the 162 policies that mention the collections of `LIAs`, 76 categorize `LIAs` as *non-personal*, whereas Google defines this as personal information. This shows a misunderstanding between developers and Google's guidelines.

6 Existing Protection Mechanisms

To the best of our knowledge, there are no existing robust mechanisms for hiding sensitive apps. Below, we present some mechanisms that can offer partial protection.

6.1 Mechanisms by Google

Android does not provide users with a mechanism to hide the existence of apps from other apps. But users can repurpose existing Android mechanisms for partially hiding apps.

Multiple Users. Android supports multiple users on a single phone by separating user accounts and app data.[19] This feature could be used to prevent fingerprinting of sensitive apps by installing sensitive apps in one or more secondary accounts, thus isolating sensitive apps from nosy apps. However, a key disadvantage of using multiple users for this purpose is that it prevents inter-app communications (e.g., intent-based interactions) among apps in different user accounts. As a result, sensitive apps' functionalities can be significantly reduced because they cannot delegate tasks to other apps. For instance, a sensitive app will not have access to a user's calendar or contacts (unless the user replicates them on each account) or access to other apps for certain tasks, e.g., sending a message or picture via Whatsapp or Facebook, accessing files in Dropbox, sending an e-mail or SMS, and authenticating users with Google or Facebook accounts. In section 10.5, we show that popular mHealth apps use inter-app communications not only for delegating tasks but also for sharing their resources with other apps. Therefore, a solution that hides sensitive apps and that still supports inter-app communications is more desirable.

Multiple user accounts could also introduce new security and privacy issues [45]. Using multiple users will significantly affect the user experience, as users will have to switch back and forth among accounts to access different types of apps and data, introducing significant delays and confusion. While the primary account is in the foreground, apps on secondary accounts are put in the background and they cannot use Bluetooth services (important for mHealth apps). Another important problem is that some popular phone manufacturers (*e.g.,* Samsung, LG, Huawei, Asus) disable multiple users in some of their devices,[20] thus affecting the availability of this solution to many users.

We have also found experimentally that the implementation of multiple users in the latest (Android 9) and earlier versions of Android does not effectively prevent nosy apps from learning what other apps are installed in different user accounts. To bypass this protection, a nosy app could do any of the following:

- On Android 7 or earlier, including an additional parameter flag (`MATCH_UNINSTALLED_PACKAGES`) in methods `getIA()` and `getIP()` will reveal the apps installed in secondary user accounts.

- On Android 9 or earlier, a nosy app can use multiple `PackageManager` methods, such as `getPackageUid()`, `getPackageGidS()`, `checkPermission()`, `checkSignatures()`, or `getApplicationEnabledSetting()`, as oracles to check if an app is installed on a secondary account or on a work profile. The nosy app only needs to include the package name of the targeted sensitive app as a parameter to these methods. Android's source code shows that these methods check the user ID of the app calling the method to show only information of apps in the same user profile, but our experimental evaluation shows that currently deployed versions of Android do not enforce such checks. This approach was tested on Android 9.

- A nosy app can guess the UIDs of the apps installed on all the accounts and work profiles, by looking at the `/proc/uid` directory to learn the ranges of current UIDs in the system. It then guesses the UIDs of other apps and uses the `getNameForUid()` method to learn the package name. This method will return a package name given a UID as a input parameter; if the app does not exist, it returns null. As a result, it can be used as an oracle to retrieve the list of installed apps on the device. This was tested on Android 6, 8.1 and 9.

- A nosy app with adb privilege can easily verify if a sensitive app is running on the device, independently of the account or profile it was installed on, by using the shell command: *pidof <PackageName>*. This approach was tested on Android 9.

- A nosy app with adb privilege can obtain the list of installed apps, which includes apps on secondary accounts and work profiles, by using the shell command *dumpsys*. This approach was tested on Android 9.

Android for Work. Android supports an enterprise solution called Android for Work; this solution separates work apps from personal apps.[21] Our tests, using similar methods as with multiple users, also confirmed that, as with multiple users accounts, it is easy to identify which apps are in the work profile. In addition, Android for Work is only available to enterprise users.

Recently, Android introduced a new feature called *Instant Apps*;[22] this feature enables users to run apps instantly without installing them. Such an approach could be used to hide sensitive apps, however, it only supports a limited subset of permissions, and it does not support features that are crucial for mHealth apps such as storing users' data or connecting to Bluetooth-enabled devices.[23]

Google classifies the list of installed apps as *personal* information hence requires apps that collect this information to include in their privacy policies the purpose of their collection. Apps that do not follow this requirement are classified as Potentially Harmful Apps (PHAs) or Mobile Unwanted Softwares (MuWS) [1, 2]. Android security services, *e.g.,* Google Play Protect [10], periodically scan users' phones and warn users if apps behave as PHAs or MuWS. Such mechanisms, however, do not seem to effectively protect against the unauthorized collection of the list of installed apps. Our analyses show that only 7.7% of the apps declare their collections of such information in their privacy policies, and some claim that a list of installed app is non-personal information (Section 5).

Furthermore, these mechanisms might fail to detect targeted attacks, *e.g.,* a nosy app might want to check if a small subset of sensitive apps exists on the phone.

6.2 Mechanisms by Third Parties

Samsung Knox[24] relies on secure hardware to offer isolation between personal and work-related apps, similar to Android for Work. Unfortunately, we were not able to evaluate the robustness of the protection offered by Knox w.r.t. hiding apps, because Samsung discontinued its support for work and personal spaces for private users; only enterprise users can use such a feature. Nevertheless, this solution is device specific and only hides apps from other apps in a different isolated environment, but not from apps in the same environment (apps in the same isolated environment can come from different, untrusted sources). That is, a solution that provides per-app isolation is preferable.

There are apps on the Google Play Store that help users to hide the icons of their sensitive apps from the Android app launcher (*e.g.,* [16]). Even though they help hide the presence of the sensitive apps from other human users (*e.g.,* nosy partners), these sensitive apps are still visible to other apps. Along the line of user-level virtualization techniques, on the Google Play Store, we found apps that use these techniques to enable users to run in parallel multiple instances of an app on their phones and to partially hide the app, (*e.g.,* [11, 17, 18]). However, these solutions require the hidden app to be installed first on the phone before protecting it, thus triggering installation and uninstallation broadcast events that can be detected by a nosy app. These apps provide only a single isolated space, *i.e.,* they do not protect apps from other apps in the same environment. Our preliminary evaluation of these apps also shows that their protection is limited, *e.g.,* the names of the hidden apps can be found in the list of running processes.

7 Our Solution: `HideMyApp`

We propose `HideMyApp` (`HMA`), a system for hiding the presence of sensitive apps w.r.t. to a nosy app on the same phone. In this section, we will present our system model, adversarial model, design goals and a high-level overview of the solution.

7.1 System Model

The scenario envisioned for `HMA` is as follows. A hospital or a hospital consortium (hereafter called hospitals) sets up an app store, called `HMA App Store`, where app developers working for the hospitals publish their mHealth apps. Hospitals want their patients to use their mHealth apps without disclosing their use to other apps on the same phone. Note that such organizations and their own app stores already exist, *e.g.,* the VA App Store set up by the U.S. Department of Veterans Affairs.

To enable the users to manage the apps provided by the `HMA App Store`, the `HMA App Store` provides the users with a client app called `HMA Manager`. This app can be distributed through any available app stores, *e.g.,* the Google Play Store. To allow the `HMA Manager` app to install apps downloaded from the `HMA App Store`, similarly to other Google Play Store alternatives *e.g.,* Amazon[25] and F-Droid [9], users need to enable the “allow apps from unknown sources" setting on their phones. Since Android 8.0, Google made this option more fine-grained by turning it into the “Install unknown apps" permission [19]. That is, users only need to grant this permission to the `HMA Manager` app to enable it to install apps from the `HMA App Store`.

7.2 Adversarial Model

We assume the Android OS on the user's phone to be trusted and secure, including its Linux kernel and its Java API framework. We assume that the `HMA App Store` and the `HMA Manager` app are trusted and secure, and that they follow the prescribed protocols of the system. We discuss mechanisms to relax the trust assumptions on the `HMA App Store` and `HMA Manager` app in Section 9.2.

We assume there is a nosy app that wants to learn if a specific app is present on the phone. The nosy app has the default app-privilege, and it is granted all dangerous permissions by its user – these are the typical capabilities of apps that users often install on their phones. In Section 9, we discuss mechanisms for preventing more advanced fingerprinting attacks by malicious apps; a malicious app has more capabilities than a nosy app, *i.e.,* it can have special permissions (*e.g.,* `PACKAGE_USAGE_STATS` or `BIND_ACCESSIBILITY_SERVICE`) and the debugging privilege (adb), thus it can perform more advanced attacks, such as fingerprinting apps using their runtime information.

We assume that apps belonging to hospitals are nosy, *i.e.,* these apps are also curious about what other apps are installed on the user's device.

7.3 Design Goals

The purpose of `HMA` is to effectively hide the presence of sensitive apps, yet preserve their usability and functionality.

- *(G1) Privacy protection.* It should be difficult for a nosy app to identify sensitive apps on the same phone.

- *(G2) No firmware modifications.* The solution should run on stock Android phones. That is, it should not require the phones to run customized versions of Android firmware, *e.g.,* extensions to Android's middleware or the Linux kernel. This also means that the solution should not require the phones to be rooted.

- *(G3) Preserving the app-isolation security model of Android.* Each app should have its own private directory and run in its own dedicated process.
- *(G4) Few app modifications.* For baseline protection against nosy apps, the solution should not require app developers to change their apps. For protection against malicious apps, apps might need to be changed or some features might not be supported.
- *(G5) Usability.* The solution should preserve the usability and the key functionalities of sensitive apps.

7.4 HMA Overview

From a high-level point of view, HMA achieves its aforementioned design goals by enabling its users to install a container app for each sensitive app (as illustrated in Fig. 1). Each container app has a generic package name and obfuscated app components. As a result, nosy apps cannot fingerprint a sensitive app by using the information about its container app. At runtime, the container app will launch the APK file of the sensitive app within its context by relying on user-level virtualization techniques. That is, the sensitive app is not registered in the OS.

To do so, HMA requires the hospitals to bootstrap the system by setting up the HMA App Store and distributing the HMA Manager app to users (Section 8.1 and 8.2). Through the HMA Manager app, users can (un)install, open, and update sensitive apps without being discovered by the OS and other apps. We detail these operations in Section 8.3.

8 HMA System Description

Here, we detail the components and operations of HMA.

8.1 HMA Manager App

Recall, to hide their presence, sensitive apps are not registered in the OS; instead, their container apps are registered. Thus, if users open their default Android app launcher, they will only see container apps with generic icons and random names. To solve this usability issue, at installation time, the HMA Manager app keeps track of the one-to-one mappings between sensitive apps and their container apps. Using the mappings, the HMA Manager app can display the container apps to the users with the original icons and labels of their sensitive apps. To provide unlinkability between users and their sensitive apps w.r.t. the HMA App Store, the HMA Manager app *never* sends any identifying information of the users to the HMA App Store, and all the communications between the HMA App Store and the HMA Manager are anonymous. This is a reasonable assumption because the HMA Manager app can be open-sourced and audited by third parties. Also, in most cases, users do not have fixed public IP addresses; they access the Internet via a NAT gateway offered by cellular providers. If needed, a VPN proxy or Tor could be used to hide network identifiers.

8.2 HMA App Store

The HMA App Store receives app-installation and app-update requests from HMA Manager apps and returns container apps to them. To reduce the delays introduced to the app-installation and app-update requests, the HMA App Store defines a set of P generic package names for container apps, *e.g.,* app-1, ..., app-P. This set of generic names is shared by all sensitive apps, thus there is no one-to-one mapping between a sensitive app and a generic name or a subset of generic names.[26] For each sensitive app, the HMA App Store can generate beforehand P container apps corresponding to P predefined generic package names and store them in its database. Below, we explain the procedure followed by the HMA App Store to create a container app. Details about the app-installation and update requests from the HMA Manager apps are explained in Section 8.3.

HMA Container-App Generation. To generate a container app for a sensitive APK, the HMA App Store performs the following steps. Note that this operation cannot be performed by the HMA Manager app, because Android does not provide tools for apps to decompile and compile other apps.

- The HMA App Store creates an empty app with a generic app icon, a random package name and label, and it imports into the app the lib and the code for the user-level virtualization, *i.e.,* to launch the APK from the container app. Note that the lib and the code are independent from the APK.
- The HMA App Store extracts the permissions declared by the sensitive app and declares them in the manifest file of the container app.
- To enable the container app to launch the sensitive APK, app components (activities, services, broadcast receivers, and content providers) declared by the sensitive app need to be declared in the manifest file of the container app. This information, however, can be retrieved by nosy apps to fingerprint sensitive apps (Section 4). To mitigate this problem, the container app declares activities, services and broadcast receivers of the sensitive app with random names. At runtime, the container app will map these random names to the real names. The intent filters declared in the components of sensitive apps are also declared in the manifest file of their sensitive apps. In Section 9, we will discuss the case of content providers.
- The HMA App Store compiles the container app to obtain its APK and signs it.

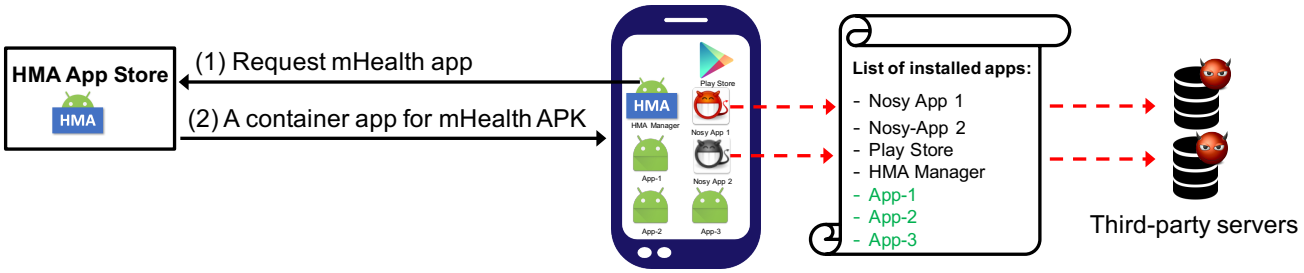


Figure 1: Overview of the HMA architecture. Nosy apps only learn the generic names of the container apps.

Note that for the sake of simplicity, here we only present a solution that protects mandatory features of Android apps. A malicious app might try to fingerprint sensitive apps based on, for instance, the runtime information produced by their container apps. We discuss this in Section 9.

HMA User-Level Virtualization. To launch the APK of a sensitive app without installing it, its container app generates a randomly named child-process in which the APK will run, *i.e.,* the APK is executed under the same UID as its container app. Thereafter, the container app loads the APK dynamically at runtime, and it intercepts and proxies the interactions between the sensitive app and the underlying system (the OS and the app framework). To do so, we rely on an open-source lib for app-virtualization called DroidPlugin [8].

8.3 HMA Operations

In this section, we detail the procedure followed by a user when she (un)installs, updates, or uses sensitive apps.

App Installation. To install a sensitive app, the user opens her `HMA Manager` app to retrieve the set of apps provided by the `HMA App Store`. Once she selects a sensitive app, the `HMA Manager` app sends to the `HMA App Store` an installation request consisting of the name of the sensitive app and her desired generic package name for the container app. The `HMA App Store` correspondingly finds in its database or creates a container app, and it sends the container app, together with the original label and icon of the sensitive app, to the `HMA Manager`. The `HMA Manager` prompts the user for her confirmation about the installation. Once the user accepts, the installation occurs as in standard app installation on Android. Also, the `HMA Manager` saves, in its private directory, a record of the package name of the container app and the package name, the original icon and the label of the sensitive app.

App Launch. To launch a sensitive app, the user opens her `HMA Manager` app to be shown with the set of container apps installed on her phone. Using the information stored in its database about the mappings between container apps and sensitive apps (Section 8.3), the `HMA Manager` displays to the user the container apps with the original labels and icons of the corresponding sensitive apps. Therefore, the user can easily identify and select her sensitive apps.

The *first time* a container app runs, it needs to obtain the sensitive APK from the `HMA App Store`; then it stores the APK in its private directory. This incurs some delays to the first launch of the sensitive app. However, it is needed to prevent the sensitive app from being fingerprinted: If the sensitive APK was included in the resources or assets folders of its container app so that the container app could copy and store the APK in its private directory at installation time, a nosy app would be able to obtain the sensitive APK. Recall, any app can obtain the resources and assets of other apps (Section 4). Also, Android does not permit apps to automatically start their background services upon installation.

At runtime, the container app dynamically loads the sensitive APK. Thereafter, it intercepts and proxies API calls and system calls between the sensitive app and the underlying system. If the version of the Android OS is at least 6.0, permissions requested by the sensitive app will be prompted by its container app at runtime. Thus, they will be shown with the generic package name of the container app. This, however, does not affect the comprehensibility of the permission requests, as shown by our user study (Section 10.6). Additionally, when an app sends an implicit intent with an `action` supported by the sensitive app, the operating system will show the sensitive app as an option for the user to choose to handle the requested action. This introduces a usability problem: the icon of the sensitive app presented to the user is a generic icon. This, however, can be solved by using the *direct share targets* feature in Android – a feature that enable apps to show finer-grained internal content in the *chooser dialog window*.[27]

App Update. When a sensitive app on the `HMA App Store` has an update, for each predefined generic container-app package name, the `HMA App Store` generates a corresponding container app for the updated sensitive app. This step is needed, because the configuration file of the container app needs to be updated w.r.t. the updates introduced by the sensitive app. The `HMA App Store` then sends a push notification to all `HMA Manager` clients to notify them about the update. If a user has the sensitive app on her phone, her `HMA Manager` sends the package name of its existing container app to the `HMA App Store`. In return, it receives the corresponding updated container app from the `HMA App Store`. It then prompts the user to confirm the installation. Once the user accepts, the updated container app is installed, similarly

to the standard app-update procedure on Android.

App Uninstallation. To uninstall a sensitive app, the user opens her HMA Manager app to be shown with the set of her container apps. Once she selects the container app, the HMA Manager prompts her to confirm the uninstallation. Thereafter, the uninstallation occurs similarly to the standard app-uninstallation procedure on Android.

9 Privacy and Security Analysis

Here, we present an analysis of HMA to show that it effectively achieves its privacy and security goals w.r.t. different capabilities of the nosy apps (*i.e.,* their granted permissions and privileges) as shown in Table 1.

9.1 Privacy

Nosy Apps without Permissions. HMA effectively protects, by default, the core attributes of sensitive apps. First, a nosy app cannot obtain the package name of a sensitive app, because the sensitive app is never registered on the system; instead, its container app with a generic package name is installed. Second, the resources, shared libraries, developers' signatures and developers' signing certificates of the sensitive app cannot be learnt by the nosy app, because they are not declared or included in the container-app's APK; instead they are dynamically loaded from the sensitive APK at runtime. Third, the nosy app cannot learn the components' names of the sensitive app, because these names are randomized. To prevent fingerprinting attacks based on the number of components declared in the container app, the HMA App Store adds dummy random components during the generation of the container app such that all the container apps declare the same number of components.

A nosy app might try to fingerprint sensitive apps by using the sets of permissions declared by their container apps. This can be mitigated if all container apps declare a union of permissions requested by sensitive apps in the HMA App Store. Note that for devices with Android 6 or later, the container app requests at runtime only the permissions needed by its sensitive app, and users can grant or decline these requests. This makes it difficult for nosy apps to fingerprint a sensitive app using the set of permissions granted to its container app.

HMA needs collaboration from app developers to prevent fingerprinting attacks based on the customized configurations of some sensitive apps, *e.g.,* themes and screen settings. The HMA App Store can define a guideline for app developers to follow such that all apps have the same configurations. This will affect the look and feel of the sensitive apps, but it is a trade-off between usability and privacy. Note that the same approach has been used in other deployed systems, *e.g.,* in the Tor browser where all the versions have the same default window size and user-agent strings.[28] To facilitate guideline compliance, the HMA App Store can also provide developers with IDE plugins to help them write guideline-compliant code; such an approach has been proposed in existing work (*e.g.,* [43] and [31]).

App developers might want to use custom features, such as custom permissions, custom actions for the intent filters of their apps' components. These features, however, can be used to fingerprint their sensitive apps, hence should not be used by app developers. An app might want to support a content provider for sharing data between its components or for sharing data with other apps. HMA can support the former case; the container apps do not need to declare the content provider in its manifest file, but it handles the requests from the components of the sensitive apps internally. HMA, however, cannot support the case of sensitive apps using content providers to share data with other apps. This is because in order to do so, the container apps need to declare the URIs of their content providers in their manifest files, and these URIs can uniquely identify apps. These are limitations of HMA, however, from our analysis, only a small number of apps is affected by these limitations (Section 10.5).

Nosy Apps with Permissions. A nosy app can fingerprint sensitive apps based on their use of the external storage (SD card), *e.g.,* unique directories and files. To prevent this, container apps can intercept and translate calls from sensitive apps associated with the creation or access of files in external storage. However, note that apps are not recommended to store data there, especially mHealth apps. To prevent an app with VPN capabilities from fingerprinting sensitive apps based on the IP addresses in the header of the IP packages, the sensitive apps can relay their traffic through the HMA App Store servers; this protection is provided at the cost of additional communication delays for the apps and it requires collaboration with app developers.

A malicious app cannot fingerprint a sensitive app by using the list of running processes, because the sensitive app runs inside the child process of its container app with a random name. To prevent malicious apps from abusing its special permissions to fingerprint sensitive apps using their runtime statistics, *e.g.,* resources consumed by their container apps, the container apps can randomly generate dummy data to obfuscate the usage statistics of sensitive apps. Note that this does not require changes to the sensitive apps. In future work, we will evaluate techniques against these side-channel attacks such as [52] and [26]. HMA cannot prevent malicious apps, with permission to accessibility services, from fingerprinting sensitive apps. Accessibility services enable access to apps' unique layout information, and it is not practical to require all sensitive apps to use a generic layout. However, Google currently bans the use of accessibility services for purposes not related to helping users with disabilities.[29] Users should grant this permission only to apps they trust.

Nosy Apps with Default App Privileges. Recall, HMA, by default, hides the package name of the sensitive apps. To pre-

vent nosy apps from fingerprinting sensitive apps by using their UI states, the container apps can also obfuscate the UI states by overlaying transparent frames on the real screens of the sensitive apps. Similarly to the case of other runtime statistics discussed above, the container apps can also randomly generate dummy data to obfuscate the memory footprints and power consumptions of the sensitive apps.

Malicious Apps with the Debugging Privilege (adb). Recall, HMA protects the package name and the process names of the sensitive apps by default. Also recall, the container apps can randomize runtime statistics of the container apps. In addition, the paths to the APK files of the container apps do not reveal any information about the sensitive apps. Also, the malicious app cannot retrieve the APK files of the sensitive apps, because the APKs are stored inside the private directories of their container apps.

To prevent advanced attacks by malicious apps, *e.g.,* fingerprinting sensitive apps by reading the log of the phone, HMA requires collaboration from app developers. Developers should not write identifying information about their sensitive apps to the log. Apps with adb privilege can take screenshots of the phone and infer apps' names from the screenshots. HMA cannot prevent this attack. However, note that this attack requires the malicious app to do extra and error-prone operations (*e.g.,* image processing) to identify sensitive apps.

9.2 Security

By using user-level virtualization techniques to launch an APK, HMA does not require users to modify the OS of the phone. The Android's app-isolation security model is also preserved, because each APK runs inside the context of its container app. Thus, it is executed in a process under the same UID as its container app, and it uses the private data directory of its container app. Similarly to other third-party stores (*e.g.,* Amazon or F-Droid), HMA requires users to enable the "allow apps from unknown sources" setting on their phones. However, apps installed from these sources are still scanned and checked by Android security services for malware [10]. Also, recently, this setting was converted to a per-app permission [19]. As a result, granting the HMA Manager app the permission to install apps from unknown sources will not give other apps on the phone the same permission.

As on the Google Play Store, with HMA, app developers register their public keys on the HMA App Store, and sign their apps before they submit to the HMA App Store. Moreover, the HMA App Store signs the container apps that it generates to vouch for the integrity of the container apps and the sensitive apps. This mechanism, however, introduces a security issue for sensitive apps: Apps from different developers are signed by the same private key of the HMA App Store, hence a dishonest app developer might exploit this same-signature property to access signature-protected components of other apps.[30] Note that requesting or declaring signature-protection permissions will facilitate fingerprinting of sensitive apps, hence HMA does not support this feature. As a result, this attack is not possible in HMA. Also note that few apps use signature-protected permissions (see Section 10.5). In future work, we will explore mechanisms for enabling container apps to verify the signatures of sensitive apps at runtime, in order to prevent unauthorized access to signature-protected components of their sensitive apps.

HMA container apps prompt users only for permissions requested by sensitive apps. To relax the trust assumptions on the HMA App Store and HMA Manager, the HMA App Store can provide an API so that anyone can implement her own HMA Manager app, or the HMA Manager app can be open-source, *i.e.,* anyone can audit the app and check if it follows the protocols as prescribed. Therefore, assuming that the metadata of the network and the lower communication layers cannot be used to identify users, *e.g.,* by using a proxy or Tor, the HMA App Store cannot link a set of sensitive apps to a user.

10 Evaluation

To evaluate HMA, we used a real dataset of free and paid mHealth apps on the Google Play Store. We looked into three evaluation criteria: (1) overhead experienced by mHealth apps, (2) HMA runtime robustness and its compatibility with mHealth apps, and (3) HMA usability.

10.1 Dataset

We selected 50 apps from the medical category on the Google Play Store, of which 42 apps are free and 8 apps are not. To have a significant and diverse dataset, we selected apps based on their popularity (more than 1000 downloads), their medical specialization, and their supported functionality. From the 50 apps, we filtered out apps that make calls to APIs that we did not support in our prototype implementations, including Google Mobile Services (GMS), Google Cloud Messaging (GCM) and Google Play Services APIs. Note that these services could be supported, similarly to other services, at the cost of additional engineering efforts. We also filtered out apps that use Facebook SDKs, because such SDKs often use custom layouts that are not yet supported by the user-level virtualization lib that HMA uses. Exploring the interaction mechanisms between custom layouts with the Android framework is an avenue for future work.

After filtering, we obtained a set of 30 apps (24 free apps and 6 paid apps, see Appendix B of our technical report at [44]) for 15 medical conditions. Also, these apps support features that are crucial for mHealth apps, *e.g.,* a Bluetooth connection with external medical devices (*e.g.,* Beurer HealthManager app [4]) and an internet connection (*e.g.,* Cancer.Net app [5]).

10.2 Implementation Details

Our prototype features the main components of HMA, including the HMA App Store and the HMA Manager app. To measure the operational delay introduced by HMA, we implemented a proof-of-concept HMA App Store on a computer (Intel Core i7, 3GHz, 16 GB RAM) with MacOS Sierra. Our HMA App Store dynamically generated container apps from APKs and relied on an open-source lib called DroidPlugin [8] for user-level virtualization. Our prototype container apps dynamically loaded the apps' classes and resources from the mHealth APKs and supported the interception and proxy of API calls commonly used by mHealth apps, *e.g.,* APIs related to Bluetooth connections and SQLite databases.

10.3 Performance Overhead

In this section, we present the delays introduced by HMA to sensitive apps during app-installation and app-launch operations.[31] For the evaluation of delays added by the user-level virtualization to commonly used API methods and system calls at runtime, we refer the readers to existing work, *e.g.,* Boxify [24] shows that such overhead is negligible (opening a camera introduces an overhead of 1.24 ms).

Results presented in this section were measured on a Google Nexus 5X phone running Android 7.0. In our experiments, the HMA App Store was connected to the phone through a micro-USB cable, hence network delays were not considered. Yet, compared to the standard use of apps, HMA incurs negligible network-delay overheads, because the only bandwidth overhead introduced by HMA is the container-app payload whose size is only several hundreds of kilobytes.

10.3.1 App Installation

When a user wants to install an mHealth app, the HMA App Store first creates a container app for it. Based on our experiments, assuming the HMA App Store decompiles the mHealth APKs beforehand, for 90% of the cases, generating a container app takes, on average, 5 s. Note that a large part of the delay comes from the compilation of the container app, and the measurement was performed on a laptop computer. Also note that the HMA App Store can always prepare in advance container apps for each mHealth app, as presented in Section 8.2. The size of the container app is only several hundreds of kilobytes, which takes less than a second for the HMA Manager app to download using a 3G or 4G Internet connection. As a result, the total delay overhead introduced by HMA would be less than 5 s in the *worst-case* scenario, and less than a second if container apps are generated beforehand, which is acceptable.

10.3.2 App Launch

On Android, apps can be launched from two different states: *cold starts* where apps are launched for the first time since the phone was booted or since the system killed the apps, and *warm starts* where the apps' activities might still reside in memory, and the system only needs to bring them to the foreground, hence faster than cold starts.

Experiment Set-Up. For cold-start delays, we rely on Android's official launch-performance profiling method [13]. For each app, we installed its container app, copied its APK file to its container app's private directory, and launched the container app through adb. We then extracted the time information from the Displayed entry of the logcat output. To simulate a first launch, before we launched an app, we used the command adb shell pm clear [package-name] to bring the app back to its initial state. To simulate a cold start, before we launched an app, we used the command adb shell am force-stop [package-name] to kill all the foreground activities and background processes of the app. For each app, we collected 50 measurements per launch setting. For a baseline, we measured the delays when the mHealth apps were executed without HMA.

To measure warm-start delays, due to the lack of Android supports for profiling warm starts, we have to instrument the source code of the sensitive apps to log the time that the app enters different stages in its lifecycle. Because apps in our dataset are closed source, we used an open-source app.[32] To simulate a warm start, we used the command input keyevent 187 to bring the app to the background, and then we used the monkey command to bring the app back to the foreground. By subtracting the time when the onResume() method is successfully executed with the time before the monkey command is sent, we know the warm-start delay experienced by the app. We measured the warm-start delays experienced by the app in both settings (w/ and w/o HMA), 50 measurements per setting.

Results. Intuitively, in HMA, the first launch of an mHealth will experience longer delays than the subsequent cold starts, because the container app has to process the APK and store the information needed for user-level virtualization. Our experiments show that the median of this process takes 6.5 ± 0.16 s (as compared to 0.74 ± 0.07 s if the mHealth apps were launched w/o HMA). Note that this occurs only once, hence it is negligible w.r.t. the lifetime of the app.

Fig. 2 shows the bar plot of subsequent cold-start delays, with and without HMA, experienced by mHealth apps; the heights of the bars represent the mean values, and the error bars represent one standard deviation. It can be seen that the average delays are at most 3.0 ± 0.5 s and 1.3 ± 0.05 s if the apps are executed with and without HMA, respectively. For 90% of the cases, the average delay with HMA is less than 2.0 ± 0.3 s. Note that our prototype is a proof-of-concept hence not optimized. Still, the observed delays are under the

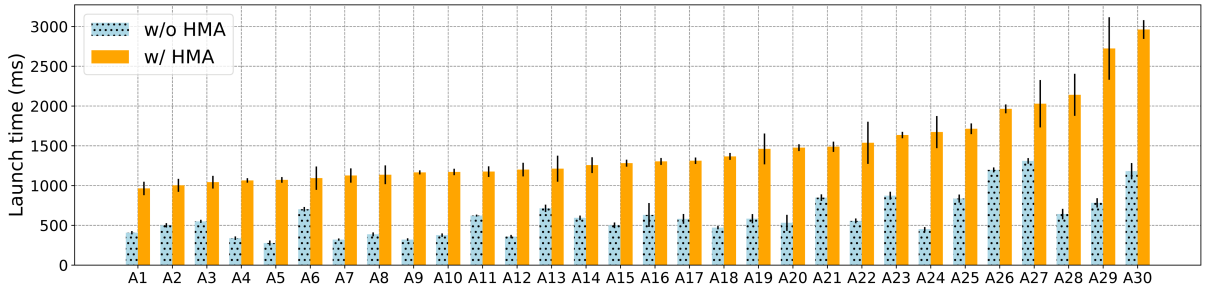


Figure 2: Cold-start delays experienced by mHealth apps when they are executed with and without HMA. Note that our HMA implementation is a proof-of-concept, hence not-optimized. The heights of the bars represent mean values and the error bars represent the standard deviation. For each setting, we collected 50 measurements per app. The full names of the apps can be found in Appendix B of our technical report at [44].

delay limit of 5 s suggested by Android [13]. Also, in our user study, 97% of participants agreed that a launch delay of 5 s is acceptable (Section 10.6).

Regarding warm-start delays, we found that the average delay experienced by our tested app, when it was launched with and without HMA, was ∼0.55 s. This is intuitive, because the app's processes were still running and the activities still resided in the phone's memory. In case the garbage collector evicts the activities from the phone's memory, warm-start delays can be longer, due to the overheads of activity initializations. We cannot simulate this case, because Android does not provide methods to control the garbage collector. However, in this case, the delay will still be less than cold-start delays (*i.e.,* at most 3 s).

10.4 HMA Robustness and Compatibility

In this section, we present the evaluation of HMA in terms of its robustness and its compatibility with Android versions.

Runtime Robustness. Following the approach used in previous work, (*e.g.,* [35] and [53]), we manually tested each app in our dataset with HMA. For each mHealth app, we extracted its APK, created a container app using HMA App Store, and installed the container app on the phone. Thereafter, we used the HMA Manager to launch the app. We manually used most of the functionality of the mHealth app, and checked if it had crashed during its execution. We found that all of the apps in our dataset worked normally, except one app that threw an error when making an SQLite connection. To determine the cause of this error, we ran an example app[33] that uses the official Android API for database access (*i.e.,* Android.database.sqlite) to insert and retrieve records from an SQLite database, and the example app ran successfully. We suspect that the mHealth app threw an error because it specifies the full path to the database (*i.e.,* /data/data/package-name/db-name). Hence, when running the app inside of the HMA container app, the hard-coded path is not longer valid. To avoid this problem, developers should specify the relative path to the database (*i.e.,* ./db-name) instead of its full path.

Compatibility. We ran HMA on a series of smartphones with Android OS from version 5.0 to 8.0, which accounts for 89% of the current Android version distribution [7]. We found that HMA can be successfully deployed on mainstream commercial Android devices. But, there are two apps (Mole Mapper and Alzheimer's Speed of Processing Game) that initially failed to run on our Nexus 5X (Android 7.1.1) due to the incompatibility between 32-bit and 64-bit systems. We fixed the problem by enabling the option -abi armeabi-v7a when installing them. From the list of 20 apps that we filtered out, we found that 3 apps (Hearing Aid, What's Up and Cardiac diagnosis) successfully ran on Android 5.0 and 6.0, but they failed to run in later versions of Android. We investigated the log of the three apps and found that API methods related to GMS services that we do not support were called in the later versions of Android. This problem could be solved if these services are hooked, as we discussed in Section 10.2. Note that, with the recent release of Android 9, Google has restricted the use of Java reflection[34] – the programming interface that all user-level virtualization techniques rely on. Therefore, for HMA to work seamlessly in Android 9, new user-level virtualization techniques need to be explored. Still, Android 9 has only less than 1% market share [7], which means HMA will be compatible with most Android devices in the coming years. Alternatively, HMA could work with rooted Android 9 devices by using dynamic customization frameworks, *e.g.,* Xposed.[35]

10.5 Inter-App Communication Support

Sharing resources with other apps via customized features (*e.g.,* custom permissions and custom intent-filters) and publicly exposing components (*e.g.,* activities, services and content providers) could facilitate fingerprinting attacks. HMA can partially support inter-app communications, but it is preferable to avoid such features to guarantee robust fingerprinting protection. Avoiding inter-app communication, however, can affect apps' functionality and backward compatibility. To estimate the effect of using HMA and avoiding inter-app communication on existing apps, we analyzed a set of popular

sensitive apps from the Google Play Store. Our results show that a small number of apps use inter-app communication features that are not supported by HMA and, in many cases, such features are not directly related to apps' key functionalities.

Dataset. We collected a total of 1045 APK files from the most popular free apps in the Medical and Health&Fitness categories in the US Google Play Store. By checking the apps' descriptions, we found that approximately 60% directly match HMA's use case (*i.e.,* health- and fitness-related apps). The rest of the apps are less related to HMA's use case, *e.g.,* apps for medical doctors and nurses, apps for managing accounts with health providers, and apps for managing gym subscriptions. From the APK files, we extracted the manifest file using the `apktool` application.

Custom Permissions. Permissions defined by apps to control access to their components can be used to fingerprint them (e.g., they typically include the app name), as any app can list the permissions of other apps. Hence, custom permissions should be avoided. We found that a total of 531 apps declared custom permissions. However, most of these permission declarations are related to deprecated services (Google Cloud Messaging and Android Maps API v2)[36] and can be replaced with newer alternatives that do not require custom permissions. Therefore, ignoring permissions associated with these deprecated services, we found that only 68 apps (6.5%) declared valid custom permissions.

Signature-Level Permissions. Signature permissions[37] are a subset of custom permissions, hence they can be used to fingerprint apps. Given that HMA container apps are all signed by the same key, a malicious app inside a container app could abuse signature permissions to access resources of sensitive apps in other container apps. Therefore, HMA currently does not declare signature permissions in the container apps. Our analysis shows that only 113 apps (10.8%) declared signature permissions and, as explained before, many of these permissions are associated with deprecated services (*e.g.,* Google Cloud Messaging and Android Maps API).

Content Providers. Any app can list the content providers of other installed apps and use this information to fingerprint them. Therefore, HMA obfuscates this information in the container app. This means that public content providers (used to share data with other apps) are not currently supported by HMA. Our analysis shows that only 84 apps (8%) declare public content providers. From these apps, 68% declare public content-providers associated with third-party frameworks (*e.g.,* Seattle Clouds) for services such as file sharing and authentication, and approximately 23% require permission to access the provider.

Intent Filters. Custom intent filters, *i.e.,* intent filters with app-specific actions, could be used to fingerprint apps. Apps cannot list the intent filters of other apps, but they can list all the activities of other apps that can be performed by a particular intent. Hence, developers should avoid using custom intent filters in their apps' activities. Our analysis shows that this is not a problem, as only 38 apps (3.6%) have activities with custom intent filters.

Activities and Explicit Intents. Explicit intents are currently not supported by HMA because container apps obfuscate the activities' names of sensitive apps; thus, direct reference to sensitive apps' activities is not possible. However, it is recommended to use only explicit intents to launch internal activities; not activities of other apps (implicit intents should be used instead). Our analysis shows that 170 apps (16.2%) declare activities that can be launched by other apps via explicit intents only, *i.e.,* no intent filters. We noticed that many apps (67) declared this type of activities to support Google's Firebase authentication services. Yet, Firebase's official documentation does not seem to mention this approach to support its services. Hence, to be compatible with HMA, these apps could evaluate alternative (official) approaches to support Firebase services or rely on other services for user authentication.

Services and Explicit Intents. Explicit intents are recommended to access services offered by other apps. But, as stated before, it is not possible to use explicit intents with HMA. Our analysis show that 367 apps (35.1%) declare public services that require explicit intents. This is a significant number, yet we noticed that a large number of apps (252) use services with explicit intents to support Google Play services for authentication (Google Sign-In user revocation). Hence, these apps could use alternative user authentication services to be compatible with HMA. We also notice that only 44 apps declared services that belong to the app itself; this indicates that most of these services are associated with third-parties and probably are not part of apps' main functionalities.

Broadcast Receivers. Whereas any app can list the broadcast receivers of other apps, HMA container apps obfuscate the names of the receivers to defend against fingerprinting attacks. The container app can declare the same intent filters for receivers that the sensitive app declares (including custom intent filters) because other apps cannot list these intent filters. As broadcast receivers offer asynchronous communication, nosy apps cannot use API methods to check if an app is receiving a particular broadcast intent. In short, broadcast receivers are supported by HMA.

10.6 HMA Usability and Desirability

To evaluate the usability of HMA and the users' interest for it, we conducted a user study that was approved by our institutional ethical committee. It involved 30 student subjects (19 males, 11 females, 22 ± 4.5 years old) from 18 areas of study. The participants were experienced Android users: 87% of them have used an Android phone for at least a year. Also, they were relatively concerned about their privacy; using the standard metric for measuring privacy perception (IUIPC) [39], we found that, on a scale from 1 to 5, 97% of participants

graded at least 3.0 and an average of 4.1.

We began the study with an entry survey about demographic information, privacy postures, users' awareness and concerns about the problem of LIA collections. Then, we provided each participant with a fresh phone and asked them to install and use two apps: a popular public-transportation app for our city and an mHealth app called Cancer.Net. To precisely measure the users' perceptions of the delay introduced by HMA, the participants were asked to use the two aforementioned apps with and without HMA; detailed instructions were provided to them. 67% of the participants had used the transportation app before, whereas only 7% of them had used the Cancer.Net app or an mHealth app. We finished the user study with an exit survey containing questions related to the usability of HMA and the users' levels of interest in HMA. The user-study session took ∼45 minutes, and we paid each participant ∼$25 (*i.e.,* 25 CHF). The transcript of survey questions and the instructions can be found at [38].

Our study shows that the participants are concerned about the privacy of health-related data: 90% of the participants would be at least concerned if their health-related information were collected by apps installed on their phones and shared with third parties, and 87% of participants would be at least concerned if third parties learned that they had used health-related apps. Indeed, our study confirms the findings from previous works (*e.g.,* [41]) that the majority of people never read privacy policies. Therefore, the current solution of using privacy policies by Google for LIA collections is not satisfactory. These findings make clear the case for HMA.

Regarding the usability of HMA, only 30% of the participants noticed a difference when the two apps ran with and without HMA. Note that the delays that users experienced in the user study were the first-launch delays, which are 4.2 ± 0.06 s and 5.1 ± 0.07 s for the transportation app and the Cancer.Net app, respectively. From the open-ended question in our exit survey, we found that the observed differences are mainly about the launching delay of the apps and the change in the app names in permission prompts. From the close-ended questions, which were coded using a five-point Likert scale, we observe the following. Almost all participants agree that these changes and delays are acceptable (97% and 93% of the participants, respectively). 93% of the participants also agree that the use of an HMA Manager to install and launch apps is at least somewhat acceptable. Also, 90% of the participants agreed that HMA does not affect the user experience of the apps that it protects, and that they are at least somewhat interested in using HMA. These results suggest that HMA is usable and desirable.

11 Conclusion

In this work, we have shown that apps can collect a significant amount of static and runtime information about other apps, to fingerprint them. Our analysis has shown that many third parties are interested in learning about the apps installed on people's phones. Moreover, we have shown that there are no existing mechanisms for hiding the presence of an app from other apps. We have proposed HMA, the first solution that addresses this problem. HMA does not require any modifications to the Android OS and preserves the key functionalities of apps. The results of our evaluation and user study suggest that HMA is usable and of interest to users.

Acknowledgements

We thank our anonymous reviewers and our shepherd Sven Bugiel for their insightful comments on this work. We are grateful to oncologist Olivier Michielin who helped us identify the problem addressed in this work. This project was supported by the grant #2017-201 of the Strategic Focal Area "Personalized Health and Related Technologies (PHRT)" of the ETH Domain. The work was carried out while Anh Pham was a PhD student at EPFL.

References

[1] Android Security 2015 Year In Review. https://source.android.com/security/reports/Google_Android_Security_2015_Report_Final.pdf. Visited: Sep. 2018.

[2] Android Security 2016 Year In Review. https://source.android.com/security/reports/Google_Android_Security_2016_Report_Final.pdf. Visited: Sep. 2018.

[3] Angry Birds. https://play.google.com/store/apps/details?id=com.rovio.angrybirds. Visited: Sep. 2018.

[4] Beurer HealthManager. https://play.google.com/store/apps/details?id=com.beurer.connect.healthmanager. Visited: Sep. 2018.

[5] Cancer.Net Mobile. https://play.google.com/store/apps/details?id=com.fueled.cancernet. Visited: Sep. 2018.

[6] DH Texas Poker - Texas Hold'em. https://play.google.com/store/apps/details?id=com.droidhen.game.poker. Visited: Sep. 2018.

[7] Distribution dashboard. https://developer.android.com/about/dashboards/. Visited: Sep. 2018.

[8] DroidPlugin. https://github.com/DroidPluginTeam/DroidPlugin. Visited: Sep. 2018.

[9] F-Droid. https://f-droid.org/en/. Visited: Sep. 2018.

[10] Help protect against harmful apps with Google Play Protect. https://support.google.com/accounts/answer/2812853?hl=en. Visited: Sep. 2018.

[11] Hide App, Private Dating, Safe Chat - PrivacyHider. https://play.google.com/store/apps/details?id=com.trigtech.privateme&hl=en. Visited: Sep. 2018.

[12] InstaSize Editor: Photo Filters and Collage Maker. https://play.google.com/store/apps/details?id=com.jsdev.instasize. Visited: Sep. 2018.

[13] Launch-Time Performance. https://developer.android.com/topic/performance/launch-time.html. Visited: Sep. 2018.

[14] MX Player. https://play.google.com/store/apps/details?id=com.mxtech.videoplayer.ad. Visited: Sep. 2018.

[15] Neon Motocross. https://play.google.com/store/apps/details?id=com.motomex.neonmotocross. Visited: Sep. 2018.

[16] Nova Launcher. https://play.google.com/store/apps/details?id=com.teslacoilsw.launcher&hl=en. Visited: Sep. 2018.

[17] Parallel Space - Multiple accounts and Two face. https://play.google.com/store/apps/details?id=com.lbe.parallel.intl&hl=en. Visited: Sep. 2018.

[18] Private Zone - Safe Vault. https://play.google.com/store/apps/details?id=com.leo.appmaster. Visited: Sep. 2018.

[19] Publish Your App. https://developer.android.com/studio/publish/index.html#publishing-unknown. Visited: Sep. 2018.

[20] Solitaire: Super Challenges. https://play.google.com/store/apps/details?id=com.cardgame.solitaire.full. Visited: Sep. 2018.

[21] Sweet Selfie - selfie camera, beauty cam, photo edit. https://play.google.com/store/apps/details?id=com.cam001.selfie. Visited: Sep. 2018.

[22] ACHARA, J. P., ACS, G., AND CASTELLUCCIA, C. On the Unicity of Smartphone Applications. In *Proc. of WPES* (2015).

[23] AITKEN, M., AND LYLE, J. Patient adoption of mhealth: use, evidence and remaining barriers to mainstream acceptance. *IMS Institute for Healthcare Informatics* (2015).

[24] BACKES, M., BUGIEL, S., HAMMER, C., SCHRANZ, O., AND VON STYP-REKOWSKY, P. Boxify: Full-fledged App Sandboxing for Stock Android. In *Proc. of USENIX Security* (2015).

[25] BIANCHI, A., FRATANTONIO, Y., KRUEGEL, C., AND VIGNA, G. Njas: Sandboxing unmodified applications in non-rooted devices running stock android. In *Proc. of SPSM* (2015).

[26] CHEN, Q. A., QIAN, Z., AND MAO, Z. M. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *Proc. of USENIX Security* (2014).

[27] CHEN, Y., JIN, X., SUN, J., ZHANG, R., AND ZHANG, Y. POWERFUL: Mobile app fingerprinting via power analysis. In *Proc. of IEEE INFOCOM* (2017).

[28] DAI, S., TONGAONKAR, A., WANG, X., NUCCI, A., AND SONG, D. NetworkProfiler: Towards automatic fingerprinting of Android apps. In *Proc. of IEEE INFOCOM* (2013).

[29] DEMETRIOU, S., MERRILL, W., YANG, W., ZHANG, A., AND GUNTER, C. A. Free for all! assessing user data exposure to advertising libraries on android. In *Proc. of NDSS* (2016).

[30] FELT, A. P., HA, E., EGELMAN, S., HANEY, A., CHIN, E., AND WAGNER, D. Android Permissions: User Attention, Comprehension, and Behavior. In *Proc. of SOUPS* (2012).

[31] FERNANDES, E., PAUPORE, J., RAHMATI, A., SIMIONATO, D., CONTI, M., AND PRAKASH, A. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *Proc. of USENIX Security* (2016).

[32] GRACE, M. C., ZHOU, W., JIANG, X., AND SADEGHI, A.-R. Unsafe exposure analysis of mobile in-app advertisements. In *Proc. of ACM WiSec* (2012).

[33] GULYÁS, G. G., ACS, G., AND CASTELLUCCIA, C. Near-Optimal Fingerprinting with Constraints. *Proceedings of Privacy Enhancing Technologies Symposium* (2016).

[34] HUANG, J., SCHRANZ, O., BUGIEL, S., AND BACKES, M. The ART of App Compartmentalization: Compiler-based Library Privilege Separation on Stock Android. In *Proc. of ACM CCS* (2017).

[35] JAEBAEK, S., DAEHYEOK, K., DONGHYUN, C., INSIK, S., AND TAESOO, K. FLEXDROID: Enforcing In-App Privilege Separation in Android. In *Proc. of NDSS* (2016).

[36] JANA, S., AND SHMATIKOV, V. Memento: Learning secrets from process footprints. In *Proc. of IEEE S&P* (2012).

[37] KOTZ, D., GUNTER, C. A., KUMAR, S., AND WEINER, J. P. Privacy and Security in Mobile Health: A Research Agenda. *Computer* (June 2016).

[38] LIN, C.-C., LI, H., ZHOU, X.-Y., AND WANG, X. Screenmilker: How to Milk Your Android Screen for Secrets. In *Proc. of NDSS* (2014).

[39] MALHOTRA, N. K., KIM, S. S., AND AGARWAL, J. Internet users' information privacy concerns (IUIPC): The construct, the scale, and a causal model. *Information systems research* (2004).

[40] MALMI, E., AND WEBER, I. You Are What Apps You Use: Demographic Prediction Based on User's Apps. In *Proc. of AAAI CWSM* (2016).

[41] MCDONALD, A. M., REEDER, R. W., KELLEY, P. G., AND CRANOR, L. F. A Comparative Study of Online Privacy Policies and Formats. In *Privacy Enhancing Technologies* (2009).

[42] NAVEED, M., ZHOU, X.-Y., DEMETRIOU, S., WANG, X., AND GUNTER, C. A. Inside Job: Understanding and Mitigating the Threat of External Device Mis-Binding on Android. In *Proc. of NDSS* (2014).

[43] NGUYEN, D. C., WERMKE, D., ACAR, Y., BACKES, M., WEIR, C., AND FAHL, S. A Stitch in Time: Supporting Android Developers in Writing Secure Code. In *Proc. of ACM CCS* (2017).

[44] PHAM, A., DACOSTA, I., LOSIOUK, E., STEPHAN, J., HUGUENIN, K., AND HUBAUX, J.-P. HideMyApp : Hiding the Presence of Sensitive Apps on Android. In *EPFL Infoscience* (2019).

[45] RATAZZI, P., AAFER, Y., AHLAWAT, A., HAO, H., WANG, Y., AND DU, W. A systematic security evaluation of android's multi-user framework. *arXiv preprint arXiv:1410.7752* (2014).

[46] SENEVIRATNE, S., SENEVIRATNE, A., MOHAPATRA, P., AND MAHANTI, A. Predicting User Traits from a Snapshot of Apps Installed on a Smartphone. *SIGMOBILE Mob. Comput. Commun. Rev. 18*, 2 (June 2014).

[47] SENEVIRATNE, S., SENEVIRATNE, A., MOHAPATRA, P., AND MAHANTI, A. Your installed apps reveal your gender and more! *SIGMOBILE Mob. Comput. Commun. Rev.* (2015).

[48] SUN, M., AND TAN, G. NativeGuard: Protecting Android Applications from Third-party Native Libraries. In *Proc. of ACM WiSec* (2014).

[49] SUNSHINE, J., EGELMAN, S., ALMUHIMEDI, H., ATRI, N., AND CRANOR, L. F. Crying Wolf: An Empirical Study of SSL Warning Effectiveness. In *Proc. of USENIX Security* (2009).

[50] TAYLOR, V. F., SPOLAOR, R., CONTI, M., AND MARTINOVIC, I. Appscanner: Automatic fingerprinting of smartphone apps from encrypted network traffic. In *Proc. of IEEE EuroS&P* (2016).

[51] TAYLOR, V. F., SPOLAOR, R., CONTI, M., AND MARTINOVIC, I. Robust Smartphone App Identification via Encrypted Network Traffic Analysis. *IEEE Trans. on Inf. Forensics and Security 13*, 1 (Jan. 2018).

[52] WANG, T., AND GOLDBERG, I. Walkie-talkie: An efficient defense against passive website fingerprinting attacks. In *Proc. of USENIX Security* (2017).

[53] WANG, X., SUN, K., WANG, Y., AND JING, J. DeepDroid: Dynamically Enforcing Enterprise Policy on Android Devices. In *Proc. of NDSS* (2015).

[54] XU, Q., LIAO, Y., MISKOVIC, S., MAO, Z. M., BALDI, M., NUCCI, A., AND ANDREWS, T. Automatic generation of mobile app signatures from traffic observations. In *Proc. of IEEE INFOCOM* (2015).

[55] ZHOU, X., DEMETRIOU, S., HE, D., NAVEED, M., PAN, X., WANG, X., GUNTER, C. A., AND NAHRSTEDT, K. Identity, location, disease and more: Inferring your secrets from android public resources. In *Proc. of ACM CCS* (2013).

Notes

[1] https://liquid-state.com/mhealth-apps-market-snapshot/. Visited: Nov. 2018.

[2] https://research2guidance.com/mhealth-app-market-getting-crowded-259000-mhealth-apps-now/. Visited: Sep. 2018.

[3] https://www.theguardian.com/technology/2014/nov/27/twitter-scanning-other-apps-tailored-content. Note that Twitter recently announced that it excludes apps dealing with health, religion and sexual orientation, https://help.twitter.com/en/safety-and-security/app-graph. Visited: Sep. 2018.

[4] https://techcrunch.com/2017/05/04/report-smartphone-owners-are-using-9-apps-per-day-30-per-month/. Visited: Sep. 2018.

[5] Now reclassified as Mobile Unwanted Software (MUwS) [2].

[6] https://play.google.com/about/developer-content-policy-print/. Visited: Sep. 2018.

[7] Additional protections by Safe Browsing for Android users, https://security.googleblog.com/2017/12/additional-protections-by-safe-browsing.html. Visited: Sep. 2018.

[8] Note that, unlike previous work (*e.g.,* [22]) that focuses on apps directly retrieving the list of installed apps, our work focuses on the fingerprintability of *a specific* app, a more general and difficult problem.

[9] https://developer.android.com/guide/topics/permissions/overview. Visited: Sep. 2018.

[10] https://developer.android.com/studio/command-line/adb.html. Visited: Sep. 2018.

[11] https://codelabs.developers.google.com/codelabs/developing-android-a11y-service/. Visited: Apr. 2019.

[12] https://cromulentlabs.wordpress.com/2016/01/15/explanation-of-canopenurl-changes-in-ios-9/. Visited: Sep. 2018.

[13] https://ibotpeaches.github.io/Apktool/. Visited: Sep. 2018.

[14] Note that we also found many occurrences of other methods presented in Section 4, but we did not know the purposes of the calling apps.

[15] http://www.zdnet.com/article/accuweather-caught-sending-geo-location-data-even-when-denied-access/. Visited: Nov. 2018.

[16] https://github.com/M66B/XPrivacy. Visited: Sep. 2018.

[17] We performed a similar analysis on a small set of paid apps, see Appendix A of our technical report at [44].

[18] https://play.google.com/about/privacy-security-deception/personal-sensitive/. Visited: Sep. 2018.

[19] https://source.android.com/devices/tech/admin/multi-user. Visited: Sep. 2018.

[20] https://www.xda-developers.com/add-multi-user-support-android/. Visited: Feb. 2019.

[21] https://www.android.com/enterprise/employees/. Visited: Sep. 2018

[22] https://developer.android.com/topic/instant-apps/index.html. Visited: Sep. 2018.

[23] https://developer.android.com/topic/instant-apps/reference.html#instantapps.InstantApps. Visited: Sep. 2018

[24] https://www.samsungknox.com/en. Visited: Sep. 2018.

[25] https://www.amazon.com/mobile-apps/b?ie=UTF8&node=2350149011. Visited: Sep. 2018.

[26] *P* is defined based on the estimation about the number of sensitive apps that users of the `HMA App Store` can have, because Android does not permit duplicate package names for apps. Average users have around 80 apps on their phones, therefore *P* is *at most* 80.

[27] https://developer.android.com/about/versions/marshmallow/android-6.0#direct-share. Visited: Feb. 2019.

[28] https://www.torproject.org/projects/torbrowser/design/. Visited: Sep. 2018.

[29] https://www.androidpolice.com/2017/11/12/google-will-remove-play-store-apps-use-accessibility-services-anything-except-helping-disabled-users/. Visited: Apr. 2018.

[30] A signature-protected permission is a permission that the system grants only if the requesting app is signed with the same certificate as the app that declared the permission.

[31] We omit the app-update operation, because app-update and app-installation operations are similar.

[32] https://github.com/commonsguy/cw-omnibus/tree/master/Activities/Lifecycle. Visited: Sep. 2018.

[33] SQLiteOpenHelper, https://github.com/commonsguy/cw-omnibus/tree/master/Database/ConstantsROWID. Visited: Sep. 2018

[34] https://developer.android.com/about/versions/pie/restrictions-non-sdk-interfaces. Visited: Sep. 2018.

[35] https://repo.xposed.info/module/de.robv.android.xposed.installer. Visited: Sep. 2018

[36] https://developers.google.com/cloud-messaging/android/android-migrate-fcm. Visited: Feb. 2019.

[37] A permission that the system grants only if the requesting application is signed with the same certificate as the application that declared the permission

[38] https://www.dropbox.com/sh/lo273jtx6jkbf1c/AAB1BtkBmBuNVOV13OAwDu-ha?dl=1

TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time

Feargus Pendlebury*†‡, Fabio Pierazzi*†‡, Roberto Jordaney†‡, Johannes Kinder§, Lorenzo Cavallaro†
† *King's College London*
‡ *Royal Holloway, University of London*
§ *Bundeswehr University Munich*

Abstract

Is Android malware classification a solved problem? Published F_1 scores of up to 0.99 appear to leave very little room for improvement. In this paper, we argue that results are commonly inflated due to two pervasive sources of experimental bias: *spatial bias* caused by distributions of training and testing data that are not representative of a real-world deployment; and *temporal bias* caused by incorrect time splits of training and testing sets, leading to impossible configurations. We propose a set of space and time constraints for experiment design that eliminates both sources of bias. We introduce a new metric that summarizes the expected robustness of a classifier in a real-world setting, and we present an algorithm to tune its performance. Finally, we demonstrate how this allows us to evaluate mitigation strategies for time decay such as active learning. We have implemented our solutions in TESSERACT, an open source evaluation framework for comparing malware classifiers in a realistic setting. We used TESSERACT to evaluate three Android malware classifiers from the literature on a dataset of 129K applications spanning over three years. Our evaluation confirms that earlier published results are biased, while also revealing counter-intuitive performance and showing that appropriate tuning can lead to significant improvements.

1 Introduction

Machine learning has become a standard tool for malware research in the academic security community: it has been used in a wide range of domains including Windows malware [12, 34, 51], PDF malware [27, 32], malicious URLs [28, 48], malicious JavaScript [11, 43], and Android malware [4, 22, 33]. With tantalizingly high performance figures, it seems malware should be a problem of the past.

Malware classifiers operate in dynamic contexts. As malware evolves and new variants and families appear over time, prediction quality decays [26]. Therefore, temporal consistency matters for evaluating the effectiveness of a classifier. When the experimental setup allows a classifier to train on what is effectively future knowledge, the reported results become biased [2, 36].

This issue is widespread in the security community and affects multiple security domains. In this paper, we focus on Android malware and claim that there is an endemic issue in that Android malware classifiers [4, 13, 18, 22, 33, 49, 56, 57] (including our own work) are not evaluated in settings representative of real-world deployments. We choose Android because of the availability of (a) a public, large-scale, and timestamped dataset (AndroZoo [3]) and (b) algorithms that are feasible to reproduce (where all [33] or part [4] of the code has been released).

We identify experimental bias in two dimensions, *space* and *time*. *Spatial bias* refers to unrealistic assumptions about the ratio of goodware to malware in the data. The ratio of goodware to malware is domain-specific, but it must be enforced consistently during the testing phase to mimic a realistic scenario. For example, measurement studies on Android suggest that most apps in the wild are goodware [21, 30], whereas for (desktop) software download events most URLs are malicious [31, 41]. *Temporal bias* refers to temporally inconsistent evaluations which integrate future knowledge about the testing objects into the training phase [2, 36] or create unrealistic settings. This problem is exacerbated by families of closely related malware, where including even one variant in the training set may allow the algorithm to identify many variants in the testing.

We believe that the pervasiveness of these issues is due to two main reasons: first, possible sources of evaluation bias are not common knowledge; second, accounting for time complicates the evaluation and does not allow a comparison to other approaches using headline evaluation metrics such as the F_1-Score or AUROC. We address these issues in this paper by systematizing evaluation bias for Android malware classification and providing new constraints for sound experiment design along with new metrics and tool support.

Prior work has investigated challenges and experimental bias in security evaluations [2, 5, 36, 44, 47, 54]. The *base-rate*

*Equal contribution.

fallacy [5] describes how evaluation metrics such as *TPR* and *FPR* are misleading in intrusion detection, due to significant class imbalance (most traffic is benign); in contrast, we identify and address experimental settings that give misleading results *regardless* of the adopted metrics—even when correct metrics are reported. Sommer and Paxson [47], Rossow et al. [44], and van der Kouwe et al. [54] discuss possible guidelines for sound security evaluations; but none of these works identify temporal and spatial bias, nor do they *quantify* the impact of errors on classifier performance. Allix et al. [2] and Miller et al. [36] identify an initial temporal constraint in Android malware classification, but we show that even results of recent work following their guidelines (e.g., [33]) suffer from other temporal and spatial bias (§4.4). To the best of our knowledge, we are the first to identify and address these sources of bias with novel, actionable constraints, metrics, and tool support (§4).

This paper makes the following contributions:

- We identify *temporal* bias associated with incorrect train-test splits (§3.2) and *spatial* bias related to unrealistic assumptions in dataset distribution (§3.3). We experimentally verify on a dataset of 129K apps (with 10% malware) that, due to bias, performance can decrease up to 50% in practice (§3.1) in two well-known Android malware classifiers, DREBIN [4] and MAMADROID [33], which we refer to as ALG1 and ALG2, respectively.
- We propose novel building blocks for more robust evaluations of malware classifiers: a set of spatio-temporal constraints to be enforced in experimental settings (§4.1); a new metric, AUT, that captures a classifier's robustness to time decay in a single number and allows for the fair comparison of different algorithms (§4.2); and a novel tuning algorithm that empirically optimizes the classification performance, when malware represents the minority class (§4.3). We compare the performance of ALG1 [4], ALG2 [33] and DL [22] (a deep learning-based approach), and show how removing bias can provide counter-intuitive results on real performance (§4.4).
- We implement and publicly release the code of our methodology (§4), TESSERACT, and we further demonstrate how our findings can be used to evaluate performance-cost trade-offs of solutions to mitigate time decay such as active learning (§5).

TESSERACT can assist the research community in producing comparable results, revealing counter-intuitive performance, and assessing a classifier's prediction qualities in an industrial deployment (§6).

We believe that our methodology also creates an opportunity to evaluate the extent to which spatio-temporal experimental bias affects security domains other than Android malware, and we encourage the security community to embrace its underpinning philosophy.

Use of the term "bias": We use *(experimental) bias* to refer to the details of an experimental setting that depart from the conditions in a real-world deployment and can have a misleading impact (*bias*) on evaluations. We do not intend it to relate to the classifier bias/variance trade-off [8] from traditional machine learning terminology.

2 Android Malware Classification

We focus on Android malware classification. In §2.1 we introduce the reference approaches evaluated, in §2.2 we discuss the domain-specific prevalence of malware, and in §2.3 we introduce the dataset used throughout the paper.

2.1 Reference Algorithms

To assess experimental bias (§3), we consider two high-profile machine learning-driven techniques for Android malware classification, both published recently in top-tier security conferences. The first approach is **ALG1** [4], a linear support vector machine (SVM) on high-dimensional binary feature vectors engineered with a lightweight static analysis. The second approach is **ALG2** [33], a Random Forest (RF) applied to features engineered by modeling caller-callee relationships over Android API methods as Markov chains. We choose ALG1 and ALG2 as they build on different types of static analysis to generate feature spaces capturing Android application characteristics at different levels of abstraction; furthermore, they use different machine learning algorithms to learn decision boundaries between benign and malicious Android apps in the given feature space. Thus, they represent a broad design space and support the generality of our methodology for characterizing experimental bias. For a sound experimental baseline, we reimplemented ALG1 following the detailed description in the paper; for ALG2, we relied on the implementation provided by its authors. We replicated the baseline results for both approaches. After identifying and quantifying the impact of experimental bias (§3), we propose specific constraints and metrics to allow fair and unbiased comparisons (§4). Since ALG1 and ALG2 adopt traditional ML algorithms, in §4 we also consider **DL** [22], a deep learning-based approach that takes as input the same features as ALG1 [4]. We include DL because the latent feature space of deep learning approaches can capture different representations of the input data [19], which may affect their robustness to time decay. We replicate the baseline results for DL reported in [22] by re-implementing its neural network architecture and by using the same input features as for ALG1.

It speaks to the scientific standards of these papers that we were able to replicate the experiments; indeed, we would like to emphasize that we do not criticize them specifically. We use these approaches for our evaluation because they are available and offer stable baselines.

We report details on the hyperparameters of the reimplemented algorithms in §A.1.

2.2 Estimating in-the-wild Malware Ratio

The proportion of malware in the dataset can greatly affect the performance of the classifier (§3). Hence, unbiased experi-

ments require a dataset with a realistic percentage of malware over goodware; on an already existing dataset, one may enforce such a ratio by, for instance, downsampling the majority class (§3.3). Each malware domain has its own, often unique, ratio of malware to goodware typically encountered in the wild. First, it is important to know if malware is a minority, majority, or an equal-size class as goodware. For example, malware is the minority class in network traffic [5] and Android [30], but it is the majority class in binary download events [41]. On the one hand, the estimation of the percentage of malware in the wild for a given domain is a non-trivial task. On the other hand, measurement papers, industry telemetry, and publicly-available reports may all be leveraged to obtain realistic estimates.

In the Android landscape, malware represents 6%–18.8% of all the apps, according to different sources: a key industrial player[1] reported the ratio as approximately 6%, whereas the AndRadar measurement study [30] reports around 8% of Android malware in the wild. The 2017 Google's Android security report [21] suggests 6–10% malware, whereas an analysis of the metadata of the AndroZoo dataset [3] totaling almost 8M Android apps updated regularly, reveals an incidence of 18.8% of malicious apps. The data suggests that, in the Android domain, malware is the minority class. In this work, we decide to stabilize its percentage to 10% (a de-facto average across the various estimates), with per-month values between 8% and 12%. Settling on an average overall ratio of 10% Android malware also allows us to collect a dataset with a statistically sound number of per-month malware. An aggressive undersampling would have decreased the statistical significance of the dataset, whereas oversampling goodware would have been too resource intensive (§2.3).

2.3 Dataset

We consider samples from the public AndroZoo [3] dataset, consisting of more than 8.5 million Android apps between 2010 and early 2019: each app is associated with a timestamp, and most apps include VirusTotal metadata results. The dataset is constantly updated by crawling from different markets (e.g., more than 4 million apps from Google Play Store, and the remaining from markets such as Anzhi and AppChina). We choose to refer to this dataset due to its size and timespan, which allow us to perform realistic space- and time-aware experiments.

Goodware and malware. AndroZoo's metadata reports the number p of positive anti-virus reports on VirusTotal [20] for applications in the AndroZoo dataset. We chose $p = 0$ for goodware and $p \geq 4$ for malware, following Miller et al.'s [36] advice for a reliable ground-truth. About 13% of AndroZoo apps can be called grayware as they have $0 < p < 4$. We exclude grayware from the sampling as including it as either goodware or malware could disadvantage classifiers whose features were designed with a different labeling threshold.

[1] Information obtained through confidential emails with the authors.

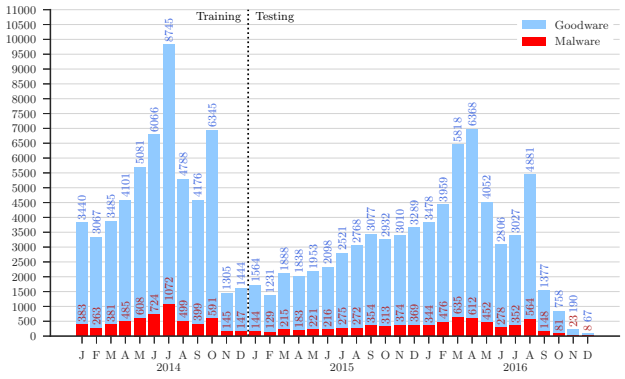


Figure 1: *Details of the dataset considered throughout this paper*. The figure reports a stack histogram with the monthly distribution of apps we collect from AndroZoo: 129,728 Android applications (with average 10% malware), spanning from Jan 2014 to Dec 2016. The vertical dotted line denotes the split we use in all time-aware experiments in this paper (see §4 and §5): training on 2014, testing on 2015 and 2016.

Choosing apps. The number of objects we consider in our study is affected by the feature extraction cost, and partly by storage space requirements (as the full AndroZoo dataset, at the time of writing, is more than 50TB of apps to which one must add the space required for extracting features). Extracting features for the whole AndroZoo dataset may take up to three years on our research infrastructure (three high-spec Dell PowerEdge R730 nodes, each with 2 x 14 cores in hyperthreading—in total, 168 vCPU threads, 1.2TB of RAM, and a 100TB NAS), thus we decided to extract features from 129K apps (§2.2). We believe this represents a large dataset with enough statistical significance. To evaluate time decay, we decide on a granularity of one month, and we uniformly sample 129K AndroZoo apps in the period from Jan 2014 to Dec 2016, but also enforce an overall average of 10% malware (see §2.2)–with an allowed percentage of malware per month between 8% and 12%, to ensure some variability. Spanning over three years ensures 1,000+ apps per month (except for the last three months, where AndroZoo had crawled less applications). We consider apps up to Dec 2016 because the VirusTotal results for 2017 and 2018 apps were mostly unavailable from AndroZoo at the time of writing; moreover, Miller et al. [36] empirically evaluated that antivirus detections become stable after approximately one year—choosing Dec 2016 as the finishing time ensures good ground-truth confidence in objects labeled as malware.

Dataset summary. The final dataset consists of 129,728 Android applications (116,993 goodware and 12,735 malware). Figure 1 reports a stack histogram showing the per-month distribution of goodware/malware in the dataset. For the sake of clarity, the figure also reports the number of malware and goodware in each bin. The training and testing splits

used in §3 are reported in Table 1; all the time-aware experiments in the remainder of this paper are performed by training on 2014 and testing on 2015 and 2016 (see the vertical dotted line in Figure 1).

3 Sources of Experimental Bias

In this section, we motivate our discussion of bias through experimentation with ALG1 [4] and ALG2 [33] (§3.1). We then detail the sources of temporal (§3.2) and spatial bias (§3.3) that affect ML-based Android malware classification.

3.1 Motivational Example

We consider a motivational example in which we vary the sources of experimental bias to better illustrate the problem. Table 1 reports the F_1-score for ALG1 and ALG2 under various experimental configurations; rows correspond to different sources of temporal experimental bias, and columns correspond to different sources of spatial experimental bias. On the left-part of Table 1, we use squares (■/■) to show from which time frame training and testing objects are taken; each square represents six months (in the window from Jan 2014 to Dec 2016). Black squares (■) denote that samples are taken from that six-month time frame, whereas periods with gray squares (■) are not used. The columns on the right part of the table correspond to different percentages of malware in the training set Tr and the testing set Ts.

Table 1 shows that both ALG1 and ALG2 perform far worse in realistic settings (bold values with green background in the last row, for columns corresponding to 10% malware in testing) than in settings similar to those presented in [4, 33] (bold values with red background). This is due to inadvertent experimental bias as outlined in the following.

Note. We clarify to which similar settings of [4, 33] we refer to in the cells with red background in Table 1. The paper of ALG2 [33] reports in the abstract performance "up to 99% F_1", which (out of the many settings they evaluate) corresponds to a scenario with 86% malware in both training and testing, evaluated with 10-fold CV; here, we rounded off to 90% malware for a cleaner presentation (we have experimentally verified that results with 86% and 90% malware-to-benign class ratio are similar). ALG1's original paper [4] relies on hold-out by performing 10 random splits (66% training and 33% testing). Since hold-out is almost equivalent to k-fold CV and suffers from the same spatio-temporal biases, for the sake of simplicity in this section we refer to a k-fold CV setting for both ALG1 and ALG2.

3.2 Temporal Experimental Bias

Concept drift is a problem that occurs in machine learning when a model becomes obsolete as the distribution of incoming data at test-time differs from that of training data, i.e., when the assumption does not hold that data is independent and identically distributed (i.i.d.) [26]. In the ML community, this problem is also known as *dataset shift* [50]. *Time decay* is the decrease in model performance over time caused by concept drift.

Concept drift in malware combined with similarities among malware within the same family causes *k-fold cross validation* (CV) to be *positively biased*, artificially inflating the performance of malware classifiers [2, 36, 37]. K-fold CV is likely to include in the training set at least one sample of each malware family in the dataset, whereas new families will be unknown at training time in a real-world deployment. The all-black squares in Table 1 for 10-fold CV refer to each training/testing fold of the 10 iterations containing at least one sample from each time frame. The use of k-fold CV is widespread in malware classification research [11, 12, 27, 31, 34, 37, 41, 49, 51, 57]; while a useful mechanism to prevent overfitting [8] or estimate the performance of a classifier in the *absence* of concept drift when the i.i.d. assumption holds (see considerations in §4.4), it has been unclear how it affects the real-world performance of machine learning techniques with non-stationary data that are affected by time decay. Here, in the first row of Table 1, we quantify the performance impact in the Android domain.

The second row of Table 1 reports an experiment in which a classifier's ability to detect past objects is evaluated [2, 33]. Although this characteristic is important, high performance should be expected from a classifier in such a scenario: if the classifier contains at least one variant of a past malware family, it will likely identify similar variants. We thus believe that experiments on the performance achieved on the detection of past malware can be misleading; the community should focus on building malware classifiers that are robust against time decay.

In the third row, we identify a novel temporal bias that occurs when goodware and malware correspond to different time periods, often due to having originated from different data sources (e.g., in [33]). The black and gray squares in Table 1 show that, although malware testing objects are posterior to malware training objects, the goodware/malware time windows do not overlap; in this case, the classifier may learn to distinguish applications from different time periods, rather than goodware from malware—again leading to artificially high performance. For instance, spurious features such as new API methods may be able to strongly distinguish objects simply because malicious applications predate that API.

The last row of Table 1 shows that the realistic setting, where training is temporally precedent to testing, causes the worst classifier performance in the majority of cases. We present decay plots and a more detailed discussion in §4.

3.3 Spatial Experimental Bias

We identify two main types of spatial experimental bias based on assumptions on percentages of malware in testing and training sets. All experiments in this section assume temporal consistency. The model is trained on 2014 and tested on 2015 and 2016 (last row of Table 1) to allow the analysis of spatial

			% mw in testing set Ts							
			10% (*realistic*)				90% (*unrealistic*)			
			% mw in training set Tr				% mw in training set Tr			
	Sample dates		10%	90%	10%	90%	10%	90%	10%	90%
Experimental setting	Training	Testing	ALG1 [4]		ALG2 [33]		ALG1 [4]		ALG2 [33]	
10-fold CV	gw: ■■■■■■ mw: ■■■■■■	gw: ■■■■■■ mw: ■■■■■■	**0.91**	0.56	0.83	0.32	0.94	0.98	0.85	**0.97**
Temporally inconsistent	gw: ■■■■■■ mw: ■■■■■■	gw: ■■■■■■ mw: ■■■■■■	0.76	0.42	0.49	0.21	0.86	0.93	0.54	**0.95**
Temporally inconsistent gw/mw windows	gw: ■■■■■■ mw: ■■■■■■	gw: ■■■■■■ mw: ■■■■■■	0.77	0.70	0.65	0.56	0.79	0.94	0.65	**0.93**
Temporally consistent (*realistic*)	gw: ■■■■■■ mw: ■■■■■■	gw: ■■■■■■ mw: ■■■■■■	**0.58**	**0.45**	**0.32**	**0.30**	0.62	0.94	0.33	0.96

Table 1: F_1-Score results that show impact of spatial (in columns) and temporal (in rows) experimental bias. Values with red backgrounds are experimental results of (unrealistic) settings similar to those considered in papers of ALG1 [4] and ALG2 [33]; values with green background (last row) are results in the realistic settings we identify. The dataset consists of three years (§2.3), and each square on the left part of the table represents a six month time-frame: if training (resp. testing) objects are sampled from that time frame, we use a black square (■); if not, we use a gray square (■).

bias without the interference of temporal bias.

Spatial experimental bias in testing. The percentage of malware in the testing distribution needs to be estimated (§2.2) and *cannot* be changed, if one wants results to be representative of in-the-wild deployment of the malware classifier. To understand why this leads to biased results, we artificially vary the testing distribution to illustrate our point. Figure 2 reports performance (F_1-Score, Precision, Recall) for increasing the percentage of malware during testing on the X-axis. We change the percentage of malware in the testing set by randomly downsampling goodware, so the number of malware remains fixed throughout the experiments.[2] For completeness, we report the two training settings from Table 1 with 10% and 90% malware, respectively.

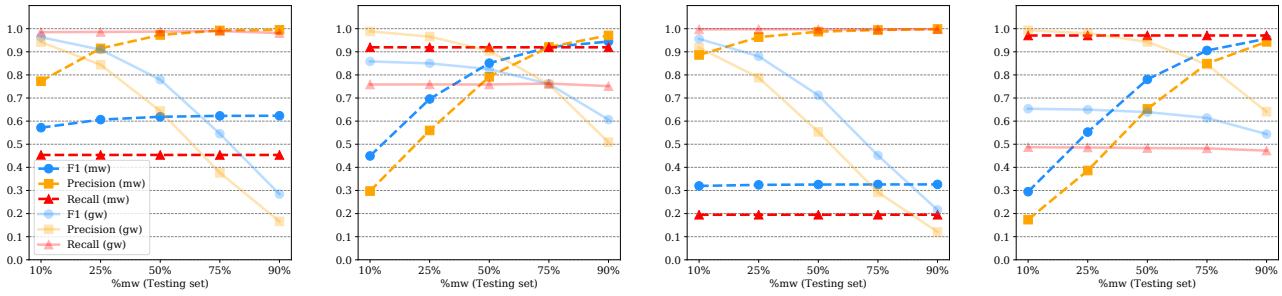
Let us first focus on the malware performance (dashed lines). All plots in Figure 2 exhibit constant Recall, and increasing Precision for increasing percentage of malware in the testing. Precision for the malware (mw) class—the positive class—is defined as $P_{mw} = {}^{TP}/_{(TP+FP)}$ and Recall as $R_{mw} = {}^{TP}/_{(TP+FN)}$. In this scenario, we can observe that TPs (i.e., malware objects correctly classified as malware) and FNs (i.e., malware objects incorrectly classified as goodware) do not change, because the number of malware does not increase; hence, Recall remains stable. The increase in number of FPs (i.e., goodware objects misclassified as malware) decreases as we reduce the number of goodware in the dataset; hence, Precision improves. Since the F_1-Score is the harmonic mean of Precision and Recall, it goes up with Precision. We also observe that, inversely, the Precision for the goodware (gw) class—the negative class—$P_{gw} = {}^{TN}/_{(TN+FN)}$ decreases (see yellow solid lines in Figure 2), because we are reducing the TNs while the FNs do not change. This example shows how considering an unrealistic testing distribution with more malware than goodware in this context (§2.2) positively inflates Precision and hence the F_1-Score of the malware classifier.

[2]We choose to downsample goodware to achieve up to 90% of malware (mw) for testing because of the computational and storage resources required to achieve such a ratio by oversampling. This does not alter the conclusions of our analysis. Let us assume a scenario in which we keep the same number of goodware (gw), and increase the percentage of mw in the dataset by oversampling mw. The precision ($P_{mw} = {}^{TP}/_{(TP+FP)}$) would increase because TPs would increase for any mw detection, and FPs would not change—because the number of gw remains the same; if training (resp. testing) observations are sampled from a distribution similar to the mw in the original dataset (e.g., new training mw is from 2014 and new testing mw comes from 2015 and 2016), then Recall ($R_{mw} = {}^{TP}/_{(TP+FN)}$) would be stable—it would have the same proportions of TPs and FNs because the classifier will have a similar predictive capability for finding mw. Hence, if the number of mw in the dataset increases, the F_1-Score would increase as well, because Precision increases while Recall remains stable.

Spatial experimental bias in training. To understand the impact of altering malware-to-goodware ratios in training, we now consider a motivating example with a linear SVM in a 2D feature space, with features x_1 and x_2. Figure 3 reports three scenarios, all with the same 10% malware in testing, but with 10%, 50%, and 90% malware in training.

We can observe that with an increasing percentage of malware in training, the hyperplane moves towards goodware. More formally, it improves Recall of malware while reducing its Precision. The opposite is true for goodware. To minimize the overall error rate $Err = {}^{(FP+FN)}/_{(TP+TN+FP+FN)}$ (i.e., maximize Accuracy), one should train the dataset with the same distribution that is expected in the testing. However, in this scenario one may have more interest in finding objects of the minority class (e.g., "more malware") by improving Recall subject to a constraint on maximum FPR.

Figure 4 shows the performance for ALG1 and ALG2, for increasing percentages of malware in training on the X-axis; just for completeness (since one cannot artificially change the test distribution to achieve realistic evaluations), we report results both for 10% mw in testing and for 90% malware in testing, but we remark that in the Android setting we have estimated 10% mw in the wild (§2.2). These plots confirm the trend in our motivating example (Figure 3), that is, R_{mw} increases but P_{mw} decreases. For the plots with 10% mw in



(a) ALG1: Train with 10% mw (b) ALG1: Train with 90% mw (c) ALG2: Train with 10% mw (d) ALG2: Train with 90% mw

Figure 2: *Spatial experimental bias in testing*. Training on 2014 and testing on 2015 and 2016. For increasing % of malware in the testing (unrealistic setting), Precision for malware increases and Recall remains the same; overall, F_1-Score increases for increasing percentage of malware in the testing. However, having more malware than goodware in testing does not reflect the in-the-wild distribution of 10% malware (§2.2), so the setting with more malware is unrealistic and reports biased results.

testing, we observe there is a point in which F_1-Score$_{mw}$ is maximum while the error for the gw class is within 5%.

In §4.3, we propose a novel algorithm to improve the performance of the malware class according to the objective of the user (high Precision, Recall or F_1-Score), subject to a maximum tolerated error. Moreover, in §4 we introduce constraints and metrics to guarantee bias-free evaluations, while revealing counter-intuitive results.

4 Space-Time Aware Evaluation

We now formalize how to perform an evaluation of an Android malware classifier free from spatio-temporal bias. We define a novel set of constraints that must be followed for realistic evaluations (§4.1); we introduce a novel time-aware metric, AUT, that captures in one number the impact of time decay on a classifier (§4.2); we propose a novel tuning algorithm that empirically optimizes a classifier performance, subject to a maximum tolerated error (§4.3); finally, we introduce TESSERACT and provide counter-intuitive results through unbiased evaluations (§4.4). To improve readability, we report in Appendix A.2 a table with all the major symbols used in the remainder of this paper.

4.1 Evaluation Constraints

Let us consider D as a labeled dataset with two classes: malware (positive class) and goodware (negative class). Let us define $s_i \in D$ as an *object* (e.g., Android app) with timestamp $time(s_i)$. To evaluate the classifier, the dataset D must be split into a training dataset Tr with a time window of size W, and a testing dataset Ts with a time window of size S. Here, we consider $S > W$ in order to estimate long-term performance and robustness to decay of the classifier. A user may consider different time splits depending on his objectives, provided each split has a significant number of samples. We emphasize that, although we have the labels of objects in $Ts \subseteq D$, all the evaluations and tuning algorithms *must* assume that labels y_i of objects $s_i \in Ts$ are unknown.

To evaluate performance over time, the test set Ts must be split into time-slots of size Δ. For example, for a testing set time window of size $S = 2$ years, we may have $\Delta = 1$ month. This parameter is chosen by the user, but it is important that the chosen granularity allows for a statistically significant number of objects in each test window $[t_i, t_i + \Delta)$.

We now formalize three constraints that must be enforced when dividing D into Tr and Ts for a realistic setting that avoids spatio-temporal experimental bias (§3). While C1 was proposed in past work [2, 36], we are the first to propose C2 and C3—which we show to be fundamental in §4.4.

C1) Temporal training consistency. All the objects in the training must be *strictly* temporally precedent to the testing ones:

$$time(s_i) < time(s_j), \forall s_i \in Tr, \forall s_j \in Ts \tag{1}$$

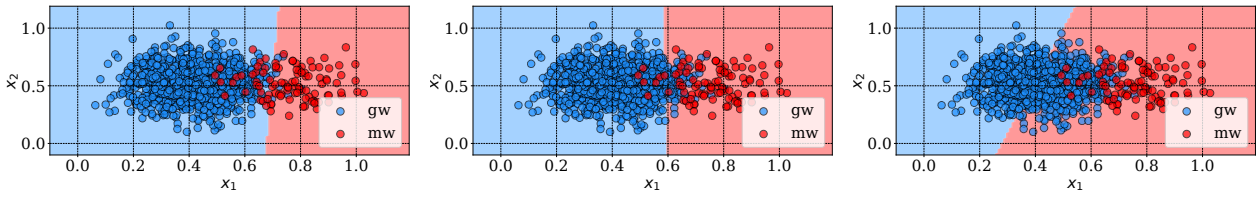
where s_i (resp. s_j) is an object in the training set Tr (resp. testing set Ts). Eq. 1 must hold; its violation inflates the results by including future knowledge in the classifier (§3.2).

C2) Temporal gw/mw windows consistency. In every testing slot of size Δ, all test objects must be from the same time window:

$$t_i^{min} \leq time(s_k) \leq t_i^{max}, \quad \forall s_k \text{ in time slot } [t_i, t_i + \Delta) \tag{2}$$

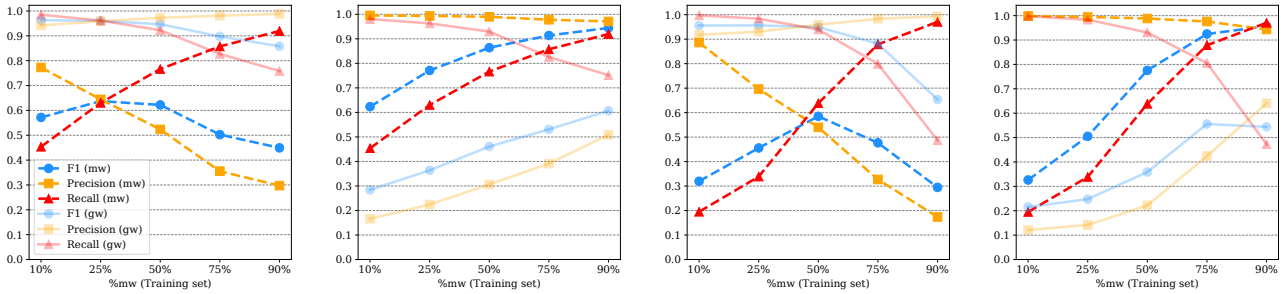
where $t_i^{min} = \min_k time(s_k)$ and $t_i^{max} = \max_k time(s_k)$. The same should hold for the training: although violating Eq. 2 in the training data does not bias the evaluation, it may affect the sensitivity of the classifier to unrelated artifacts. Eq. 2 has been violated in the past when goodware and malware have been collected from different time windows (e.g., ALG2 [33], re-evaluated in §4.4)—if violated, the results are biased because the classifier may learn and test on artifactual behaviors that, for example, distinguish goodware from malware just by their different API versions.

C3) Realistic malware-to-goodware ratio in testing. Let us define φ as the average percentage of malware in training



(a) Training 10% mw; Testing 10% mw (b) Training 50% mw; Testing 10% mw (c) Training 90% mw; Testing 10% mw

Figure 3: Motivating example for the intuition of *spatial experimental bias in training* with Linear-SVM and two features, x_1 and x_2. The training changes, but the testing points are fixed: 90% gw and 10% mw. When the % of malware in the training increases, the decision boundary moves towards the goodware class, improving Recall for malware but decreasing Precision.



(a) ALG1: Test with 10% mw (b) ALG1: Test with 90% mw (c) ALG2: Test with 10% mw (d) ALG2: Test with 90% mw

Figure 4: *Spatial experimental bias in training.* Training on 2014 and testing on 2015 and 2016. For increasing % of malware in the training, Precision decreases and Recall increases, for the motivations illustrated in the example of Figure 3. In §4.3, we devise an algorithm to find the best training configuration for optimizing Precision, Recall, or F_1 (depending on user needs).

data, and δ as the average percentage of malware in the testing data. Let $\hat{\sigma}$ be the estimated percentage of malware in the wild. To have a realistic evaluation, the average percentage of malware in the testing (δ) must be as close as possible to the percentage of malware in the wild ($\hat{\sigma}$), so that:

$$\delta \simeq \hat{\sigma} \tag{3}$$

For example, we have estimated that in the Android scenario goodware is predominant over malware, with $\hat{\sigma} \approx 0.10$ (§2.2). If C3 is violated by overestimating the percentage of malware, the results are positively inflated (§3.3). We highlight that, although the testing distribution δ cannot be changed (in order to get realistic results), the percentage of malware in the training φ may be tuned (§4.3).

4.2 Time-aware Performance Metrics

We introduce a time-aware performance metric that allows for the comparison of different classifiers while considering time decay. Let Θ be a classifier trained on Tr; we capture the performance of Θ for each time frame $[t_i, t_i + \Delta)$ of the testing set Ts (e.g., each month). We identify two options to represent per-month performance:

- **Point estimates** (*pnt*): The value plotted on the Y-axis for $x_k = k\Delta$ (where k is the test slot number) computes the performance metric (e.g., F_1-Score) only based on predictions $\hat{y}_i$ of Θ and true labels y_i in the interval $[W + (k-1)\Delta, W + k\Delta)$.
- **Cumulative estimates** (*cml*): The value plotted on the Y-axis for $x_k = k\Delta$ (where k is the test slot number) computes the performance metric (e.g., F_1-Score) only based on predictions $\hat{y}_i$ of Θ and true labels y_i in the cumulative interval $[W, W + k\Delta)$.

Point estimates are always to be preferred to represent the real performance of an algorithm. The cumulative estimates can be used to highlight a smoothed trend and to show overall performance up to a certain point, but can be misleading if reported on their own if objects are too sparsely distributed in some test slots Δ. Hence, we report only point estimates in the remainder of the paper (e.g., in §4.4), while an example of cumulative estimate plots is reported in Appendix A.3.

To facilitate the comparison of different time decay plots, we define a new metric, *Area Under Time* (**AUT**), the area under the performance curve over time. Formally, based on the trapezoidal rule (as in AUROC [8]), AUT is defined as follows:

$$AUT(f, N) = \frac{1}{N-1} \sum_{k=1}^{N-1} \frac{[f(x_{k+1}) + f(x_k)] \cdot (1/N)}{2} \tag{4}$$

where: $f(x_k)$ is the value of the point estimate of the performance metric f (e.g., F_1) evaluated at point $x_k := (W + k\Delta)$; N is the number of test slots, and $1/(N-1)$ is a normalization factor so that AUT $\in [0, 1]$. The perfect classifier with robustness to time decay in the time window S has AUT $= 1$. By

default, AUT is computed as the area under point estimates, as they capture the trend of the classifier over time more closely; if the AUT is computed on cumulative estimates, it should be explicitly marked as AUT_{cml}. As an example, $\text{AUT}(F_1, 12m)$ is the point estimate of F_1-Score considering time decay for a period of 12 months, with a 1-month interval. We highlight that the simplicity of computing the AUT should be seen as a benefit rather than a drawback; it is a simple yet effective metric that captures the performance of a classifier with respect to time decay, de-facto promoting a fair comparison across different approaches.

> **AUT**(f,N) is a metric that allows us to evaluate performance f of a malware classifier against time decay over N time units in realistic experimental settings—obtained by enforcing C1, C2, and C3 (§4.1). The next sections leverage AUT for tuning classifiers and comparing different solutions (§4.4).

4.3 Tuning Training Ratio

We propose a novel algorithm that allows for the adjustment of the training ratio φ when the dataset is imbalanced, in order to optimize a user-specified performance metric (F_1, Precision, or Recall) on the minority class, subject to a maximum tolerated error, while aiming to reduce time decay. The high-level intuition of the impact of changing φ is described in §3.3. We also observe that ML literature has shown ROC curves to be misleading on highly imbalanced datasets [14, 25]. Choosing different thresholds on ROC curves *shifts* the decision boundary, but (as seen in the motivating example of Figure 3) re-training with different ratios φ (as in our algorithm) also changes the *shape* of the decision boundary, better representing the minority class.

Our tuning algorithm is inspired by one proposed by Weiss and Provost [55]; they propose a progressive sampling of training objects to collect a dataset that improves AUROC performance of the minority class in an imbalanced dataset. However, they did not take temporal constraints into account (§3.2), and heuristically optimize only AUROC. Conversely, we enforce C1, C2, C3 (§4.1), and rely on AUT to achieve three possible targets for the malware class: higher F_1-Score, higher Precision, or higher Recall. Also, we assume that the user already has a training dataset Tr and wants to use as many objects from it as possible, while still achieving a good performance trade-off; for this purpose, we perform a *progressive subsampling* of the goodware class.

Algorithm 1 formally presents our methodology for tuning the parameter φ to find the value $\varphi^*_{\mathbb{P}}$ that optimizes $\mathbb{P}$ subject to a maximum error rate E_{max}. The algorithm aims to solve the following optimization problem:

$$\text{maximize}_{\varphi}\{\mathbb{P}\} \quad \text{subject to: } E \leq E_{max} \tag{5}$$

where $\mathbb{P}$ is the target performance: the F_1-Score (F_1), Precision (Pr) or Recall (Rec) of the malware class; E_{max} is the maximum tolerated error; depending on the target $\mathbb{P}$, the error rate E has a different formulation:

- if $\mathbb{P} = F_1 \rightarrow E = 1 - \text{Acc} = {}^{(FP+FN)}/_{(TP+TN+FP+FN)}$
- if $\mathbb{P} = Rec \rightarrow E = FPR = {}^{FP}/_{(TN+FP)}$
- if $\mathbb{P} = Pr \rightarrow E = FNR = {}^{FN}/_{(TP+FN)}$

Each of these definitions of E is targeted to limit the error induced by the specific performance—if we want to maximize F_1 for the malware class, we need to limit both FPs and FNs; if $\mathbb{P} = Pr$, we increase FNs, so we constrain FNR.

Algorithm 1 consists of two phases: *initialization* (lines 1–5) and *grid search* of $\varphi^*_{\mathbb{P}}$ (lines 6–14). In the initialization phase, the training set Tr is split into a proper training set $ProperTr$ and a validation set Val; this is split according to the space-time evaluation constraints in §4.1, so that all the objects in $ProperTr$ are temporally anterior to Val, and the malware percentage δ in Val is equal to $\hat{\sigma}$, the in-the-wild malware percentage. The maximum performance observed P^* and the optimal training ratio $\varphi^*_{\mathbb{P}}$ are initialized by assuming the estimated in-the-wild malware ratio $\hat{\sigma}$ for training; in Android, $\hat{\sigma} \approx 10\%$ (see §2.2).

The grid-search phase iterates over different values of φ, with a learning rate μ (e.g., $\mu = 0.05$), and keeps as $\varphi^*_{\mathbb{P}}$ the value leading to the best performance, subject to the error constraint. To reduce the chance of discarding high-quality points while downsampling goodware, we prioritize the most uncertain points (e.g., points close to the decision boundary in an SVM) [46]. The constraint on line 6 ($\hat{\sigma} \leq \varphi \leq 0.5$) is to ensure that one does not under-represent the minority class (if $\varphi < \hat{\sigma}$) and that one does not let it become the majority class (if $\varphi > 0.5$); also, from §3.3 it is clear that if $\varphi > 0.5$, then the error rate becomes too high for the goodware class. Finally, the grid-search explores multiple values of φ and stores the best ones. To capture time-aware performance, we rely on AUT (§4.2), and the error rate is computed according to the target $\mathbb{P}$ (see above). Tuning examples are in §4.4.

4.4 TESSERACT: Revealing Hidden Performance

Here, we show how our methodology can reveal hidden performance of ALG1 [4], ALG2 [33], and DL [22] (§2.1), and their robustness to time decay.

We develop TESSERACT as an open-source Python framework that enforces constraints C1, C2, and C3 (§4.1), computes AUT (§4.2), and can train a classifier with our tuning algorithm (§4.3). TESSERACT operates as a traditional Python ML library but, in addition to features matrix X and labels y, it also takes as input the timestamp array t containing dates for each object. Details about TESSERACT's implementation and generality are in §A.5.

Figure 5 reports several performance metrics of the three algorithms as point estimates over time. The X-axis reports the testing slots in months, whereas the Y-axis reports different scores between 0 and 1. The areas highlighted in blue correspond to the $\text{AUT}(F_1, 24m)$. The black dash-dotted horizontal lines represent the best F_1 from the original papers [4, 22, 33],

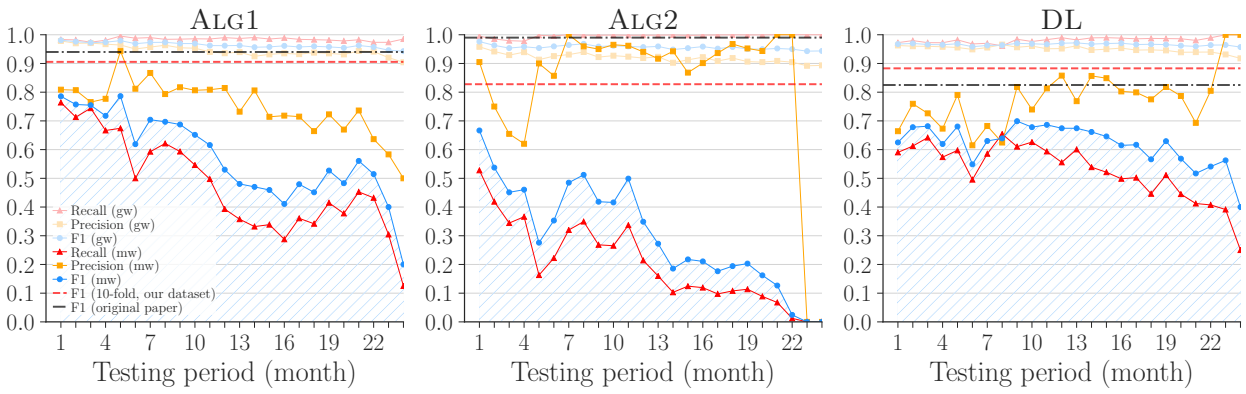


Figure 5: Time decay of ALG1 [4], ALG2 [33] and DL [22]—with AUT$(F_1, 24m)$ of 0.58, 0.32 and 0.64, respectively. Training and test distribution both have 10% malware. The drop in the last 3 months is also related to lower samples in the dataset.

Algorithm 1: Tuning φ.

Input: Training dataset Tr
Parameters: Learning rate μ, target performance $\mathbb{P} \in \{F_1, Pr, Rec\}$, max error rate E_{max}
Output: $\varphi^*_{\mathbb{P}}$, optimal percentage of mw to use in training to achieve the best target performance $\mathbb{P}$ subject to $E < E_{max}$.

Split the training set Tr into two subsets: actual training (*ProperTr*) and validation set (*Val*), while enforcing C1, C2, C3 (§4.1), also implying $\delta = \hat{\sigma}$
Divide *Val* into N non-overlapped subsets, each corresponding to a time-slot Δ, so that $Val_{array} = [V_0, V_1, ..., V_N]$
Train a classifier Θ on *ProperTr*
$P^* \leftarrow$ AUT($\mathbb{P}$,N) on Val_{array} with Θ
$\varphi^*_{\mathbb{P}} = \hat{\sigma}$
for $(\varphi = \hat{\sigma};\ \varphi \leq 0.5;\ \varphi = \varphi + \mu)$ **do**
    Downsample gw in *ProperTr* so that percentage of mw is φ
    Train the classifier Θ_φ on *ProperTr* with φ mw
    performance $P_\varphi \leftarrow$ AUT($\mathbb{P}$, N) on Val_{array} with Θ_φ
    error $E_\varphi \leftarrow$ Error rate on Val_{array} with Θ_φ
    if $(P_\varphi > P^*)$ **and** $(E_\varphi \leq E_{max})$ **then**
        $P^* \leftarrow P_\varphi$
        $\varphi^*_{\mathbb{P}} \leftarrow \varphi$
return $\varphi^*_{\mathbb{P}}$;

corresponding to results obtained with 10 hold-out random splits for ALG1, 10-fold CV for ALG2, and random split for DL; all these settings are analogous to k-fold from a temporal bias perspective, and violate both C1 and C2. The red dashed horizontal lines correspond to 10-fold F_1 obtained on our dataset, which satisfies C3.

Differences in 10-fold F_1. We discuss and motivate the differences between the horizontal lines representing original papers' best F_1 and replicated 10-fold F_1. The 10-fold F_1 of ALG1 is close to the original paper [4]; the difference is likely related to the use of a different, more recent dataset. The 10-fold F_1 of ALG2 is much lower than the one in the paper. We verified that this is mostly caused by **violating C3**: the best F_1 reported in [33] is on a setting with 86% malware—hence, spatial bias increases even 10-fold F_1 of ALG2. Also **violating C2** tends to inflate the 10-fold performance as the classifier may learn artifacts. The 10-fold F_1 in DL is instead slightly higher than in the original paper [22]; this is likely related to a hyperparameter tuning in the original paper that optimized Accuracy (instead of F_1), which is known to be misleading in imbalanced datasets. Details on hyperparameters chosen are in §A.1. From these results, we can observe that even if an analyst wants to estimate what the performance of the classifier would be in the *absence* of concept drift (i.e., where objects coming from the same distribution of the training dataset are received by the classifier), she still needs to enforce C2 and C3 while computing 10-fold CV to obtain valid results.

Violating C1 and C2. Removing the temporal bias reveals the real performance of each algorithm in the presence of concept drift. The AUT$(F_1, 24m)$ quantifies such performance: 0.58 for ALG1, 0.32 for ALG2 and 0.64 for DL. In all three scenarios, the AUT$(F_1, 24m)$ is lower than 10-fold F_1 as the latter violates constraint C1 and may violate C2 if the dataset classes are not evenly distributed across the timeline (§4).

Best performing algorithm. TESSERACT shows a counter-intuitive result: the algorithm that is most robust to time decay and has the highest performance over the 2 years testing is the DL algorithm (after removing space-time bias), although for the first few months ALG1 outperforms DL. Given this outcome, one may prefer to use ALG1 for the first few months and then DL, if retraining is not possible (§5). We observe that this strongly contradicts the performance obtained in the presence of temporal and spatial bias. In particular, if we only looked at the best F_1 reported in the original papers, ALG2 would have been the best algorithm (because spatial bias was present). After enforcing C3, the k-fold on our dataset would have suggested that DL and ALG1 have similar performance (because of temporal bias). After enforcing C1, C2 and C3, the AUT reveals that DL is actually the algorithm most robust to time decay.

Different robustness to time decay. Given a training dataset, the robustness of different ML models against perfor-

mance decay over time depends on several factors. Although more in-depth evaluations would be required to understand the theoretical motivations behind the different robustness to time decay of the three algorithms in our setting, we hereby provide insights on possible reasons. The performance of ALG2 is the fastest to decay likely because its feature engineering [33] may be capturing relations in the training data that quickly become obsolete at test time to separate goodware from malware. Although ALG1 and DL take as input the same feature space, the higher robustness to time decay of DL is likely related to feature representation in the *latent feature space* automatically identified by deep learning [19], which appears to be more robust to time decay in this specific setting. Recent results have also shown that linear SVM tends to overemphasize a few important features [35]—which are the few most effective on the training data, but may become obsolete over time. We remark that we are *not* claiming that deep learning is always more robust to time decay than traditional ML algorithms. Instead, we demonstrate how, in this specific setting, TESSERACT allowed us to highlight higher robustness of DL [22] against time decay; however, the prices to pay to use DL are lower explainability [23, 42] and higher training time [19].

Tuning algorithm. We now evaluate whether our tuning (Algorithm 1 in §4.3) improves robustness to time decay of a malware classifier for a given target performance. We first aim to maximize $\mathbb{P} = F_1$-Score of malware class, subject to E_{max}= 10%. After running Algorithm 1 on ALG1 [4], ALG2 [33] and DL, we find that $\varphi^*_{F_1} = 0.25$ for ALG1 and DL, and $\varphi^*_{F_1} = 0.5$ for ALG2. Figure 6 reports the improvement on the test performance of applying $\varphi^*_{F_1}$ to the full training set Tr of 1 year. We remark that the choice of $\varphi^*_{F_1}$ uses only training information (see Algorithm 1) and no test information is used—the optimal value is chosen from a 4-month validation set extracted from the 1 year of training data; this is to simulate a realistic deployment setting in which we have no a priori information about testing. Figure 6 shows that our approach for finding the best $\varphi^*_{F_1}$ improves the F_1-Score on malware at test time, at the cost of slightly reduced goodware performance. Table 2 shows details of how total FPs, total FNs, and AUT changed by training ALG1, ALG2, and DL with $\varphi^*_{F_1}$, φ^*_{Prec}, and φ^*_{Rec} instead of $\hat{\sigma}$. These training ratios have been computed subject to $E_{max} = 5\%$ for φ^*_{Rec}, $E_{max} = 10\%$ for $\varphi^*_{F_1}$, and $E_{max} = 15\%$ for φ^*_{Prec}; the difference in the maximum tolerated errors is motivated by the class imbalance in the dataset—which causes lower FPR and higher FNR values (see definitions in §4.3), as there are many more goodware than malware. As expected (§3.3), Table 2 shows that when training with $\varphi^*_{F_1}$ Precision decreases (FPs increase) but Recall increases (because FNs decrease), and the overall AUT increases slightly as a trade-off. A similar reasoning follows for the other performance targets. We observe that the AUT for Precision may slightly differ even with a similar number of total FPs—this is because AUT$(Pr, 24m)$ is sensitive to the time at which FPs occur; the same observation is valid for total FNs and AUT Recall. After tuning, the F_1 performance of ALG1 and DL become similar, although DL remains higher in terms of AUT. The tuning improves the AUT$(F_1, 24m)$ of DL only marginally, as DL is already robust to time decay even before tuning (Figure 5).

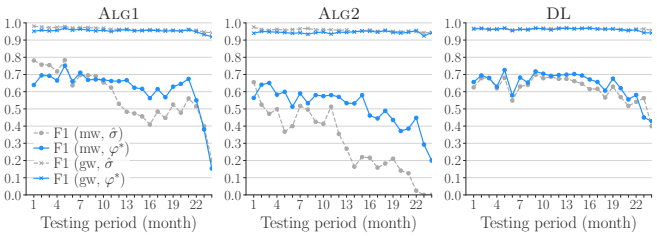


Figure 6: Tuning improvement obtained by applying $\varphi^*_{F_1} = 25\%$ to ALG1 and DL, and $\varphi^*_{F_1} = 50\%$ to ALG2. The values of $\varphi^*_{F_1}$ are obtained with Algorithm 1 and one year of training data (trained on 8 months and validated on 4 months).

Algorithm	φ	FP	FN	AUT($\mathbb{P}$,24m)		
				F_1	Pr	Rec
ALG1 [4]	10% ($\hat{\sigma}$)	965	3,851	0.58	0.75	0.48
	25% ($\varphi^*_{F_1}$)	2,156	2,815	0.62	0.65	0.61
	10% (φ^*_{Pr})	965	3,851	0.58	0.75	0.48
	50% (φ^*_{Rec})	3,728	1,793	0.64	0.58	0.74
ALG2 [33]	10% ($\hat{\sigma}$)	274	5,689	0.32	0.77	0.20
	50% ($\varphi^*_{F_1}$)	4,160	2,689	0.53	0.50	0.60
	10% (φ^*_{Pr})	274	5,689	0.32	0.77	0.20
	50% (φ^*_{Rec})	4,160	2,689	0.53	0.50	0.60
DL [22]	10% ($\hat{\sigma}$)	968	3,291	0.64	0.78	0.53
	25% ($\varphi^*_{F_1}$)	2,284	2,346	0.65	0.66	0.65
	10% (φ^*_{Pr})	968	3,291	0.64	0.78	0.53
	25% (φ^*_{Rec})	2,284	2,346	0.65	0.66	0.65

Table 2: Testing AUTs performance over 24 months when training with $\hat{\sigma}$, $\varphi^*_{F_1}$, φ^*_{Pr} and φ^*_{Rec}.

The next section focuses on the two classifiers less robust to time decay, ALG1 and ALG2, to evaluate with TESSERACT the performance-cost trade-offs of budget-constrained strategies for delaying time decay.

5 Delaying Time Decay

We have shown how enforcing constraints and computing AUT with TESSERACT can reveal the real performance of Android malware classifiers (§4.4). This *baseline AUT performance* (without retraining) allows users to evaluate the general robustness of an algorithm to time decay. A classifier may be retrained to update its model. However, *manual labeling* is costly (especially in the Android malware setting), and the ML community [6, 46] has worked extensively on mitigation strategies—e.g., to identify a limited number of *best* objects to label (active learning). While effective at postponing time decay, strategies like these can further complicate the fair evaluation and comparison of classifiers.

In this section, we show how TESSERACT can be used

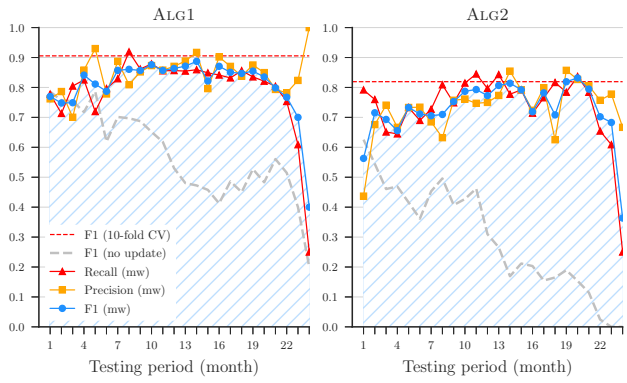


Figure 7: Delaying time decay: incremental retraining.

Delay method	Costs L ALG1	Costs L ALG2	Costs Q ALG1	Costs Q ALG2	Performance P: AUT(F_1,24m) $\varphi=\hat{\sigma}$ ALG1	$\varphi=\hat{\sigma}$ ALG2	$\varphi=\varphi^*_{F_1}$ ALG1	$\varphi=\varphi^*_{F_1}$ ALG2
No update	0	0	0	0	0.577	0.317	0.622	0.527
Rejection ($\hat{\sigma}$)	0	0	10,283	3,595	0.717	0.280	–	–
Rejection ($\varphi^*_{F_1}$)	0	0	10,576	24,390	–	–	0.704	0.683
AL: 1%	709	709	0	0	0.708	0.456	0.703	0.589
AL: 2.5%	1,788	1,788	0	0	0.738	0.509	0.758	0.667
AL: 5%	3,589	3,589	0	0	0.782	0.615	0.784	0.680
AL: 7.5%	5,387	5,387	0	0	0.793	0.641	0.801	0.714
AL: 10%	7,189	7,189	0	0	0.796	0.656	0.802	0.732
AL: 25%	17,989	17,989	0	0	0.821	0.674	0.823	0.732
AL: 50%	35,988	35,988	0	0	0.817	0.679	0.828	0.741
Inc. retrain	71,988	71,988	0	0	0.818	0.679	0.830	0.736

Table 3: Performance-cost comparison of delay methods.

to compare and evaluate the trade-offs of different budget-constrained strategies to delay time decay. Since DL has shown to be more robust to time decay (§4.4) than ALG1 and ALG2, in this section we focus our attention these to show performance-cost trade-offs of different mitigations.

5.1 Delay Strategies

We do not propose novel delay strategies, but instead focus on how TESSERACT allows for the comparison of some popular approaches to mitigating time decay. This shows researchers how to adopt TESSERACT for the fair comparison of different approaches when proposing novel solutions to delaying time decay under budget constraints. We now summarize the delay strategies we consider and show results on our dataset. For interested readers, we include additional background knowledge on these strategies in §A.4.

Incremental retraining. We first consider an approach that represents an ideal upper bound on performance, where *all* points are included in retraining every month. This is likely unrealistic as it requires continuously labeling all the objects. Even assuming a reliance on VirusTotal, there is still an API usage cost associated with higher query rates and the approach may be ill-suited in other security domains. Figure 7 shows the performance of ALG1 and ALG2 with monthly incremental retraining.

Active learning. Active Learning (AL) strategies investigate how to select a subset of test objects (with unknown labels) that, if manually labeled and included in the training set, should be the most valuable for updating the classification model [46]. Here, we consider the most popular AL query strategy, *uncertainty sampling*, in which the points with the most uncertain predictions are selected for retraining, under the intuition that they are the most relevant to adjust decision boundaries. Figure 8 reports the active learning results obtained with uncertainty sampling, for different percentages of objects labeled per month. We observe that even with 1% AL, the performance already improves significantly.

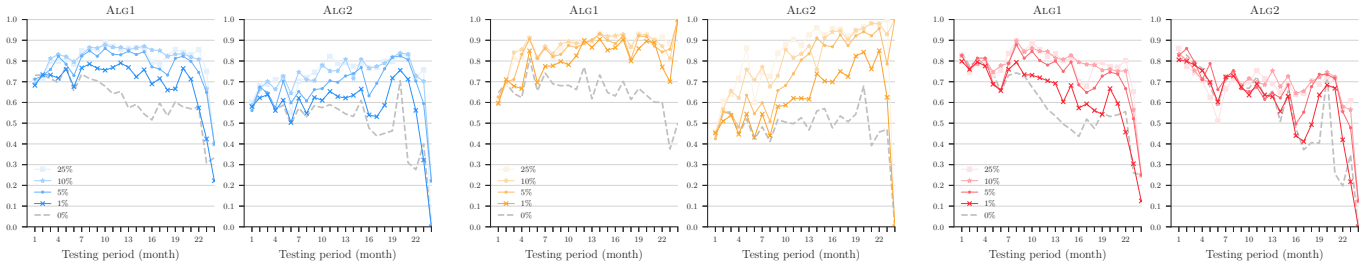
Classification with rejection. Another mitigation strategy involves rejecting a classifier's decision as "low confidence" and delaying the decision to a future date [6]. This isolates the rejected objects to a *quarantine* area which will later require manual inspection. Figure 9 reports the performance of ALG1 and ALG2 after applying a reject option based on [26]. In particular, we use the third quartile of probabilities of incorrect predictions as the rejection threshold [26]. The gray histograms in the background report the number of rejected objects per month. The second year of testing has more rejected objects for both ALG1 and ALG2, although ALG2 overall rejects more objects.

5.2 Analysis of Delay Methods

To quantify performance-cost trade-offs of methods to delay time decay without changing the algorithm, we characterize the following three elements: **Performance** (P), the performance measured in terms of AUT to capture robustness against time decay (§4.2); **Labeling Cost** (L), the number of testing objects (if any) that must be labeled—the labeling must occur periodically (e.g., every month), and is particularly costly in the malware domain as manual inspection requires many resources (infrastructure, time, expertise, etc)—for example, Miller et al. [36] estimated that an average company could manually label 80 objects per day; **Quarantine Cost** (Q), the number of objects (if any) rejected by the classifier—these must be manually verified, so there is a cost for leaving them in quarantine.

Table 3, utilizing AUT(F_1,24m) while enforcing our constraints, reports a summary of labeling cost L, quarantine cost Q, and two performance columns P, corresponding to training with $\hat{\sigma}$ and $\varphi^*_{F_1}$ (§4.3), respectively. In each row, we highlight in purple cells (resp. orange) the column with the highest AUT for ALG2 (resp. ALG1). Table 3 allows us to: (i) examine the effectiveness of the training ratios $\varphi^*_{F_1}$ and $\hat{\sigma}$; (ii) analyze the AUT performance improvement and the corresponding costs for delaying time decay; (iii) compare the performance of ALG1 and ALG2 in different settings.

First, let us compare $\varphi^*_{F_1}$ with $\hat{\sigma}$. The first row of Table 3 represents the scenario in which the model is trained only once at the beginning—the scenario for which we originally



(a) F_1-Score (b) Precision (c) Recall

Figure 8: Delay time decay: performance with active learning based on uncertainty sampling.

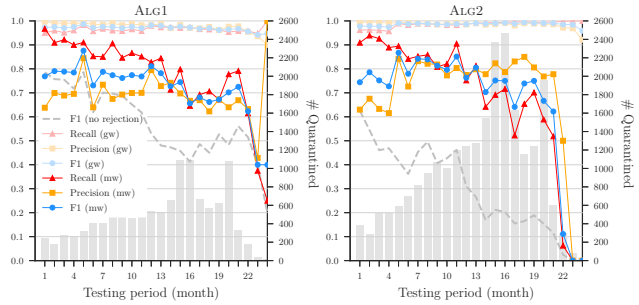


Figure 9: Delay time decay: classification with rejection.

designed Algorithm 1 (§4.3 and Figure 6). Without methods to delay time decay, $\varphi^*_{F_1}$ achieves better performance than $\hat{\sigma}$ for both ALG1 and ALG2 at no cost. In all other configurations, we observe that training $\varphi = \varphi^*_{F_1}$ always improves performance for ALG2, whereas for ALG1 it is slightly advantageous in most cases except for rejection and AL 1%—in general, the performance of ALG1 trained with $\varphi^*_{F_1}$ and $\hat{\sigma}$ is consistently close. The intuition for this outcome is that $\varphi^*_{F_1}$ and $\hat{\sigma}$ are also close for ALG1: when applying the AL strategy, we re-apply Algorithm 1 at each step and find that the average $\varphi^*_{F_1} \approx 15\%$ for ALG1, which is close to 10% (i.e., $\hat{\sigma}$). On the other hand, for ALG2 the average $\varphi^*_{F_1} \approx 50\%$, which is far from $\hat{\sigma}$ and improves all results significantly. We can conclude that our tuning algorithm is most effective when it finds a $\varphi^*_{\mathbb{P}}$ that differs from the estimated $\hat{\sigma}$.

Then, we analyze the performance improvement and related cost of using delay methods. The improvement in F_1-Score granted by our algorithm comes at no labeling or quarantine cost. We can observe that one can improve the in-the-wild performance of the algorithms at some cost L or Q. It is important to observe that objects discarded or to be labeled are not necessarily malware; they are just the objects most uncertain according to the algorithm, which the classifier may have likely misclassified. The labeling costs L for ALG1 and ALG2 are identical (same dataset); in AL, the percentage of retrained objects is user-specified and fixed.

Finally, Table 3 shows that ALG1 consistently outperforms ALG2 on F_1 for all performance-cost trade-offs. This confirms the trend seen in the realistic settings of Table 1.

This section shows that TESSERACT is helpful to both researchers and industrial practitioners. Practitioners need to estimate the performance of a classifier in the wild, compare different algorithms, and determine resources required for L and Q. For researchers, it is useful to understand how to reduce costs L and Q while improving classifiers performance P through comparable, unbiased evaluations. The problem is challenging, but we hope that releasing TESSERACT's code fosters further research and widespread adoption.

6 Discussion

We now discuss guidelines, our assumptions, and how we address limitations of our work.

Actionable points on TESSERACT. It is relevant to discuss how both researchers and practitioners can benefit from TESSERACT and our findings. A *baseline AUT performance* (without classifier retraining) allows users to evaluate the general robustness of an algorithm to performance decay (§4.2). We demonstrate how TESSERACT can reveal true performance and provide counter-intuitive results (§4.4). Robustness over extended time periods is practically relevant for deployment scenarios without the financial or computational resources to label and retrain often. Even with retraining strategies (§5), classifiers may not perform consistently over time. Manual labeling is costly, and the ML community has worked on mitigation strategies to identify a limited number of *best* objects to label (e.g., active learning [46]). TESSERACT takes care of removing spatio-temporal bias from evaluations, so that researchers can focus on the proposal of more robust algorithms (§5). In this context, TESSERACT allows for the creation of comparable baselines for algorithms in a time-aware setting. Moreover, TESSERACT can be used with different time granularity, provided each period has a significant number of samples. For example, if researchers are interested in increasing robustness to decay for the upcoming 3 months, they can use TESSERACT to produce bias-free comparisons of their approach with prior research, while considering time decay.

Generalization to other security domains. Although we used TESSERACT in the Android domain, our methodology

generalizes and can be immediately applied to any machine learning-driven security domain to achieve an evaluation without spatio-temporal bias. Our methodology, although general, requires some domain-specific parameters that reflect realistic conditions (e.g., time granularity Δ and test time length). This is not a weakness of our work, but rather an expected requirement. In general, it is reasonable to expect that spatio-temporal bias may afflict other security domains when affected by concept drift and i.i.d. does not hold—however, further experiments in other domains (e.g., Windows malware, code vulnerabilities) are required to make any scientific conclusion. TESSERACT can be used to understand the extent to which spatio-temporal bias affects such security domains; however, the ability to generalize requires access to large timestamped datasets, knowledge of realistic class ratios, and code or sufficient details to reproduce baselines.

Domain-specific in-the-wild malware percentage $\hat{\sigma}$. In the Android landscape, we assume that $\hat{\sigma}$ is around 10% (§2.2). Correctly estimating the malware percentage in the testing dataset is a challenging task and we encourage further representative measurement studies [30, 53] and data sharing to obtain realistic experimental settings.

Correct observation labels. We assume goodware and malware labels in the dataset are correct (§2.3). Miller et al. [36] found that AVs sometimes change their outcome over time: some goodware may eventually be tagged as malware. However, they also found that VirusTotal detections stabilize after one year; since we are using observations up to Dec 2016, we consider VirusTotal's labels as reliable. In the future, we may integrate approaches for *noisy oracles* [15], which assume some observations are mislabeled.

Timestamps in the dataset. It is important to consider that some timestamps in a public dataset could be incorrect or invalid. In this paper, we rely on the public AndroZoo dataset maintained at the University of Luxembourg, and we rely on the `dex_date` attribute as the approximation of an observation timestamp, as recommended by the dataset creators [3]. We further verified the reliability of the `dex_date` attribute by re-downloading VirusTotal [20] reports for 25K apps[3] and verifying that the `first_seen` attribute always matched the `dex_date` within our time span. In general, we recommend performing some sanitization of a timestamped dataset before performing any analysis on it: if multiple timestamps are available for each object, consider the most reliable timestamp you have access to (e.g., the timestamp recommended by the dataset creators, or the VirusTotal's `first_seen` attribute) and discard objects with "impossible" timestamps (e.g., with dates which are either too old or in the future), which may be caused by incorrect parsing or invalid values of some timestamps. To improve trustworthiness of the timestamps, one could verify whether a given object contains time inconsistencies or features not yet available when the app was released [29]. We encourage the community to promptly notify dataset maintainers of any date inconsistencies. In the TESSERACT's project website (§8), we will maintain an updated list of timestamped datasets publicly available for the security community.

[3] We downloaded only 25K VT reports (corresponding to about 20% of our dataset) due to restrictions on our VirusTotal API usage quota.

Choosing time granularity (Δ). Choosing the length of the time slots (i.e., time granularity) largely depends on the sparseness of the available dataset: in general, the granularity should be chosen to be as small as possible, while containing a statistically significant number of samples—as a rule of thumb, we keep the buckets large enough to have at least 1000 objects, which in our case leads to a monthly granularity. If there are restrictions on the number of time slots that can be considered (perhaps due to limited processing power), a coarser granularity can be used; however if the granularity becomes too large then the true trend might not be captured.

Resilience of malware classifiers. In our study, we analyze three recent high-profile classifiers. One could argue that other classifiers may show consistently high performance even with space-time bias eliminated. And this should indeed be the goal of research on malware classification. TESSERACT provides a mechanism for an unbiased evaluation that we hope will support this kind of work.

Adversarial ML. Adversarial ML focuses on perturbing training or testing observations to compel a classifier to make incorrect predictions [7]. Both relate to concepts of *robustness* and one can characterize adversarial ML as an artificially induced worst-case concept drift scenario. While the adversarial setting remains an open problem, the experimental bias we describe in this work—endemic in Android malware classification—must be addressed prior to realistic evaluations of adversarial mitigations.

7 Related Work

A common experimental bias in security is the *base rate fallacy* [5], which states that in highly-imbalanced datasets (e.g., network intrusion detection, where most traffic is benign), TPR and FPR are misleading performance metrics, because even FPR $= 1\%$ may correspond to *millions* of FPs and only *thousands* of TPs. In contrast, our work identifies experimental settings that are misleading *regardless* of the adopted metrics, and that remain incorrect even if the right metrics are used (§4.4). Sommer and Paxson [47] discuss challenges and guidelines in ML-based intrusion detection; Rossow et al. [44] discuss best practices for conducting malware experiments; van der Kouwe et al. [54] identify 22 common errors in system security evaluations. While helpful, these works [44, 47, 54] do not identify temporal and spatial bias, do not focus on Android, and do not *quantify* the impact of errors on classifiers performance, and their guidelines would not prevent all sources of temporal and spatial bias we identify. To be precise, Rossow et al. [44] evaluate the percentage of objects—in previously adopted datasets—that are "incorrect" (e.g., goodware labeled as malware, malfunctioning malware),

but without evaluating impact on classifier performance. Zhou et al. [58] have recently shown that Hardware Performance Counters (HPCs) are not really effective for malware classification; while interesting and in line with the spirit of our work, their focus is very narrow, and they rely on 10-fold CV in the evaluation.

Allix et al. [2] broke new ground by evaluating malware classifiers in relation to time and showing how future knowledge can inflate performance, but do not propose any solution for comparable evaluations and only identify C1. As a separate issue, Allix et al. [1] investigated the difference between in-the-lab and in-the-wild scenarios and found that the greater presence of goodware leads to lower performance. We systematically analyze and explain these issues and help address them by formalizing a set of constraints (jointly considering the impact of temporal and spatial bias), introducing AUT as a unified performance metric for fair time-aware comparisons of different solutions, and offering a tuning algorithm to leverage the effects of training data distribution. Miller et al. [36] identified *temporal sample consistency* (equivalent to our constraint C1), but not C2 or C3—which are fundamental (§4.4); moreover, they considered the test period to be a uniform time slot, whereas we take time decay into account. Roy et al. [45] questioned the use of recent or older malware as training objects and the performance degradation in testing real-world object ratios; however, most experiments were designed without considering time, reducing the reliability of their conclusions. While past work highlighted some sources of experimental bias [1, 2, 36, 45], it also gave little consideration to classifiers' aims: different scenarios may have different goals (not necessarily maximizing F_1), hence in our work we show the effects of different training settings on performance goals and propose an algorithm to properly tune a classifier accordingly (§4.3).

Other works from the ML literature investigate imbalanced datasets and highlighted how training and testing ratios can influence the results of an algorithm [9, 25, 55]. However, not coming from the security domain, these studies [9, 25, 55] focus only on some aspects of spatial bias and do *not* consider temporal bias. Indeed, concept drift is less problematic in some applications (e.g., image and text classification) than in Android malware [26]. Fawcett [16] focuses on challenges in spam detection, one of which resembles spatial bias; no solution is provided, whereas we introduce C3 to this end and demonstrate how its violation inflates performance (§4.4). Torralba and Efros [52] discuss the problem of *dataset bias* in computer vision, distinct from our security setting where there are fewer benchmarks; moreover in images the negative class (e.g., "not cat") can grow arbitrarily, which is less likely in the malware context. Moreno-Torres et al. [38] systematize different *drifts*, and mention *sample-selection bias*; while this resembles spatial bias, they do not propose any solution/experiments for its impact on ML performance. Other related work underlines the importance of choosing appropriate performance metrics to avoid an incorrect interpretation of the results (e.g., ROC curves are misleading in an imbalanced dataset [14, 24]). In this paper, we take imbalance into account, and we propose actionable constraints and metrics with tool support to evaluate performance decay of classifiers over time.

Summary. Several studies of bias exist and have motivated our research. However, none of them address the entire problem in the context of evolving data (where the i.i.d. assumption does not hold anymore). Constraint C1, introduced by Miller et al. [36], is by itself insufficient to eliminate bias. This is evident from the original evaluation in MaMaDroid [33], which enforces only C1. The evaluation in §4.4 clarifies why our novel constraints C2 and C3 are fundamental, and shows how our AUT metric can effectively reveal the true performance of algorithms, providing counter-intuitive results.

8 Availability

We make TESSERACT's code and data available to the research community to promote the adoption of a sound and unbiased evaluation of classifiers. The TESSERACT project website with instructions to request access is at https://s2lab.kcl.ac.uk/projects/tesseract/. We will also maintain an updated list of publicly available security-related datasets with timestamped objects.

9 Conclusions

We have identified novel temporal and spatial bias in the Android domain and proposed novel constraints, metrics and tuning to address such issues. We have built and released TESSERACT as an open-source tool that integrates our methods. We have shown how TESSERACT can reveal the real performance of malware classifiers that remain hidden in wrong experimental settings in a non-stationary context. TESSERACT is fundamental for the correct evaluation and comparison of different solutions, in particular when considering mitigation strategies for time decay. We are currently working on integrating a time-varying percentage of malware in our framework to model still more realistic scenarios, and on how to use the slope of the performance decay curve to better differentiate algorithms with similar AUT.

We envision that future work on Android malware classification will use TESSERACT to produce realistic, comparable and unbiased results. Moreover, we also encourage the security community to adopt TESSERACT to evaluate the impact of temporal and spatial bias in other security domains where concept drift still needs to be quantified.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Roya Ensafi, for their constructive feedback, which has improved the overall quality of this work. This research has been partially sponsored by the UK EP/L022710/1 and EP/P009301/1 EPSRC research grants.

References

[1] Kevin Allix, Tegawendé F. Bissyandé, Quentin Jérome, Jacques Klein, Radu State, and Yves Le Traon. Empirical Assessment of Machine Learning-Based Malware Detectors for Android. *Empirical Software Engineering*, 2016.

[2] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Are Your Training Datasets Yet Relevant? In *ESSoS*. Springer, 2015.

[3] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting Millions of Android Apps for the Research Community. In *Mining Software Repositories*. ACM, 2016.

[4] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *NDSS*, 2014.

[5] Stefan Axelsson. The Base-Rate Fallacy and the Difficulty of Intrusion Detection. *ACM TISSEC*, 2000.

[6] Peter L Bartlett and Marten H Wegkamp. Classification with a reject option using a hinge loss. *JMLR*, 2008.

[7] Battista Biggio and Fabio Roli. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition*, 2018.

[8] Christopher M Bishop. *Pattern Recognition and Machine Learning*. 2006.

[9] Nitesh V Chawla, Nathalie Japkowicz, and Aleksander Kotcz. Special Issue on Learning From Imbalanced Data Sets. *ACM SIGKDD Explorations Newsletter*, 2004.

[10] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[11] Charlie Curtsinger, Benjamin Livshits, Benjamin G Zorn, and Christian Seifert. ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection. In *USENIX Security*, 2011.

[12] George E Dahl, Jack W Stokes, Li Deng, and Dong Yu. Large-Scale Malware Classification Using Random Projections and Neural Networks. In *Int. Conf. Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2013.

[13] Santanu Kumar Dash, Guillermo Suarez-Tangil, Salahuddin Khan, Kimberly Tam, Mansour Ahmadi, Johannes Kinder, and Lorenzo Cavallaro. Droidscribe: Classifying Android Malware Based on Runtime Behavior. In *MoST-SPW*. IEEE, 2016.

[14] Jesse Davis and Mark Goadrich. The Relationship Between Precision-Recall and ROC Curves. In *Proceedings of the 23rd international conference on Machine learning*, pages 233–240. ACM, 2006.

[15] Jun Du and Charles X Ling. Active Learning with Human-Like Noisy Oracle. In *ICDM*. IEEE, 2010.

[16] Tom Fawcett. In vivo spam filtering: a challenge problem for kdd. *ACM SIGKDD Explorations Newsletter*, 2003.

[17] Giorgio Fumera, Ignazio Pillai, and Fabio Roli. Classification with reject option in text categorisation systems. In *Int. Conf. Image Analysis and Processing*. IEEE, 2003.

[18] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural Detection of Android Malware using Embedded Call Graphs. In *AISec*. ACM, 2013.

[19] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*. MIT press Cambridge, 2016.

[20] Google. VirusTotal, 2004.

[21] Google. Android Security 2017 Year In Review. `https://source.android.com/security/reports/Google_Android_Security_2017_Report_Final.pdf`, March 2018.

[22] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial examples for malware detection. In *ESORICS*. Springer, 2017.

[23] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. LEMNA: Explaining Deep Learning based Security Applications. In *CCS*. ACM, 2018.

[24] David J Hand. Measuring Classifier Performance: a Coherent Alternative to the Area Under the ROC Curve. *Machine Learning*, 2009.

[25] Haibo He and Edwardo A Garcia. Learning From Imbalanced Data. *IEEE TKDE*, 2009.

[26] Roberto Jordaney, Kumar Sharad, Santanu Kumar Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. Transcend: Detecting Concept Drift in Malware Classification Models. In *USENIX Security*, 2017.

[27] Pavel Laskov and Nedim Šrndić. Static Detection of Malicious JavaScript-Bearing PDF Documents. In *ACSAC*. ACM, 2011.

[28] Sangho Lee and Jong Kim. WarningBird: Detecting Suspicious URLs in Twitter Stream. In *NDSS*, 2012.

[29] Li Li, Tegawendé Bissyandé, and Jacques Klein. MoonlightBox: Mining Android API Histories for Uncovering Release-time Inconsistencies. In *Symp. on Software Reliability Engineering*. IEEE, 2018.

[30] Martina Lindorfer, Stamatis Volanis, Alessandro Sisto, Matthias Neugschwandtner, Elias Athanasopoulos, Federico Maggi, Christian Platzer, Stefano Zanero, and Sotiris Ioannidis. AndRadar: Fast Discovery of Android Applications in Alternative Markets. In *DIMVA*. Springer, 2014.

[31] Federico Maggi, Alessandro Frossi, Stefano Zanero, Gianluca Stringhini, Brett Stone-Gross, Christopher Kruegel, and Giovanni Vigna. Two Years of Short URLs Internet Measurement: Security Threats and Countermeasures. In *WWW*. ACM, 2013.

[32] Davide Maiorca, Giorgio Giacinto, and Igino Corona. A Pattern Recognition System for Malicious PDF Files Detection. In *Intl. Workshop on Machine Learning and Data Mining in Pattern Recognition*. Springer, 2012.

[33] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models. In *NDSS*, 2017.

[34] Zane Markel and Michael Bilzor. Building a Machine Learning Classifier for Malware Detection. In *Anti-malware Testing Research Workshop*. IEEE, 2014.

[35] Marco Melis, Davide Maiorca, Battista Biggio, Giorgio Giacinto, and Fabio Roli. Explaining Black-box Android Malware Detection. *EUSIPCO*, 2018.

[36] Brad Miller, Alex Kantchelian, Michael Carl Tschantz, Sadia Afroz, Rekha Bachwani, Riyaz Faizullabhoy, Ling Huang, Vaishaal Shankar, Tony Wu, George Yiu, et al. Reviewer Integration and Performance Measurement for Malware Detection. In *DIMVA*. Springer, 2016.

[37] Bradley Austin Miller. *Scalable Platform for Malicious Content Detection Integrating Machine Learning and Manual Review*. University of California, Berkeley, 2015.

[38] Jose G Moreno-Torres, Troy Raeder, RocíO Alaiz-RodríGuez, Nitesh V Chawla, and Francisco Herrera. A unifying view on dataset shift in classification. *Pattern Recognition*, 2012.

[39] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-Learn: Machine Learning in Python. *JMLR*, 2011.

[40] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. POSTER: Enabling Fair ML Evaluations for Security. In *CCS*. ACM, 2018.

[41] Babak Rahbarinia, Marco Balduzzi, and Roberto Perdisci. Exploring the Long Tail of (Malicious) Software Downloads. In *DSN*. IEEE, 2017.

[42] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Why Should I Trust You?: Explaining the Predictions of Any Classifier. In *KDD*. ACM, 2016.

[43] Konrad Rieck, Tammo Krueger, and Andreas Dewald. Cujo: Efficient Detection and Prevention of Drive-By-Download Attacks. In *ACSAC*. ACM, 2010.

[44] Christian Rossow, Christian J Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten Van Steen. Prudent Practices for Designing Malware Experiments: Status Quo and Outlook. In *Symp. S&P*. IEEE, 2012.

[45] Sankardas Roy, Jordan DeLoach, Yuping Li, Nic Herndon, Doina Caragea, Xinming Ou, Venkatesh Prasad Ranganath, Hongmin Li, and Nicolais Guevara. Experimental Study with Real-World Data for Android App Security Analysis Using Machine Learning. In *ACSAC*. ACM, 2015.

[46] Burr Settles. Active Learning Literature Survey. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 2012.

[47] Robin Sommer and Vern Paxson. Outside the Closed World: On Using Machine Learning for Network Intrusion Detection. In *Symp. S&P*. IEEE, 2010.

[48] Gianluca Stringhini, Christopher Kruegel, and Giovanni Vigna. Shady Paths: Leveraging Surfing Crowds to Detect Malicious Web Pages. In *CCS*. ACM, 2013.

[49] Guillermo Suarez-Tangil, Santanu Kumar Dash, Mansour Ahmadi, Johannes Kinder, Giorgio Giacinto, and Lorenzo Cavallaro. DroidSieve: Fast and Accurate Classification of Obfuscated Android Malware. In *CODASPY*. ACM, 2017.

[50] Masashi Sugiyama, Neil D Lawrence, Anton Schwaighofer, et al. *Dataset Shift in Machine Learning*. The MIT Press, 2009.

[51] Gil Tahan, Lior Rokach, and Yuval Shahar. Mal-id: Automatic malware detection using common segment analysis and meta-features. *JMLR*, 2012.

[52] Antonio Torralba and Alexei A Efros. Unbiased look at dataset bias. In *CVPR*. IEEE, 2011.

[53] Phani Vadrevu, Babak Rahbarinia, Roberto Perdisci, Kang Li, and Manos Antonakakis. Measuring and Detecting Malware Downloads in Live Network Traffic. In *ESORICS*. Springer, 2013.

[54] Erik van der Kouwe, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Gernot Heiser. Benchmarking Crimes: An Emerging Threat in Systems Security. *arXiv preprint*, 2018.

[55] Gary M Weiss and Foster Provost. Learning when Training Data Are Costly: The Effect of Class Distribution on Tree Induction. *Journal of Artificial Intelligence Research*, 2003.

[56] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. Droid-sec: Deep learning in android malware detection. In *SIGCOMM Computer Communication Review*. ACM, 2014.

[57] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-Aware Android Malware Classification Using Weighted Contextual Api Dependency Graphs. In *CCS*. ACM, 2014.

[58] Boyou Zhou, Anmol Gupta, Rasoul Jahanshahi, Manuel Egele, and Ajay Joshi. Hardware Performance Counters Can Detect Malware: Myth or Fact? In *ASIACCS*. ACM, 2018.

A Appendix

A.1 Algorithm Hyperparameters

We hereby report the details of the hyperparameters used to replicate ALG1, ALG2 and DL.

We replicate the settings and experiments of ALG1 [4] (linear SVM with C=1) and ALG2 [33] (package mode and RF with 101 trees and max depth of 64) as described in the respective papers [4, 33], successfully reproducing the published results. Since on our dataset the ALG1 performance is slightly lower (around 0.91 10-fold F_1), we also reproduce the experiment on their same dataset [4], achieving their original performance of about 0.94 10-fold F_1. We have used SCIKIT-LEARN, with *sklearn.svm.LinearSVC* for ALG1 and *sklearn.ensemble.RandomForestClassifier* for ALG2.

We then follow the guidelines in [22] to re-implement DL with KERAS. The features given as initial input to the neural network are the same as ALG1. We replicated the best-performing neural network architecture of [22], by training with 10 epochs and batch size equal to 1,000. To perform the training optimization, we used the stochastic gradient descent class *keras.optimizers.SGD* with the following parameters: *lr*=0.1, *momentum*=0.0, *decay*=0.0, *nesterov*=False. Some low-level details of the hyperparameter optimization were missing from the original paper [22]; we managed to obtain slightly higher F_1 performance in 10-fold setting (§4.4) likely because they have performed hyperparameter optimization on the Accuracy metric [8]—which is misleading in imbalanced datasets [5] where one class is prevalent (goodware, in Android).

A.2 Symbol table

Table 4 is a legend of the main symbols used throughout this paper to improve readability.

Symbol	Description
gw	Short version of goodware.
mw	Short version of malware.
ML	Short version of Machine Learning.
D	Labeled dataset with malware (mw) and goodware (gw).
Tr	Training dataset.
W	Size of the time window of the training set (e.g., 1 year).
Ts	Testing dataset.
S	Size of the time window of the testing set (e.g., 2 years).
Δ	Size of the test time-slots for time-aware evaluations (e.g., months).
AUT(f,N)	Area Under Time, a new metric we define to measure performance over time decay and compare different solutions (§4.2). It is always computed with respect to a performance function f (e.g., F_1-Score) and N is the number of time units considered (e.g., 24 months)
$\hat{\sigma}$	Estimated percentage of malware (mw) in the wild.
φ	Percentage of malware (mw) in the training set.
δ	Percentage of malware (mw) in the testing set.
$\mathbb{P}$	Performance target of the tuning algorithm in §4.3; it can be F_1-Score, Precision (Pr) or Recall (Rec).
$\varphi^*_{\mathbb{P}}$	Percentage of malware (mw) in the training set, to improve performance $\mathbb{P}$ on the malware (mw) class (§4.3).
E	Error rate (§4.3).
E_{max}	Maximum error rate when searching $\varphi^*_{\mathbb{P}}$ (§4.3).
Θ	Model learned after training a classifier.
L	Labeling cost.
Q	Quarantine cost.
P	Performance; depending on the context, it will refer to AUT with F_1 or Pr or Rec.

Table 4: Symbol table.

A.3 Cumulative Plots for Time Decay

Figure 10 shows the cumulative performance plot defined in §4.2. This is the cumulative version of Figure 5.

A.4 Delay Strategies

We discuss more background details on the mitigation strategies adopted in Section 5.

Incremental retraining. Incremental retraining is an approach that tends towards an "ideal" performance P^*: *all* test objects are periodically labeled manually, and the new knowledge introduced to the classifier via retraining. More formally, the performance of month m_i is determined from the predictions of a model Θ trained on: $Tr \cup \{m_0, m_1, ..., m_{i-1}\}$, where $\{m_0, m_1, ..., m_{i-1}\}$ are testing objects, which are manually labeled. The dashed gray line represents the F_1-Score *without* incremental retraining (i.e., stationary training). Although incremental retraining generally achieves optimal performance throughout the whole test period, it also incurs the highest labeling cost L.

Active learning. Active learning is a field of machine learning that studies *query strategies* to select a small number of testing points close to the decision boundaries, that, if included in the training set, are the most relevant for updating the classifier. For example, in a linear SVM the slope of the decision boundary greatly depends on the points that are closest to it, the *support vectors* [8]; all the points further from the SVM decision boundary are classified with higher confidence, hence have limited effect on the slope of the hyperplane.

We evaluate the impact of one of the most popular active learning strategies: *uncertainty sampling* [36, 46]. This query strategy selects the most points the classifier is least certain about, and uses them for retraining; we apply it in a time-aware scenario, and choose a percentage of objects to retrain per month. The intuition is that the most uncertain elements are the ones that may be indicative of concept drift, and new, correct knowledge about them may better inform the decision boundaries. The number of objects to label depends on the user's available resources for labeling.

More formally, in binary classification uncertainty sampling gives a score x^*_{LC} (where LC stands for *Least Confident*) to each sample [46]; this score is defined as follows[4]:

$$x^*_{LC} := \text{argmax}_x\{1 - P_\Theta(\hat{y}|x)\} \quad (6)$$

where $\hat{y} := \text{argmax}_y P_\Theta(y|x)$ is the class label with the highest posterior probability according to classifier Θ. In a binary classification task, the maximum uncertainty for an object is achieved when its prediction probability equal to 0.5 for both classes (i.e., equal probability of being goodware or malware). The test objects are sorted by descending order of uncertainty x^*_{LC}, and the top-n most uncertain are selected to be labeled for retraining the classifier.

Depending on the percentage of manually labeled points, each scenario corresponds to a different labeling cost L. The labeling cost L is known a priori since it is user specified.

Classification with rejection. Malware evolves rapidly over time, so if the classifier is not up to date, the decision region may no longer be representative of new objects. Another approach, orthogonal to active learning, is to include a *reject option* as a possible classifier outcome [17, 26]. This discards the most uncertain predictions to a *quarantine* area for manual inspection at a future date. At the cost of rejecting some objects, the overall performance of the classifier (on the remaining objects) increases. The intuition is that in this way only high confidence decisions are taken into account. Again, although performance P improves, there is a quarantine cost Q associated with it; in this case, unlike active learning, the cost is not known a priori because, in traditional classification with rejection, a threshold on the classifier confidence is applied [17, 26].

[4] In multi-class classification, there is a query strategy based on the entropy of the prediction scores array; in binary classification, the entropy-based query strategy is proven to be equivalent to the "least confident" [46].

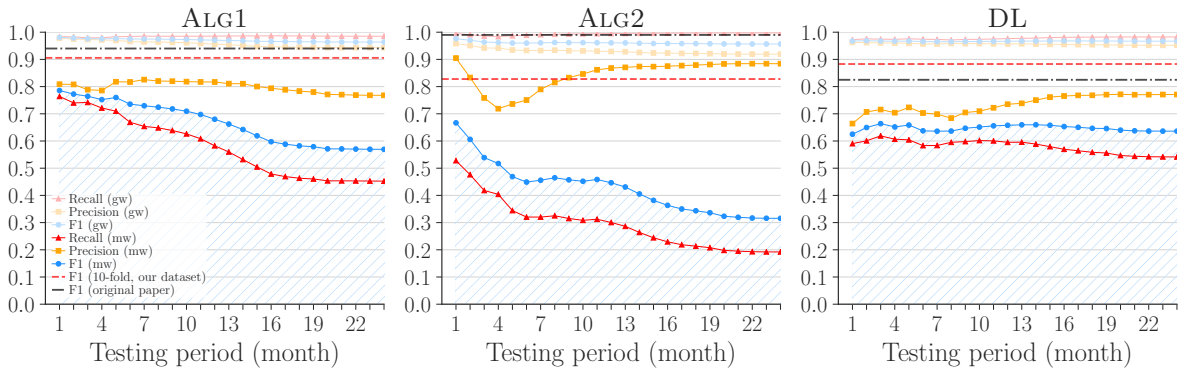


Figure 10: Performance time decay with cumulative estimate for ALG1, ALG2 and DL. Testing distribution has $\delta = 10\%$ malware, and training distribution has $\varphi = 10\%$ malware.

A.5 TESSERACT Implementation

We have implemented our constraints, metrics, and algorithms as a Python library named TESSERACT, designed to integrate easily with common workflows. In particular, the API design of TESSERACT is heavily inspired by and fully compatible with SCIKIT-LEARN [39] and KERAS [10]; as a result, many of the conventions and workflows in TESSERACT will be familiar to users of these libraries. Here we present an overview of the library's core modules while further details of the design can be found in [40].

temporal.py While machine learning libraries commonly involve working with a set of predictors `X` and a set of output variables `y`, TESSERACT extends this concept to include an array of `datetime` objects `t`. This allows for operations such as the time-aware partitioning of datasets (e.g., `time_aware_partition()` and `time_aware_train_test_split()`) while respecting temporal constraints C1 and C2.

spatial.py This module allows the user to alter the proportion of the positive class in a given dataset. `downsample_set()` can be used to simulate the natural class distribution $\hat{\sigma}$ expected during deployment or to tune the performance of the model by over-representing a class during training. To this end we provide an implementation of Algorithm 1 for finding the optimal training proportion φ^* (`search_optimal_train_ratio()`). This module can also assert that constraint C3 (§4.1) has not been violated.

metrics.py As TESSERACT aims to encourage comparable and reproducible evaluations, we include functions for visualizing classifier assessments and deriving metrics such as the accuracy or total errors from slices of a time-aware evaluation. Importantly we also include `aut()` for computing the AUT for a given metric (F_1, Precision, AUC, etc.) over a given time period.

evaluation.py Here we include the `predict()` and `fit_predict_update()` functions that accept a classifier, dataset and set of parameters (as defined in §4.1) and return the results of a time-aware evaluation performed across the chosen periods.

selection.py *and* **rejection.py** For extending the evaluation to testing model update strategies, these modules provide hooks for novel query and reject strategies to be easily plugged into the evaluation cycle. We already implement many of the methods discussed in §5 and include them with our release. We hope this modular approach lowers the bar for follow-up research in these areas.

A.6 Summary of Datasets Evaluated by Prior Work

As a reference, Table 5 reports the composition of the dataset used in our paper (1st row) and of the datasets used for experimentally biased evaluations in prior work ALG1 [4], ALG2 [33], and DL [22] (2nd and 3rd row). In this paper, we always evaluate ALG1, ALG2 and DL with the dataset in the first row (more details in §2.3 and Figure 1), because we have built it to allow experiments without spatio-temporal bias by enforcing constraints C1, C2 and C3. Details on experimental settings that caused spatio-temporal bias in prior work are described in §3 and §4. *We never use the datasets of prior work [4, 22, 33] in our experiments.*

Work	Apps	Date Range	# Objects	Total	Violations
TESSERACT (this work)	Benign	Jan 2014 - Dec 2016	116,993	116,993	-
	Malicious		12,735	12,735	
[4], [22]	Benign	Aug 2010 - Oct 2012	123,453	123,453	C1
	Malicious		5,560	5,560	
[33]	Benign	Apr 2013 - Nov 2013	5,879	8,447	(C1) C2 C3
		Mar 2016	2,568		
	Malicious	Oct 2010 - Aug 2012	5,560	35,493	
		Jan 2013 - Jun 2013	6,228		
		Jun 2013 - Mar 2014	15,417		
		Jan 2015 - Jun 2015	5,314		
		Jan 2016 - May 2016	2,974		

Table 5: Summary of datasets composition used by our paper (1st row) and by prior work [4, 22, 33] (2nd and 3rd row).

Devils in the Guidance: Predicting Logic Vulnerabilities in Payment Syndication Services through Automated Documentation Analysis

Yi Chen[1,3]*, Luyi Xing*[2]*, Yue Qin*[2]*, Xiaojing Liao*[2]*, XiaoFeng Wang*[2]*, Kai Chen*[1,3]*, Wei Zou*[1,3]
[1]*{CAS-KLONAT*[†]*, BKLONSPT*[‡]*, SKLOIS*[§]*},Institute of Information Engineering, CAS,* [2]*Indiana University Bloomington,*
[3]*School of Cyber Security, University of Chinese Academy of Sciences*
{luyixing, qinyue, xliao, xw7}@indiana.edu, {chenyi, chenkai, zouwei}@iie.ac.cn

Abstract

Finding logic flaws today relies on the program analysis that leverages the functionality information reported in the program's documentation. Our research, however, shows that the documentation alone may already contain information for *predicting* the presence of some logic flaws, even before the code is analyzed. Our first step on this direction focuses on emerging syndication services that facilitate integration of multiple payment services (e.g., Alipay, Wechat Pay, PayPal, etc.) into merchant systems. We look at whether a syndication service will cause some security requirements (e.g., checking payment against price) to become unenforceable due to losing visibility of some key parameters (e.g., payment, price) to the parties involved in the syndication, or bring in implementation errors when required security checks fail to be communicated to the developer. For this purpose, we developed a suite of Natural Language Processing techniques that enables automatic inspection of the syndication developer's guide, based upon the payment models and security requirements from the payment service. Our approach is found to be effective in identifying these potential problems from the guide, and leads to the discovery of 5 new security-critical flaws in popular Chinese merchant systems that can cause circumvention of payment once exploited.

1 Introduction

Logic vulnerabilities are a category of security defects caused by faulty program logic, which have long been known to be hard to analyze, due to their close ties to specific functionalities of a system. Finding these defects relies on evaluating the target system's behaviors, at the code level, against a set of invariants describing its function-related security properties (e.g., expected authentication and authorization operations). A great source for such invariants is the system's documentation, which states its security goals and has been leveraged by prior research for the purposes such as model checking on the system's code [27]. In the meantime, the documentation also provides detailed accounts of how the system is designed to achieve the goals and how it should be used to remain secure. Such information can be valuable for *predicting* whether security-critical logic flaws are present in the system or applications that use the system: for example, prior research shows that logic vulnerabilities could be related to low-quality documentations, such as those missing explanations about implicit assumptions for secure integration of authentication SDKs [49]; also a conflict between the system's operations, as described in its documentation, and its security goals may indicate the existence of a design flaw. However, never before has any effort been made to dig into the details contained in a myriad of software documentations to understand their security implications, not to mention any attempt to exploit their full value for logic flaw detection.

Logic vulnerabilities in payment syndication. In this paper, we present preliminary evidence that system documentations indeed carry abundant vulnerability-related indicators, which can help identify logic flaws *even before the code of the system has been analyzed*. Further we show that this documentation-based approach can be automated, enabling more effective vulnerability detection through providing guidance to program analysis such as fuzzing. This first step has been made possible by a study on *syndication services* that facilitate integration of various payment services (e.g., Alipay [2], WeChat Pay [17], PayPal [25]) in mobile apps. These payment services have different APIs and SDKs, and an app often needs to use all of them to offer its customers different payment options. To simplify integration of these options, a syndication service encapsulates each payment service with a *wrapper* that exposes to the developer a uniform interface. This, however, injects the syndicator as a proxy into the already complicated payment interactions among the payer (the app user), the payee (the merchant) and the payment service provider (e.g., Paypal). Logic flaws can therefore be induced

*Work was done when the first author was at Indiana University Bloomington.
[†]Key Laboratory of Network Assessment Technology, CAS.
[‡]Beijing Key Laboratory of Network Security and Protection Technology
[§]State Key Laboratory of Information Security, IIE, CAS

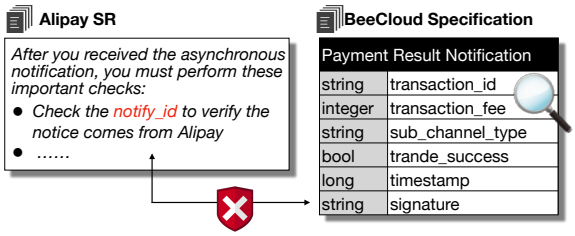


Figure 1: Incosistency example.

at the design level, when the wrapper causes some security checks hard to proceed (e.g., verification of payment against price), or at the implementation level, when the developer fails to perform security checks correctly, due to incomplete instructions given by the syndicator.

These payment syndication services today become increasingly popular: according to the reports [35, 36], their total transactions in year 2018 have reached 21.1 trillion yuan (3.15 trillion dollars). Any vulnerability inside these services, once exploited, will have significant impacts, affecting 251 million syndication users world-wide. We believe that their documentations, developer's guide in particular, contain information that can help predict or even detect the logic flaws in their customers' systems. As an example, Figure 1 demonstrates the inconsistency discovered when comparing an Alipay's security requirement, which asks for inspecting `notify_id`, with the payment notification issued by BeeCloud [6] (a popular service syndicating Alipay and other payment services), which does *not* include this information, as discovered from its developer's guide; therefore, the merchant server receiving the notification will *not* be able to perform the required check. This logic problem has been *confirmed* in our research. A question here is how to systematically identify such logic vulnerabilities from documentation. Also, developer's guides are typically long and complicated, including a lot of irrelevant information (e.g., instructions for using the syndicator's tools like dashboard). For example, Ping++ has 1093 KB text documents online with at least 278 KB related technical content [12]. Inspecting all the content manually is both time-consuming and error-prone. Automated techniques therefore need to be developed to help vulnerability discovery.

Document analysis for flaw detection. In our research, we developed *Dilution* (Documentation Inspector for Logic Vulnerability Prediction), a new technique that automatically analyzes the developer's guide of a payment syndication to infer potential security flaws in the merchant systems integrating the service. Dilution is designed to predict missing security checks in the integration, which is caused by either improper encapsulation of a payment service that renders its security checks impossible to perform through the wrapper, or failure in communicating the necessary checks to the developer through the guide. For this purpose, we utilize natural language processing (NLP) to automatically recover semantic information from the wrapper's integration instructions documented by the developer's guide and compared it against the finite state machine (FSM) of the payment service encapsulated. Note that this payment FSM was manually extracted but considered as one-time efforts (Section 3.1). Our analysis automatically infers the relation between the syndication payment process and the payment FSM, maps important payment states to the related instructions in the guide and further recovers the parameters for required security checks (e.g., Alipay key for signature verification, price) at the state from the text. By analyzing the merchant or syndicator's visibility of the parameters, we can determine whether these checks can still be performed by the merchant or the syndicator. Also from related descriptions in the guide, our approach can automatically find out whether the developer is informed about these security requirements when integrating the wrapper. Missing such instructions indicates the possible absence of security checks in the merchant's code.

We implemented Diluation using a suite of NLP techniques, including dependency parsing and word embedding, and evaluated it on labeled content extracted from the developer's guides of real-world syndication services. Our study shows that Dilution accurately caught logic flaws and went through 182 KB text document content within 3.18 seconds, averagely.

Finding. Further, we ran Dilution on the documentations of eight popular syndication services, including Ping++ [12], Paymax [11], BeeCloud [6], etc., which have tens of thousands of merchants each and power the apps with millions of users. From 1,456 KB documentations, our approach automatically predicted totally 41 potential issues, including 11 highly likely to be logic flaws from five syndication services: Fuqianla [8], BeeCloud, TrPay [15], UMF Pay [16] and 66zhifu [1]. All the issues reported were found to be accurate by our manual inspection of the documentations. Despite the challenges in finding the apps integrating these services, due to the obfuscations these services suggest [20, 23, 26], we collected 17 popular Chinese apps using two of these syndication services. Through a black-box testing on these apps and their merchant systems, we concluded that all 5 logic flaws related to these syndication services predicted by Dilution are indeed present in either the syndicators' systems or their customers' code. All such confirmed flaws are security-critical, and once exploited, will have serious consequences, allowing the adversary to shop at a lower price or even for free. We reported our findings to the providers of the syndications and the merchants who are affected, and they all acknowledged the importance of the problems we discovered. Now we are in the process of helping them fix these vulnerabilities. Video demos of our attacks are posted online [4].

Contributions. The contributions of the paper are outlined as follows:

• *New direction*. We explore the potential to predict the presence of logic vulnerabilities in a software system from its documentation. Our preliminary study on payment syndica-

tion services shows that this is indeed feasible. Research along this line could bring in a new perspective to software security analysis, enabling more effective and intelligent vulnerability detection and helping enhance software security quality.

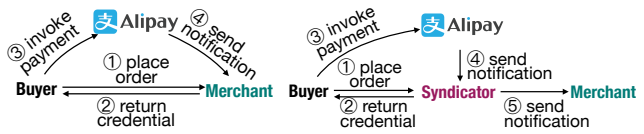
• *New techniques*. We developed Dilution, the first semantics-based documentation analyzer, to automatically inspect the developer's guide and infer possible security fallacies in the merchant's integration of the syndication service. Our approach includes a suite of NLP techniques tuned towards software documentation, which are found to be effective and efficient, as demonstrated by our evaluation.

• *New findings*. We analyzed the developer's guides of 8 most popular syndication services using Dilution and discovered potential security issues. Among these we can validate, we confirmed that all 5 logic flaws predicted by our approach are indeed present in syndicator or merchant systems. These vulnerabilities, once exploited, allows the adversary to shop at an arbitrarily low price he set or completely for free, affecting millions of users. We are working with affected syndicators and merchants to fix these problems.

2 Background

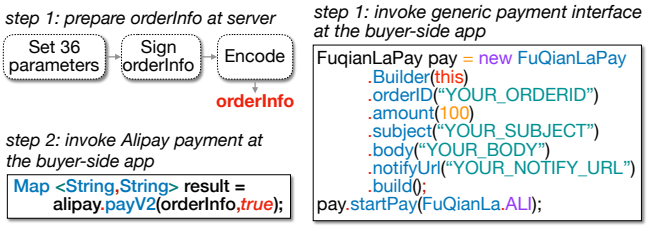
Payment and syndication service. A third-party payment service (aka. a *payment processor*) like PayPal [25] is an Internet service to help handle transactions between the buyer (the payer) and the seller (the payee) [43]. Such a service simplifies transaction managements on both the payer and the payee sides and therefore plays an important role in e-commerce. Figure 2(a) shows how the service works using Alipay [2], the largest online payment processor with over 1 billion users around the world [47], as an example. The buyer first places an order through the app (①) and receives a payment-related `credential` from the merchant (②), including order ID, price, seller account and others, and then forwards it to Alipay (③). After the order is paid by the buyer, Alipay issues a notification to inform the merchant the completion of the transaction (④). The interfaces exposed by these payment services tend to be complicated. Figure 3(a) further details the process the app developer (working for the merchant) is supposed to do for using Alipay : the merchant server generates an Alipay specific argument `orderInfo` (part of `credential`) with 36 entries once the buyer places an order; then the buyer-side app of the merchant invokes Alipay's payment service.

To simplify this integration process, syndication services emerge to wrap different payment processors into a uniform interface for the developer to conveniently incorporate them into her app. Figure 2(b) and 3(b) shows a common syndication payment process for Alipay. Instead of asking merchant developer to implement anything specific to Alipay, the syndicator receives an order from the buyer on behalf of the merchant (①), construct the `credential` (②) and then invokes Alipay payment service from the buyer's app (③). Note that,



(a) Payment process (b) Syndication process

Figure 2: Examples of payment and syndication process.



(a) To integrate Alipay (b) To integrate a syndicator Fuqianla

Figure 3: The implementation examples.

③ only requires the app to invoke a generic payment interface provided by the syndicator's SDK in the app, and the target payment processor, i.e., Alipay in this case, will be invoked by the syndicator's SDK. Once the payment is done, the syndicator receives the notification from the the payment processor (④) and restructures the message to a uniform format before forwarding it to the merchant server (⑤). In this way, a simple integration of a single syndication service on both the app end and the merchant server end will allow the merchant to work with multiple payment processors supported by the syndicator. The developer is only supposed to follow a single set of instructions from the syndicator to ensure the secure payment process.

Security requirements. Online payment is a security-critical process, so it is safeguarded by various security checks performed both by the payment processor and by the merchant, as required by the payment processor on its developer's guide. We call these checks *security requirements* (SRs)[1] throughout the paper. For instance, most third-party payments ask their merchants to verify the payment amount on the notification against the price of the purchase, the seller account information to ensure that the merchants are intended payees, etc. In Section 4.2, we present more examples for common security requirements. In the case that the payment syndication is used, we expect that either the developer or the syndicator is still at the position to perform these required checks, and also in the former case, the developer should be properly informed through the guide provided by the syndicator.

Natural language processing. In our research, we utilized two NLP techniques to automatically analyze the documentation of the payment syndication service: *dependency parsing* and *word embedding*, as explained below.

[1] All the abbreviation's explanation are summarized in Table 5 at Appendix for convenience.

Table 1: Examples of the relations between linguistic units

Example sentence: *She gave me a very happy smile and a hug.*

Abbreviation	Description	Relation example
SBV	Subject-verb	She <- gave
VOB	Verb-object	gave ->smile, gave ->hug
IOB	Indirect object	gave ->her
ATT	Attribute	happy <- smile
ADV	Adverbial	very <- happy
COO	Coordinate	smile ->hug

Dependency parsing is an NLP technique to reveal the syntactic structure of a sentence by analyzing the grammatical relations between linguistic units such as words. Examples of such relations include subject-verb(SBV), verb-object(VOB), indirect object(IOB), attribute(ATT), adverbial(ADV), coordinate(COO) and others (see detailed explanation and examples of these relations in Table 1). The result of the dependency parsing is represented as a rooted parsing tree. At the center of the tree is the verb of a clause structure, which is linked, directly or indirectly, by other linguistic units. The state-of-the-art dependency parser (e.g., Stanford parser [31]) can achieve a 92.2% accuracy in grammatical relation discovery from a sentence. In our study, we leveraged the parsing tree generated from sentences in a developer's guide to locate the parties involved in a payment process and the content transmitted between them.

Word Embedding is a set of language modeling and feature learning techniques that map text (words or phrases) from a vocabulary to high-dimensional vectors of real numbers. Such a mapping can be implemented in different ways. The state-of-the-art word embedding tool, *Word2vec* [37], initializes word representations by random values and uses as its input a joint probability distribution of words' context by applying a continuous Bag-of-Words or a skip-gram model. This distribution is then utilized during the training of a neural network, in which word vectors are continuously updated to maximize the joint probability. The outcome of the training ensures that related words are given approximate vectors for their similar contexts while irrelevant words are mapped into different vectors. In our study, we leveraged *Word2vec* to generate a semantic vector for each word and measured their semantic difference by cosine distance between vectors.

Threat model. In our research, we consider a malicious buyer who intends to get a product for free or at a lower price by exploiting the vulnerabilities in the payment process, particularly the logic flaws caused by incorrect or inadequate security checks on the merchant or the syndicator side. This adversary has the capability to modify or forge the messages delivered to both the merchant server and the syndicator.

3 Dilution: Design

3.1 Overview

We believe that a critical security goal of a payment syndication is to ensure that all the security requirements made by the payment processor it wraps are met through proper

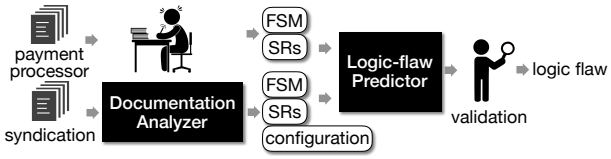


Figure 4: Architecture of our approach.

security checks, either by the merchant integrating the service or by the sydnicator itself. The purpose of Dilution, therefore, is to identify from the developer's guide of the syndication *indicators* that some SRs may fall through the cracks.

To this end, our approach is designed to compare the information observed by the syndicator and the merchant (including its app) with the SRs expected at individual states of the original payment process (e.g., that of Alipay), to determine whether either of the syndicator or the merchant has enough information to fulfill these requirements at the states. Also analyzed is whether these SRs have been explicitly stated in the developer's guide provided by the syndicator, when related checks need to be done on the merchant's side. More specifically, we first extract an FSM from the payment process, with some of its states associated with SRs and the parameters they are predicated on. This FSM is then extended to include the operations performed by the wrapper, based upon the information automatically recovered from the guide. Further using the text content related to each state of the extended FSM, we evaluate whether all parameters of each SR at the state are still visible to the merchant or the syndicator. Finally, our approach automatically analyzes the content of the guide to determine whether the SRs are explained to the developer.

Architecture. Figure 4 illustrates the architecture of Dilution, including a preprocessing component, *Documentation Analyzer* (DoA), *Logic-flaw Predictor* (LfP) and a validation component. The preprocessing step is done manually in our current system, which involves extraction of the FSM and label of SRs for a payment processor. Since our focus is the syndication service, so we consider this step as a one-time effort: for each third-party payment, the information identified can be used to automatically analyze tens of syndication services, each with a development guide containing hundreds of thousands of words. Such documentations are inspected by DoA, which utilizes NLP techniques to extend the payment FSM with the states representing the wrapper's actions, and further recover the description for each SR from the guide. The extended FSM and the parameters are then analyzed by the LfP to identify the SRs that cannot be fulfilled by the merchant and the syndicator, and those that have not been properly explained to the developer. Also, the logic flaws predicted by LfP are validated on the merchant's integration of the syndicator (the merchant's app and its server-side component) to confirm the presence of these vulnerabilities.

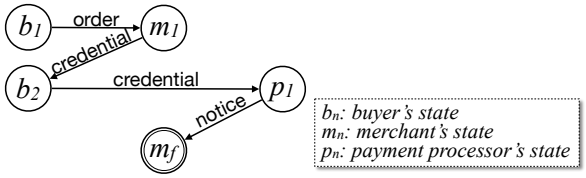
Example. Here, we use an example to describe how our approach works. Figure 5(a) shows Alipay's FSM, with one of

its SRs at state m_f being verification of the payment amount against the price of an item. Analyzing the developer's guide from Fuqianla (a popular syndicator), DoA discovers that the order has been sent to the syndicator, instead of the merchant ("The merchant client sends a payment query to the Fuqianla server"), and the notification from Alipay is delivered to the syndicator ("The Fuqianla server will receive a payment notification from the payment service"), before it is forwarded to the merchant . The semantics recovered from the text is then used to replace state m_1 in Figure 5(a) with state w_1 and add state w_2 to the FSM, converting it to the syndication FSM as shown in Figure 5(b). This extended FSM and all the parameters it carries is then further inspected by LfP. Here let us look at the aforementioned SR at m_f. At this state, LfP concludes that the syndicator cannot check the payment amount, as it gets the price information from the buyer not the merchant, which cannot be trusted. On the other hand, though the merchant is at the position to make the security check, LfP cannot find a sentence, right after the description of the last communication (from w_2 to m_f), informing the developer about the SR, through a dependency analysis on all the sentences involving terms related to the subject (the merchant), the object (price and payment amount) and the expected action of the security check. This leads to the conclusion that the SR may not be communicated to the developer, and so may not be fulfilled on the merchant side.

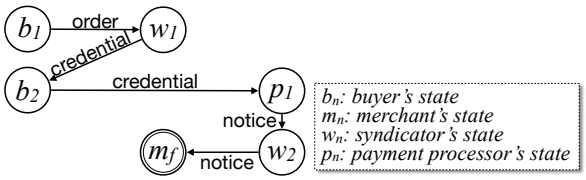
3.2 Preprocessing

As mentioned earlier, our approach involves a one-time preprocessing step in which the FSMs of major payment processors (Alipay, Wechat, PayPal, etc.) are constructed and the SRs for different states are identified. Such information can typically be found from these payment services' integration guides. For example, Figure 11 at Appendix shows the excerpts from Alipay's documents. Its payment process is clearly described through a diagram, which can be easily converted into an FSM. From the figure, we can also see the content for SRs, their parameters and the relations with the payment process. Following we present the FSM that models the payment process and the SRs. From such content, we extract the payment process model and security checks, as formally described below.

Payment model. The FSM for a payment process is described as a 5-tuple: ($\boldsymbol{S}$, $\boldsymbol{D}$, $\boldsymbol{E}$, b_1, m_f). Here $\boldsymbol{S}$ is a set of payment states, in each of which an actor (buyer b, merchant m or syndicator w) can send out a message $d \in D$; $E : S \times D \rightarrow S$ is a function that drives the transition from one payment state to the next, given a specific message $d \in D$ sent out from the former; b_1 is the initial state in which the buyer places an order to start the whole payment transaction, and m_f is the final state that the merchant receives the last message and the transaction is complete. For example, Figure 5(a) illustrates such an FSM for Alipay.



(a) The FSM of the payment processor using Alipay



(b) The FSM of the syndication using Fuqianla

Figure 5: FSM examples of the payment processor and syndication.

Security requirement. A security requirement SR describes a security check that needs to be performed at a certain state: for example, at m_f of Alipay's FSM (Figure 5(a)), the merchant is supposed to verify $payment = price$. As we can see from the example, central to the SR are the state (m_f), subject (*merchant*), object (*payment*), a verification function (the *equal function* in the example) and additional parameters for the verification (*price*). However, in the context of the payment FSM, each state is bound to an actor who is actually the subject performing a security check. Further since all we care is the feasibility of fulfilling the SR at a given state, we just need to know whether all the inputs of the verification function (object and other parameters) are visible to the subject at the state, not the function itself. Therefore, we can simply model an SR as a 3-tuple: (SR_{state}, SR_{obj}, SR_{para}), to represent its state, object and other parameters for the expected security check. Note that the object and parameters here are *types of information*, i.e., the *key* part of a key-value pair. This is because all we want to know is whether the merchant at a state can see these keys (*payment* and *price*) so as to perform the required check; the outcome of the check, which depends on their specific values, is not important for our purpose.

3.3 Syndication FSM Discovery

With the FSM and SRs collected from a payment processor, we can run DoA to automatically analyze the developer's guides of different syndication services that wrap the processor. Most important here is to discover the syndicated payment process through extending the payment FSM, which further allows us to find out whether the parameters of required security checks are still visible to the new states supposed to perform the checks, and how these security requirments are explained to the developer (Section 3.4).

FSM extension structures. Fundamentally, a syndication service is meant to support a merchant's interactions with the payment service, in terms of generating the input for the

service and converting its output to a unified form easy for the merchant to interpret. This observation allows us to come up with a set of possible *extension structures*, which are then confirmed by the evidence extracted from the syndication's document (its developer's guide).

Specifically, given a payment processor's FSM, we consider two types of states[2] that the syndicator can help: m where the merchant generates an input for the payment service, and m_f when the merchant receives the final notification from the payment service. More specifically, the operations at the state m can be assisted by a syndication state for input construction; payment notification the merchant finally receives at m_f can be converted by a syndication state first. Following this line of thinking, we can identify all possible extensions of m and m_f, which are present in Figure 6. As we can see here, for m, the extensions include the situations when either the merchant or the syndicator produces the full output of m, that is, the input for the payment service (see a.1 and a.4), and those when they jointly create the output (see a.2, a.3, a.5 and a.6). The latter can be further broken down into the cases when the same party receives and issues the messages related to m (see a.2 and a.5), or when different parties do (see a.3 and a.6). For m_f, since the payment process must end at this merchant state, only three situations exist: no extension (see b.1), the merchant getting the input (see b.2) and the syndicator receiving the original input (see b.3). DoA automatically inspects every merchant state and evaluates the consistency between each possible extension structure and the evidence discovered from the developer's guide through NLP, to find out how the syndication indeed happens.

Taking the syndicator Fuqianla as an example (see Figure 5(b)), through inspecting it documentation, DoA determines that Fuqianla uses the extension structures a.4 and b.3 to wrap the payment processor. Specifically, its state w_1 replaces the original state m_1 to generate credential, which is an input for invoking the payment processor; the syndicator at state w_2 receives the payment notification from the merchant and converts it to a unified form (across different payment services it supports) before forwarding it to the merchant at state m_f.

Extension discovery from document. The evidence collected from the syndication developer's guide is the sentence that describes the message transferred from a sender to a receiver in the FSM (the buyer, the merchant, the payment provider and the syndicator). For example, from the sentence "the merchant client sends a payment query to the Fuqianla server", we know that the payment query from the buyer has been sent to the syndication server, not the merchant, which confirms the existence of a transition from b to w in the extension structure a.5 and a.6 (Figure 6). Our idea is to find all

[2]Note that we are only interested in these states because per payment services' guides [2,17,25], the inputs they accept are supposed to be generated by the merchant server and the outcome of a transaction will be delivered to the server.

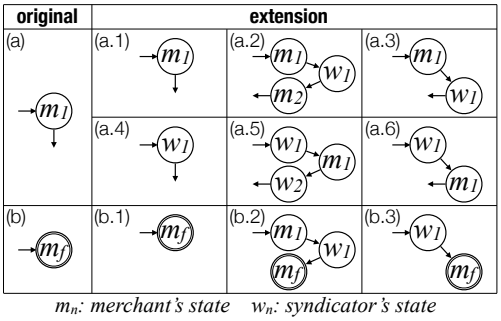


Figure 6: FSM possible extension structures.

such descriptions and extract their transition-related *semantic information*, in the form of (*Sender*, *Receiver*, *Content*), to identify all transitions introduced by the syndication and further determine the way merchant states have been extended.

For this purpose, we utilized a suite of NLP techniques to first find out all sentences related to transmission activities (e.g., including predicates like "send", "receive", etc.), then performs a syntactic analysis on each sentence and further converts detected syntactic elements to a *semantic triplet* describing the parties involved in a transition as well as the message sent (*Sender*, *Receiver*, *Content*). The challenges here come from the ambiguity of the descriptions and diversity of sentence structures in the developer's guide. Particularly, we found that a variety of terms are used to describe message delivery and reception: not only common synonyms like "transmit", "dispatch", etc., but also those specific to the integration domains such as "call" (a remote function) and "invoke" (a remote client).

To identify those synonymous terms, we leverage the observation that such expressions, no matter how diverse they are, all share the similar context. For example, from the sentences "call the merchant API to place an order" and "send the order to the merchant", we know that "call" and "send" are semantically close given their relations with 'merchant' and 'order'. So in our research, we trained a *word embedding* model [18] over the documentations of two syndicators, Ping++ [12] and Fuqianla [8], and two payment service providers, Alipay [2] and Wechat Pay [17]. The model maps each word to a vector that represents its context. So the cosine distance between the vectors quantifies their semantic similarity. In our research we first manually collected a small set of "seeds", words semantically related to "send" and "receive", such as "call" and "invoke", and then built a synonym list for these words with the embedding model we trained over the aforementioned documentations, using LTP [19] to segment Chinese words. These lists are utilized to identify sentences in a developer's guide involving these transmission-related terms.

On each sentence discovered, DoA needs to extract its semantics – the triplet. For this purpose, we come up with a unique technique that utilizes dependency parsing (LTP [19]) to first identify a sentence's syntactic elements (subject, object, etc.) and then determine their semantics (*Sender*, *Receiver*,

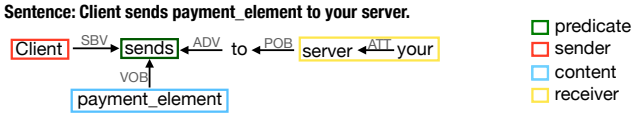


Figure 7: Entities in data-transmission related sentence.

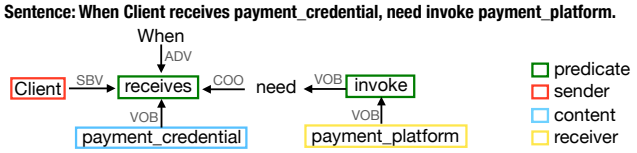


Figure 8: Entities in complex sentence.

Content). This is quite intuitive for a simple sentence. For example, Figure 7[3] shows the dependency relations between the predicate "send" and other words or phrases. As we can see here, the subject of the predicate ("client") is *Sender*, direct object ("payment_element") is *Content* (the message delivered), and indirect object ("your server") is the receiver. These elements can then be mapped to a transition on the FSM, based upon their semantic similarity with payment parties, as measured by the distances between their vectors. However, the semantics of the syntactic elements become more difficult to determine in the presence of more complicated sentence structures. For example, Figure 8 shows a complex sentence with multiple clauses, including both "When Client receives payment_credential" and "(Client) needs to invoke payment_platform". In this case, the subject of "receives" ("Client") becomes *Sender*, the object of "invoke" ("payment_platform") is *Receiver* and the *Content* in the sentence is found to be the object of "receives" ("payment_credential"). To address this challenge, we trained an SVM classifier on a labeled dataset with transmission related sentences discovered from payment documentations and syndication documentations. The model uses the predicates and their relations (e.g., order) as features to predict their subject, object and indirect object's semantic class labels (*Sender*, *Receiver* or *Content*).

Using the triplets recovered from the sentences, DoA continues to inspect each merchant state to determine whether and how it is extended by a syndicator. Specifically, for each transition in the extension structures described in Figure 6, denoted by $s' = E(s,d)$, we try to align *Sender* to the name of the actor at s (e.g., "merchant"), *Receiver* to the actor of s' (e.g., "syndicator") and *Content* to d (representing message *name* here, such as "order"). Note that in the case any of these entities is described by a phrase, instead of a word (e.g., "payment element"), we calculate its *phase vector* as the average of its individual word vectors (e.g., those of "payment" and "element"). We consider that an alignment succeeds when all these elements are found to be similar to their counterparts on the transition. When this happens, we believe that this transition exists in the extended FSM. To discover more transitions, our approach also leverages partial information collected, when only two elements of the triplet has been recovered. If one of these elements is *Content*, DoA still compares them against the transitions and confirms the presence of a transition if an alignment is found.

Based upon all the transitions discovered, our approach further determines the extension structure used by a syndication: the one contains all these transitions is selected. When there are more than one such structures, we consider that all such extensions are possible and predict the presence of potential logic flaws if one of them is found to be problematic.

3.4 SR Information Discovery

The syndication FSM discovered tells us how a payment transaction proceeds among the buyer, the merchant and the payment service provider, in the presence of the syndicator. To find out whether all security checks required by the payment service can be performed on this new FSM, we need to take a further look at the information visible to the states that need to fulfill these security requirements. Also to be understood is how the SRs are presented to the developers who are supposed to integrate these checks in merchant-side code. All such information has been automatically recovered from the syndication developer's guide, as elaborated below.

API parameter discovery. At a syndication or merchant state a security check is expected to happen (SR_{state}), the information required for the check (SR_{obj}, SR_{para}) either *comes from the message it receives (e.g., payment_result) or is already in the possession of the syndicator or the merchant*. In the former case, the communication with the syndicator always goes through its APIs, as integrated in the merchant's client or server side code. These APIs are documented in the developer's guide and their attributes describe the information passed by a message. A question is how to determine which API is used in a transition. Such an API is explicitly mentioned in some sentences, such as "call the Creating Charge to invoke a payment processor". In our research, we utilized a SVM model for labeling syntactic elements to detect the API names, which serve as the object of "call".

However this approach turns out to be inadequate, since more often than not, a transition-related sentence in the developer's guide does not include any API name: for example,"initiate payment". In this case, we found that the semantics of the message name (d in the transition and its corresponding *Content*) is always highly related to the name of the API to be used. In the above example, an API "initiate payment" is responsible for sending the message of payment requirement. This is understandable since the guide is supposed to inform the developer how to establish communication with the syndicator. If this has not been done explicitly, using semantically related API names is an implicit way to do so. Our DoA automatically identifies such APIs using our

[3]Figure 7, Figure 8 and Figure 10 show Chinese grammatical relations between words. The words shown in the figure were translated from Chinese.

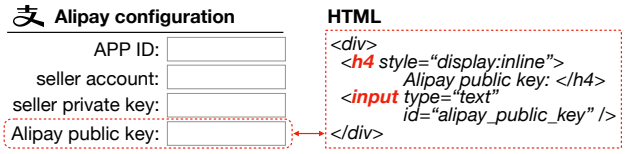


Figure 9: An excerpt of configuration HTML.

word-embedding model to compare the phrase similarity between each API name and d and *Content*. Again, here we utilized LTP [19] for Chinese word segmentation and word extraction from API names. Whenever an API is found to be semantically similar to either d or its related *Content*, our approach collects all its attributes and consider that their values have been exposed to the actor at the state receiving d.

Configuration information extraction. As mentioned earlier, the information for a security check can also be provided to the actor in a certain state before a transaction happens. For example, the merchant has her private key for signing a payment credential and she can also delegate this task to the syndicator by configuring her account on the syndication website. Such preconfigured information is documented by the developer's guide. However, the details are often included in images, together with those irrelevant ones for explaining the payment or syndication service. Content extraction from these images using OCR [39] did not work well in our study. So our DoA is designed to discover the configuration data directly from the syndication website, the one from which we collect the developer documentation.

Specifically, our approach first searches for the entry link labeled with "Alipay configuration", "Wechat Pay management", "PayPal setup", etc., the standard names for the configuration page on a syndication site, using named entity recognition and keywords (e.g., Alipay) together with synonyms for "configuration", etc. On such a page, we inspect its HTML tags, looking for the input type – the entry item for the merchant to enter her data, and its inline header, which is the *key* for the data. Figure 9 shows part of the configuration page of Ping++ [12]. From the text entry identified, we can recover its header "Alipay public key" under the tag *h4*. Also as we can see from the example, other keys that can be found from the configuration page includes APP ID, seller account, etc. The information is gathered for comparing with SR parameters for a given payment service such as Alipay.

SR description recovery. Also important to logic vulnerability discovery is to find out whether required security checks have been properly explained to the developer. For this purpose, DoA has also been made to search for the description of the SRs for a given payment service. Such an SR is typically presented in a sentence: e.g., "The merchant should verify whether the payment amount equals to the order price." From the sentence, we know that the merchant is the party responsible for this SR, so the check is supposed to happen on a merchant state (SR_{state}), payment amount is the object

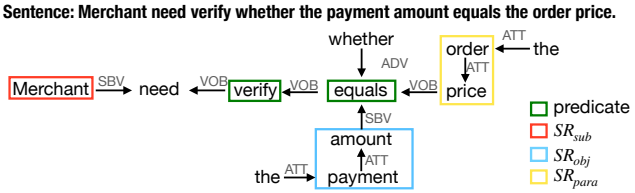


Figure 10: Entities in sentences describing SR.

(SR_{obj}) and the order price is the additional parameter for the security check (SR_{para}). Our idea is to automatically discover all SR-related sentences based upon the actions to be taken, as we did in the FSM extension discovery (see Section 3.3), and then perform a syntactic analysis to discover SR_{obj} and SR_{para}, before finally determining the state of the potential check (SR_{state}).

Specifically, to find all SR-related sentences, our approach first utilizes a small set of *seed action terms* including "check", "match", "verify", etc., and runs our word-embedding model on the training documents (guides for syndication and payment services) to extend these seeds with their synonyms. Then DoA inspects a given document to collect all the sentences containing the term(s) on the list. Each of them is analyzed using dependency parsing to label the subject, object and indirect object of the action term ("verify", "check", etc.), as demonstrated by the example in Figure 10. Given the fact that the developer's guide is meant to explain implementation details to the developer working for the merchant, we expect that an SR-related subject should be "merchant", "developer", "you", and their synonyms (e.g, "seller","verdor", etc.). Also we consider the merchant to be the subject of all imperative sentences, e.g., "please make sure that the payment amount equals to the order price". From these sentences, we further identify the object and indirect object (if exists) of the predicate, and label them as potential SR_{obj} and SR_{para}.

Before we can report possible SR-related description, we also need to determine the state for the potential security check (SR_{state}). Our approach is to look at where the sentence is found: intuitively a reminder of a security check should appear under the context of state transition. For example, the sentence "The merchant needs to verify whether the payment amount matches the order price." comes right after "The syndicator will send a Webhook request to the merchant server. " in Beecloud [6]. So for each potential SR-related sentence, DoA tries to locate a transition-related sentence in the same paragraph, before the SR sentence. Once the transition is found, we further check whether its destination is a *merchant* state, so the merchant is supposed to perform the security inspection mentioned in the sentence. In this case, we set SR_{state} to that merchant state.

The only other place where security check description can be found is the specification of the API used for the state transition. For example, under the API "Transaction-result-notification" (iAppPay [9]), there is a note that reminds the developer to inspect payment: "Please verify the transaction

payment is the same as the product price". So when the SR-related sentence is discovered in such a specification, we look at the state the API leads to, and make it SR_{state} if it is a merchant state.

3.5 Logic-flaw Prediction and Validation

From the FSM and the SR information discovered from the syndication documentation, LfP infers the possible presence of logic flaws: the SRs expected by the payment processor (e.g., Alipay) that cannot be fulfilled in the syndication FSM, and those that have not been explained to the developer. Following we explicate how to determine the states expected to perform the required security checks, how to evaluate whether the checks can take place, and how to capture the SRs that have not been properly communicated to the developer.

Security goals. Consider a syndicator W that wraps a payment service P. We believe that W needs to achieve the following two security goals:

• *Secure Design (SD):* for any security requirement SR to be enforced at a state s of P, there exists an *enforceable* SR' at the state s' of W such that SR and SR' are equivalent except their states, and s' is a state in the extension structures for s (Figure 6). Intuitively, this means that every payment SR should still be fulfilled after syndication.

• *Secure Implementation (SI):* every SR of W is correctly implemented by either the merchant or the syndicator.

To achieve SD, for every SR of the state s in the payment FSM, LfP first identifies all extended states of s and then inspects each of them s' to determine whether the state has the *visibility* of the object (SR_{obj}) and other parameters (SR_{para}) of SR. As mentioned earlier, s' is identified by the extension structures (Figure 6). Any state among these replacing s is considered to be a possible location for enforcing SR. As an example, consider *b.3*, which is an extension of b, and $SR = (m_f, payment, price)$, which checks $payment = price$. This inspection can happen at either m_f or w, when at least one of them can observe both *payment* from the payment service (e.g., Alipay) and *price* from the merchant.

For SI, without looking at the code of the extended FSM, LfP goes through the developer's guide for the indicators that could lead to implementation flaws. The most important one used in our current design is the absence of the explanations about SRs, a clear signal that the related security checks might not be implemented by the uninformed developer. Also we are concerned about the SRs that can only be enforced by the syndication state, since they are out of the merchant's control. So in both cases, LfP will predict potential logic flaws and suggests a code-level validation.

Design flaws identification. When none of the states in the extension structure of a payment state can observe SR_{obj} and SR_{para} for a security requirement SR, we can conclude that the syndication service contains a design flaw. To detect such a flaw, LfP needs to analyze visibility of data at each extension state. As mentioned earlier, such data either comes from the message a given state receives, whose content is described by all the attributes of the API used for transmitting the message (Section 3.4), or preconfigured by the merchant in her syndication account, with all data attributes (APP ID, public key, seller account, etc.) discovered by DoA. Our approach directly compares these attributes with the SR information. Note that *only the data delivered through a secure channel and from a trusted source can be used in a security check*: for example, the payment amount should be signed by the payment provider and the price should come from the merchant (through preconfiguration, signed message or local storage).

Specifically, at a given extension state, let TA be a set of *trusted* message attributes (e.g., signed by the payment provider) as collected from related API specifications, and when the state is controlled by the syndicator, TC be a set of collected attributes for preconfigured merchant information. Also, we abuse notations a little bit, using SR_{obj} and SR_{para} to represent the *sets* for the object and for the additional parameters, respectively, of a given security requirement SR at that state. The objective of LfP is to find out whether there exists an extension state such that for a given SR on the corresponding payment state, $SR_{obj} \cup SR_{para} \subseteq TA \cup TC$ for the TA and TC of the extension state (TC includes all the local data for a merchant state). If none of such an extension state can be found, LfP reports that the SR can no longer be enforced and therefore a design flaw is detected.

For a data attribute (e.g., "price", "public_key", etc.) $a \in SR_{obj} \cup SR_{para}$, it is *nontrivial* to determine whether it is also in $TA \cup TC$, simply because the attribute names of the SR collected from a payment service (e.g., Alipay) may not match those included in the message API and the configuration web page. Our solution here, again, is using our word-embedding model to product a semantic vector for a and then find whether there is an attribute in $TA \cup TC$ whose vector is sufficiently close to that of a. When every attribute of the SR can find its counterpart in TA or TC, we consider that the SR is enforceable at the current extension state.

Implementation flaws prediction. As mentioned earlier, implementation flaws can also be predicted when a required security check has not been properly communicated to the developer. LfP is designed to inspect the SR descriptions recovered by DoA to identify such missing security guidance.

Specifically, for each SR enforced at the state s in the payment FSM, LfP searches across all security requirements discovered from the guide by DoA for those associated with the extension states of s. Let SR' be such a requirement. Our approach tries to determine whether $SR_{obj} = SR'_{obj}$ and $SR_{para} = SR'_{para}$. Again, here we need to deal with the inconsistency in attribute names during the comparisons, which has been addressed in our implementation using our word-embedding model and distance measurement between semantic vectors. If SR' here indeed matches SR, we have reason to believe that the developer knows the security check required

by the payment provider. If such SR' cannot be found, then we know that such information has not been conveyed to the developer. When the SR can only be enforced by the merchant, LfP will raise the alarm since we doubt that an uninformed developer can make the protection right.

Even when all the SRs are found to be enforceable on the syndication FSM and all merchant-side SRs are properly mentioned in the developer's guide, we are still concerned about the security checks that can only be performed by the syndicator, who does not mention that this has actually been implemented. Given the fact that the merchant essentially loses the control of these security checks that they could do without the syndication, we believe that these SRs should be evaluated to ensure that they have been put in place. So our current implementation of LfP also reports all such SRs, which are evaluated during the validation step.

Validation. Fully automated verification of our predicted flaws is possible but nontrivial, due to the requirements of entering user credentials (password, fingerprint) to trigger a payment process and handling diverse user interfaces in different mobile apps for entering purchase information (product, quantity, address, etc.). Although existing GUI testing tools could be enhanced to serve this purpose and likely industry-grade fuzzers can already support these operations, building such techniques are outside the scope of our research. So we manually validated all the flaws predicted by Dilution. Specifically, based upon the specific security requirement that we consider hard to enforce, we acted as a malicious buyer to adjust the payment parameters to find out whether the predicted flaw can indeed be exploited. As an example, consider the payment process in Figure 5(b). Dilution predicts an implementation flaw that the merchant does not check the payment amount. In our research, we set a lower price in the *orderInfo* given to an app. This transaction got through (Section 5.2), which confirmed the presence of the flaw we predicted.

4 Implementation and Evaluation

4.1 Implementation

Dilution is implemented in a prototype. In the DoA, we employ Language Technology Platform (LTP) [19], for word segmenting, POS tagging and dependency parsing to analyze the sentences. To adopt the open domain toolkit to payment document analysis, we craft external dictionaries containing 48 domain-specific terms (e.g., payment element and payment credential) in the payment process to improve the performance of word segmenting and embedding. Taking the results of dependency parsing as features we further implement the classifier for entity recognition with LIBSVM [30] in version 3.23. To map words into vectors we utilize the word2vec model in Gensim library [18] in 3.7.1 version. Moreover, we ran the crawler *Scrapy* [44] in version 1.6 to crawl all the web pages of syndications' official websites, and then utilize the *BeautifulSoup* [42] in version 4.4.0 to parse webpages and extract the developer's guides and configuration. For the LfP, we implement with 404 lines of Python code for interring the supposed SRs and inspecting each of them to predict flaws. We are going to release the source code of Dilution online [14].

4.2 Experiment Settings

Dataset. In our research, we utilized four datasets for model training and evaluation:

• *Groundtruth set*. The groundtruth set was used for logic flaw detection, entity recognition and phrase alignment.

For logic flaw detection, the groundtruth set includes two syndication documents (Ping++ [12] and Fuqianla [8]) and their corresponding 17 potential logic flaws. In particular, we manually analyzed the documents and identified 11 implementation flaws in the documentations of syndication Ping++, 1 design flaw and 5 implementation flaws in the documentation of syndication Fuqianla, as elaborated in Section 5.

The groundtruth set for entity recognition in the payment process consists of 574 entities (148 *Sender*, 175 *Receiver* and 251 *Content*) from 242 sentences describing data transmission. These sentences were collected from a training corpus including documents of two payment services Alipay and Wechat and two syndicators Ping++ and Fuqianla. We implemented a 2-fold cross validation with half of the data as the training set each time.

We manually labelled the groundtruth set for phrase alignment upon the syndication documents of Ping++ and Fuqianla, which contain 14 data-transmission sentences, 103 APIs and 1,986 parameters in total. The groundtruth set includes 63 and 203 positive pairs, and 45 and 40,866 negative pairs in transition mapping for extension discovery and API parameter discovery, respectively.

• *Unknown syndication documents*. To evaluate *Dilution*, we ran our prototype on the developer's guides of six syndications, including Paymax [11], BeeCloud [6], iAppPay [9], Trpay [15], UMF Payment [16] and 66zhifu [1]. These documents consist of 3,613 sentences and 46,098 words in total. They are all publicly available, well-written documentations including the description of data transmission in payment processes, API parameter explanations, SRs and configuration information. Note that those services are popular, serving tens of thousands apps and millions users.

• *Third-party payment documents*. The third-party payment documents we manually analyzed to extract FSMs and SRs come from Alipay, WeChat Pay and PayPal, which are the three most popular mobile payment services in the world [45]. We read all the related documentations about the payment process and searched keywords related to SRs including "security", "requirement", "check", "inspect", "compare" to collect all the SRs. Three experts spent 2 days to finish the extraction and the SRs are validated across all three experts. The detailed SRs are summarized in Table 2.

Table 2: Security requirements of payment service

$notify_{id}$: The id of a notification, $seller_{id}$: The id of a seller, txn_{id}: The id of a transaction,
$receiver_{email}$: The account of the merchant, mc_{gross}: The amount of a payment, $mc_{currency}$: The currency of a payment.

Payment Service	No.	SR description	*SR*
Alipay	SR1	Check the signature in the notification.	$(m2, notification, key)$
	SR2	Check the $notify_{id}$ to verify the message comes from Alipay.	$(m2, notify_{id}, \emptyset)$
	SR3	Check the price in the notification is the same with the amount in the order.	$(m2, payment, price)$
	SR4	Check the $seller_{id}$ represents the supposed merchant.	$(m2, seller_{id}, merchant)$
WeChat Pay	SR5	Verify the signature in the payment notification message.	$(m2, notification, key)$
	SR6	Check the price in the notification equals the price in the order.	$(m2, payment, price)$
PayPal	SR7	Verify the message came from PayPal.	$(m2, message, \emptyset)$
	SR8	Check the txn_{id} against the previous PayPal transaction that you processed to ensure the IPN message it not duplicate.	$(m2, txn_{id}, previous\ txn_{id})$
	SR9	Check that the $receiver_{email}$ is the email address registered in your account.	$(m2, receiver_{email}, registered\ email)$
	SR10	Check that the price carried in mc_{gross} are correct for the item.	$(m2, payment, price)$
	SR11	Check that the currency carried in $mc_{currency}$ are correct for the item.	$(m2, receipt\ currency, supposed\ currency)$

• *Payment corpora for word embedding model training*. For training the word embedding model, we built a corpus for payment service by combining two payment documentations (Alipay, Wechat) and two syndication documentations (Ping++, Fuqianla), which were crawled from the corresponding websites. After word segmenting, the training corpus contains 1715.2 KB text with 23,576 sentences and 306,680 words.

Parameters. The parameters for our implementation are set as follows:

• *Entity classifier*. We implemented the classifier for entity recognition with LIBSVM [30]. The classifier was trained with the following settings: c=8.0, g=0.5 and default settings.

• *Word2vec*. We utilized skip-gram with negative sampling as the framework of the word2vec model, which was trained with the following parameters: sg=1, size=100, sample=0.0001, window=10, iter=5, min_count=1, negative=20 and other default settings.

• *Threshold*. We utilized phrase similarity to find out whether two phrases are semantically close. For payment-related expressions (e.g.,payment_element, payment_credential), the threshold used in transition mapping, API name matching and parameter matching were set to 0.91, 0.91, and 0.97 respectively. As for other phrases, the threshold in three tasks were set to 0.87, 0.87, and 0.96, respectively.

Platform. All the experiments in our study were conducted on the macOS with 2.3GHz CPU, 16GB memory and 512GB hard drivers using a single process.

4.3 Effectiveness

We first evaluated the overall effectiveness of our prototype in predicting potential logic flaws from documentations. Running on both the groundtruth set (Ping++ and Fuqinala) and the unknown dataset containing six syndicators, Dilution achieved 100% accurate predictions. More specifically, on the six unknown documentations, our system predicted 1 design flaw and 16 potential implementation flaws. We manually verified each of them and found that all reports were correct (based upon the descriptions in the documentations).

Further, we evaluated the two internal modules of DoA: entity recognition and API parameter discovery. The evaluation for entity recognition was run on the groundtruth set for entity recognition (242 data-transmission related sentences with 574 entities including 148 *Sender*, 175 *Receiver* and 251 *Content*). Under the two-fold cross validation, our model achieved a precision of 89.38%, 93.28%, 94.57% and a recall of 96.88%, 97.72%, 96.81% when taking *Sender*, *Receiver*, *Content* as the positive class, respectively. The effectiveness of our model is acceptable since our algorithm for discovering the state extension is capable of addressing the false positives and false negatives induced by entity recognition through alignment with the transitions in the extension structures (Section 3.3). As for the API parameter discovery, the experiment results on the guides of all eight syndications show that all phrase pairs aligned by DoA are accurate.

4.4 Performance

We ran Dilution on the developer's guide of 8 syndications (1,456KB) to predict the presence of logic flaws. Averagely, our system spent merely 3,177.8 ms to go through the whole process on one syndication. The time of the analysis ranges from 2,743.2ms to 3,873.8 ms, with the medium being 3,099.1 ms. More specifically, DoA spent 2,738.8 ms to 3,840.0 ms with an average of 3,166.2 ms. LfP took 2.9 ms to 33.8 ms with an average of 11.6 ms. This result offers strong evidence that Dilution can easily scale to the level expected for processing a large amount of documentation.

5 Discoveries in the Wild

In this section, we report the logic flaws predicted by Dilution from the developer's guides of real-world syndication services and the end-to-end exploits to validate the predictions through popular merchant apps. We show that our document-only predictions are indeed accurate, leading to the discovery of security-critical vulnerabilities.

5.1 Finding from Documentations

There are more than 30 syndication services, with the number continuing to grow. However, most of them provide developer's guides to paid users only. Actually we found that just

Table 3: Summary of predictions by Dilution

(DF: design flow, IF: implementation flow, CI: cases of interest)

(a) design & implementation

Syndication	Type	SR No.
Fuqianla	DF	1
	IF	3, 6
BeeCloud	DF	1
	IF	11
TrPay	IF	3, 6
UMF Pay	IF	3, 6
66zhifu	IF	3, 6
Total	DF	**2**
	IF	**9**

(b) cases of interest

Syndication	Type	SR No.
Ping++	CI	1 - 11
Fuqianla	CI	2, 4, 5
Paymax	CI	1 - 6
BeeCloud	CI	2, 4, 5, 7, 8, 9
iAppPay	CI	2
TrPay	CI	2
UMF Pay	CI	2
66zhifu	CI	2
Total	CI	**30**

8 of them have well-documented guides publicly available. These syndicators are all popular, with hundreds of millions of users. In our research, we ran Dilution on all of their guides (over 1.4 MB), which reported its findings in a few seconds.

Landscape. Table 3 shows all the syndications we analyzed and the logic flaws predicted. Specifically, Dilution reported 41 potential issues from all the syndications. Among them, 11 are highly likely to be logic flaws, including 2 design flaws (in BeeCloud and Fuqianla), in which required security checks cannot be done, and 9 likely implementation flaws, with critical security checks missed in the guides. In addition, the remaining 30 are "cases of interest", since their SRs can only be or should be fulfilled by syndicators if merchants cannot achieve them or do not be told. Therefore they are considered to be risky and need to be validated.

Design flaws. Among all the syndication services, BeeCloud and Fuqianla are found to contain a design flaw each (SR1 in Table 2). Specifically, Dilution reported that these syndicators receive payment notifications from Alipay on behalf of their merchants, helping them finish the final security checks before informing them of the completion of the transactions. The problem is that the merchants of these services cannot configure their Alipay's verification keys[4] to the syndicators. As a result, the syndicators cannot check the authenticity of the messages, nor can their merchants, since they do not get the signed notifications. We further found that the practice of processing payment messages for the merchants without forwarding them the original messages is very common across all syndications we studied. This is because the syndication aims to unify the merchant-side interfaces with different payment services to reduce the complexity in integrating them, which, however, makes the syndicator-side operations complicated and error-prone. Although only two design flaws were revealed by Dilution, due to the small number of syndicators we evaluated, we believe that the practice likely brings in design lapses to other syndication services.

Implementation flaws. Dilution predicted 9 potential implementation flaws by the merchant developers in 5 syndication services. Specifically, our approach found that the syndicators Fuqianla, TrPay, UMP Pay and 66zhifu cannot verify the payment amount since they do not have access to the price of a purchase.In the meantime, they fail to remind their merchant developers of enforcing the security requirements (SR3 and SR6) through verifying the amount. Similarly, BeeCloud encapsulates PayPal but does not tell its merchant that the currency type for a purchased item needs to be checked. As a result, we believe that very likely the required security checks will fall through the cracks.

[4]Each merchant has a unique key-pair for verifying the messages from Alipay.

5.2 Attacks on Real-World Systems

Challenges in validating predicted flaws. To find out whether the predicted logic flaws are indeed present in syndicators or the merchant-side code, we need to validate them through merchant apps. This attempt, however, faces two challenges. First, finding the apps integrating a given syndication is difficult. Even though these services are popular (e.g., at least 25K apps using Ping++), with tens of millions of users according to their websites, rarely do they provide a list of the merchants that use their services. Actually, most syndicators ask their merchant developers to obfuscate their code, possibly for the purpose of IP protection [29]. Second, even given an app integrating a syndication, exploiting its logic flaws may need additional resources we do not have and some of the flaws may not even be exploitable in the absence of other flaws. Particularly, in 7 out of the 30 cases of interest reported by Dilution, we need to produce Alipay's signature on the payment notification to confirm whether the syndicator (e.g., Paymax) indeed fails to perform a security check (e.g., on the payment amount), since the syndicator may still verify Alipay's signature. The exploit can only be executed with the help of a merchant under our control: we can make a purchase from our own merchant and use the notification to determine whether the syndicatior indeed verifies its attributes (e.g., payment amount). We manually analyzed our findings and believe that if the predicted flaws are there, we can exploit them in this way. However, merchant registration (with Alipay) is complicated, which we did not do in our research.

Despite the challenges, still we were able to find 17 apps to confirm 5 logic flaws across 2 syndication services and their merchants. These 17 apps were found from over 50K apps we analyzed. Most importantly, *for every merchant app that could be analyzed, every single logic flaw or case of interest predicted by Dilution has been confirmed*. Specifically, we randomly crawled over 50K apps from the Baidu Market, a top Chinese app market [5], and ran Apktool [3] to reserve-engineer them. The 17 were found because the names of their syndication SDKs or the domains of syndication servers have not been obfuscated. Among them, 16 use BeeCloud [6] and 1 uses TrPay [15]. Both the syndication services and the apps are very popular. As shown in Table 4 in Appendix, BeeCloud claims to have tens of thousands merchants, and these apps have hundreds of millions of users. In our experiment, we ran a proxy called Burp Suite [7] and a network API testing tool called Postman [13] to modify or forge the messages

delivered from the apps or our site to the merchant or the syndicator server.

Attacks on design flaws. As mentioned earlier (Section 5.1), Dilution reported two design flaws, one for BeeCloud and the other for Fuqianla, in which the syndicator cannot verify Alipay's signature on a payment notification due to its lack of the verification key. In our research, we could not find the app using Fuqianla possibly due to the obfuscation it suggests to its users [23]. The 16 apps using BeeCloud, however, were predicted to all have the *same* logic flaw. Therefore, we just randomly picked one of them, a Chinese education app called Chuangyebang [21], for the validation. Specifically, we placed an order for an online class provided by the app without payment, then used Postman to forge an unsigned Alipay's notification and delivered to the BeeCloud server. As predicted, the syndicator accepted the fake message and we successfully got the digital product for free [5]. This demonstrates that the design flaw is real and exploitable, which has a serious consequence given the fact that BeeCloud is serving tens of thousands of merchants [28]. Actually, the 16 apps we collected have 82.8 million downloads in total, selling products ranging from 0.1 dollars to 2,000 dollars. Even Changyebang is reported 490,000 installs.

Attacks on implementation flaws. Also we were able to find a Chinese tool app called OffPhone [24] (with 830,000 downloads) that integrates TrPay, another popular syndication service [15] wrapping WeChat and Alipay payment services. This enabled us to validate 2 potential implementation flaws that Dilution predicted. Both problems happen at the final state m_f (one for WeChat and the other for Alipay), where the merchant is supposed to check the payment amount, which the developer's guide of TrPay fails to mention. Since in both cases, the syndicator generates the credential (including the price of the order) for payment at WeChat or Alipay, based upon the order issued through the app, the adversary can control the app to provide wrong price information to mislead TrPay into producing a credential with a lower price. If the payment amount reported by WeChat Pay or Alipay has not been verified by the merchant (OffPhone) at m_f, the adversary can get the item with the price she set. In our experiment, we modified the order placed through the OffPhone app to reduce the price of a VIP membership from 5 dollars to just 1 cent, causing TrPay to send a credential to the payment services. In the end, both transactions went through, which indicates that indeed the payment amount has *not* been checked by the merchant, as Dilution predicted.

Furthermore, Dilution also reported 2 "cases of interest" in BeeCloud, where the syndicator is supposed to perform some security checks on behalf of the merchant. Specifically, we found that on receiving the payment notification from Alipay, BeeCloud only passes some of the notification content to the merchant. An attribute not being forwarded is *notify_id*, the merchant's identity issued by Alipay for finding out whether the merchant is the right recipient of the notification. Also, although another attribute, *seller_id*, serving the same purpose is indeed sent to the merchant, BeeCloud fails to mention in its developer's guide that the attribute should be verified. Both attributes are *required* to be inspected by Alipay [2]. Here, we tried to find out whether the checks have been done by BeeCloud on the merchant's behalf. For this purpose, we randomly selected another app from the 16 apps, called Clean [22], a memory cleaner with 850,000 downloads, to find out whether the SR2 and SR4 in Table 2 have been enforced. In the experiment, we ran Postman to fake a payment notification including a random *notify_id* and *seller_id*, which did not prevent our payment transaction from getting through. In the end, we got a paid version of the app (without Ads) for free. The same attack also succeeded on Chuangyebang. Evidently, not only does BeeCloud fail to verify *notify_id* and *seller_id*, but Clean (the merchant) does not check *seller_id* either, in line with the predictions made by Dilution. These flaws open the door for an attack in which one pays for her own merchant while getting product from a different store [49]. Also note that since the merchant is not given *notify_id*, remaining 15 apps and others using BeeCloud are certain to have the same flaw.

Responsible experiment design. We carefully designed our end-to-end attack in a responsible manner. The entire study was conducted under the guidance of a lawyer at our University. We strictly followed the principles below when attacking the real-world apps and services: (1) we performed no intrusion of either merchant servers and syndicator servers; (2) we ensured that no financial damage caused by returning items, paying the shipping costs, not getting refunded, not using electronic products, paying for items hard to return; (3) we reported all security flaws to affected apps and syndications and did what we could to help them improve their systems. All the flaws discovered have been acknowledged by the syndicators and merchants, who are very grateful for our help.

6 Discussion

Limitations. Our research demonstrates that logic flaws can be predicted from the developer's documentation, even before the system code is inspected. This finding can lead to new techniques that make full use of information available for enhancing software security. Our current design and implementation, however, are still preliminary. We only focus on the payment syndication service with limited targets on missing security checks. More complicated flaws, such as policy enforcement weaknesses (possibly caused by inaccurate guidance), and more complicated service procedures (e.g., refund, bonus, etc.), are all missing from the picture, not to mention documentation-based analysis on other security-critical systems. Further, the NLP techniques underlying Dilution can only deal with well-written documents, not those containing

[5] All the exploit video demos are post online [4].

typos, grammatical errors, ambiguous sentence structures, etc. as observed in real-world developer's guides. Also, our current approach still needs human involvements: particularly, we extracted SRs and FSMs of payment processors manually, which has two reasons. First, the number of payment processors is small. Second, we want the extracted SRs and FSMs to be precise, based on which we built syndicators' FSMs and predicted logic flaws. However, we envision it's possible to automate this process using open-source NER tools. All these issues need to be addressed in the future research.

Future research. Down the road, we expect more explorations on this new direction, toward the end of an intelligent, semantics-based methodology that combine documentation-level and code-level analysis together for more effective flaw discovery. In addition to the direct improvement of our approach mentioned above, we envision that document-based flaw prediction will be applied to secure other syndication services, those for single sign-on services in particular, such as MobSDK [10]. More importantly, we believe that with the help of machine learning and automatic inference, other subtle, semantics-dependent weaknesses only detectable by experienced analysts today will become increasingly manageable by automatic techniques, leading to a significant improvement in software security quality.

7 Related Work

Numerous studies have looked into logic flaw detection in various applications. For example, [49] discovered logic flaws of the payment service. [48, 50] investigated logic flaws on authorization. Traditional logic flaws discovery heavily relies on domain experts [33]. Recent year witnesses the trend of automatic logic flaw detection, mainly based on model checking. The typical approach based on model check first standardizes a logic process, and then detects whether the application violates the predefined logic. For instance, both [41] and [32] automatically extracted a model from a number of correct behavioral patterns. Then, they checked the source code statically, using model checking over symbolic input to identify violated program paths. [34] manually summarized the correct usage of OpenSSL APIs and then statically analyzed whether an application violates the correct usage. Different from previous researches, our approach does not touch program code and automatically utilizes only documentation to predict logic flaws.

The closest to our study are the works of payment logic flaw assessment [49] [51]. [49] is the first work, which relies on human effort, to discover serious payment logic vulnerabilities and reveal their security implications. A set of follow-up studies identified different types of payment logic flaws in various applications. For instance, [51] extended the work of [49] to investigate the logic flaws in mobile payment and detected seven security rule violations to the payment in Android apps. [46] detected violations of the invariant in secure checkout processes, which revealed 11 new logic vulnerabilities in web payment modules. Given these studies, a bunch of policies and guidelines for secure online shopping were investigated [38, 40]. In contrast to previous works, which assessed logic flaws by manual efforts, we report the first work towards automatic payment logic flaw discovery. Also, we investigate a novel payment service, payment syndication, which has never been studied before.

8 Conclusion

In this paper, we report the first step towards automatic documentation-based logic flaw discovery. Our study on the emerging payment syndication services shows that their developer's guide contains abundant information that can be leveraged to predict the presence of logic flaws in their customers' systems. Using a suite of NLP techniques, our approach effectively analyzed over 1.4 MB of technical documentations from real-world syndicators within seconds, and accurately predicted 5 new security-critical flaws in the Chinese merchant systems with millions of users. The research demonstrates that software documentations can be more effectively used to help find the security risks hard to automatically detect today.

9 Acknowledgment

We would like to thank our shepherd Adam Doupé and the anonymous reviewers for their insightful comments. The IU authors are supported in part by NSF-1527141, 1618493, 1838083, 1801432 and 1850725, ARO W911NF-16-1-0127 and Indiana University FRSP-SF. IIE authors are supported in part by NSFC U1836211, U1836209, 61728209, 61602470, 61802394, National Top-notch Youth Talents Program of China, Youth Innovation Promotion Association CAS, Beijing Nova Program, Beijing Natural Science Foundation (No.JQ18011), National Frontier Science and Technology Innovation Project (No. YJKYYQ20170070), Strategic Priority Research Program of the CAS (XDC02040100, XDC02030200, XDC02020200), National Key Research and Development Program of China (2016QY071405) and the Program of Beijing Municipal Science & Technology Commission (NO. D181100000618004).

References

[1] 66zhifu. https://www.66zhifu.com.

[2] Alipay. https://www.alipay.com.

[3] apktool. https://ibotpeaches.github.io/Apktool/.

[4] Attack demos. https://sites.google.com/view/dilution/home/attack-demos.

[5] Baidu mobile assistant. https://shouji.baidu.com.

[6] Beecloud. https://beecloud.cn.

[7] Burp suite. https://portswigger.net/burp.

[8] Fuqianla. https://fuqianla.net.

[9] iapppay. https://www.iapppay.com.

[10] Mobsdk. http://www.mob.com.

[11] Paymax. https://paymax.cc.

[12] Ping++. https://www.pingxx.com.

[13] Postman. https://www.getpostman.com.

[14] Source code of Dilution:. https://github.com/ccy1991911/Dilution.

[15] Trpay. http://pay.trsoft.xin/front/index.html.

[16] UMF pay. https://xy.umfintech.com.

[17] WeChat pay. https://pay.weixin.qq.com.

[18] Gensim. https://github.com/rare-technologies/gensim, 2018.

[19] LTP. https://github.com/HIT-SCIR/pyltp, 2018.

[20] 66zhifu obfuscation guide. https://www.66zhifu.com/show/help, 2019.

[21] Chuangyebang download link. http://m.cyzone.cn/app/, 2019.

[22] Clean download link. https://shouji.baidu.com/software/25240151.html, 2019.

[23] Fuqianla obfuscation guide. https://fuqianla.net/docs.html?Android_SDK, 2019.

[24] Offphone download link. http://offphone.net, 2019.

[25] Paypal. https://www.paypal.com, 2019.

[26] Trpay obfuscation guide. http://pay.trsoft.xin/front/documentation.html, 2019.

[27] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking for real-time systems. In *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 414–425. IEEE, 1990.

[28] BeeCloud. Beecloud news. https://beecloud.cn/about/#honor, 2019.

[29] Chandan Kumar Behera and D Lalitha Bhaskari. Different obfuscation techniques for code protection. *Procedia Computer Science*, 70:757–763, 2015.

[30] Chih-Chung Chang and Chih-Jen Lin. Libsvm: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2011.

[31] Danqi Chen and Christopher Manning. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 740–750, 2014.

[32] Viktoria Felmetsger, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Toward automated detection of logic vulnerabilities in web applications. In *USENIX Security Symposium*, volume 58, 2010.

[33] OWASP Testing Guide. Testing for business logic. https://www.owasp.org/index.php/Testing_for_business_logic/, 2019.

[34] Boyuan He, Vaibhav Rastogi, Yinzhi Cao, Yan Chen, VN Venkatakrishnan, Runqing Yang, and Zhenrui Zhang. Vetting ssl usage in applications with sslint. In *2015 IEEE Symposium on Security and Privacy (SP)*, pages 519–534. IEEE, 2015.

[35] Prospective Industry Research Institute. China's syndication payment industry market prospects and investment strategic planning analysis report for 2018-2023. https://bg.qianzhan.com/report/detail/1703301644253052.html, 2018.

[36] iyiou. China syndication payment industry development report in 2018. https://www.iyiou.com/p/88682.html, 2018.12.28.

[37] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. *neural information processing systems*, pages 3111–3119, 2013.

[38] Chandan Kumar Giri MimansaGantayat. Security issues, challenges and solutions for e-commerce applications over web.

[39] Shunji Mori, Hirobumi Nishida, and Hiromitsu Yamada. *Optical character recognition*. John Wiley & Sons, Inc., 1999.

[40] M Niranjanamurthy and DR Dharmendra Chahar. The study of e-commerce security issues and solutions. *International Journal of Advanced Research in Computer and Communication Engineering*, 2(7), 2013.

[41] Giancarlo Pellegrino and Davide Balzarotti. Toward black-box detection of logic flaws in web applications. In *NDSS*, 2014.

[42] Leonard Richardson. Beautiful soup. https://www.crummy.com/software/BeautifulSoup/, 2019.

[43] David Sacks. System and method for third-party payment processing, February 7 2002. US Patent App. 09/901,962.

[44] Scrapinghub. Scrapy. https://scrapy.org, 2019.

[45] Statista. Number of users of leading mobile payment platforms worldwide as of august 2017. https://www.statista.com/statistics/744944/mobile-payment-platforms-users/, 2017.

[46] Fangqi Sun, Liang Xu, and Zhendong Su. Detecting logic vulnerabilities in e-commerce applications. In *NDSS*, 2014.

[47] TechNode. Briefing: Alipay now has over 1 billion users worldwide. https://technode.com/2019/01/10/alipay-1-billion-users/, 2019.

[48] Rui Wang, Shuo Chen, and XiaoFeng Wang. Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 365–379. IEEE, 2012.

[49] Rui Wang, Shuo Chen, XiaoFeng Wang, and Shaz Qadeer. How to shop for free online–security analysis of cashier-as-a-service based web stores. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 465–480. IEEE, 2011.

[50] Luyi Xing, Xiaolong Bai, Tongxin Li, XiaoFeng Wang, Kai Chen, Xiaojing Liao, Shi-Min Hu, and Xinhui Han. Cracking app isolation on apple: Unauthorized cross-app resource access on mac os. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 31–43. ACM, 2015.

[51] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Hui Liu, Qing Wang, Yueheng Zhang, and Dawu Gu. Show me the money! finding flawed implementations of third-party in-app payment in android apps. In *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)*, 2017.

APPENDIX

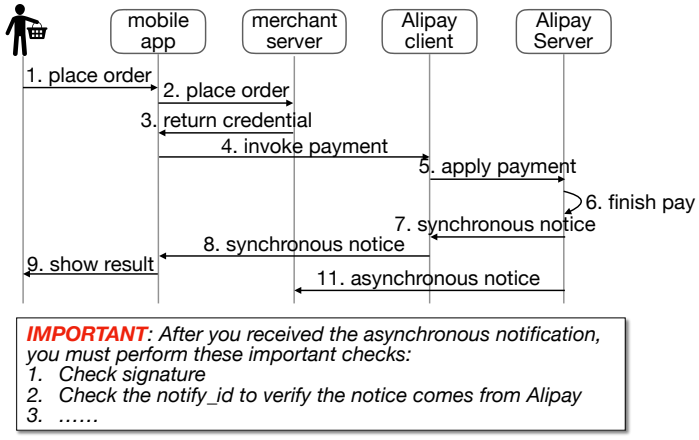


Figure 11: An excerpt of the Alipay diagram and security requirements.

Table 4: Collected apps

App	Package name	Category	Syndication	Downloads
Max+	com.dotamax.app	Game	BeeCloud	1,300,000
Zhihuiwuxi	com.hoge.android.wuxiwireless	Business	BeeCloud	1,160,000
Zhongyizhiku	com.zk120.aportal	Life	BeeCloud	1,100,000
Clean	com.ktls.fileinfo	Tool	BeeCloud	850,000
Yuebanchuxing	com.ynwl.yueban	Social	BeeCloud	770,000
Jiaoshisuishixue	cn.ixunke.suishixue	Education	BeeCloud	660,000
Chuangyebang	com.cyzone.news	Education	BeeCloud	490,000
Yikeweiqi	com.indeed.golinks	Game	BeeCloud	360,000
Hediandian	com.hoge.android.app.hdd	Business	BeeCloud	350,000
Zhihuiyancheng	com.hoge.android.yancwireless	Business	BeeCloud	170,000
Huiyouhui	com.huiuhuisc.zoj	Business	BeeCloud	90,000
Wuxianhuaian	com.hoge.android.huaian	Business	BeeCloud	70,000
Zhongyiguji	com.zk120.ji	Health	BeeCloud	50,000
Zhongyiyian	com.zk120.an	Health	BeeCloud	10,000
Quandashi	com.dream.ipm	Business	BeeCloud	10,000
Shuohua	com.etang.talkart	Art	BeeCloud	10,000
OffPhone	com.alion.silent	Tool	TrPay	830,000

Table 5: Abbreviation summary in alphabetical order.

Abbreviation	Denote
ADV	Adverbial
ATT	Attribute
b,b_i	Buyer state (No.i)
b_1	The initial state
B	Buyer
COO	Coordinate
d	A message transmitted among a buyer, a merchant and a syndicator
D	A set of messages
DoA	Documentation Analyzer
E	A function that drives the transition from one payment state to the next
FSM	Finite state machine
LfP	Logic-flaw Predictor
m, m_i	Merchant state (No.i)
m_f	The final state
M	Merchant
p, p_i	Payment processor state (No.i)
P	Payment processor
POB	Preposition-object
s	A state in an FSM
S	A set of payment states
SBV	Subject-verb
SD	Secure design goal
SI	Secure implementation goal
SR	Security Requirement
SRi	The No.i Security Requirement in Table 2
SR_{obj}	The object to check for an SR
SR_{para}	The parameters for an SR when checking
SR_{state}	An SR's corresponding state
TA	A set of trusted message attributes as collected from related API specifications
TC	A set of collected attributes for preconfigured merchant information
VOB	Verb-object
w, w_i	Syndicator state (No.i)
W	Syndicator

Understanding iOS-based Crowdturfing Through Hidden UI Analysis

Yeonjoon Lee[1]*, Xueqiang Wang[1]*, Kwangwuk Lee[1], Xiaojing Liao[1]
XiaoFeng Wang[1], Tongxin Li[2], Xianghang Mi[1]
[1]*Indiana University Bloomington,* [2]*Peking University*

Abstract

A new type of malicious crowdsourcing (a.k.a., crowdturfing) clients, mobile apps with hidden crowdturfing user interface (UI), is increasingly being utilized by miscreants to coordinate crowdturfing workers and publish mobile-based crowdturfing tasks (e.g., app ranking manipulation) even on the strictly controlled Apple App Store. These apps hide their crowdturfing content behind innocent-looking UIs to bypass app vetting and infiltrate the app store. To the best of our knowledge, little has been done so far to understand this new abusive service, in terms of its scope, impact and techniques, not to mention any effort to identify such stealthy crowdturfing apps on a large scale, particularly on the Apple platform. In this paper, we report the first measurement study on iOS apps with hidden crowdturfing UIs. Our findings bring to light the mobile-based crowdturfing ecosystem (e.g., app promotion for worker recruitment, campaign identification) and the underground developer's tricks (e.g., scheme, logic bomb) for evading app vetting.

1 Introduction

Crowdturfing is a term coined for underground crowdsourcing [44], in which an illicit actor (typically a cybercriminal) hires a large number of small-time workers to perform questionable and often malicious tasks online. Supporting such an operation is a crowdturfing platform, the underground counterpart of Amazon Mechanical Turk [1] that acts as an intermediary for the cybercriminal to recruit small-time workers for the hit jobs like creating fake accounts on an online store, posting fake Yelp reviews, spreading rumors through Twitter, etc. These attacks damage the quality of online social media, manipulate political opinions, etc., thereby threatening the public confidence in the cyberspace, which is the very foundation of the open web ecosystem.

Mobile crowdturfing. With the fast growth of mobile markets today, crowdturfing is extending its reach to mobile computing, serving illegal missions like inflation of an app's rating or mass collection of coupons or other bonus during a sales promotion. For this purpose, a mobile client (app) needs to be deployed to a large number of underground workers. Such an app, however, is prohibited by both Apple and Android app stores according to their guidelines [21, 28], and will be taken down once detected. Although dissemination of the crowdturfing apps is still possible in the fragmented Android world, through less regulated third-party stores, on the Apple platform, cybercriminals find it hard to reach out to the iPhone users, due to the centralized app vetting and installation enforced by the Apple App Store. To circumvent this security check, it has been reported [52] that crowdturfing Trojans have been increasingly used to infiltrate the iOS App Store, through embedding stealthy crowdturfing user interfaces (UI) in innocent-looking iOS apps. An example is shown in Figure 1. Compared with *web-based crowdturfing* [37, 45, 46, 49], these apps are used to deliver mobile based crowdturfing tasks, such as fake app review and app ranking manipulation. Also, they are characterized by utilizing hidden UI techniques to bypass app vetting and deliver tasks for their small-time workers, which raise the challenges for finding them. So far, little has been done to systematically discover and analyze such hidden crowdturfing apps, not to mention any effort to understand the underground ecosystem behind them.

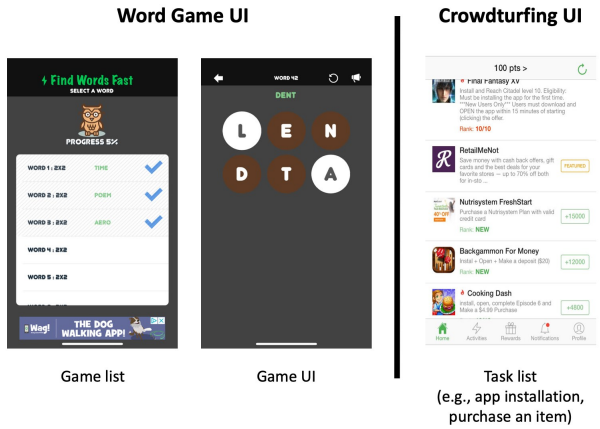


Figure 1: A Word Game with hidden crowdturfing UIs.

Finding crowdturfing apps. In this paper, we report the first measurement study on iOS crowdturfing apps. The study relies on the discovery of such malicious apps from the Apple App Store, which is challenging, due to the difficulty in identifying their elusive hidden UIs. These UIs are under the cover of benign ones and can only be invoked under some specific conditions (e.g., time, commands from C2 servers).

*The two lead authors contributed equally to this work.

Even when they indeed show up, likely they operate similarly as the legitimate UIs: no malware downloading, no illicit use of private APIs, etc. To capture their illegitimacy, one needs to read their content and understand their semantics. This, however, requires human involvement and therefore does not scale during app vetting. The attempt to detect such UIs becomes even more complicated for the third party, who does not have the source code of the related apps and therefore needs to work on binary executables. Indeed, our research has brought to light almost 100 such apps already published on Apple App Store, completely bypassing its vetting protection.

To address these challenges, we come up with a new triage methodology, *Cruiser*, that identifies the iOS apps likely to contain hidden crowdturfing UIs for further manual inspection. A key observation here is that such apps are characterized by their conditionally triggered UIs (e.g., triggered not by user actions but by network events), as demonstrated through UI transitions. Also, the content of such hidden UIs is related to crowdturfing semantically (e.g., app ranking manipulation), which is inconsistent with their hosting app's public description. These unique features make it possible to detect these iOS apps through a combination of binary, UI layout and content analyses. From 28,625 iOS apps covering 25 app categories, our system reports 102 most likely involving hidden crowdturfing UIs; considering the large scale of Apple App Store (2 million apps [3]) and the relatively high false detection rate (8.8%) of our tool, we manually examined all the 102 flagged apps, and found that 93 apps indeed contain hidden crowdturfing UIs.

Measurement and discoveries. Looking into the apps with hidden crowdturfing UIs reported by *Cruiser*, we are surprised to find that this new threat is indeed trending, with a big impact on today's mobile ecosystem. More specifically, from the 93 apps detected, we discover 67 different mobile crowdturfing platforms, which handle a variety of crowdturfing tasks, such as app ranking manipulation, fraud account registration, fake reviews, online blog reposting, and order scalping, etc. Also importantly, these apps are found to bypass app vetting several times and have a long lifetime. Such apps are popular, having been installed by a large number of users (32.4 million in total). Some of them even appear on the Apple leaderboards, with 25 of them ranked among the top 100 in their corresponding categories.

Also interesting is the ecosystem of mobile crowdturfing, as discovered in our study, which includes app promotion for worker recruitment, campaign identification, etc. In particular, crowdturfing platform owners are found to advertise their apps through multiple channels, including crowdturfing app gateway sites, in-app promotion and a pyramid (or referral) scheme that rewards the individuals for recommending crowdturfing apps to other users. In the crowdturfing app gateway sites, we observe that around 50% of hidden crowdturfing apps have been downloaded more than 18K times; there are 32.4 million downloads in total. Also, we find that the app with hidden UIs is in high demand from the underground market: e.g., cybercriminals are willing to pay hundreds of dollars for developing such an app to circumvent Apple's vetting.

Furthermore, we analyze the evasion techniques employed by the crowdturfing apps, and bring to light new techniques that utilize complicated conditions to trigger their malicious behaviors: such apps not only know whether they have passed Apple's review so they can change their behaviors accordingly, but also protect their hidden UIs with the conditions involving user interactions or communication with a malicious website. Up to our knowledge, such techniques have not been reported to the Apple platform before, and therefore bring new challenges to its vetting process. Further discovered in our research is the way underground developers reuse their product and work with each other: we see that different developers inject different crowdturfing UIs to similar apps, and the same developer hides the same UIs into her different products. Also interestingly, almost identical apps, with both over and cover UIs, are found to be submitted to the store under different developer IDs. We disclosed our findings to Apple, which acknowledged us and has removed all reported apps from the App Store, though new attack apps of this type continue to pop up due to Apple's lack of effective means to detect them; also upon Apple's request, we provided a list of fingerprints for eliminating the apps similar to the malicious ones.

Contributions. The contributions of the paper are outlined as follows:

• *New methodology*. We developed a novel approach that utilizes a binary-code analysis on UI hierarchy and Natural Language Processing (NLP) analysis on UI semantics to detect the iOS apps with hidden crowdturfing UI.

• *New findings*. *Cruiser* helps us gain new insights into the mobile crowdturfing ecosystem and exposes the underground developer's new tricks for evading Apple's app vetting. Also importantly, our study sheds light on a new attack vector that has long been ignored: use of hidden UIs to evade even most restrictive app vetting to distribute illicit content.

Roadmap. The rest of the paper is organized as follows: Section 2 provides background information for our study; Section 3 elaborates on the design of *Cruiser*; Section 4 presents our measurement study and new findings; Section 5 discusses the limitations of our current design and potential future research; Section 6 reviews related prior research and Section 7 concludes the paper

2 Background

Crowdturfing platform. As mentioned earlier, crowdturfing, also called malicious crowdsourcing, is an illicit business model, in which cybercriminals (i.e., *intermediaries*) recruit *small-time workers* to carry out malicious tasks (e.g., app ranking manipulation) for *dishonest third parties* (e.g., app

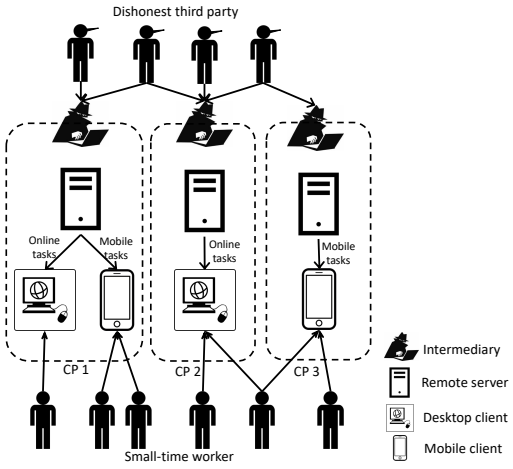


Figure 2: Overview of modern crowdturfing platforms, where "CP" represents a crowdturfing platform.

owner). Moving from the desktop browser-based clients (e.g., Zhubajie [5] and Sandaha [4]) to mobile devices, crowdturfing today increasingly happens through the apps deployed to workers' smartphones. As an example, consider a dishonest app owner who intends to inflate the app's installation volume and therefore seeks help from a crowdturfing platform; through the platform, the owner can pay workers to download and install his app so as to fake its popularity. Other hit jobs performed by the platform include the spread of fake reviews, defamatory rumors, etc.

Figure 2 illustrates modern crowdturfing platforms supporting both desktop browser-based clients and mobile clients. Such a platform, generally created and maintained by intermediaries, is designed to coordinate crowdturfing tasks and organize small-time criminals (workers) to do the tasks. As shown in the figure, a crowdturfing platform consists of servers to distribute crowdturfing tasks, and desktop browser-based clients or mobile clients to interact with the workers (e.g., publishing the tasks and checking the quality of the work). Unlike the platforms with browser-based clients, those with mobile clients mainly aimed at mobile-related crowdturfing (e.g., app ranking manipulation).

However, mobile crowdturfing clients, in the form of apps, are widely considered to be illicit by app stores, including Apple App Store [21] and reputable Android App stores like Google Play [28]. Especially for iOS crowdturfing clients, it is extremely hard for such apps to get through Apple's restrictive vetting process. Actually, from the underground forum, we find that some intermediaries seek experienced developers to build apps capable of infiltrating the Apple store, by hiding their crowdturfing UIs (Section 4.4). Also interestingly, due to the difficulty in publishing crowdturfing apps, we find from the Apple store that multiple servers even share one client (Section 4).

iOS UI design. The UIs of an iOS app include view, view controller (VC) and data: *view* defines the UI elements to be displayed (e.g., button, image, and shape); *data* is the information delivered through the defined UI elements; and a *VC* controls both views and their data to present a UI. All the VCs of an app and their relations, which describe the transitions between different UIs, form a *VC hierarchy*, with its root (called *anchor*) being the initial VC of the app or the VC launched by the iOS object *AppDelegate*. Implementing a VC hierarchy can be done using either VC transition APIs (e.g., *pushViewController:animated*), or storyboard [19], a visual tool in the Xcode interface builder. In the storyboard, a sequence of scenes are used to represent VCs, and they are connected by *segue objects*, which describe transitions between VCs. iOS employs *layout files* (a.k.a., nib files) to implement UIs, which can be generated using storyboard.

Over a VC hierarchy, developers commonly define two kinds of transitions between a pair of VCs: *Modal* and *Push*. A *modal VC* does not contain any navigation bar or tab bar, and is used when developers create outgoing connections between two UIs. To present a modal VC, the developer can directly use APIs (e.g., *presentViewController:animated:completion:*), or define a modal segue object [20] in a storyboard. An API needs to be called in order to dismiss such a modal VC. On the other hand, *Push* uses a navigation interface for VC transitions. Selecting an item in the VC pushes a new VC onscreen, thereby hiding the previous VC. Tapping the back button in the navigation bar removes the top VC and reveals the background VC. More specifically, developers can display the view of a VC by pushing it to the navigation stack using the *pushViewController:animated:* API, or define a push segue in a storyboard. In the meantime, tapping the back button will pop up the top VC from the navigation stack and makes the new top displayed.

In our research, we observe that hidden crowturfing UIs exhibits conditionally triggered navigation patterns in an app's VC hierarchy, including multiple root VCs as entry UIs, entry VC not triggered by the users nor dismissed by itself, etc. (Section 3.2).

Natural language processing. The semantic information our system relies on is automatically extracted from UIs using Natural Language Processing (NLP). Below we briefly introduce the key NLP techniques used in our research.

• *Word embedding*. Word Embedding is an NLP technique that maps text (words or phrases) to high-dimensional vectors. Such a mapping can be done in different ways, e.g., using the continual bag-of-words model or the skip-gram technique to analyze the context in which the words show up. Such a vector representation ensures that synonyms are given similar vectors and antonyms are mapped to different vectors. Tools such as *Word2vec* [50] could be used to generate such vectors. Word2vec takes a corpus of text (e.g., Wikipedia dataset) as inputs, and assigns a vector to each unique word in the

corpus by training a neural network. In our study, we leverage Word2vec to quantify the semantic similarity between the words based on the cosine distance of their vectors.

• *Topic model for keyword extraction*. Topic model is a statistical model for finding the abstract "topics" of a document, and topic modeling is a common text-mining tool for discovering keywords from corpora. Among various topic modeling approaches, Latent Dirichlet Allocation (LDA) [13] is one of the most popular methods. The basic idea is that documents are represented as random mixtures over latent topics, where a topic is characterized by a distribution over words, and the statistically significant words are selected to represent the topic. In our study, we leverage the LDA implementation of *Stanford Topic Modeling Toolbox* [48] for keyword extraction.

Threat Model. In our research, we consider an adversary who tries to publish iOS apps carrying hidden crowdturfing content on Apple App Store. Examples of such crowdturfing activities include fake review posting, app ranking manipulation and order scalping [15], etc. For this purpose, the adversary creates iOS apps with hidden crowdturfing UIs. These UIs are meant for displaying the tasks assigned by a crowdturfing platform and providing guidance on how to accomplish the tasks, so typically they do not ask for additional capabilities (guarded on iOS by entitlements). To publish such apps, the adversary is supposed to be knowledgeable about Apple's vetting process. Use of private APIs or side-loading are the focus of Apple's vetting and therefore not considered in our research. Also, in our research, we only cover native iOS apps. The cross-platform framework (e.g., react native) based apps, which are built using different languages (e.g., javascript), are out of the scope of this work.

3 Methodology

Here we elaborate on the design and implementation of a new technique for identifying apps with hidden crowturfing UIs. We begin with an overview of the idea behind *Cruiser*, and then present the design details of each component.

3.1 Overview

Architecture. Figure 3 illustrates the architecture of *Cruiser*, which includes a *Structure Miner* and a *Semantic Analyzer*. After fetching and decrypting iOS apps from App Store, *Structure Miner* takes as its input a set of decrypted iOS apps, and disassemble them. The disassembled apps are then utilized by the Structure Miner to construct a VC hierarchy for identifying the VCs with conditionally triggered UIs (e.g., two entry UIs). Here we define *checkpoint VCs* as all VCs associated with conditionally triggered UIs and their corresponding children VCs (see detail in Section 3.2). We also consider children VC, since VCs with conditionally triggered patterns

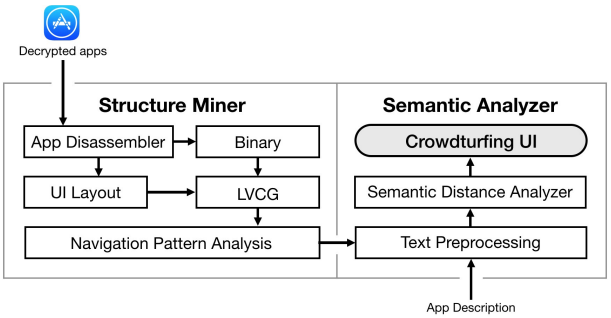


Figure 3: Architecture of *Cruiser*.

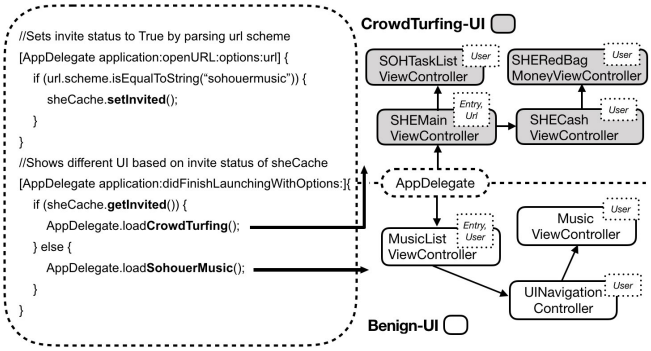


Figure 4: Pseudocode and simplified LVCG with conditionally triggered UIs.

sometimes may not contain sufficient texts for semantic analysis. On each checkpoint VC, the Semantic Analyzer further extracts texts from it, and evaluates its content through a set of NLP techniques to determine whether it is used for crowdturfing.

Example. To explain how *Cruiser* works, here we walk through its workflow (Figure 3) using a Music Player app with a hidden app ranking manipulation UI, *com.sohouer.music*. *Cruiser* first automatically decrypts the app and disassembles it into binary, UI layout files and resource files. Meanwhile, we also crawl the app's metadata (i.e., description for the Music Player app) from the App Store as the input for the Semantic Analyzer.

The Structure Miner processes the binary and UI layouts of the *com.sohouer.music* app and creates a VC hierarchy in the form of a labeled view controller graph (LVCG) (as shown in Figure 4). From the LVCG, our approach extracts VCs with conditionally triggered UIs and marks them as the checkpoint VCs. More specifically, the Structure Miner identifies VCs and VC transitions from the app binary and UI layout files to construct the LVCG (Figure 4). From the LVCG, the Structure Miner discovers the conditionally triggered UIs: two root VCs of *MusicListViewController* and *SHEMainViewController*, which indicate that there are two entry UIs for the app. Depending on whether the app has received a particular scheme invocation before, different main UIs will be

displayed when the app is launched. Therefore, these two VCs are labeled as checkpoint VCs for the follow-up semantic analysis.

Once the checkpoint VCs are found, the Semantic Analyzer then processes their text data to identify semantic features: the *MusicListViewController* VC contains a series of *Music Player* related words, such as $\{album, singer, shuffle, song, music, radio\}$, which are consistent with the app's description. On the other hand, the topic words under *SHEMainViewController* are $\{task, cash, earn, withdrawal, join, pay, reward\}$. Given the semantic inconsistency discovered, the Semantic Analyzer flags the app as a crowdturfing client.

Data collection. In our research, we collected 28,625 iOS apps for discovering new hidden crowdturfing apps, which we call *unknown set*. Specifically, we scanned the entire iOS app list from iTunes Preview website [34] using an app crawler running on an iPhone, and then selected the apps updated after Jan. 1, 2016 to download and decrypt. This is because apps with hidden crowdturfing UI is an emerging threat, and recently updated apps tend to have more active users. In this way, we collected 28,625 iOS apps, which cover 25 app categories.

3.2 Structure Miner

The Structure Miner is designed to identify the VCs with conditionally triggered UIs from an app's disassembled code and UI layout files. Examples of such patterns include two different main UIs, as discovered from *com.sohouer.music*, and the UI that can only be invoked by a specific network or other events, not directly by the user, indicating the potential presence of evasive behaviors. To discover such patterns, we first construct a VC hierarchy in the form of an LVCG through analyzing the app's binary and retrieving UIs from the UI layout files to identify their corresponding VCs and establish their transition relations among them. Then, from the LVCG, we search for predefined conditionally triggered UIs and mark those having these UIs as checkpoint VCs for further analysis.

LVCG. LVCG is a directed graph as shown in Figure 4, in which each node is a VC and each directed edge describes a transition from one VC (corresponding to a UI) to another.

Definition 1. An LVCG is a directed graph $G = (V, E, \alpha)$ over a node label space Ω, where:

1. V is a node set, with each node being a VC;
2. Edge set $E \subseteq V \times V$ is a set of transitions between VCs;
3. Node labeling function $\alpha : V \rightarrow \Omega$ marks each node with its UI properties and text data. Each node is given four property labels: *entry*, *user*, *url*, *others*. Table 1 shows the definition of each property and the corresponding method names.

LVCG construction. The construction of an LVCG requires both an app's binary and its UI layout files. This is because the VC of a UI is in the code and even the UI itself can be programmed through APIs (e.g., *initWithFrame:* API in *UIView*) so becoming part of the VC, and in the meantime, all the UIs built through storyboard can only be found in the layout files, including the transitions between them. To address this complexity, *Cruiser* builds two LVCGs, one from the binary and the other from the layout files, before combining them together.

Specifically, on the binary code, we look for system VC class names (e.g., *UIViewController*) and method names (e.g., setNavigationBarHidden), which help identify individual VCs and their properties (see Table 1). Then we track the data flows from a VC to another to recover the transitions between the detected VCs. For this purpose, our approach first maps the addresses in the binary code to symbols (e.g., class name, method name) using a binary analysis tool Capstone [7], and then uses a set of targeted system VC class names (e.g., UIViewController) and method names (e.g., setNavigationBarHidden) to recognize VCs and their properties (e.g., *entry*) from the symbols. After that, the Structure Miner performs a data-flow analysis using an implementation similar to the prior techniques [18, 23], to connect the transition APIs (*performSegueWithIdentifier:sender:*) discovered in a VC to another one, the transition target.

To construct a LVCG on the layout files under the storyboard folder generated by Apple's interface builder, we need to extract VCs and VC transitions from the files. The former can be found from the storyboard plist file that includes the mappings from VC names to the obfuscated names of nib files. The latter is recorded by the nib files, each of which carries a subset of a VC's properties, e.g., the types of some elements (such as botton, textbox, etc.) and the transitions between VCs.

Our approach directly recovers VCs from the plist file and further detects each VC's nib files from the mappings it records. More challenging here, however, is to identify the transitions between the VCs, since objects included in a nib file are undocumented. To enable the Structure Miner to interpret the file, we reverse-engineered part of its format relevant to the transition and content extraction. Specifically, we started from the interface builder, through which one can define one or multiple scenes to represent a UI and a *Segue* to describe a transition. Through a differential analysis, we compared the compiled nib files with and without a specific transition to pinpoint the nib objects corresponding to different *Segue* types (e.g., push, modal, unwind), such as *ClassSwapper*. From such objects, the Structure Miner is then able to collect the transitioning data, in the form of *src, dst, type, etc.*. This allows us to restore the recorded transition information and build up the LVCG of an app.

Given the LVCGs generated from the binary and the layout files, our approach automatically combines them together, based on the relations between the VCs on these graphs: particularly, when a transition is found from a VC in the layout to the one defined in the code, two LVCGs can then

Table 1: LVCG node properties and their corresponding method names.

Property	Definition	Method/Class names
entry	root VC	setRootViewController:
user	VC triggered by a user interaction	addTarget:action:forControlEvents:
url	VC rendering web content	openURL:, UIWebViewController
others	other properties (e.g., self-dismiss)	dismissViewControllerAnimated:completion:

be linked together through this VC pair. On the combined LVCG, further we remove the dead VCs introduced by the part of libraries and other shared code not used by an app. To this end, our approach performs a test to find out all the VCs that cannot be reached from the app's entry points (such as *AppDelegate*, the initial VC of the main storyboard) and drops them. In this way, we remove 1,053,161 dead VCs (55.4%) from the 28,625 iOS apps we collect (see Section 3.1).

Conditionally triggered UI extraction. Given 17 apps with hidden crowdturing UI collected from *91ssz* [8] (see detail in Section 3.4), without loss of generality, in our study, we consider two types of conditionally triggered UIs on the LVCG, as elaborated below:

• *More than one root VCs*. We consider an LVCG to be suspicious if it has more than one root VCs, i.e., app has two entry points, that is, two different root UIs. The root VC is the first one launched (by *AppDelegate*) when an app starts running. One evasion trick the adversary often plays is to run two root VCs, one legitimate and the other illicit, depending on some trigger conditions (e.g., the app's execution environment). For example, in the app *com.sohouer.music* (see Section 3.1), besides the benign UI (i.e., MusicListViewController), the hidden crowdturfing UI (i.e., SHEMainViewController) can also be invoked by *AppDelegate*). Such a pattern can be described as $|\alpha(v) == `root'| \geq 2$. In this case, we label the two VCs and their corresponding children VCs as checkpoint VCs for further semantic analysis.

• *VC not triggered by users*. If an *entry* VC or intermediate VC is not triggered by the user, but by other external events (e.g., network), i.e., $\alpha(v)[`entry'] = True \wedge \alpha(v)[`user'] = False$ or $\alpha(v)[`user'] = False \wedge \alpha(v)[`url'] = True$, we consider it as suspicious, since such UI is difficult to be triggered during app vetting. In such a case, we mark such a VC v and its children VCs as checkpoint VCs.

Looking into all 28,625 apps, we discover 34,679 checkpoint VCs using conditionally triggered UIs. These VCs are further evaluated by the Semantic Analyzer. Our evaluation (see Section 3.4) shows that the Structure Miner maintains a good coverage on hidden crowdturfing UIs while filtering out most legitimate apps.

3.3 Semantic Analyzer

The Semantic Analyzer determines whether checkpoint VCs are crowdturfing UIs. Serving this purpose is a set of NLP based semantic analysis techniques: we first extract UI texts from the VCs, and then find out whether they are related to crowdturfing by calculating the semantic distance between the texts and crowdturfing keywords.

Text discovery. As mentioned earlier, the format of the UI layout files (the nib files) is undocumented. However, they can be converted into the XML form using ibtool [42]. From their XML content, we can find plain-text strings under *NSString* objects, a property of UI element objects like button, table, textbox, font, color, etc. Some of these strings are part of the content a UI displays, while the others are not, depending on the type of the UI element objects. For example, *UIFont* and *UIColor* carry strings such as *".HelveticaNeueInterface-Regular"* and *"blackColor"* for defining fonts and UI color, respectively. To extract UI content from the nib files, we come up with a blacklist of UI element objects that do not include UI texts, and use that list to filter out irrelevant text strings. More specifically, we randomly sampled 70 iOS apps from our unknown set, which gives us 1,307 nib files including 28,469 *NSString* objects. We clustered them based on the types of their UI element objects, and manually went over all 103 types discovered. In this way, we constructed a blacklist with 21 patterns that cover 64 object types that do not contain any meaningful UI texts. Table 8 in Appendix shows the blacklist. When analyzing a given app, the Semantic Analyzer locates all *NSString* objects from its checkpoint VCs and further recovers their host UI element objects from the app's UI object tree (i.e., a tree built on layout files). If the element is on the blacklist, we ignore its *NSString* object.

In addition to the text strings in the *NSString* objects, other UI content can be embedded in images and therefore cannot be easily extracted. To collect more semantic information for crowdturfing UI detection, we utilize an app's meaningful variable names (e.g., *_album_id*), class names (e.g., *TicketDetailViewController*) and method names (e.g., *setSongIdsArrayM*), which are preserved in the binary's symbol table by the Object-C compiler. These human-readable symbols are recovered by our approach from the variables, class names, etc. output by Capstone [7] for each checkpoint VC. Also for the VCs with Web UIs (e.g., *UIWebViewController*), we include the text content collected from the URL embedded in the VC. An example of the data gathered from both UI layouts and a binary is presented in Table 2.

Crowdturfing UI identification. Given the UI content recovered from each checkpoint VC, we analyze whether such data is semantically associated with crowdturfing: to this end,

Table 2: Sample text data.

Object Type	Text Data
UILabel	"Proceed to checkout"
NSLocalizableString	"start making money"
Class Name	"TaxiViewController", "GameView" "TicketDetailViewController"
Method Name	"setSongIdsArrayM:", "setBuyAllProductId:"
Instance Variables	"_album_id", "_uploadMedia ", "_btnPaid"
CFString	"Select photo from photo library" "more clear free voice calls"
URL	*booking.com* "hotel" "city", "trip", "taxi"

we first preprocess the texts to address the issues like multi-language, noisy words, and then identify the keywords representing their semantics. In the meantime, we crawl a set of popular crowdturfing websites (e.g., Zhubajie [5] and Sandaha [4]) to build a crowdturfing word list. Words on the list are compared with the UI keywords using Word2vec [50] to find out their semantic distances. When such a distance becomes sufficiently small, the checkpoint VC is then flagged as a hidden crowdturfing UI. In the following, we elaborate on each step of this analysis.

At the preprocessing step, our approach runs Google Translate [2] to convert content in other languages into English. For the text in the languages without delimiters, Chinese in particular, we first use open source tools [27, 30] to segment texts into words before the translation; for the class/method names extracted from the binary, we tokenize them using regular expressions that cover common naming conventions (e.g., *CamelCase* style). Further, we drop all common stop words (e.g., NLTK stop words), and the frequent words from iOS frameworks and programming languages (e.g., "UIViewController", "ignoreTouch:forEvent:" and "raiseException"), as well as program language and debugging related texts (e.g., "socket", "connection", "memory", "allocation"). These words come from 74 framework-libraries of iOS 8.2.1, and are gathered in our research from sections such as *__cfstring* and *__objc_methname*. Selected from these documents are 1,806 frequent words whose inverse document frequency (IDF) values are larger than a threshold (we use $log(5)$ in our implementation). Also 1,031 program language and debugging related words are hand-picked for *Objective-C*, *Swift*, and *Javascript*.

After removing these words from a checkpoint VC, the remaining words are then analyzed using affinity propagation [26], which clusters them based upon their semantics (represented by the vectors computed using an embedding technique) and reports the most significant cluster. The words in such a cluster are then used by our approach to represent the semantics of their hosting VC.

To collect crowdturfing keywords, we crawl 280 web pages from the popular crowdturfing websites (i.e., Zhubajie [5] and Shandaha [4]). From these pages, we identify their topic keywords using the Latent Dirichlet Allocation (LDA) method. In this way, we build a crowdturfing list of 214 words. A problem for directly using these words is the observation that some of the crowdturfing words may also appear in legitimate apps: for example, "coupon" is certainly a meaningful word for a shopping app, not necessarily referring to the illicit task of bounty hunting. To address this problem, we compare these words with the keywords extracted from an app's description, dropping those related to the app's publicly stated functionality before the comparison below.

Given keywords discovered from the checkpoint VCs and the list of crowdturfing words, we run *Word2vec* [50] on each of these words, which maps the word to a vector that describes its semantics. Using these vectors, our approach measures the semantic relations between the UI keywords and the crowdturfing keywords by calculating their vectors' cosine similarities. For each UI keyword, its average similarity with all the crowdturfing keywords is used to determine its relevance with crowdturfing. We find that when the average relevance score of all the keywords of a checkpoint VC reaches 0.525 or above, the VC is nearly certain to be a crowdturfing UI.

3.4 Challenges in Identification

Here we evaluate *Cruiser* and elaborate on the challenges in crowdturfing app identification.

Evaluation with ground-truth set and unknown set. We evaluated *Cruiser* over the following *ground-truth datasets*: for the bad set, we collected the apps with hidden crowdturfing UIs from *91ssz* [8]. *91ssz* is a website that hosts the apps with the features (e.g., spam forums, earn money) violating Apple's guidelines. We manually examined 290 apps and confirmed 17 with hidden crowdturfing UIs (the other 273 apps do not have hidden UIs and are *only* accessible through third-party black markets). The good set were gathered from the top paid app list found from Apple App Store charts, which are considered to be mostly clean. We randomly sampled 17 of them (the same size of the bad set) to build the good set. Note that we manually examined those apps and verified that they are indeed benign. Running on these sets, *Cruiser* shows a precision of 88.9% and a recall of 94.1%.

Next we further report the results when running our approach on the unknown set, including all the apps collected from the Apple App Store (Section 3.1), at each stage of our analysis pipeline. We statically analyzed disassembled code and UI layout files over the 28,625 iOS apps, and discovered 34,679 checkpoint VCs, which are related to 3,999 (14.0%) apps using conditionally triggered UIs. Then, we executed the Semantic Analyzer, which flagged 102 apps. We manually examined all of them and found that 93 apps indeed contain hidden crowdturfing UIs. This gives us a precision

of 91.2%. The 9 falsely detected apps, though not including crowdturfing UIs, also turned out to be less legitimate. Below we elaborate on the missed apps and the falsely detected apps.

Missed apps. On the ground-truth set, only one crowdturfing app was missed by *Cruiser*. The app fell through the cracks due to inadequate semantic content extracted from their UIs. It is found to construct the URL for the content to be displayed during its runtime and dynamically loads crowdturfing pages through the URL. While *Cruiser* can find the suspicious view controller, it cannot statically gather semantic content from the crowdturfing pages and therefore fail to provide enough semantic information for the Semantic Analyzer to make a decision.

Determining the number of missed crowdturfing apps in the unknown set (with 28K iOS apps) is challenging. Given the low density of such malicious apps in the dataset, we could not randomly sample from the set hoping to capture ones missed by our methodology. So what we did in our study is to lower down the threshold used by the Semantic Analyzer for detection, which improved the recall, at the expense of precision. With the threshold decreasing from 0.525 to 0.513, our approach flagged 313 more apps. We manually analyzed all these apps and found only 3 new crowdturfing apps (false negatives), while the remaining 310 were all false positives. Looking into these 3 missed apps, interestingly we found that they were all web-based apps that dynamically download crowdturfing content from the web during their runtime, as we observed on the ground-truth set.

Falsely detected apps. All false detections reported come from the apps indeed carrying conditionally triggered UIs. These apps are not only structurally but also semantically related to a true crowdturfing app. More specifically, their hidden UIs all contain monetary content, which is one of the semantic features for crowdturfing apps. For example, among the 9 false detections, 7 are about "Health & Fitness" but actually include hidden lottery UIs. The remaining two are "Education" apps, which declare to be free but later display a remotely controllable UI asking for payment. Note that all these UIs are potentially unwanted, since they are undocumented (in the apps' description) and forbidden by Apple's guideline [21]. We consider these apps (with illicit UIs) as false detections, just because they are not directly related to crowdturfing.

Legitimate use of conditionally triggered UI. In Section 3, we report the observation of 14% apps including conditionally triggered UIs. Through a manual analysis, we found that these apps use two entry UIs to display notifications, a tour or a guide for the app, special events (e.g., New Year) and etc. All their hidden UIs cannot be reached through user interactions. This demonstrates the importance of the Semantic Analyzer, which utilizes NLP to determine the irrelevance of these apps to crowdturfing, thereby controlling the FDR of our approach.

3.5 Comparison to Other Approaches

***NaiveCruiser*: Semantic analysis on all VCs**. *Cruiser* is characterized by a two-step analysis (by the Structure Miner and then the Semantic Analyzer), first filtering out the VCs with normal navigation pattern and then analyzing the semantics of suspicious VCs. This strategy is designed to minimize the overheads incurred by the Semantic Analyzer, which is crucial for making our system scalable for analyzing the 28K apps in the wild. In the meantime, there is a concern whether the performance benefit comes with an impact on the technique's effectiveness, making it less accurate. To understand the problem, we compared our implementation of *Cruiser* with an alternative solution, called *NaiveCruiser*, which conducts a semantic analysis on all VCs in the app. This approach is fully tuned toward effectiveness, completely ignoring the performance impact.

In particular, we also evaluated the *NaiveCruiser* over the same *ground-truth datasets* we used to evaluate *Cruiser*. Running on these sets, *NaiveCruiser* shows a precision of 90.9% and a recall of 93.2%, which is in line with *Cruiser* (precision of 88.9% and recall of 94.1%). This indicates that our two-step design does not affect the effectiveness of detection. We also show the large performance degrade of *NaiveCruiser*, compared to *Cruiser*, in Appendix.

Crowdturfing keyword search. Simply searching for crowdturfing keywords is not effective. This is because the words used in crowdturfing UI (e.g., money, withdrawal, cash) are common, which often appear on other legitimate UIs (e.g., stock apps, accounting apps). Therefore, a simple keyword-based approach would bring in a high FDR (see below). Our approach utilizes a suite of techniques (e.g., looking for structural features of conditionally triggered UIs and corresponding VCs, removing words related to app descriptions) to avoid false reporting of legitimate UIs.

To understand how effective these techniques are, we evaluated the baseline – the naive keyword search on the 28K iOS apps. Specifically, we automatically extracted keywords from crowdturfing content collected from our ground-truth set, and then manually crafted a list of 32 most representative keywords for crowdturfing tasks (e.g., reward, task and installation). In the experiment, we studied the effectiveness of these keywords by first searching for the apps containing individual words and then analyzing their combinations (those including 2, 3, $\cdots$, 32 words). The more keywords an app includes, the more likely it is problematic but the fewer such apps would be found. In the end, we did not see any app involving more than 8 keywords. Among those carrying no more than 8 words, the highest precision achieved was 15.38% (an FDR of 84.62%), for those with 8 words. In this case, only 5 apps were reported. By comparison, our approach achieved a precision of 91.2%, reporting 93 malicious apps on the unknown set. This result demonstrates that the naive keyword search is indeed inadequate.

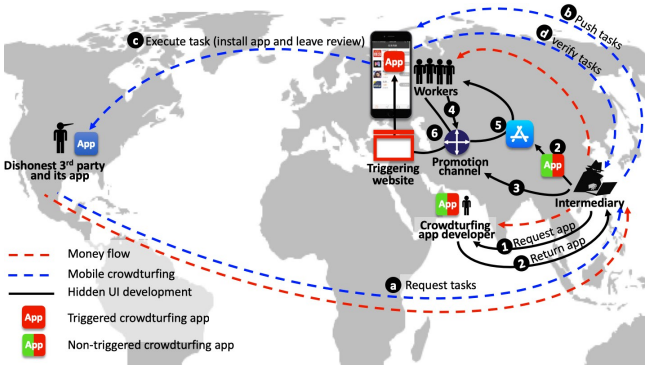


Figure 5: Overview of modern crowdturfing value chain, which consists of hidden crowdturfing app development (❶-❻) and mobile crowdturfing operations (ⓐ-ⓓ).

4 Understanding iOS-based Crowdturfing

Based on the detected crowdturfing apps, we further performed a measurement study to understand the iOS-based crowdturfing ecosystem. In this section, we first present as an example a real value chain of modern crowdturfing (Section 4.1), and then describe the scope and magnitude of this malicious activity as discovered in our research (Section 4.2), before elaborating the two key components of the value chain: *crowdturfing app development and promotion* (Section 4.3) and *mobile crowdturfing operations* (Section 4.4).

4.1 Mobile-Crowdturfing Value Chain

Before coming to the details of our measurement findings, first let us summarize the mobile-crowdturfing value chain discovered in our research.

A cybercriminal (i.e., intermediary), who owns a modern crowdturfing platform *chinazmob*, intends to publish a mobile client, which is downloadable from the App Store, to publish crowdturfing tasks and coordinate with small-time workers. Hence, the intermediary seeks underground app developers to build an app with hidden crowdturfing UIs (❶, see Section 4.3). The hidden crowdturfing UI will only be triggered when app users visit the website *ioswall.chinazmob.com*. Once done, the app *Pleasant Music* (id115****781), which disguises as a music player, passes the vetting of the App Store and is published (❷). Then, the intermediary promotes this app on social networks (❸) with links to the App Store and the triggering website *ioswall.chinazmob.com*. Small-time workers, who observe the promotion (❹) and download the *Pleasant Music* app (❺), will access the mobile crowdturfing client after triggering the hidden UI (❻) to execute crowdturfing tasks. Meanwhile, in the underground business of mobile crowdturfing, a dishonest mobile app owner of *Anjuke* who plans to inflate the app's installation volume reported by the App Store, pays for a crowdturfing platform *chinazmob* to manage crowdsourced app downloading tasks (ⓐ). Then, the intermediary will publish a task on its mobile client and recruit small-time workers (ⓑ) to do the task. These workers will install *Anjuke* and write fake reviews for the app (ⓒ). Once done and verified by the crowdturfing platform (ⓓ), the workers will get commissions from the platform.

In the rest of the section, we discuss the security implication introduced by these hidden UI apps, considering both crowdturfing app development and promotion and mobile crowdturfing operations in the value chain. As evidence for their impacts, those apps successfully infiltrated the App Store, even reached a high rank and bypassed the app vetting multiple times. In addition, we discovered various hidden UI techniques and the underground services that support the development of such apps. In particular, we revealed a set of techniques (e.g., logic bomb, scheme) deployed by the cybercriminals, as well as the underground services that are willing to pay $450 for developing such iOS apps. For app promotion, we identified 40 crowdturfing app gateway sites used by cybercriminals to promote 67.7% of such apps, which also enabled us to estimate the volume of the users. Furthermore, we report the findings related to *mobile-based* crowdturfing and discuss their insights, which have never been done before. For example, in contrast to the web-based crowdturfing dominated by a small number of platforms, on the mobile side we observed a fragmented crowdturfing market and a stealthy iOS crowdturfing ecosystem: we detected 93 hidden crowdturfing apps related to 9 campaigns, after clustering them based on similar app information, code structure and network behavior. Finally, we report a case study on an app with a hidden app ranking manipulation UI.

4.2 Landscape

Scope and magnitude. Our study reveals that apps with hidden crowdturfing UI are indeed trending in the Apple App store. Altogether, *Cruiser* detected 93 apps with hidden crowdturfing UIs, which are related to 67 crowdturfing platforms. To the best of our knowledge, this is the largest finding on mobile crowdturfing ever reported.

Apps with hidden crowdturfing UI, as discovered in our experiment, are found in 15 categories of the Apple App Store. As shown in Table 3, over 77.4% of the apps are in the categories of *Music*, *Utilities*, *LifeStyle*, and *Entertainment*. These apps are often built upon existing open source projects (see Section 4.4). Surprisingly, we found that some crowdturfing apps are of high ranks: six apps, including the wifi helper app (*cn.qimai2014.polarbearwifi*), the recorder utility app (com.amzhushou.app), the Temple Run style app (com.funinteract.ballgame) and several word guessing game apps reached top 20 of the leaderboard across different countries (e.g., *China*, *Laos*), based on the ranking data available from App Annie [6]; also, we observed that at least 14 apps were once ranked within the top 50, and 25 apps were in the

Table 3: Top 5 app store categories of apps with hidden crowdturfing UI.

Category	# apps	Benign UI examples
Music	32 (34.4%)	Ringtones, Piano Pieces
Utilities	15 (16.1%)	Recorder, File Manager
LifeStyle	15 (16.1%)	Story Teller
Entertainment	10 (10.8%)	Web Browsers, *Jeopardy*-style Quiz
Games	5 (5.4%)	Word Guess, Fruit Cutting

top 100 of their corresponding categories.

Impact of hidden crowdturfing apps. Furthermore, our study shows that the apps with hidden crowdturfing UIs have indeed successfully infiltrated App Store. Figure 6 illustrates the *Version* distribution of the crowdturfing apps. Most of them (73%) have only few updates, with a version number in the range from 0 to 1.5. However, still a non-negligible portion of apps (27% apps have *Version* $\geq$ 2.0) seem to be capable of carrying their suspicious payloads even to their higher versions. This is interesting since apps need to go through Apple's inspection for every new version submitted to the App Store.

Then, we analyzed the trend of the infiltration performed by the crowdturfing apps. Figure 7 shows the distribution of the number of such apps on the Apple App store over their release date. The trend-line based on the linear forecast regression indicates that those apps are still on the rise and require further attention. We observed that the newly-released apps with hidden crowdturfing UI have increased by 150% from Jan. 2015 to Jun. 2017.

4.3 App Development and Promotion

App development. Apparently, the development of crowdturfing apps is in strong demand on the underground market. Our research shows that one could get an illicit app, with desired hidden UIs, on the App Store for $450 [25]. Specifically, a quick search on Google yields dozens of recruitment posts for such app development; e.g.,*freelancer* [24,25], *Code Mart* [17], *witmart* [51], *dongcoder* [22], *Code4App* [16]. As shown in the task description [25], the illicit app to be developed should be capable of displaying a benign UI during app vetting, and switching to an illicit UI once it is published on the App Store.

Also illicit app developers tend to minimize the effort to develop the benign UIs for covering the crowdturfing ones. One common approach they take is to hide the crowdturfing UIs to the app built upon an open source project ([31,35,43]). In particular, we extracted strings from the benign VCs of the detected crowdturfing apps, and then searched them in leading code repositories (e.g., Github). Interestingly, we found that the benign UIs of six crowdturfing apps come from two open source projects: *ESTMusicPlayer* and *LittleFrog-MusicPlayer*. Note that according to Apple's guidelines [21] (4.3 and 4.2.6), such template apps should have been rejected. However, we observe that Apple seems to loosen its policy, which makes developing such illicit apps easier. To verify the observation, we designed a hidden crowdturfing app by utilizing one of the open source projects, ESTMusicPlayer [35], as the benign template. The app successfully got into the App Store in two days (we removed the app immediately before any user downloaded it).

UI hiding techniques: Triggers. We found that such illicit apps utilize a spectrum of UI hiding techniques to evade app vetting, which are described as follows:

• *Logic bomb*. Apparently, the adversary tends to trigger hidden crowdturfing UI when certain conditions are met (e.g., after app vetting). Some detected hidden crowdturfing apps contain logic bombs; e.g., the app sets off the hidden crowdturfing UI when a specified time (e.g., after "2017-01-18 00:00:00"), location (e.g., "isCN"), or device information (e.g., connected to cellular) conditions are met. For instance, the crowdturfing UI in *cn.music.s3b* is only activated when the device is connected to network and has its area/language code set to "zh".

• *C2 server*. Like bots, apps with hidden crowdturfing UI are also found to leverage command and control servers (C2 servers) to trigger their hidden UIs. For instance, *com.catTestPlay.app* retrieves a "status" code from its web server *http://[domain]/itunes_app/sound_dog* to decide whether to switch to its hidden UI.

• *Scheme*. Another interesting observation is that the app developers utilize extremely sophisticated triggering conditions, which even require the user to take certain actions. An interesting example is that a hidden crowdturfing UI can only be invoked by a specific scheme. Those apps promoted themselves on the social networks or websites; when users download those hidden crowdturfing apps from the App Store, the promoted sites provide the users an activation link to trigger the hidden crowdturfing UIs. More specifically, when the activation link is clicked, a scheme (e.g., *babyforring://[params]*), that releases the illicit UI, is sent to the app.

• *Others*. Several other techniques are also used to differentiate normal users' devices and vetting environment. As an example, we observe that a UI is hidden by the combination of scheme and logic bomb: the app *com.qianying.music* will first determine whether a user has logged into her WeChat app on the device, and then release its illicit UI only when receiving a scheme from a specific website.

App Clones. We observed that illicit app owners resubmitting clones of removed or existing illicit apps by only changing their bundle IDs through different Apple developer IDs; e.g., after *com.cloud.NHCore* was removed from App Store, it was quickly resubmitted as *com.good.jingling*. Developers also submitted multiple repackaged apps containing the same hidden crowdturfing UI; e.g., two apps, music

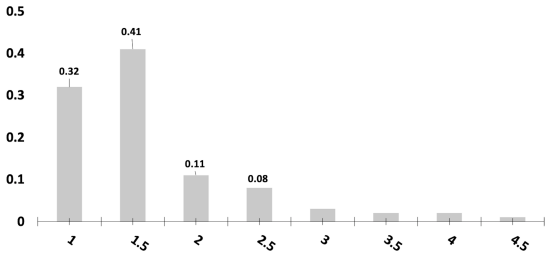


Figure 6: Version distribution of apps with hidden crowdturfing UIs.

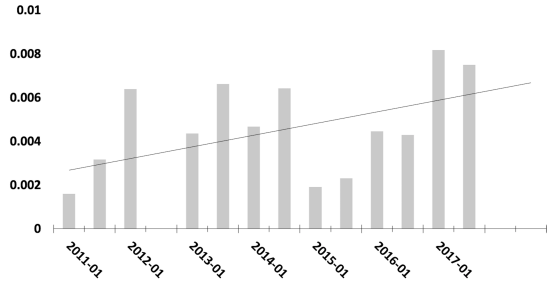


Figure 7: Release date distribution of apps with hidden crowdturfing UIs.

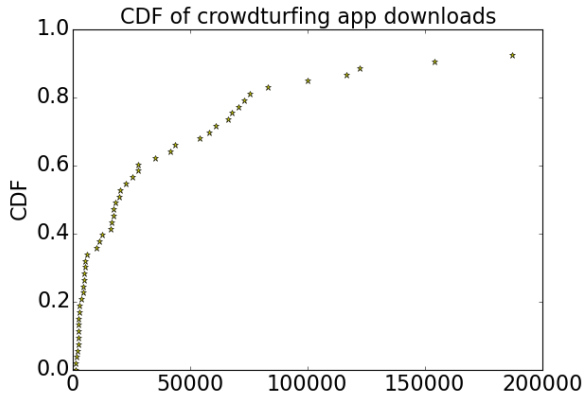


Figure 8: Cumulative distribution of crowdturfing app downloads.

player *com.yueyuemusic* and eBook reader *com.Qingyu* app, were found to integrate the identical crowdturfing platform (i.e., *rehulu.com*). To mitigate the threat of such persistent infiltration attempts, we provided a list of words that could help to fingerprint such apps upon Apple's request, and meanwhile are actively collecting resubmitted/repackaged hidden crowdturfing apps.

App Promotion and worker recruitment. To understand how crowdturfing platform owners disseminate such apps and recruit workers, we searched for the apps' names on the search engine and manually analyzed top-10 results to identify their promotion websites. In this way, we gathered 50 websites advertising 78 (83.9%) hidden crowdturfing apps. We found that the owners of these hidden crowdturfing apps promote their apps through multiple channels: advertising on the online communities (e.g., *BBS*, *tieba*), social networks (e.g., youtube, weibo), and crowdturfing app gateway sites (e.g., *app522.com*, *i8i3.com*).

Of particular interest is the crowdturfing app gateway sites, which refer the visitors to multiple hidden crowdturfing apps. We identified 40 such gateway sites that promoted 63 (67.7%) hidden crowdturfing apps. For example, the *com.cq.diaoqianyaner.pro.bookstore* app was found to be promoted on eight crowdturfing sites: *qisw123.com*, *ydzapp.com*, *eshiwan.com*, etc. Most intriguing is the discovery that all the apps *actively* promoted on those gateway websites have been detected by *Cruiser* from the unknown set. Since those websites record apps' download volume, we were able to estimate the number of these apps' users. Figure 8 illustrates the cumulative distribution of the number of downloads per crowdturfing app. As shown in the figures, around 50% of the crowdturfing apps were downloaded more than 18K times, with 32.4 million downloads in total.

Another interesting promotion channel is the referral bonus policy, which is provided through the app: the app's owner pays workers (users) if they invite other workers to use this app for crowdturfing. We found that 23% of the crowdturfing apps are using such a channel to recruit workers.

4.4 Mobile Crowdturfing Operations

Crowdturfing tasks. Table 4 illustrates the top-6 most common illicit crowdturfing tasks found in the apps with hidden crowdturfing UIs. As we can see here, most of them are mobile based crowdturfing tasks. According to our findings, app ranking manipulation is supported by a significant portion (88.2%) of crowdturfing apps, followed by fraud account registration, and fake review. Figure 9 illustrates the cumulative distribution of the task categories per app. We observe that about 62.5% apps only provide one kind of crowdturfing tasks, among which 86.7% are designed for iOS app ranking manipulation. Surprisingly, when analyzing apps seeking crowdturfing for iOS app ranking manipulation, we observe several popular and reputable apps. Examples include a calendar app, which ranked Top 10 in the App Store category of Utilities across 15 countries, and a restaurant review app, which ranked Top 10 in Lifestyle category across 49 countries.

To measure the task volume of an app (i.e., number of tasks $\times$ number of required workers per task), we crawled five apps' task information and the number of required workers through their crowdturfing UIs. Table 5 presents the average daily task volume for each app. For instance, the app ranking manip-

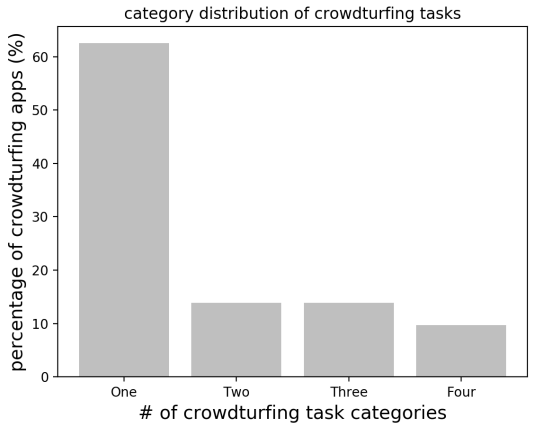


Figure 9: Distribution of the categories of crowdturfing tasks per app.

Table 4: Top-6 most common illicit crowdturfing tasks in apps with hidden crowdturfing UIs.

Crowdturfing tasks	# apps	# download (K)	Highest ranking
App ranking manipulation	82	32,268	5
Fraud account registration	28	15,618	64
Fake review	13	1,218	79
Bonus scalping	11	13,990	18
Online blog reposting	9	14,602	19
Order scalping	9	601	122

ulation app *com.zhang.samusic* has a daily task volume of 42,064 for manipulating 24 apps. Given an average task price of $0.14, the revenue for all those tasks is around $5.88K.

Furthermore, we analyze network traffic of such apps to study their servers, which distribute the tasks to the apps (see Figure 2). Interestingly, due to the difficulty in publishing crowdturfing apps, we find that multiple servers even share one client. In particular, besides their own servers, six apps are found to receive crowdturfing tasks from seven other servers (e.g., *qumi.com* and *domob.cn*) and all these tasks are related to app ranking manipulation.

Campaign discovery. In contrast to the web-based crowdturfing platforms [49], which are dominated by a few popular websites, we observed that the iOS-based crowdturfing platforms are more diverse. To study the relations among these crowdturfing apps, we built a graph for campaign discovery and further manually analyzed large campaigns identified. In the graph, each app is regarded as a node, and an edge connecting two apps represents that they are all from the same developer, with similar code or similar network behaviors. In particular, we crawled apps' developer information from the iTunes Preview website [34]. Then, we checked the common strings referenced by different apps' hidden crowdturfing UIs. If the strings from two different apps have more than 90% in common, we link them together. To capture the network behavior, we triggered all these apps by signing onto their platforms. If two apps' hidden crowdturfing UIs connect to the same server, we consider them to belong to the same campaign.

Table 5: Task volume and price of five apps with hidden crowdturfing UIs

App	# tasks	task volume	Per task price
com.zhang.samusic	24	42,064	0.14
com.roidmi.mifm	29	29,000	0.12
com.miaolaierge.iosapp	12	7,500	0.13
com.applyape.yycuimian	15	16,715	0.11
com.jialiang.weka	8	10,000	0.14

Table 6: Top-3 campaigns with most apps with hidden crowdturfing UI.

Campaign	# apps	Remote server
uxiaowei	9	uxiaowei.com
apptyk	6	apptyk.com laizhuan.com diaoqianyaner.com.cn
rehulu	6	rehulu.com

Table 6 shows top-3 campaigns with most crowdturfing apps. The largest one includes nine apps with hidden app ranking manipulation UIs, and all of them connect to the server *uxiaowei.com*. Interestingly, we observe that seven crowdturfing app owners (e.g., id109****906, id110****416, id110****262, id114****820) are related to this campaign. This campaign enjoyed a long lifetime, from May 2016 to March 2018.

4.5 Case Study

Here we introduce a typical app with hidden crowdturfing UI *sohouermusic*, which disguises as a music player, but also receives app ranking manipulation tasks (download, install, make up fake reviews, etc.). We observed that triggering the illicit service is surprisingly difficult, and such triggering process is designed to evade app vetting. Specifically, the *sohouermusic* app is promoted on popular social networks (e.g., WeChat), which redirect users to a website (*play.sohouer.com*). Only when a user visits the website on his iPhone and requires an invitation scheme *sohouermusic://invite=[serial number]* to be sent, will the app load its hidden UI. However, before the UI is actually rendered, the *sohouer* app checks whether it has passed the vetting process via its server, and the hidden crowdturfing UI shows up only when the remote server re-

sponds with *"isreview: 0"* and a *scripturl*. Besides acting as a client of a crowdturfing platform, such an app also stealthily collects user's data ; e.g., device type, version, jailbreak status, location. Another interesting observation is that the *sohouermusic* developers are persistent: after the *sohouermusic* app was removed (after we reported to Apple), the hidden crowdturfing UI was quickly repackaged into a *sohouercamera* app and was submitted through a different developer account.

5 Discussion

Evasion. The current implementation of *Cruiser* is based on identifying two types of conditionally triggered UIs for further semantic analysis (see Section 3.2). Hence, to evade *Cruiser*, the adversary may use the hidden crowdturfing UI, which is triggered by users and also avoids the root UI. Such evasion techniques, however, will cause the possible crowdturfing UIs to be triggered during app vetting. This is because all clickable elements may be triggered by Apple employee's manual or automatic analysis during app vetting [47]. This defeats the purpose of hidden UI.

The adversary may play other evasion tricks, by hiding semantic texts on the hidden crowdturfing UI to downgrade the accuracy of the Semantic Analyzer. In particular, the adversary can show crowdturfing related texts in the images, or obfuscate class names and method names, even dynamically fetch the crowdturfing related content. One possible solution is to run an Optical Character Recognition (OCR) tool [36] to extract the texts from images in the resource files, which enables to identify enough UI semantic even when the code is obfuscated. Considering the dynamically fetched hidden crowdturfing content, the adversary may deliver it on runtime using dynamic code loading (e.g., JSPatch [12]). However, Apple regulates and carefully monitors those dynamic code enabling techniques (e.g., hot patching frameworks) to minimize the attack vector; recently, Apple even bans or rejects any apps that use hot patch [39] from their App Store.

Limitations. Although *Cruiser* can already achieve a precision around 90%, still human involvement is needed to ensure that the apps reported are indeed problematic. Therefore, in the current form, it can only serve as a triage tool, instead of a full-fledged detection system. Also, as mentioned earlier, our current design is focused on iOS based apps, since cybercriminals have more intentions to utilize hidden UI to infiltrate the iOS app store than that of Android: centralized app vetting and installation make it hard for the crowdturfing app to reach out to the iPhone users. In the meantime, based on our observations, such hidden crowdturfing apps exist, though less pervasive, in the Android world. In particular, we conducted a small-scale study to find whether our detected apps have Android versions by searching for app names on Google Play, third-party stores and app download portals, and further manually examining them. We did not find any hidden crowdturfing apps, but did observe blatant crowdturfing apps (without hidden UIs) in less regulated third-party Android app stores.

Moreover, besides crowdturfing, we do think that cybercriminals can use hidden UI techniques for other abusive services, such as delivering unauthorized content, or even malware. When looking into such apps (those found in our research to carry hidden UIs but not perform crowdturfing), we found instances such as covering a phishing UI behind a travel app. A natural follow-up step is to investigate all abusive services exploring hidden UI to infiltrate the iOS app store and characterize the underground markets behind them. We will leave this as our future work.

Ethical issue. Our research only involved analysis of pre-existing code and app content and did not collect new data during the study. Therefore, it is just a secondary analysis of already published materials, which does not constitute human-subject research. Another ethical concern comes from the potential that *Cruiser* could be used to identify possible benign hidden UIs; e.g., for censorship circumvention. Here we clarify that *Cruiser* is just a methodology for discovery and understanding of a new type of cybercrime, and during our study, we did not observe any such censorship evasion attempts. We acknowledge that any evasion detection techniques, including ours, could also be used for censorship. In the meantime, our methodology has been tailored towards crowdturfing detection: e.g., the features used by the structure miner are based upon the structures of real-world crowdturfing apps, the Word2vec model and other NLP components are all built on crowdturfing data. We are not sure how effective our approach would be when applying it to detect other types of hidden content, and how much additional effort is needed to make it a full-fledged censorship tool.

Responsible disclosure. Since the discovery of apps with hidden crowdturfing UI, we have been in active communication with Apple. So far, we have reported all the apps detected in our research to Apple, who has removed all of them from the App Store; also upon Apple's request, we provided a list of fingerprints for eliminating the similar apps.

6 Related Work

Study on crowdturfing. The ecosystem of *web-based* crowdturfing has been studied for long. Motoyama et al. [37] identified the labor market Freelance involved in service abuse (e.g., fraud account creation) and characterized how pricing and demand evolved in supporting this activity. Wang et al. [49] studied two Chinese online crowdturfing platforms and also revealed the impact of the crowdturfing followers task on those platforms to microblogging sites. Stringhini el al. [45] investigated five Twitter follower markets to study the size of these markets and the price distribution of their service. Su et al. [46] studied the spamming activity of "Add To Fa-

vorites" by collecting the several "Add To Favorites" tasks information from one crowdturfing platform. In our research, to the best of our knowledge we for the first time investigate the crowdturfing platforms on the mobile devices, and reveal several unique characteristics; e.g., fragmented crowdturfing markets, mobile targeted crowdturfing tasks, stealthy worker recruitment channel, hidden crowdturfing UI techniques.

Illicit iOS app detection. Compared with Android, the Apple platforms are much less studied in terms of their security protection. Egele et al [23] proposed PiOS, which uses control flow analysis to detect privacy leaks in iOS apps. Deng et al [18] presented an approach to detect private API abuse by binary instrumentation and static analysis. Chen et al. [14] determines potentially harmful iOS libraries by looking for their counterparts on Android. Bai et al. [11] and Xing et al. [53] uncovered several zero configuration and cross-app resource sharing vulnerabilities, and proposed the corresponding detection methods. Understanding the security implications of hidden crowdturfing UI in iOS apps has never been done before. Also, none of the prior research provides a UI based detection mechanism to identify illicit iOS apps with hidden UI.

Text analysis for mobile security. Numerous studies have looked into *apps' UI texts* to detect mobile threats such as task jacking, mobile phishing attack, ransomware, or to protect user privacy. AsDroid [33] checks the coherence between the semantics of the UI text (e.g., text of button) and program behavior associated with the UI (e.g., button) to detect malicious behavior (e.g., sensitive API) in Android apps such as sending short messages and making phone calls. Heldroid [10] uses a supervised classifier to detect threatening *sentences* from Android apps to detect ransomware. SUPOR [32], UIPicker [38] and UiRef [9] identify sensitive user inputs within user interfaces to protect user privacy. In particular, SUPOR [32] extracts layouts by modifying the static rendering engine of the Android Developer Tool (ADT). UIPicker [38] operates directly on the XML specification of layouts. UiRef [9] resolves the semantics of user-input widgets by analyzing the GUIs of Android applications. It improves the accuracy of SUPOR by addressing ambiguity of descriptive text through word embedding. In addition to UI texts, researchers intensively leverage Natural Language Processing (NLP) to process *app descriptions* for mobile security research. Examples include WHYPER [40] and AutoCog [41], which check whether an Android app properly indicates its permission usage in its app description, CHABADA [29] applied topic modeling technique on an app's text description to help infer user's expectation of security and privacy relevant actions. Different from previous works, our work compared the semantics of conditionally triggered UI texts of iOS apps, crowdturfing keywords and app descriptions to identify hidden crowdturfing apps. Also, sensitive or private APIs are not used for detection in our work as the illicit behavior of the app we detect are based on UI not API. Also, different from SUPOR, UIPicker and UiRef, we extract UI texts from UI hierarchies (LVCG) we generated from iOS apps.

7 Conclusion

In this paper, we report our study on illicit iOS apps with hidden crowdturfing UIs, which introduce conditionally triggered UIs and a large semantic gap between hidden crowdturfing UI and other UIs in the app. Exploiting these features, our crowdturfing UI scanner for iOS, *Cruiser*, utilizes iOS UI hierarchy analysis technique and NLP techniques to automatically generate a UI hierarchy from binary and UI layout files and investigate conditionally triggered UI and the semantic gap to identify such illicit apps. Our study shows that *Cruiser* introduces a reasonable false detection rate (about 11.1%) with over 94.1% coverage. Running on 28K iOS apps, *Cruiser* automatically detects 93 apps with hidden crowdturfing UIs, which brings to light the significant impact of such illicit apps: they indeed successfully infiltrate App Store, even bypassing app vetting several times. What is worse, we observed an increasing trend of the number of such apps in App Store. Our research further uncovers a set of unique characteristics of iOS crowdturfing, which has never been revealed before: for example, we observe several remote crowdturfing servers share one iOS crowdturfing app as a client, which may be due to the difficulty of infiltration; also, such illicit apps were promoted by crowdturfing gateway sites to recruit workers, etc. Moving forward, we further investigate the hidden UI techniques providing by illicit app developers, including logic bomb, command and control infrastructure, and scheme technique etc.

8 Acknowledgements

We are grateful to our shepherd Gianluca Stringhini and the anonymous reviewers for their insightful comments. This work is supported in part by NSF CNS-1801365, 1527141, 1618493, 1801432, 1838083 and ARO W911NF1610127.

References

[1] Amazon mechanical turk: Access a global, on-demand, 24x7 workforce. https://www.mturk.com.

[2] Google translate. https://translate.google.com.

[3] Number of apps available in leading app stores 2018. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.

[4] Sandaha. http://sandaha.cc.

[5] Zhubajie. https://www.zbj.com.

[6] App annie. https://www.appannie.com/en/, Mar. 2010.

[7] Capstone: The ultimate disassembler. http://www.capstone-engine.org, Nov. 2013.

[8] 91ssz. A website that provides ios apps with illicit features. http://www.91ssz.com/app/iphone/, Mar. 2017.

[9] B. Andow, A. Acharya, D. Li, W. Enck, K. Singh, and T. Xie. Uiref: analysis of sensitive user inputs in android applications. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 23–34. ACM, 2017.

[10] N. Andronio. *Heldroid: Fast and Efficient Linguistic-Based Ransomware Detection*. PhD thesis, 2015.

[11] X. Bai, L. Xing, N. Zhang, X. Wang, X. Liao, T. Li, and S.-M. Hu. Staying secure and unprepared: understanding and mitigating the security risks of apple zeroconf. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 655–674. IEEE, 2016.

[12] bang590. Jspatch: bridging objective-c and javascript using the objective-c runtime. https://github.com/bang590/JSPatch, May 2015.

[13] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[14] K. Chen, X. Wang, Y. Chen, P. Wang, Y. Lee, X. Wang, B. Ma, A. Wang, Y. Zhang, and W. Zou. Following devil's footprints: Cross-platform analysis of potentially harmful libraries on android and ios. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 357–376. IEEE, 2016.

[15] G. Cheng. *7 Winning Strategies For Trading Forex: Real and actionable techniques for profiting from the currency markets*. Harriman House Limited, 2007.

[16] Code4App. Code4app: Looking for ios chameleon app developer. http://www.code4app.com/thread-14820-1-1.html, Sep. 2017.

[17] coding mart. Recruitement for ios chameleon app developer. https://mart.coding.net/project/11325, Nov. 2017.

[18] Z. Deng, B. Saltaformaggio, X. Zhang, and D. Xu. iris: Vetting private api abuse in ios applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 44–56. ACM, 2015.

[19] A. Developer. Storyboard: Guides and sample code. https://developer.apple.com/library/content/documentation/General/Conceptual/Devpedia-CocoaApp/Storyboard.html, Sep. 2013.

[20] A. Developer. Using segues. https://developer.apple.com/library/content/featuredarticles/ViewControllerPGforiPhoneOS/UsingSegues.html, Sep. 2015.

[21] A. Developer. App store review guidelines. https://developer.apple.com/app-store/review/guidelines/, Dec. 2017.

[22] dongcoder. In demand of chameleon for app vetting. http://www.dongcoder.com/detail-678294.html, Sep. 2017.

[23] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. Pios: Detecting privacy leaks in ios applications. In *NDSS*, pages 177–183, 2011.

[24] Freelancer. Freelancer: looking for developer for lottery chameleon app. https://www.freelancer.com/projects/php/app-edt-15321896/, Apr. 2017.

[25] Freelancer. We need to do a universal application on ios, and then display our url through the interface. https://www.freelancer.com/projects/iphone/need-universal-application-ios-then/, Apr. 2017.

[26] B. J. Frey and D. Dueck. Clustering by passing messages between data points. *science*, 315(5814):972–976, 2007.

[27] fxsjy. Jieba chinese text segmentation. https://github.com/fxsjy/jieba, Jul. 2013.

[28] Google. Developer policy center. https://play.google.com/about/developer-content-policy/#!?modal_active=none, Dec. 2017.

[29] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1025–1035. ACM, 2014.

[30] T. S. N. L. P. Group. Stanford word segmenter. https://nlp.stanford.edu/software/segmenter.shtml, May 2006.

[31] hellclq. ios app: Happy english sentences 8k. https://github.com/helloclq/HappyEnglishSentences8000, Aug. 2013.

[32] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang. Supor: Precise and scalable sensitive user input detection for android apps. In *USENIX Security Symposium*, pages 977–992, 2015.

[33] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang. Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1036–1046. ACM, 2014.

[34] A. Inc. itunes preview (app store). https://itunes.apple.com/genre/ios/id36?mt=8, Jul. 2008.

[35] P. King. Estmusicplayer. https://github.com/Aufree/ESTMusicPlayer, Nov. 2015.

[36] S. Mori, H. Nishida, and H. Yamada. *Optical character recognition*. John Wiley & Sons, Inc., 1999.

[37] M. Motoyama, D. McCoy, K. Levchenko, S. Savage, and G. M. Voelker. Dirty jobs: The role of freelance labor in web service abuse. In *Proceedings of the 20th USENIX conference on Security*, pages 14–14. USENIX Association, 2011.

[38] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang. Uipicker: User-input privacy identification in mobile applications. In *USENIX Security Symposium*, pages 993–1008, 2015.

[39] T. C. P. N. NETWORK. Apple removes 45,000 apps in china. http://www.asiaone.com/digital/apple-removes-45000-apps-china, Jun. 2017.

[40] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. Whyper: Towards automating risk assessment of mobile applications. In *USENIX Security Symposium*, pages 527–542, 2013.

[41] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen. Autocog: Measuring the description-to-permission fidelity in android applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1354–1365. ACM, 2014.

[42] D. Quesada. ios interface builder utility. https://github.com/davidquesada/ibtool.

[43] SimonLo. Hulumusic. https://github.com/SimonLo/HuluMusic, Apr. 2017.

[44] J. Song, S. Lee, and J. Kim. Crowdtarget: Target-based detection of crowdturfing in online social networks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 793–804. ACM, 2015.

[45] G. Stringhini, G. Wang, M. Egele, C. Kruegel, G. Vigna, H. Zheng, and B. Y. Zhao. Follow the green: growth and dynamics in twitter follower markets. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 163–176. ACM, 2013.

[46] N. Su, Y. Liu, Z. Li, Y. Liu, M. Zhang, and S. Ma. Detecting crowdturfing add to favorites activities in online shopping. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*, pages 1673–1682. International World Wide Web Conferences Steering Committee, 2018.

[47] M. Tabini. How apple is improving mobile app security. https://www.macworld.com/article/2047567/how-apple-is-improving-mobile-app-security.html, SEP 2013.

[48] S. T. M. Toolbox. Stanford word segmenter. https://nlp.stanford.edu/software/tmt/tmt-0.4/, May 2006.

[49] G. Wang, C. Wilson, X. Zhao, Y. Zhu, M. Mohanlal, H. Zheng, and B. Y. Zhao. Serf and turf: crowdturfing for fun and profit. In *Proceedings of the 21st international conference on World Wide Web*, pages 679–688. ACM, 2012.

[50] Wikipedia. Word2vec: a model to produce word embeddings. https://en.wikipedia.org/wiki/Word2vec, Feb. 2018.

[51] witmart. Buy covering ios apps for 30,000 cny. http://www.witmart.com/cn/app-software/jobs/jobid_34788.html, Oct. 2017.

[52] C. Xiao. Pirated ios app store's client successfully evaded apple ios code review. https://researchcenter.paloaltonetworks.com/2016/02/pirated-ios-app-stores-client-successfully-evaded-apple-ios-code-review/, Feb. 2016.

[53] L. Xing, X. Bai, T. Li, X. Wang, K. Chen, X. Liao, S.-M. Hu, and X. Han. Cracking app isolation on apple: Unauthorized cross-app resource access on mac os. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 31–43. ACM, 2015.

9 Appendix

9.1 Performance evaluation of *Cruiser* and *NaiveCruiser*

To understand the performance of *Cruiser*, we measured the time it takes to process all the apps in the unknown set, on our Red Hat server using 14 processes. The breakdowns of the delays observed at each stage (Structure Miner and Semantic Analyzer) are reported in Table 7. As we can see here, on average, 27.4 seconds were spent on each app. The results demonstrate that *Cruiser* scales well and can easily process a large number of iOS apps. Furthermore, we evaluated the performance of *NaiveCruiser* (Table 7). As we can see, in the absence of the conditionally triggered UI detection step to first filter out legitimate VCs, the performance overhead of the Semantic Analyzer became overwhelming: introducing a delay at least 14 times as large as our original approach, which makes it difficult to scale. In addition, we evaluated the performance of app collection. On average, downloading an app took 15 seconds and decrypting it took 10 seconds; however, the time varied greatly depending on the network speed, program sizes and etc. In total collecting and decrypting apps took 3 months.

Table 7: Running time at different stages, where SM means Structure Miner and SA means Semantic Analyzer.

Cruiser	Average time (s/app)	*NaiveCruiser*	Average time (s/app)
SM	18.88	LVCG construction	16.2
SA	8.56	SA	122.95
Total	27.43	Total	139.15

9.2 UI element objects without semantic UI texts

Table 8: UI element objects without semantic UI texts

Pattern type	UI element object	Parent UI element object[1]
A[3]	NSKey	*[2]
A	UIColor	*
A	UIFont	*
A	UINibKeyValuePair	*
A	NS.rectval	*
A	UIViewContentHuggingPriority	*
A	UIViewContentCompressionResistancePriority	*
A	UIOriginalClassName	*
A	UINibName	*
A	UIDestinationViewControllerIdentifier	*
A	UIActionName	*
A	UISource	*
A	UIDestination	*
A	UIStoryboardIdentifier	*
A	NSLayoutIdentifier	*
B[4]	UIProxiedObjectIdentifier	UIProxyObject
B	UIAction	UIStoryboardUnwindSegueTemplate
B	UIKeyPath	_UIAttributeTraitStorage
B	_UILayoutGuideIdentifier	_UILayoutGuide
B	UIKeyPath	_UIRelationshipTraitStorage
B	runtimeCollectionClassName	UIRuntimeOutletCollectionConnection

[1] Parent UI element object: The parent UI object of UI element object.
[2] *(asterisk): Any Object.
[3] Type A: The string of a UI element object will be removed regardless its parent.
[4] Type B: The string of a UI element object will be removed only if its parent UI element object also matches.

BITE: Bitcoin Lightweight Client Privacy using Trusted Execution

Sinisa Matetic
ETH Zurich

Karl Wüst
ETH Zurich

Moritz Schneider
ETH Zurich

Kari Kostiainen
ETH Zurich

Ghassan Karame
NEC Labs

Srdjan Capkun
ETH Zurich

Abstract

Blockchains offer attractive advantages over traditional payments such as the ability to operate without a trusted authority and increased user privacy. However, the verification of blockchain payments requires the user to download and process the entire chain which can be infeasible for resource-constrained devices like mobile phones. To address this problem, most major blockchain systems support so called lightweight clients that outsource most of the computational and storage burden to full blockchain nodes. However, such verification leaks critical information about clients' transactions, thus defeating user privacy that is often considered one of the main goals of decentralized cryptocurrencies.

In this paper, we propose a new approach to protect the privacy of light clients in Bitcoin. Our main idea is to leverage the trusted execution capabilities of commonly available SGX enclaves. We design and implement a system called BITE where enclaves on full nodes serve privacy-preserving requests from light clients. However, as we will show, naive processing of client requests from within SGX enclaves still leaks client's addresses and transactions. BITE therefore integrates several private information retrieval and side-channel protection techniques at critical parts of the system. We show that BITE provides significantly improved privacy protection for light clients without compromising the performance of the assisting full nodes.

1 Introduction

Since its inception in 2008, Bitcoin has fueled considerable interest in decentralized currencies and other blockchain applications. The main goals of blockchains include a distributed trust model and increased user privacy. Several other blockchain platforms, such as Ethereum [4], leverage the same open or permissionless model as Bitcoin, while platforms like Hyperledger [15], Ripple [10] and R3 [9], enable closed or permissioned blockchains. Most blockchains implement a decentralized time-stamping mechanism that ensures eventual consistency of *transactions* by collecting them from the underlying peer-to-peer (P2P) network, verifying their correctness, and including them in connected blocks. This process imposes heavy requirements on bandwidth, computing, and storage resources of blockchain nodes that need to fetch all transactions and blocks issued in the blockchain, locally index them, and verify their correctness against all prior transactions. For instance, a typical Bitcoin installation requires more than 200 GB of storage today, and the sizes of popular blockchains are growing fast [12, 5]. Therefore, users operating resource-constrained clients like mobile devices cannot afford to run their own full node.

Lightweight clients and privacy. To address such heavy resource requirements, most open blockchain platforms support *lightweight clients*, targeted for devices like smartphones, that only download and verify a small part of the chain. As a matter of fact, according to [24], in $73-85\%$ of $5.8-11.5$ million active Bitcoin wallets users control keys. Since there are $\sim 10,000$ full nodes [11], estimated 4.2-9.8 million wallets are lightweight clients. For example, Bitcoin provides the BitcoinJ [2], PicoCoin [8] and Electrum [3] clients implementing the Simple Payment Verification (SPV) mode [44], where the clients connect to a full node that has access to the complete chain and assists the client in transaction confirmation. Transactions contain inputs and outputs that are bound to *addresses* owned by users. As the full node has to learn all transactions issued and received by the requesting client to confirm them, such payment verification obviously violates user privacy.

To improve user privacy, several clients support *filters* (e.g., Bitcoin's BIP37 [31] and Ethereum's LES [6]). The goal of filters is to allow the client to define an anonymity set in an attempt to hide its real addresses from the full node. For instance, BIP37 supports Bloom filters [18] that allow the client to define a set of transactions, with false positives, that are requested from the full node. Essentially, this approach presents a trade-off between communication efficiency and privacy: a filter that returns many false positives provides a

larger anonymity set but requires more communication. Although such filters can be configured to be efficient, recent studies have shown that in practice they offer almost no privacy [25]. Ergo, none of the current light clients provides adequate privacy with practical performance overhead.

Our solution. Our goal is to improve the privacy of Bitcoin lightweight clients without compromising the performance of the assisting full nodes. The starting point of our solution is to leverage the commonly available trusted computing capabilities of SGX enclaves [23] on full nodes. We propose BITE (for ***BI****tcoin lightweight client privacy using* ***T****rusted* ***E****xecution*), a solution in which a potentially untrusted entity runs a full node with an SGX enclave that serves transaction confirmation requests from clients. Since SGX provides code integrity and data confidentiality for enclaves, such a solution can preserve privacy (confidentiality) and completeness (integrity) of client requests.

Unfortunately, simple usage of trusted computing is not sufficient to solve our problem. While SGX prevents an adversary that controls malicious software from directly accessing enclave's memory, secret-dependent access patterns to external storage, such as transaction databases, can reveal the client's address. SGX is also susceptible to side-channel attacks, where malicious software on the same platform infers secret-dependent enclave data access patterns or control flow by monitoring shared resources like caches [20, 43, 27, 50]. Thus, the simple usage of SGX would still leak the client's addresses to malicious full node.

Given such limitations of SGX, *the primary research problem and contribution of this paper is how to design and implement a solution that enables private processing of light client request in the presence of enclave leakage without compromising the system's overall performance.* To address this non-trivial challenge, we carefully select and apply known private information retrieval (PIR) and side-channel protection techniques and combine them into a novel solution that meets our performance requirements. We emphasize that in our application the assisting full node needs to process a large blockchain database to serve client requests, and thus straightforward usage of generic SGX side-channel protection systems, such as Raccoon [47], Cloak [28] or ZeroTrace [49], would result in either excessive performance overhead or imperfect side-channel protection. Instead of using such systems directly, we pick low-level primitives and apply them at critical points in our system to achieve more complete protection and better performance.

We design two variants of our solution. Our first variant, *Scanning Window*, is similar to the current SPV clients that verify transactions using block headers and Merkle paths received from the full node. To prevent leakage from file accesses and message sizes, we design a customized chain access mechanism that hides the client's transactions and the relationship between the size of the response and the number of read blocks. Our second variant, *Oblivious Database*, allows the client to verify the amount of coins associated with its addresses by querying a specially-crafted version of the unspent transaction output (UTXO) database. To prevent leakage from database accesses, we leverage a well-known Oblivious RAM (ORAM) algorithm [52]. (Prior to us, usage of ORAM from enclaves has been proposed in systems like ZeroTrace [49].) This variant allows even *lighter* clients that no longer need to download and verify Merkle paths.

To prevent software-based side-channels, we adopt further protections from recent SGX research. The basic building block for our control-flow hiding is the `cmov` instruction [7] that enables building oblivious execution of branches. (We adopt this technique from the Raccoon system [47].) To prevent leakage from data access patterns we apply additional defenses, such as iterating over the entire data structure when an element is accessed based on the protected client address.

Results. We show that our solution provides strong privacy protection. In both of our variants, the external data access patterns are independent of the protected client address. The side-channel protections in the Oblivious Database variant also make the enclave's memory accesses (both code and data) independent of the address, thus preventing leakage caused by known SGX side-channels [20, 43, 27, 50, 58, 36]. While similar protections can also be used for the Scanning Window variant, they impose a high overhead, which is why we recommend using Oblivious Database if side-channels are a concern. Our solutions also fail gracefully: even if the used SGX processor would be completely broken (e.g., through a physical attack), the adversary cannot double spend or steal users' coins or wallets.

In terms of performance, our solution is comparable to the SPV scheme. The Oblivious Database variant increases the full node's *storage* moderately (e.g., additional 4 GB). The required *communication* is significantly lower (e.g., 12 KB instead of 17 MB per client request). The *processing* cost for incoming client requests is reduced (e.g., 0.5s instead of 1.1s), but the processing cost for new blocks is higher (79s instead of 2s). Even compared to SPV without privacy protection, our solution adds no processing time or communication overhead (in fact, BITE's processing is faster by 0.1s and the response size is 2kB smaller). The full node can be easily made responsive for incoming client request during block updates by using two enclave instances in parallel. The Scanning Window variant requires no additional storage and its communication cost is lower than in SPV. The processing cost is also comparable when full side-channel protection is not used.

We argue that BITE emerges as the *first* practical solution that provides strong privacy protection for lightweight Bitcoin clients. Our solution can be integrated into existing full nodes and lightweight clients with minor modifications to the existing software. While BITE is designed for Bitcoin,

we stress that it finds direct applicability in various other blockchain platforms as well.

Contributions. In summary, in this paper we make the following contributions:

• *Novel approach.* We propose leveraging commonly available trusted execution capabilities of SGX enclaves for improved lightweight Bitcoin client privacy.

• *New system.* We design and implement a system called BITE that carefully combines a number of PIR and side-channel protection techniques to prevent leakage.

• *Evaluation.* We show that BITE significantly improves client privacy without compromising full node performance. We argue that BITE is the first practical way to provide strong privacy for lightweight Bitcoin clients.

The remainder of this paper is organized as follows. Section 2 describes our problem and Section 3 outlines our approach. Section 4 explains the details of our system BITE. Section 5 covers security analysis and Section 6 provides performance results. We provide discussion in Section 7, review related work in Section 8, and conclude in Section 9.

For readers unfamiliar with SGX and ORAM, we provide brief introductions in Appendices A and B.

2 Problem Statement

In this section, we provide background on Bitcoin lightweight clients, explain the limitations of known approaches and define requirements for our solution.

2.1 Bitcoin Lightweight Clients

Bitcoin [44] is the first and still most popular cryptocurrency based on blockchain technology. It enables users to perform payments by issuing *transactions* that transfer Bitcoins (BTC) from one or more transaction inputs to one or more outputs. Each of the outputs is bound to an *address* that is derived from a user's public key. A user that knows the corresponding private key is able to spend the Bitcoin contained in the transaction output.

When a user wants to perform a payment, she creates a transaction that contains inputs, outputs, and the signatures that allow her to spend the inputs. Subsequently, the transaction is propagated to all nodes using a peer-to-peer network created by the system's participants. Miners, a special type of nodes, collect valid transactions into blocks and solve Proof-of-Work (PoW) puzzle to make the contained transactions hard to revert. A miner that successfully finds a valid PoW, broadcasts the block to all other nodes, who then verify its correctness and include it in their copy of the chain.

To verify transactions, Bitcoin users, or clients, need to store the full history of all Bitcoin transactions. This approach puts a heavy load on client implementations in terms of network and storage, and as a consequence, makes transaction confirmation on mobile clients infeasible. To address this concern, the original Bitcoin paper proposed a solution called *Simplified Payment Verification* (SPV) [44]. In this technique, light clients store only block headers, check their PoW puzzles and then request their own transactions and the Merkle paths that are needed to verify their presence in the blocks from a full node that stores the entire chain.

Improvement proposal BIP 37 [31] introduced Bloom filters [18] that allow a light client to request a subset of all transactions to preserve some privacy without needing to download all transactions for each block. A Bloom filter [18] is a probabilistic data structure that consists of a set of hash functions and a bit array where each bit is set to one if one of the hash functions hashes one of its inputs to the index of the bit in the array. This allows checking if a value is contained in the filter by hashing the value with each of the hash functions and checking whether the corresponding bit is set. If this is not true, the value was not an input. If it is true, however, the value *might* have been an input or a false positive. The false positive rate can be set by the creator of the filter.

In Bitcoin light clients, Bloom filters are used to encode transactions or addresses, and allow a full node to determine which transactions to send to a lightweight client without letting the full node know the exact addresses. A lightweight client prepares a Bloom filter to which she adds all of her addresses and sends it to the full node. The full node then checks for incoming (or past, if requested) transactions whether they match the Bloom filter. If they match, she sends them to the client together with the Merkle path needed for verification. The client can adjust the false positive rate to increase her privacy. If the false positive rate is higher, the client will receive more irrelevant transactions, in an attempt to hide her true addresses with a larger anonymity set.

2.2 Limitations of Known Solutions

The use of Bloom filters to receive Bitcoin transactions from an assisting full node inherently creates a trade off between performance and privacy. If a client increases the false positive rate she receives more transactions which provides increased privacy, as any of the matching addresses could be her real addresses, but it also means that she needs the network capacity to download all of these transactions. In the extreme cases, the filter matches everything, i.e., the client downloads the full blocks, or the filter only matches the client's addresses, i.e., she has no privacy at all.

Gervais et al. [25] have shown that using Bloom filters in Bitcoin light clients leaks more information than was previously thought. In particular, if the Bloom filter only contains a moderate number of addresses, the attacker is able to guess addresses correctly with high probability. For example, with 10 addresses the probability for a correct guess is 0.99. They also show that, even with a larger number of

addresses, the attacker is able to correctly identify a client's addresses with high probability if she is in possession of two distinct Bloom filters from the same client (e.g., due to a client restart). Hearn [30] later expanded on why solving these issues is hard (e.g., need for resizing). Furthermore, it is likely that an attacker using additional de-anonymization heuristics, such as the ones described in [16, 41], could further increase the probability to guess correctly.

Finally, a lightweight client cannot be sure that she receives *all* transactions that fit her filter from a full node. While the full node cannot include faulty transactions in the response, as this would be detected by the client when recomputing the Merkle root, the client cannot detect whether she has received all requested transactions. This problem can be solved by requesting transactions from multiple nodes, which again imposes more network load on the client.

Another solution would be to run the SPV protocol with Bloom filters over a network anonymity mechanism such as Tor. While this would prevent the full node from learning the IP address of the client, the full node could still correlate queries from the same client based on the addresses that leak from Bloom filters. We argue that query unlinkability is a useful privacy property in systems like Bitcoin and can be considered similar to the transaction unlinkability in systems like Zcash (that provide more advanced privacy protection).

2.3 Requirements

The high-level goal of this paper is to develop a solution that provides better privacy for lightweight clients without compromising the system performance. More precisely, our solution should meet the following requirements:

(**R1**) **Privacy.** Lightweight clients should be able to verify that their transactions are confirmed on the blockchain or check the amount of coins associated with their addresses without revealing their addresses to the potentially untrusted entity that controls the assisting full node. The full nodes should not be able to link queries from the same light client that could allow them to incur additional information regarding the client's transactional pattern or behavior.

(**R2**) **Completeness.** The verification process should guarantee that no valid transactions have been omitted.

(**R3**) **Performance.** The performance of the system should be comparable to or better than current light client schemes.

3 Our Approach

The main idea behind our approach is to leverage commonly available Trusted Execution Environments (TEEs) such as Intel's SGX enclaves [33, 23] running within full nodes to provide a privacy-preserving verification service to light clients. Besides increased privacy, TEEs can enable better performance in terms of reduced processing and bandwidth, and guarantee completeness of received responses.

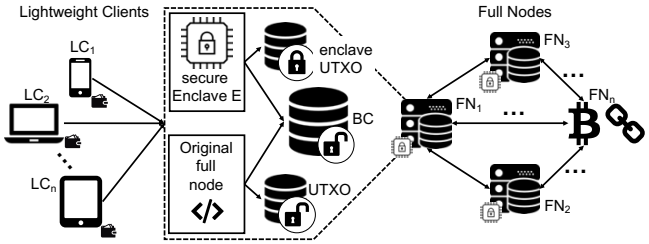


Figure 1: **System model.** Lightweight clients request transaction verification from enclaves hosted on full nodes.

In short, SGX provides a set of security enhancements in the processor that allow creation of small applications, called *enclaves*, whose data confidentiality and code integrity is protected from any malicious software running on the same platform, including the privileged OS.

A simple way to leverage SGX would be a solution where the light client sends its wallet private key to an enclave on the assisting full node. Using that key, the enclave can perform any operation on behalf of the user, including transaction verification. However, such simple solution has a critical drawback. If the used enclave is compromised, the adversary can steal all user's coins. Such approach might give the owners of full nodes an undesirable economic incentive to break their own SGX processors, e.g., using physical attacks.

To avoid such incentives, we choose a different approach. In our solution, when a client needs to verify a transaction or check the amount of coins associated with the user's addresses, the client connects to one of the full nodes that supports our service. The client performs remote attestation and establishes a secure channel to the enclave. Then, the lightweight client sends the addresses that the user is interested in to the enclave. The enclave obtains all the required verification information from the locally stored blockchain or custom unspent transactions database (UTXO) and sends back a response to the client that can verify it. Importantly, the client's private key is never shared with the enclave which enables safe adoption of our solution.

We envision two types of deployment for our system. In the first example deployment, a well-recognized company could provide such a verification service. In the second example, any volunteer currently running a Bitcoin full node could adopt our extension and start providing the service to lightweight clients. In both cases, to incentivize deployment by the full nodes, the service could be run in exchange for some small renumeration (i.e., verification fees).

3.1 System Model

Figure 1 shows our system model that consists of full nodes $FN_1 ... FN_m$ and lightweight clients $LC_1 ... LC_n$. When a lightweight client LC_i wants to acquire information about its

transactions or addresses, it can connect to any full node FN_j that supports our service and hosts an enclave E_j. Full nodes download and store the entire blockchain (BC) locally and based on that maintain a database that contains all unspent transaction outputs (UTXO). Our system additionally maintains a specially-crafted version of the UTXO, called *enclave UTXO*, in an encrypted (sealed) form.

In SGX, enclave memory is limited to 128MB. Although swapping memory pages is supported (swapping requires expensive encryption and integrity verification [17]), the complete blockchain (BC) and the database of unspent transaction outputs (UTXO) are significantly larger (as of Jan 2019, 200GB [12] and 2.8GB [13] or more, respectively) than the enclave's memory limits . Therefore, these databases are stored on the local persistent storage.

3.2 Adversary Model

We consider an adversary who controls the OS and any other privileged software on the full node. For example, the adversary could be a malicious administrator or an external attacker who has remotely compromised the OS on the full node. Since the adversary controls the OS, she can schedule and restart enclaves, start multiple instances, and block, delay, read, or modify all messages sent by enclaves, either to the OS itself or to other entities over the network. We assume that the adversary cannot break the hardware security enforcements of Intel SGX. That is, the adversary cannot access processor-specific keys (e.g., attestation or sealing key) and she cannot access enclave runtime memory that is encrypted and integrity-protected by the CPU. (Although we consider SGX trusted, in Section 5 we discuss enclave compromise and show that our solution can handle it without any financial loss.) Finally, we assume that common cryptographic primitives like encryption or signatures are secure.

3.3 Challenges

Secure and practical realization of our approach under the defined attacker model involves several technical challenges.

Leakage through external accesses. Since the adversary controls the OS, she can observe access patterns to any *external* resources, such as files or databases stored on the disk. Although externally stored data can be sealed (encrypted by the CPU such that only the same enclave can decrypt), the OS can infer information about the accessed element by observing access patterns to individual records, such as files or database entries. In a simple implementation of our approach, the adversary could infer the client's addresses by observing which entries the enclave reads from a (sealed) UTXO database when processing a client request.

Similarly, enclaves rely on the OS to perform communication operations which allows it to infer information about the enclave's communication patterns. Even if messages are encrypted by the enclave, the message sizes, frequency and destination can leak information. In our case, the adversary could determine how many transactions are included to the response by observing response sizes.

Leakage through side channels. The SGX architecture is also susceptible to *internal* leakage. Numerous, recently demonstrated side-channel attacks against SGX show that internal leakage is a relevant concern. For example, by monitoring CPU caches the OS can infer secret-dependent data and code accesses inside the enclave's memory [20, 43, 27, 50]. The OS can also infer enclave's secrets by monitoring the memory pages that the enclave requests [58]. Researchers have also demonstrated side-channel attacks using the CPU's branch prediction functionality [36]. In a simple implementation of our approach, the adversary can monitor address-dependent branching in the enclave's control flow and data accesses and thus determine the client's addresses.

4 BITE System

In this section we present a system called BITE that realizes the above approach securely and addresses the aforementioned challenges. In particular, we present two variants of the same approach that serve slightly different purposes.

Our first variant, *Scanning Window*, can be seen as an extension to the current SPV verification mode, but without reliance on bloom filters. Based on the client request, an enclave on the full node *scans* the blockchain and replies with a set of Merkle paths that the client can use to verify its transactions using downloaded block headers. This variant allows the client to check that each of its transactions are confirmed on the blockchain. As Bitcoin provides only eventual consensus, the client may want to additionally verify that the blocks where its transactions are placed have been extended with a sufficient number of valid blocks (e.g., six).

Our second variant, *Oblivious Database* is a completely new verification mode for lightweight clients. In this variant, the enclave on the full node maintains a specially-crafted version of the unspent transaction outputs (UTXO) database and when a client sends a verification request, it checks for the presence of client's outputs in this database using oblivious database access (ORAM [52]) and responds accordingly. Such verification allows the client to check how many coins are currently associated to its addresses, with significant performance improvements over SPV.

In both variants, the client performs remote attestation and establishes a TLS connection to the enclave. We note that current light clients communicate with the full nodes without encryption. Existing full node functionality, such as participation in the P2P network and mining, remain unaffected. Therefore, our system can be seen as a simple add-on to ex-

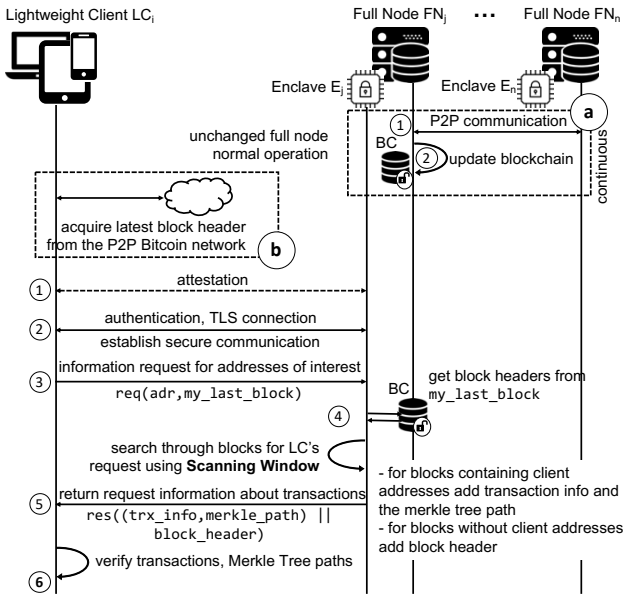


Figure 2: **Scanning Window operation.** Light client creates a secure connection to an enclave on full node and sends a request with its address and last known block. The enclave scans the locally stored chain and prepares a response with the size proportional to the number of scanned blocks.

isting full nodes. For clients, payment execution remains unchanged. Payment verification requires minor additions (attestation and TLS) when Scanning Window is used or slightly bigger changes with Oblivious Database variant.

4.1 Scanning Window Variant

In our first variant, we want to improve the privacy of the current SPV verification mode. When a client needs to verify transactions, it constructs a request that specifies the addresses of interest and the last block that it has in its internal state and sends that to the secure enclave residing on the full node. The enclave reads the locally stored blockchain database using a custom scanning technique that normalizes the relationship between response sizes and actually accessed data to hide the data/block access patterns and ensure client privacy. Figure 2 shows the operation of this variant, and we describe the details as follows:

Initialization and continuous operation.

(**a**) On initialization the Full Node FN_j connects to the Bitcoin network (**a-1**) and downloads the full blockchain (**a-2**). Similarly, the locally stored blockchain database is updated for each new block that is appended to the chain (i.e., as new blocks are received over the P2P network).

(**b**) The lightweight client installation package includes a checkpoint block header from a recent date. When the client is started for the first time, it downloads all newer block headers from the peer-to-peer network and verifies that (i) they all have correct Proof of Work and (ii) the hash chain of the downloaded headers leads to the checkpoint. Once the client's internal state it synchronized with the peer-to-peer network, it stores a small number of the newest headers (e.g., six blocks from the head of the chain to handle shallow forks). The client can update its internal state by downloading newest block headers periodically or before each transaction verification request. The network and storage requirements of this process are minor and easily met even by clients with severe resource constraints.[1]

Client request handling.

(**1**) The Lightweight Client LC_i performs attestation with the secure Enclave E_j residing on the full node FN_j.

(**2**) If the attestation was successful, the Lightweight Client LC_i establishes a secure communication channel to the Enclave E_j using TLS.

(**3**) The Lightweight Client LC_i sends a request containing the addresses of interest and a block number that specifies how deep in the chain transactions should be searched for verification. Typically, this number would be saved from the previous interaction with a full node or in the case of the first transaction verification the number could roughly match the date when the client started using Bitcoin.

(**4**) The Enclave E_j starts *scanning* its locally stored copy of the blockchain (BC) for the requested address and range of blocks using a scanning technique described in detail below.

(**5**) In preparation of the response, the Enclave E_j does the following: for blocks containing client addresses it adds the full transaction information and the corresponding Merkle tree path to the response, while for blocks without client addresses it only adds the block header.

(**6**) The Lightweight Client LC_i verifies that (i) the received block headers match its internal state and (ii) the received transactions and Merkle Tree paths match to the block headers. The client considers such received transactions as confirmed (assuming that they are sufficiently deep in the chain). The client updates its internal state regarding the latest verified block number and closes the connection to the enclave.

Block scanning details. As explained in Section 3.3, enclave execution can leak information in various ways. For example, if our solution would simply return each matching transaction (and the corresponding Merkle Tree) in the specified range of blocks, based on the size of the response the adversary could deduce how much information of interest

[1] For example, obtaining block headers for a checkpoint that is one month old, would require 300 kB of downloaded data (one-time operation) and updating the block headers once per day would require 10 kB of communication per day. Storing the latest six headers takes less than 1 kB of storage.

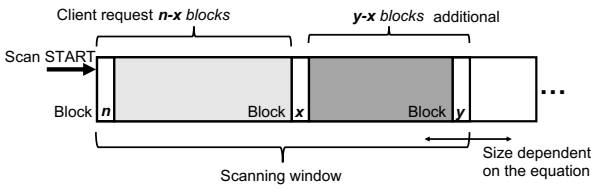


Figure 3: **Block reading in Scanning Window**. Depending on the number of requested blocks (up to x) and the number of matching transaction in them, we read potentially extraneous blocks (up to y) to keep the ratio between the read blocks and the response message size constant.

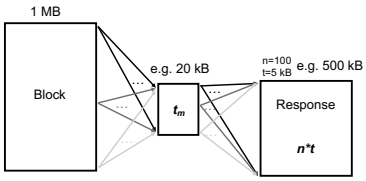


Figure 4: **Oblivious copying in Scanning Window.** The data is copied in an oblivious fashion from the block to a temporary array, i.e., every transaction is conditionally moved using `cmov` to every possible destination. The data contained in the temporary array is then copied to the response in an oblivious fashion, again using `cmov` to conditionally copy everything to all possible locations in the response.

for the client was contained within the scanned blocks. Over a period of time, by tracking requests and response sizes, the adversary could gain significant information about the client's addresses and transactions.

We address such leakage by using a custom-made block scanning scheme. The main goal of the scheme is to fully hide the ratio between the response size (that indicates the number of transactions returned to the client) and the number of scanned blocks. When this ratio is constant, the adversary cannot deduce any meaningful information.

Figure 3 depicts the details of our scanning scheme. The newest block in the blockchain observed by the Bitcoin network is n. A client's request contains an addresses of interest and the number block x indicating how deep the chain should be scanned. The enclave starts scanning from n and moves towards x. It stores intermediate responses and when it reaches block x it performs a check. The total size of the response, r, is divided by the threshold size, t. The threshold indicates the maximum response size per block such that if we are to scan $n-x$ blocks, the maximum response size for the client can be $r = (n-x)*t$. If the given response size r is greater, then the enclave has to scan up to block y (or $y-x$ more blocks), such that $r = (n-y)*t$. If the response size is smaller, i.e., if after scanning $n-x$ blocks $r \leq (n-x)*t$, we pad the response size such that $r = (n-x)*t$. The exact size of the threshold is empirically determined in Section 6.

Side-channel protection. The scanning technique described above prevents external leakage through response sizes and disk accesses. However, if the adversary is able to mount high-granularity digital side-channel attacks (e.g., one that allows her to observe execution paths with instruction-level granularity), she will be able to determine the transactions that were accessed, and thus infer the client's addresses.

To make our system more robust against such attacks, we optionally add side-channel protections at the expense of performance (cf. Section 6). To protect against timing leakage we compute the Merkle path for all transactions in each of the scanned block in contrast to only computing the path for the transactions of client's interest. For protection against control-flow side channels we make use of the `cmov` assembly instruction to hide execution paths. `cmov` is a conditional move such that "*If the condition specified in the opcode (cc) is met, then the source operand is written to the destination operand. If the source operand is a memory operand, then regardless of the condition, the memory operand is read*" [7]. We use the `cmov` instruction in form of a wrapper (originally presented in [47]) that allows us to remove branches from our code resulting in the same control flow with no leakage.

The same technique is also used in previous side-channel protection solutions like Raccoon [47]. However, since using such a general purpose side-channel defense system directly would incur an extremely high performance overhead in our particular setting (due to large amounts of accessed data), we customize these techniques to our setting. Specifically, we apply the following modifications, as per Figure 4:

(i) Instead of continuing to scan the chain if the size of the response exceeds the threshold, we stop scanning after the specified number of blocks. If not all transactions fit in the response, the client does not receive all transactions and is informed of this through a flag in the response. This allows the allocation of a response array that does not change size during processing. The client can request the remaining transactions in a new query (potentially from a different node).

(ii) For each block, we allocate a temporary array of size t_m (see Figure 4), where t_m is a threshold that specifies the maximum data per block, as opposed to the threshold t that specifies the average data per block. While the block is parsed, each transaction is moved to the temporary array in an oblivious fashion, i.e., we use the `cmov` instruction to conditionally move each word of each transaction to every entry in the array. This means that for every transaction we access every entry in the array and since the same instruction is used for each possible copy – independent of whether the data is actually copied – even an attacker with an instruction level view of the control flow cannot determine which data is actually copied. After processing the block, the temporary array is traversed and all entries are copied to the response array (see

Figure 4). This is again done in an oblivious fashion, i.e., each entry is copied conditionally using the `cmov` method to every possible position in the response array.

This method of copying transactions from the block to the response is required to efficiently keep the data accesses oblivious. Specifically, for a block of size m, a temporary array of size t_m and n requested blocks, this method requires $\mathcal{O}(m \cdot t_m + t_m \cdot n \cdot t)$ instead of $\mathcal{O}(m \cdot n \cdot t)$ operations when naively copying the data obliviously from the block to the response. Since t_m is usually much smaller than m and $n \cdot t$, this method is in practice orders of magnitude faster.

4.2 Oblivious Database Variant

In our second variant, we focus on reducing the load of lightweight clients in terms of computation and network while offering even better privacy preservation (namely, the block number that specifies how deep the chain should be searched does not leak). The main idea behind this variant is to allow lightweight clients to send requests containing addresses of their interest and directly receive information regarding unspent outputs, without the need to verify block headers and Merkle tree paths.

In order to achieve such verification, a new indexed database of unspent transactions (denoted as *enclave UTXO*) is created and searched for every client request using an Oblivious RAM algorithm. Figure 5 shows the operation of this variant, and we describe the details as follows:

Initialization and continuous operation.

(**a**) Similar to a standard full node, on initialization the full node FN_j connects to the peer-to-peer network and downloads and verifies the entire blockchain. After initialization, when new blocks are available in the peer-to-peer network, FN_j downloads and verifies them.

(**b**) During initialization Enclave E_j reads the locally stored blockchain and verifies each block. The enclave builds its own *enclave UTXO* database that is a special version of the original structure present in standard full nodes. In particular, this UTXO set is encrypted on the disk as sealed storage, indexed for easy and fast access depending on the client request, and accessed using ORAM to prevent information leakage through disk accesses. After initialization, the enclave updates this UTXO using ORAM when new blocks are available in the locally stored blockchain.

(**c**) As in the Scanning Window variant, the client obtains the latest block headers from the peer-to-peer network.

Client request handling.

(**1**) The Lightweight Client LC_i performs an attestation with the secure Enclave E_j residing on the full node FN_j.

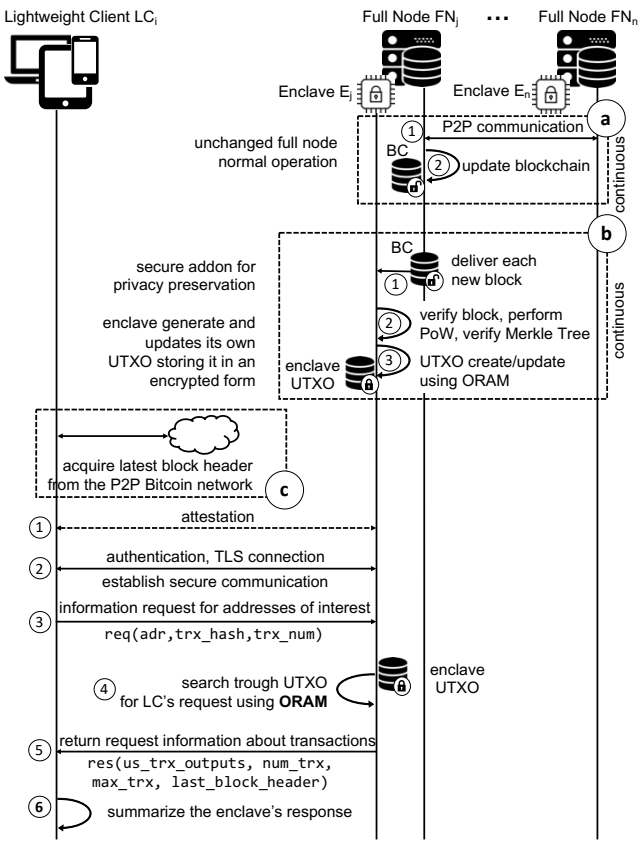


Figure 5: **Oblivious Database operation.** Lightweight client sends a request containing its address and the last transaction to an enclave on full node. Enclave queries a specially-constructed UTXO database using ORAM and provides a response back to the client.

(**2**) LC_i establishes a secure communication channel to the Enclave E_j using TLS.

(**3**) LC_i sends a request containing the addresses of interest, along with the hash and number of the latest transaction known to the client. The last two parameters are needed in case the number of unspent outputs contained by an address is larger than the maximum size of the message. For example, LC_i receives the first response containing x transaction outputs with an indication that there is more, and in a consequent request specifies the same address as in the first request along with the x–th transaction hash and transaction number. This gives an indication to the enclave to respond with the second batch of outputs starting from that transaction. The process repeats (possibly with a different node) until the client is satisfied. To prevent information leakage through the message sizes, requests are always of constant size, i.e., the client pads shorter requests and splits up larger queries. The size is defined to accommodate the majority of requests. Since a lightweight client can choose any available node to connect to, she can choose to send requests to

different nodes to hide the number of sent requests.

(**4**) The Enclave E_j reads the enclave UTXO database to get the unspent transaction output information in respect to the client's request. E_j uses ORAM and the previously created index to access the enclave UTXO in an oblivious fashion.

(**5**) In preparation of the response, E_j includes the relevant information as explained in step **(3)**, which encompasses the currently included and maximum number of unspent transactions found for a specific address. When these numbers match, the LC_i knows that she has received all the unspent outputs of a specific address. The enclave additionally includes the block hash of the last known block from the local blockchain (longest chain). With this information the client can deduce whether the enclave has been served with the latest block and that the enclave's database is fully updated. Responses are always of constant size, i.e., shorter responses are padded and if a response is too large, the client is informed of missing outputs, such that she can later retrieve the rest of the outputs (e.g., from a different node). The size of the response is chosen such that it accommodates the majority of responses.

(**6**) The Lightweight Client LC_i can summarize the unspent transaction outputs received from the Enclave E_j. The enclave guarantees completeness in terms of transaction confirmation and the current state of the chain, so the client does not have to perform any additional checks by herself. Successful update of the client's internal state results in the connection termination between the enclave and the client.

Oblivious Database details. In this variant, we use an ORAM algorithm called Path ORAM [52] to protect data access patterns of our enclaves. For readers unfamiliar with this algorithm, a brief description is in Appendix B.

Database Initialization. The ORAM database is initialized by creating dummy buckets on disk and filling the *position map* with randomized entries. The *stash* is also filled with dummy chunks. After that the ORAM database is fully initialized and can be used to add new unspent outputs from the blockchain. To ensure that the enclave always uses the latest version of the sealed UTXO database, SGX counters or rollback-protection systems such as ROTE [39] can be used.

Database Update. When a new Bitcoin block is added, the enclave first verifies the proof of work. It then extracts all transaction inputs and outputs and bundles them by address. For each address found in the block, the UTXO database entry is requested and then updated with the new information. If too many entries are added, resulting in the chunk getting too big, the chunk is split into two and the index is updated to reflect the changes made to the UTXO database. All accesses are performed using the ORAM algorithm and, therefore, do not leak any information about the access patterns.

Database Access. Accesses to the ORAM database follow the normal procedure described in [52] and in Appendix B.

Side-channel protection. While the usage of ORAM protects against all external leakage, side-channel attacks, and thus, internal leakage remains a challenge. If we consider the most powerful attacker that can perform all digital side-channel attacks (see Section 3.3), this variant would be forfeit due to the leakage of the code access patterns, specifically, execution paths in the *if* statements when the stash, indexes and the position map is being accessed. This would leak the exact address which is used to search for the unspent transactions in the internal database.

To remedy internal leakage, we deploy several mechanisms that protect our code and execution. First, when accessing the security critical data structures, specifically, the position map, stash, and the indexes containing information about which chunks contain unspent transactions of a certain Bitcoin address we pass over them entirely in the memory to hide the memory access pattern. Second, to hide the execution paths we remove all branching in the code that accesses these data structures and deploy the `cmov` assembly instruction (see Section 4.1). Observation of the control flow and memory access does not leak whether the operation performed by the enclave was a read or a write, and since there is a single control flow without creating multiple branches depending on the condition, we effectively hide the execution and thus protect this variant from internal leakage in full.

5 Security Analysis

In this section, we provide an informal security analysis. First, we analyze our solution with respect to our adversary model where SGX security enforcements cannot be broken. In particular, we show that our solution ensures confidentiality of the requested client addresses, as the attacker cannot infer the requested address from disk access patterns, response sizes, side-channels, or a combination thereof. Second, we discuss implication of potential SGX compromise and show that our solution can handle such cases gracefully.

5.1 External Leakage Protection

Scanning window. This variant scans complete blocks from the blockchain database, instead of accessing individual transactions within them, and thus prevents direct information leakage from disk access patterns. The constant ratio of response size to scanned blocks prevents information leakage from the response size. The adversary may only infer the number of blocks that are accessed and not which addresses are sent by the client or how many transactions are returned.

Oblivious Database. To protect against information leakage attacks on the disk access, our second variant utilizes the well-studied Path ORAM [52] algorithm. Our setting is slightly different than the typical client-server model considered in ORAM. In our case, the enclave corresponds to

the client. Because the adversary can run the enclave freely, she can use it as an oracle, i.e., she can influence the data that is written (by delivering blocks to the enclave) and can query for values himself. Regardless of that, due to the unlinkability property of ORAM, the attacker learns nothing about what is accessed and the probability to guess correctly which ORAM block was accessed is equal to that of a random guess, as shown in [52]. Also, the adversary learns nothing from responses as they are of constant size.

5.2 Side-channel Protection

Most known side-channel attacks on SGX provide imperfect data-access or control-flow traces and require many repetitions to filter out noise [20, 43, 27, 50]. In BITE, queries from legitimate clients cannot be replayed due to the authenticated TLS channel and since the enclave is either stateless across power cycles or protected against rollback. The adversary can create his own client and send requests to the enclave, but this will not result in any advantage against legitimate clients. For these reasons, mounting side-channel attacks against BITE is more challenging than performing side-channel attacks against enclaves in general. To analyze our solution against future adversaries that may be able to mount more precise attacks, below we consider the worst case scenario, i.e., side-channel attacks that obtain perfect data access and control flow traces from enclave's execution.

Scanning Window. To harden our Scanning Window variant against side-channels, we provide optional protections that incur significant performance penalty. When the enclave scans through both the temporary array and the final response array in their entirety, it performs `cmov` operations for all possible transactions. This allows replacing branches in our code with a few instructions resulting in the same control flow with no leakage to the attacker since all data is accessed and the same operation is executed every time.

Oblivious Database. For our Oblivious Database variant we always include side-channel protections to our solution, since the performance overhead is negligible. When accessing the security critical data structures such as stash, indexes and the position map, we pass over them entirely to hide the memory access pattern. Second, to hide the execution paths, we remove all branching in the code that accesses these data structures and replace them with `cmov` assembly instructions (see Section 4.2). Observation of the control flow and memory access does not leak whether the operation performed by the enclave was a read or a write, and since there is a single control flow without creating multiple branches depending on the condition, we effectively hide the execution path and thus protect this variant from internal leakage in full.

The usage of `cmov` for protecting against digital side-channel and internal leakage was previously studied in Raccoon [47] and with respect to protecting ORAM-based systems it was studied in other SGX-related works [49, 14]. These works show the effectiveness of `cmov` in protecting against internal leakage. Our solution uses the same techniques, and thus directly inherits the security guarantees that successfully protect against the same type of attacks, i.e., those based on digital side-channel leakage.

5.3 Completeness

In the Scanning Window variant, the client herself performs the verification of the block headers, Merkle paths and transactions. Since the client can retrieve block headers from the P2P network and the enclave returns all transactions from its view of the chain, the client can ensure completeness of the response by checking that she received data from the longest chain. In the Oblivious Database variant, the enclave performs all verifications for the client. To ensure completeness, the client can compare the latest block hash from received response to information from other sources.

An adversary that controls the OS of the full node server can deliver incomplete blocks to BITE enclave or decide to not deliver specific new blocks to the enclave. However, this would be noticed by the light clients. Remember that light clients are required to obtain the latest block hash from an alternative source in order to verify the completeness of BITE responses. (Another approach to solve this would be to use systems such as TownCrier [59] or TLS-N [48], that enable the enclave to get an authenticated feed that could confirm the correctness of the blocks received from the full node.)

5.4 Implications of a Full SGX break

Our adversary model assumes that side-channel leakage from enclave's execution may happen, but the adversary cannot fully break SGX, i.e., the adversary cannot read all enclave's secrets and modify its control flow arbitrarily. However, SGX was never intended to provide tamper resistance against physical attacks and recent research has demonstrated that platform vulnerabilities like Spectre [35] and Meltdown [38] can be adapted to extract attestation keys from SGX processors [21, 54]. Therefore, it becomes relevant to ask how BITE handles a full SGX compromise.

In the Scanning Window variant, the client only loses the privacy protections provided by our system and all of his funds remain secure. Since the client still performs SPV, the security is otherwise not affected and our system provides the same guarantees as current light clients, i.e., a node may omit transactions, but cannot steal funds or make a client falsely accept a payment.

In the Oblivious Database variant, a compromised enclave could make the client accept false payments by sending invalid UTXOs. However, we argue that this will not be a realistic threat since it would require the client to sell some

System	Our implementation		Libraries	Total
	Bitcoin[1]	Network[2]	*mbed-tls*	
Scanning Window	1'876	1'613	53'831	57'320
Oblivious Database	4'117	1'613	53'831	59'561

[1] Processing the Bitcoin blockchain.
[2] Parsing responses from the client over TLS.

Table 1: Trusted Computing Base in LOC.

goods or service to the provider of the node, i.e. this is not a realistic issue for most users. Merchants that see a full break of SGX as a realistic threat can instead use the Scanning Window variant. Additionally, such an attack would be easily detectable after the fact and result in loss of reputation of the provider of our service and would thus likely only be profitable for high value transactions for which most merchants would probably run a full node.

We conclude that BITE can provide as much security and privacy as traditional lightweight clients even given a full break of SGX. This is in contrast to the naive solution of storing the clients' private keys in the enclave and using it as a remote wallet. Lastly, we emphasize that our approach and BITE as a solution are not limited to SGX. Our main ideas could most likely be applied to other TEEs as well, such as the open-source Keystone TEE [1], thus reducing the reliance on SGX (and thereby Intel) even further.

6 Performance Evaluation

In this section, we describe our implementation and provide performance evaluation results.

6.1 Implementation Details

The centerpiece of our system is an original blockchain parser. For TLS connections we use the *mbed-tls* library from ARM [37]. Table 1 shows the trusted computing base.

Scanning Window. The implementation of Scanning Window is very small since it only involves scanning the blockchain and does not have to keep state. The network code including the *mbed-tls* library contributes the most to the TCB with over 96%. The same work for matching and non-matching transactions is performed in order to keep the scanning time per block constant for all requests.

The response size per block allows for around 5 transactions. We believe this is a reasonable choice that satisfies common usage patterns for light clients. For n included and N total transactions in the block, an upper bound for the Merkle path size is $n*\log(N)$ and each entry is 32 bytes long. This results in an approximate upper bound of 2.2kB for $N = 4000$, the current limit in Bitcoin. As of today (November 2018) the average transaction size is around 500 bytes, therefore, a response size per block of 5kB is enough to fit around 5 transactions ($5*500\text{B}+2200\text{B} < 5\text{kB}$). If more or larger transactions are found, following from Section 4, the enclave scans more blocks of the blockchain until the response can fit all requested transactions.

Oblivious Database. The implementation of Oblivious Database is more complex than Scanning Window and the enclave has to keep state and store a large UTXO set on disk. At the time of writing, the UTXO size (indexed by Bitcoin address) is around 3GB while our ORAM overhead accounts for 2 times the original size, totaling around 6GB.

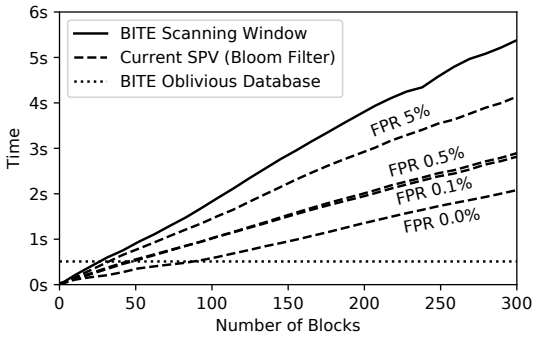
We use Path ORAM to store the UTXO set and have evaluated various chunk sizes for the implementation. The chosen chunk size accounts for 32kB, meaning a single chunk can fill up to 32kB with outputs from one address. If an address has more unspent outputs, the outputs are stored in multiple chunks. Assuming an average output size of 100B, one ORAM read can return up to 320 outputs for one address. The outputs are grouped by the receiving address and then ordered alphabetically. This is necessary in order to keep the size of the index small enough to fit in the enclave's memory. In the worst case the maximum index size involves the lower and upper limits for addresses (20B) and transaction hashes (32B) for every ORAM block resulting in a maximum of $(8\text{GB}/32\text{kB}) \cdot (32\text{B}*2+20\text{B}*2) \approx 19.5\text{MB}$.

To set the response size, we analyzed the typical unspent outputs per active address in the Bitcoin network. Our results show that 95% of all addresses have 5 or fewer unspent outputs and 98% have fewer than 12 outputs. Based on this data, we settled on 12 average outputs per request, resulting in around 1.2kB.

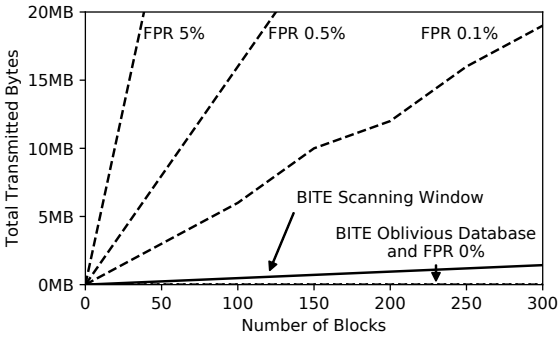
6.2 Performance Results and Comparison

In this section, we evaluate both variants of BITE and compare them to the current SPV performance using *python-bitcoinlib* [53]. The focus is put on three different metrics: processing time, communication overhead, and storage requirements. Processing time encompasses both the request handling from the client to the enclave as well as the time needed for the enclave to update the UTXO for new blocks. Communication overhead is evaluated through the response size, thus directly affecting the client's necessary bandwidth. Lastly, we report the necessary storage requirements on the full nodes that these system need for operation. A summary of all reported results can be found later on in Table 3.

Note that in all our data points, the TLS handshake times are omitted. Matetic et al. [40] report around 100ms for a new handshake and <10ms for TLS session resumption using *mbed-tls* in SGX. We do not evaluate the performance of a client since the client-side storage and network overhead are insignificant. We tested our implementation on an Intel i7-8700k with a Samsung 960 SSD for local storage.



(a) **Processing cost (client request)** for Scanning Window, Oblivious Database and current SPV protocols using Bloom filters.



(b) **Communication cost** for Scanning Window, Oblivious Database and current SPV protocols using bloom filters.

Figure 6: Performance evaluation of Scanning Window and Oblivious Database.

		t_m		
		5kB	10kB	20kB
Blocks	100	0.7s (± 0.2s)	1.3s (± 0.5s)	2.7s (± 0.9s)
	200	0.7s (± 0.2s)	1.4s (± 0.5s)	2.8s (± 0.9s)
	300	0.7s (± 0.2s)	1.5s (± 0.5s)	3.0s (± 0.9s)

Table 2: Processing time per block with oblivious execution for Scanning Window depending on the number of requested blocks and the temporary size, averaged over 100 blocks.

Processing. Figure 6a shows the processing cost to filter blocks for BITE and current SPV protocols. Note that the measurements in Figure 6a do not account for the network speed. For client update requests over the last 100 blocks, the current SPV mode takes 0,62s, 1.06s, 1.06s, 1.5s, with the false positive rates of Bloom filters set to 0.0% 0.1%, 0.5% and 5%, respectively. Note that the numbers regarding standard SPV with the Bloom filter false positive rate of 0.0% actually indicates a solution with no privacy, e.g. the light client sends only his addresses in the request without any masquerading.

For the Scanning Window variant without side-channel protections we report 1.9s, corresponding to an 81% overhead compared to the SPV with FPR 0.1% and 0.5%. If the side-channel protection is added to Scanning Window, the oblivious execution and memory access adds a significant overhead. Table 2 shows the time per block for various requests and t_m size. Higher t_m allows to cope with high variance of relevant activity within the requested blocks. Note that the blocks vary in size, and thus the time per block fluctuates a lot leading to a high standard deviation. Synchronizing 100 blocks with $t_m = 5$kB takes around 73 seconds corresponding to an overhead of approximately 40x. Note that the oblivious Scanning Window variant is not shown in Figure 6a due to its size.

In our Oblivious Database variant, the unspent outputs are directly fetched from the enclave UTXO and the individual blocks are not scanned. Thus, the performance does not depend on the client's last known block, but only on the ORAM database access times. A request that fetches the information regarding 10 client addresses accounts only for 0.5s and is completely independent on the number of requested blocks, thus making it even faster than the standard SPV mode used without any privacy protections.

Contrary to the Scanning Window, in the Oblivious Database variant, the enclave needs to update its UTXO set after each new block arrives in the ORAM database which takes 78.5s. To reach permanent availability we propose to use 2 systems in parallel which update with an offset between each other. If a user requests the result from a node that is not fully up to date, the remaining blocks can be scanned by utilizing oblivious Scanning Window. The number of clients that can be served by a single SGX enclave can be estimated by using around 120s (pessimistic estimate) for updating the state and then the remaining 8 out of 10 minutes (Bitcoin block interval) to continuously answer client requests, leading to an approximate 10000 clients per enclave.

Communication. Figure 6b shows the bandwidth comparison between all discussed protocols. Our variants use significantly smaller response sizes compared to SPV since they do not need to hide relevant information with false positives. A device with a decent 4G connection that operates at 100Mbit/s additionally requires around 1.4s to retrieve 100 blocks (17MB) with the current SPV protocol and a 0.5% false positive rate while Scanning Window only takes 0.04s (500kB). The Oblivious Database variant reduces the communication overhead even more and accounts only for 0.0001s (only 12kB), which is insignificantly small since only unspent outputs are included and not the entire transaction information along with the Merkle paths. The SPV with no privacy protections performs slight less effective than the Oblivious Database of BITE as it was the case when the processing performance was compared.

	Processing		Communication	Storage		
	Request	UTXO Update	Response	Blockchain	UTXO	Leakage Protection
Scanning Window[1]	1.9s	-	500kB	200GB	0	✓/✗[4]
Oblivious SW[1]	73s	-	500kB	200GB	0	✓
Oblivious Database[3]	0.5s	78.5s	12kB	50MB[5]	6GB	✓
Stan. SPV FPR 0.5%[1]	1.1s	≈2s	17MB	200GB	2.8GB	✗
Stan. SPV FPR 0.0%[1,2]	0.6s	≈2s	14kB	200GB	2.8GB	✗

[1] For 100 blocks. [2] SPV with no privacy protection. [3] For 10 addresses.
[4] Protects against external leakage but not side-channels.
[5] Only the block headers need to be stored.

Table 3: Performance comparison and requirements on the full node for supporting light clients.

Storage. The SPV mode has to store both the whole blockchain (200GB) and the UTXO set (2.8GB), while our Scanning Window variant only needs to store the blockchain. Moreover, our Oblivious Database variant does not need the whole blockchain (except during initialization) but only the block headers (50MB in total) and the special enclave UTXO stored in the ORAM database. This database accounts to 6GB, a 100% overhead compared to the regular UTXO set, due to the ORAM algorithm requirements. It is clear that both our variants require less storage, and our Oblivious Database variant's requirements are insignificant compared to all mentioned solutions.

Comparison of BITE variants. Table 3 shows a performance comparison between all our variants and the standard SPV mode from the full node's perspective. The performance of Scanning Window is heavily dependent if side-channels are a concern. The original Scanning Window offers a slightly worse performance than the standard SPV but offers increased privacy, protecting against external leakage, and requires significantly less bandwidth. Adding protection from side-channels greatly increases the processing time, while the communication load stays the same. Oblivious Database, on the other hand, offers the same full privacy guarantees as the oblivious Scanning Window, and has the smallest footprint in both the processing time and the network overhead. The enclave UTXO does require regular updating affecting the uptime. However, the previously mentioned solution of having two parallel enclaves with operation offset effectively removes this limitation. In conclusion, we have shown that our variants offer comparable or better performance with increased end client's privacy.

Comparison to side-channel protection systems. Finally, we compare the performance and security of BITE to previous SGX side-channel protection systems. For our comparison we use Raccoon [47] that addresses internal leakage due to secret-dependent memory accesses and

	Leakage			Performance Overhead
	External	Internal	Response Size	
Raccoon[47]	✗	✓	✗	~ 100x[1]
Obliviate[14]	✓	✗	✗	> 4x[1]
Raccoon[47] + Obliviate[14]	✓	✓	✗	$100x - 400$x[2]
BITE Scanning Window [3]	✓	✓	✓	40x
BITE Oblivious Database	✓	✓	✓	1x

[1] Based on the performance evaluation of [47] and [14].
[2] Combination of the two primitives can yield an overhead in this range.
[3] Fully oblivious Scanning Window variant.

Table 4: Performance overhead and security comparison between existing primitives and BITE.

Obliviate [14] that addresses external leakage due to file accesses. We note that ZeroTrace [49] also provides similar external leakage protection as Obliviate, but since the ZeroTrace paper does not report performance overhead numbers suitable for comparison, we exclude it from our discussion.

Table 4 summarizes our comparison. By applying Raccoon to the target enclave code, the performance overhead of the enclave's execution can range up to 1000x depending on the complexity of the original code that is made oblivious. In our case, the complexity of the original code matches the examples that report the overhead of around 100x. Applying techniques from Obliviate can cause a performance overhead of >4x. Neither Raccoon nor Obliviate alone provide full leakage protection, as Raccoon prevents only from internal leakage and Obliviate protects only against external leakage. Neither of these two systems protects against leakage from response sizes. The combination of Raccoon and Obliviate would protect both internal and external leakage, but still not leakage from response sizes. We estimate that the combined overhead of these two protection tools would amount to 100x-400x.

Our fully oblivious Scanning Window variant has a performance overhead of 40x while the Oblivious Database variant has practically no overhead. More importantly, both of the BITE variants protect against external leakage, internal leakage, and leakage from the response sizes, and therefore achieve more complete protection than any of the previous solutions.

7 Discussion

Usage and long-term privacy. Lightweight clients can use BITE in different ways and the chosen usage model can have implications on the clients' long-term privacy. For example, in what we consider *non-recommended usage*, the client (i) performs payment verification requests only when the payment appears in the ledger, (ii) always uses the same full node for verification, and (iii) only uses a single or few Bitcoin address. If all of the above conditions are met, although the adversary controlling the full node does not learn the client's address from a single verification request, he might be able to *correlate* the timing of the verification re-

quest events and the Bitcoin addresses visible in the ledger at roughly the same time, and thus construct a set of candidate addresses that may belong to the served client. We acknowledge that our solution cannot eliminate this type of correlation completely. However, we stress that such correlation would require long-term tracking of verification requests from the adversary and that the same limitation applies to any light client payment verification scheme.

In *recommended* usage of BITE, the client (i) uses different full nodes for payment verification, (ii) regularly uses fresh Bitcoin addresses (e.g., using an HD wallet [57]), and (iii) introduces unpredictability to the timing pattern of payment verification requests like a small number of extra requests at random time points. Following such a usage model, the above mentioned correlation becomes very difficult.[2]

Large responses. Some client requests might result in a larger response than our defined threshold for message size. As our performance analysis shows, the number of these requests is almost negligible. However, our mechanism still allows these types of request with the distinctive factor that the client would have to request them in batches. For example, if a client in the Scanning Window variant requests transactions for 10 of his addresses from the last 300 blocks using the full-side-channel protection, there might be more transaction data then the $300 * t$ kB message size. In this case, the enclave sets a flag indicating there is more information to be delivered. After receiving the response, the client can repeat the request with the defined flag and receive the rest of the information. The protocol operates in the same way, thus no distinction between these two requests can be observed by the attacker. However, the attacker can see the repeated request and infer that the specific client has more transactions of interest in the designated blocks. To mitigate this problem one could wait a period of time before requesting the rest of the response, obfuscate the IP address or change to a completely different service provider (another enclave) for finishing the request.

Denial of service. A malicious user might attempt DoS by asking for a very long scan window, incurring large processing times for full nodes and making the service momentarily unavailable for other clients. DoS (and spam) are common in systems where there is no significant cost involved (e.g., sending 1M emails is practically free). In our setting, one could easily remedy such denial of service attacks by applying fees based on the nature of the request. Large balance updates for lightweight clients would incur higher costs than just frequent updates, thus limiting the attacker from performing "free" DoS attacks. On the other hand, a malicious node can easily block all enclave messages or interrupt enclave execution, thereby preventing the enclave to access the blockchain, update its UTXO or serve client requests. This however falls in a domain which is impossible to fully prevent. If this would occur, the light client can just send its request to another enclave hosted by another entity.

Unbounded enclave memory. The performance of our system is mostly bounded by the slower disk operations. However, if future versions of Intel SGX would allow more enclave memory (i.e., currently the limit is 128MB without the expensive page swapping) ranging up to the RAM limit on the residing platform, one could keep the UTXO database and all other security critical data in the memory and not on the disk, similar to recently proposed SGX-based in-memory database systems like EnclaveDB [46].

8 Related Work

Lightweight client privacy. The idea of light clients for Bitcoin was already included in the Bitcoin paper by Satoshi Nakamoto [44] in the form of *Simple Payment Verification* (SPV). Hearn and Corallo later introduced Bloom filters [18] in BIP 37 [31] that allow a client to probabilistically request a subset of all transactions in a block to mask which addresses are owned by the client. Gervais et al. later showed that the information leaked by the use of Bloom filters in Bitcoin can in many cases enable the identification of client addresses [25]. Hearn later expanded on these issues and discussed the difficulties of solving them [30].

Osuntokun et al. recently proposed modifications to Bitcoin nodes and lightweight clients that move the application of the filter to the client [45]. Full nodes create a filter (with a low false positive rate) for the set of all transactions in a block. A lightweight client then fetches the filter from one or more full nodes and can then check whether the block contains transactions that she is interested in. If that is the case, the client will request the full block from any node.

This approach suffers from a number of shortcomings. First, the gained privacy largely depends on the client behavior and how well the client is connected to distinct entities. If the client does not request the filter headers from multiple entities and then requests the blocks from a different one, she can be easily tricked into revealing her addresses by using forged filters: A node prepares a filter matching half of all addresses and sends it to the client. If the client requests the block, at least one of her addresses lies within that set, otherwise all of her addresses lie in the other half. The node can then further reduce the possible set using binary search by sending modified filters for the following blocks, allowing bitwise recovery of client addresses. Second, depending on how often a transaction is of interest to the client, she might

[2] To quantify how accurately the adversary can correlate the client's addresses with these best practices, would be an interesting direction for future work. As building an accurate model would require collecting significant amount data about the behavioral patterns of light clients, we consider this task a research project on its own and outside our scope.

end up downloading the full blockchain after all. Since the client always either requests the full block or nothing at all, she will download almost every block if a large fraction of blocks contain at least one transaction that is of interest.

Other research on Bitcoin privacy shows that using different heuristics, large parts of the Bitcoin transaction graph can be deanonymized [16, 41]. These techniques are orthogonal to the problem of light client privacy and out of our scope.

Lastly, there exist alternative solutions that tackle limited computation abilities of light clients, such as VerSum [55]. The main idea is that the complex computation is outsourced to a set of remote servers. Even though these solutions do not focus on privacy preservation directly, they do offer alternative ways to construct support systems for light clients that do not require the creation of UTXO type databases for proving correctness.

SGX Leakage Protection. During the last few years, the research community has studied information leakage from SGX enclaves extensively and proposed a number of defenses. In this section we explain why none of the existing systems solves our problem directly and which prior systems use similar protective primitives as our solution.

Raccoon [47] addresses both internal and external information leakage for both code and data accesses. For control-flow obfuscation, Raccoon uses taint analysis to determine execution paths that should be hidden and transforms enclave code such that it executes extraneous decoy paths to hide the enclave's actual control flow. The basic building block for such control-flow obfuscation is the `cmov` instruction that we use as well. Raccoon also uses Path ORAM to hide external secret-dependent data accesses and "streaming" over data structures (i.e., accessing every element) in the internal enclave memory. The main difference between Raccoon and our solution is that by tailoring our implementation, we avoid the need for taint analysis and extra decoy paths enabling a more efficient solution.

Other related systems include Cloak [28] that prevents cache leakage using hardware-based transactional memory features in processors; ZeroTrace [49] and Obliviate [14] that provide a library for data structures protected using ORAM; DR.SGX [19] that randomizes and periodically re-randomizes all data locations in enclave's memory with cache-line granularity; and, T-SGX [51] and Deja Vu [22] that detect and prevent side-channel attacks based on repeated interrupts. The main limitation of Cloak is that it requires hardware features that are not available on all SGX CPUs and it only prevents cache-based leakage. ZeroTrace and Obliviate are limited to data access protection and does not prevent leakage from secret-dependent control flow. DR.SGX is also limited to data accesses and imposes a high performance overhead when configured to prevent all leakage. T-SGX and Deja Vu are limited to attacks that perform repeated interrupts (subset of known attacks).

Oblix [42] presents a new ORAM algorithm tailored to SGX. We use Path ORAM, but our solution is agnostic to the used ORAM algorithm and we could easily replace it.

9 Conclusion

Improved user privacy is one of the main goals of decentralized currencies like Bitcoin. However, payment verification requires downloading and processing the entire chain which is impossible for most mobile clients. Therefore, all popular blockchains support simplified verification modes where lightweight clients can verify transactions with the help of full nodes. Unfortunately, such payment verification does not preserve user privacy and thus defeats one of the main benefits of using systems like Bitcoin. In this paper, we have proposed a new approach to improve the privacy of lightweight clients using trusted execution. We have shown that our solution provides strong privacy protection and additionally improves performance of current lightweight clients. We argue that BITE is the first practical solution to ensure privacy for light clients, such as mobile devices, in Bitcoin.

Acknowledgments

The research work leading to these results has been supported by Zurich Information Security and Privacy Center (ZISC). We would also like to thank our shepherd Rob Jansen for his insightful comments.

References

[1] Keystone: Open-source Secure Hardware Enclave.

[2] BitcoinJ, 2018. `https://bitcoinj.github.io/`.

[3] Electrum, 2018. `https://electrum.org/#home`.

[4] Ethereum, 2018. `https://www.ethereum.org/`.

[5] Etherscan.io, 2018. `https://etherscan.io`.

[6] Light Ethereum Subprotocol (LES), 2018. `https://github.com/zsfelfoldi/go-ethereum/wiki/Light-Ethereum-Subprotocol-%28LES%29`.

[7] OpCodes: CMOV, 2018. `http://www.rcollins.org/p6/opcodes/CMOV.html`.

[8] PicoCoin, 2018. `https://github.com/jgarzik/picocoin`.

[9] R3, 2018. `https://www.r3.com/`.

[10] Ripple, 2018. `https://ripple.com/`.

[11] Bitnodes, 2019. `https://bitnodes.earn.com/`.

[12] Blockchain.info, 2019. `https://blockchain.info`.

[13] Statoshi.info, 2019. `https://statoshi.info`.

[14] AHMAD, A., KIM, K., SARFARAZ, M. I., AND LEE, B. OBLIVIATE: A Data Oblivious File System for Intel SGX. In *NDSS* (2018).

[15] ANDROULAKI, E., BARGER, A., BORTNIKOV, V., CACHIN, C., CHRISTIDIS, K., DE CARO, A., ENYEART, D., FERRIS, C., LAVENTMAN, G., MANEVICH, Y., ET AL. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the 13th EuroSys Conference* (2018), ACM.

[16] ANDROULAKI, E., KARAME, G. O., ROESCHLIN, M., SCHERER, T., AND CAPKUN, S. Evaluating User Privacy in Bitcoin. In *International Conference on Financial Cryptography and Data Security* (2013), Springer.

[17] ARNAUTOV, S., TRACH, B., GREGOR, F., KNAUTH, T., MARTIN, A., PRIEBE, C., LIND, J., MUTHUKUMARAN, D., O'KEEFFE, D., STILLWELL, M., ET AL. SCONE: Secure Linux Containers with Intel SGX. In *11th USENIX Symposium on Operating Systems Design and Implementation (USENIX OSDI)* (2016).

[18] BLOOM, B. H. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM 13*, 7 (1970), 422–426.

[19] BRASSER, F., CAPKUN, S., DMITRIENKO, A., FRASSETTO, T., KOSTIAINEN, K., MÜLLER, U., AND SADEGHI, A. DR.SGX: Hardening SGX Enclaves against Cache Attacks with Data Location Randomization, 2017.

[20] BRASSER, F., MULLER, U., DMITRIENKO, A., KOSTIAINEN, K., CAPKUN, S., AND SADEGHI, A.-R. Software Grand Exposure: SGX Cache Attacks Are Practical. In *11th USENIX Workshop on Offensive Technologies (WOOT)* (2017).

[21] CHEN, G., CHEN, S., XIAO, Y., ZHANG, Y., LIN, Z., AND LAI, T. H. SgxPectre Attacks: Leaking Enclave Secrets via Speculative Execution. *Computing Research Repository (CoRR), arXiv abs/1802.09085* (2018).

[22] CHEN, S., ZHANG, X., REITER, M. K., AND ZHANG, Y. Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjá Vu. In *Proceedings of the 12th ACM ASIA Conference on Computer and Communications Security (ASIACCS)* (2017).

[23] COSTAN, V., AND DEVADAS, S. Intel SGX explained. In *Cryptology ePrint Archive, Report 2016/086* (2016).

[24] FOR ALTERNATIVE FINANCE, C. C. Global Cryptocurrency Benchmarking Study, 20187. `https://goo.gl/7B99Ev`.

[25] GERVAIS, A., CAPKUN, S., KARAME, G., AND GRUBER, D. On the Privacy Provisions of Bloom Filters in Lightweight Bitcoin Clients. In *Proceedings of the 30th Annual Computer Security Applications Conference* (2014), ACM.

[26] GOLDREICH, O., AND OSTROVSKY, R. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM (JACM) 43*, 3 (1996), 431–473.

[27] GÖTZFRIED, J., ECKERT, M., SCHINZEL, S., AND MÜLLER, T. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security* (2017), ACM.

[28] GRUSS, D., LETTNER, J., SCHUSTER, F., OHRIMENKO, O., HALLER, I., AND COSTA, M. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *USENIX Security* (2017).

[29] HALDERMAN, J. A., SCHOEN, S. D., HENINGER, N., CLARKSON, W., PAUL, W., CALANDRINO, J. A., FELDMAN, A. J., APPELBAUM, J., AND FELTEN, E. W. Lest We Remember: Cold-boot Attacks on Encryption Keys. *Communications of the ACM 52*, 5 (2009), 91–98.

[30] HEARN, M. Bloom Filter Privacy and Thoughts on a Newer Protocol, 2015. `https://groups.google.com/forum/#!msg/bitcoinj/Ys13qkTwcNg/9qxnhwnkeoIJ`.

[31] HEARN, M., AND CORALLO, M. Connection Bloom Filtering. *Bitcoin Improvement Proposal 37* (2012). `https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki`.

[32] INTEL. Intel SGX, Ref. No.: 332680-002, 2015. `https://software.intel.com/sites/default/files/332680-002.pdf`.

[33] INTEL. Intel Software Guard Extensions - Developer Zone - Details, 2017. `https://software.intel.com/en-us/sgx/details`.

[34] KAUER, B. OSLO: Improving the Security of Trusted Computing. In *USENIX Security* (2007).

[35] KOCHER, P., HORN, J., FOGH, A., , GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (SP)* (2019).

[36] LEE, S., SHIH, M.-W., GERA, P., KIM, T., KIM, H., AND PEINADO, M. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. In *USENIX Security* (2017).

[37] LIMITED, A. mbedTLS (formerly known as PolarSSL), 2015. `https://tls.mbed.org/`.

[38] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., FOGH, A., HORN, J., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security* (2018).

[39] MATETIC, S., AHMED, M., KOSTIAINEN, K., DHAR, A., SOMMER, D., GERVAIS, A., JUELS, A., AND CAPKUN, S. ROTE: Rollback Protection for Trusted Execution. In *USENIX Security* (2017).

[40] MATETIC, S., SCHNEIDER, M., MILLER, A., JUELS, A., AND CAPKUN, S. DELEGATEE: Brokered Delegation Using Trusted Execution Environments. In *USENIX Security* (2018).

[41] MEIKLEJOHN, S., POMAROLE, M., JORDAN, G., LEVCHENKO, K., MCCOY, D., VOELKER, G. M., AND SAVAGE, S. A Fistful of Bitcoins: Characterizing Payments among Men with No Names. In *Proceedings of the 2013 conference on Internet Measurement Conference* (2013), ACM.

[42] MISHRA, P., PODDAR, R., CHEN, J., CHIESA, A., AND POPA, R. A. Oblix: An Efficient Oblivious Search Index. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (SP)* (2018).

[43] MOGHIMI, A., IRAZOQUI, G., AND EISENBARTH, T. Cachezoom: How SGX Amplifies the Power of Cache Attacks. In *International Conference on Cryptographic Hardware and Embedded Systems* (2017), Springer.

[44] NAKAMOTO, S. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008.

[45] OSUNTOKUN, O., AKSELROD, A., AND POSEN, J. Client Side Block Filtering. *Bitcoin Improvement Proposal 157* (2017). `https://github.com/bitcoin/bips/blob/master/bip-0157.mediawiki`.

[46] PRIEBE, C., VASWANI, K., AND COSTA, M. EnclaveDB: A Secure Database using SGX. IEEE.

[47] RANE, A., LIN, C., AND TIWARI, M. Raccoon: Closing Digital Side-channels Through Obfuscated Execution. In *USENIX Security* (2015).

[48] RITZDORF, H., WÜST, K., GERVAIS, A., FELLEY, G., ET AL. Tls-n: Non-repudiation over tls enabling ubiquitous content signing. In *Network and Distributed System Security Symposium (NDSS)* (2018).

[49] SASY, S., GORBUNOV, S., AND FLETCHER, C. ZeroTrace: Oblivious memory primitives from Intel SGX. In *NDSS* (2017).

[50] SCHWARZ, M., WEISER, S., GRUSS, D., MAURICE, C., AND MANGARD, S. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2017), Springer.

[51] SHIH, M.-W., LEE, S., KIM, T., AND PEINADO, M. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *NDSS* (2017).

[52] STEFANOV, E., VAN DIJK, M., SHI, E., FLETCHER, C., REN, L., YU, X., AND DEVADAS, S. Path ORAM: an extremely simple oblivious RAM protocol. In *Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2013).

[53] TODD, P. python-bitcoinlib, 2018. `https://github.com/petertodd/python-bitcoinlib`.

[54] VAN BULCK, J., MINKIN, M., WEISSE, O., GENKIN, D., KASIKCI, B., PIESSENS, F., SILBERSTEIN, M., WENISCH, T. F., YAROM, Y., AND STRACKX, R. FORESHADOW: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security* (2018).

[55] VAN DEN HOOFF, J., KAASHOEK, M. F., AND ZELDOVICH, N. Versum: Verifiable computations over large public logs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 1304–1316.

[56] WOJTCZUK, R., AND RUTKOWSKA, J. Attacking SMM Memory via Intel CPU Cache Poisoning. *Invisible Things Lab* (2009).

[57] WUILLE, P. Hierarchical Deterministic Wallets. *Bitcoin Improvement Proposal 32* (2012). `https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki`.

[58] XU, Y., CUI, W., AND PEINADO, M. Controlled-channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (SP)* (2015).

[59] ZHANG, F., CECCHETTI, E., CROMAN, K., JUELS, A., AND SHI, E. Town Crier: An Authenticated Data Feed for Smart Contracts. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2016).

A Intel SGX

Intel's SGX [23, 32] entails a security enhancement for new Intel CPUs in form of a TEE for security-critical applications in commodity PC platforms. The SGX architecture enables protected applications, called *enclaves* that are *isolated* from software running outside of the enclave. This isolation protects the integrity and confidentiality of the enclave's execution from any malicious software running on the same system, including BIOS, OS and hypervisor, or even malicious peripherals such as compromised network cards [56, 34, 29]. Enclave memory is handled in plaintext only inside the processor and is encrypted by the processor whenever it leaves the CPU (e.g., to DRAM) to ensure that neither the OS nor malicious hardware can access it.

Even though the OS is untrusted, it is responsible for starting and managing enclaves. To protect the integrity of the execution, the CPU securely records all initialization actions to create a *measurement* that records the code and initial state of the enclave. This can be later used by a third party to verify that the correct code is running on the system supported by SGX. This process is called *remote attestation*. A system service called Quoting Enclave signs the attestation statement – which contains the mentioned measurements – for remote verification. Using an online attestation service run by Intel, the verifier can check that signature. An enclave can attach data to the attestation statement, such as a public key, that it sends to the verifier. This can be used to establish a secure communication channel to an enclave.

In addition, SGX enables enclaves to store data for persistent storage in an encrypted form through a process called *sealing*. The processor provides a sealing key that can only be accessed by the same enclave running on the same platform, i.e. only the enclave that sealed data can later unseal it. This provides confidentiality and integrity for the stored data, but it does not protect from so called rollback attacks [39] when the enclave is restarted. Finally, enclaves cannot execute system calls and do not have access to secure peripherals. For this reason, software using SGX has to be split into two parts, a protected enclave and an unprotected component that runs in normal user space and handles communication with the OS, i.e. operations concerning networking and file accesses. For further details, we refer the reader to [23, 32].

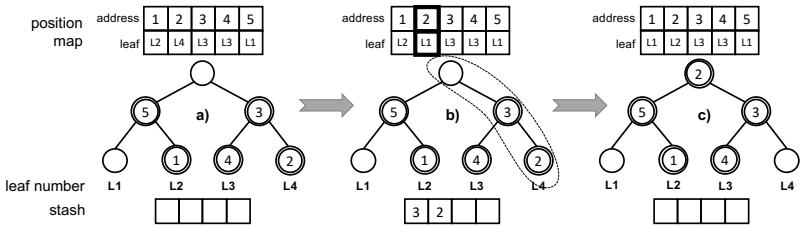


Figure 7: a) The client wants to access the chunk 2 that is stored in Path ORAM. b) The position map specifies that the chunk 2 is on the path to leaf 4. Therefore, the server reads all entries on the path into the stash and re-randomizes the position map entry of the requested chunk. c) The server writes back as many chunks as possible on the previously read path.

B Oblivious RAM

Oblivious RAM (ORAM) [26], is a well-known technique that hides access patterns to an encrypted storage medium. A typical ORAM model is one where a trusted client wants to store sensitive information on an untrusted server. Encrypting each data record before storing it on the server provides confidentiality, but access patterns to stored encrypted records can leak information, such as correlation of multiple accesses to the same record. The intuition behind the security definition of ORAM is to prevent the adversary from learning anything about the access pattern. In ORAM, the adversary does not learn any information about which data is being accessed and when, whether the same data is being repeatedly accessed (i.e., unlinkability), the pattern of the access itself, and lastly the purpose, type of the access (i.e., write or read). However, one should note that ORAM techniques cannot hide access timing.

In this work, we use a popular and simple algorithm called Path ORAM [52] that provides a good trade-off between client side storage and bandwidth. The storage is organized as a binary tree with buckets containing Z chunks each. The position of each chunk is stored in a *position map* that maps a database entry to a leaf in the tree, and for every access the leaf of the accessed entry is re-randomized. A small amount of entries is stored in a local (i.e., memory) structure – *stash*.

Every access involves reading all buckets of a path from the root to a leaf into the *stash* and then writing back new or old re-randomized data from the *stash* to the same path resulting in an overhead of $O(\log N)$ read/write operations. If the requested chunk is already in the stash, an entire path still gets read and written. The summary of ORAM operations is:

(**1**) get leaf from *position map* and generate new random leaf for the database entry. Insert it into the *position map*, read all buckets along the path to the leaf and put them into the *stash*

(**2**) if access is a write, replace the specified chunk in the stash with the new chunk

(**3**) write back some chunks from the *stash* to the path. Chunks can only be put into the path if their leaf from the *position map* allows it. Chunks are pushed down as far as possible into the tree to minimize *stash* capacity.

(**4**) return requested chunk

FASTKITTEN: Practical Smart Contracts on Bitcoin

Poulami Das* Lisa Eckey* Tommaso Frassetto§ David Gens§
Kristina Hostáková* Patrick Jauernig§ Sebastian Faust* Ahmad-Reza Sadeghi§
Technische Universität Darmstadt, Germany
** first.last@cs.tu-darmstadt.de*
§ first.last@trust.tu-darmstadt.de

Abstract

Smart contracts are envisioned to be one of the killer applications of decentralized cryptocurrencies. They enable self-enforcing payments between users depending on complex program logic. Unfortunately, Bitcoin – the largest and by far most widely used cryptocurrency – does not offer support for complex smart contracts. Moreover, simple contracts that can be executed on Bitcoin are often cumbersome to design and very costly to execute. In this work we present FASTKITTEN, a practical framework for executing arbitrarily complex smart contracts at low costs over decentralized cryptocurrencies which are designed to only support simple transactions. To this end, FASTKITTEN leverages the power of trusted computing environments (TEEs), in which contracts are run off-chain to enable efficient contract execution at low cost. We formally prove that FASTKITTEN satisfies strong security properties when all but one party are malicious. Finally, we report on a prototype implementation which supports arbitrary contracts through a scripting engine, and evaluate performance through benchmarking a provably fair online poker game. Our implementation illustrates that FASTKITTEN is practical for complex multi-round applications with a very small latency. Combining these features, FASTKITTEN is the *first* truly practical framework for complex smart contract execution over Bitcoin.

1 Introduction

Starting with their invention in 2008, decentralized cryptocurrencies such as Bitcoin [51] currently receive broad attention both from academia and industry. Since the rise of Bitcoin, countless new cryptocurrencies have been launched to address some of the shortcomings of Nakamoto's original proposal. Examples include Zerocash [47] which improves on Bitcoin's limited anonymity, and Ethereum [16] which offers complex smart contract support. Despite these developments, Bitcoin still remains by far the most popular and intensively studied cryptocurrency, with its current market capitalization of $109 billion which accounts for more than 50% of the total cryptocurrency market size [2].

A particular important shortcoming of Bitcoin is its limited support for so-called smart contracts. Smart contracts are (partially) self-enforcing protocols that allow emitting transactions based on complex program logic. Smart contracts enable countless novel applications in, e.g., the financial industry or for the Internet of Things, and are often quoted as a glimpse into our future [9]. The most prominent cryptocurrency that currently allows to run complex smart contracts is Ethereum [16], which has been designed to support Turing complete smart contracts. While Ethereum is continuously gaining popularity, integrating contracts directly into a cryptocurrency has several downsides as frequently mentioned by the advocates of Bitcoin. First, designing large-scale secure distributed systems is highly complex, and increasing complexity even further by adding support for complex smart contracts also increases the potential for introducing bugs. Second, in Ethereum, smart contracts are directly integrated into the consensus mechanics of the cryptocurrency, which requires in particular that all nodes of the decentralized system execute all contracts. This makes execution of contracts very costly and limits the number and complexity of applications that can eventually be run over such a system. Finally, many applications for smart contracts require confidentiality, which is currently not supported by Ethereum.

There has been significant research effort in addressing these challenges individually. Some works aim to extend the functionality of Bitcoin by showing how to build contracts over Bitcoin by using multiparty computation (MPC) [37, 38, 40], others focus on achieving privacy-preserving contracts (e.g., Hawk [35], Ekiden [19]) by combining existing cryptocurrencies with trusted execution environments (TEEs). However, as we elaborate in Section 2, all of these solutions suffer from various deficiencies: they cannot be integrated into existing cryptocurrencies such as Bitcoin, are highly inefficient (e.g., they use heavy cryptographic techniques such as non-interactive zero-knowledge proofs or general MPC), do not support money mechanics, or have significant financial costs due to complex transactions and high collateral (money blocked by the parties in MPC-based solutions).

In this work, we propose FASTKITTEN, a novel system that leverages trusted execution environments (TEEs) utilizing well-established cryptocurrencies, such as Bitcoin, to offer full support for arbitrary complex smart contracts. We emphasize that FASTKITTEN does not only address the challenges discussed above, but is also highly efficient. It can be easily integrated into existing cryptocurrencies and hence is ready to use today. FASTKITTEN achieves these goals by using a TEE to isolate the contract execution inside an enclave, shielding it from potentially malicious users. The main challenges of this solution, such as for instance how to load and validate blockchain data inside the enclave or how to prevent denial of service attacks, are discussed in Section 3.1. Moving the contract execution into the secure enclave guarantees correct and private evaluation of the smart contract even if it is not running on the blockchain and verified by the decentralized network. This approach circumvents the efficiency shortcoming of cryptocurrencies like Ethereum, where contracts have to be executed in parallel by thousands of users. Most related to our work is the recently introduced Ekiden system [19], which uses a TEE to support execution of multiparty computations but does support contracts that handle coins. While Ekiden is efficient for single round contracts, it is not designed for complex reactive multi-round contracts, and their off-chain execution. The latter is one of the main goals of FASTKITTEN.

We summarize our main goals and contributions below.

- **Smart Contracts for Bitcoin:** We support arbitrary multi-round smart contracts executed amongst any finite number of participants, where our system can be run on top of any cryptocurrency with only limited scripting functionality. We emphasize that Bitcoin is only one example over which our system can be deployed today; even cryptocurrencies that are simpler than Bitcoin can be used for FASTKITTEN.
- **Efficient Off-Chain Execution:** Our protocol is designed to keep the vast majority of program execution off-chain in the standard case if all parties follow the protocol. Since our system incentivizes honest behavior for most practical use cases, FASTKITTEN can thus run in real-time at low costs.
- **Formal Security Analysis:** We formally analyze the security of FASTKITTEN in a strong adversarial model. We prove that either the contract is executed correctly, or all honest parties get their money back that they have initially invested into the contract, while a malicious party loses its coins. Additionally, the service provider who runs the TEE is provably guaranteed to not lose money if he behaves honestly.
- **Implementation and benchmarking:** We provide an in-depth analysis of FASTKITTEN's performance and costs and evaluate our framework implementation with respect to several system parameters by offering benchmarks on real-world use cases. Concretely, we show that online poker can run with an overall match latency of 45ms and costs per player are in order of magnitude of one USD, which demonstrates FASTKITTEN's practicality.

We emphasize that FASTKITTEN requires only a single TEE which can be owned either by one of the participants or by an external service provider which we call the *operator*. In addition, smart contracts running in the FASTKITTEN execution framework support private state and secure inputs, and thus, offer even more powerful contracts than Ethereum. Finally, we stress that FASTKITTEN can support contracts that may span over multiple different cryptocurrencies where each participant may use her favorite currency for the money handled by the contract.

2 Related Work

Support for execution of arbitrary complex smart contracts over decentralized cryptocurrencies was first proposed and implemented by the Ethereum cryptocurrency. As pointed out in Section 1, running smart contracts over decentralized cryptocurrencies results in significant overheads due to the replicated execution of the contract. While there are currently huge research efforts aiming at reducing these overheads (for instance, via second layer solutions such as state channels [24, 49], Arbitrum [34] or Plasma [55], outsourcing of computation [58], or permissioned blockchains [46]), these solutions work only over cryptocurrencies with support complex smart contracts, e.g. over Ethereum. Another line of work, which includes Hawk [36] and the "Ring of Gyges" [33], is addressing the shortcoming that Ethereum smart contracts cannot keep private state. However, also these solutions are based on complex smart contracts and hence cannot be integrated into popular legacy cryptocurrencies such as Bitcoin, which is the main goal of FASTKITTEN.

In this section we will focus on related work, which considers smart contract execution on Bitcoin. We separately discuss multiparty computation based smart contracts and solutions using a TEE. We provide a more detailed discussion on how the above-mentioned Ethereum based solutions compare to FASTKITTEN in Appendix A. Additionally, in Section 8 we discuss some exemplary contract use cases and compare their execution inside FASTKITTEN with the execution over Ethereum.

Multiparty computation for smart contracts An interesting direction to realize complex contracts over Bitcoin is to use so-called multiparty computation with penalties [38–40]. Similar to FASTKITTEN these works allow secure m-round contract execution but they rely on the claim-or-refund functionality [39]. Such a functionality can be instantiated over Bitcoin and hence these works illustrate feasibility of generic contracts over Bitcoin. Unfortunately, solutions supporting generic contracts require complex (and expensive) Bitcoin transactions and high collateral locked by the parties which makes them impractical for most use-cases. Concretely, in

Approach	Minimal # TX	Collateral	Generic Contracts	Privacy
Ethereum contracts	$\mathcal{O}(m)$	$\mathcal{O}(n)$	✓	×
MPC [38–40]	$\mathcal{O}(1)$	$\mathcal{O}(n^2 m)$	✓	✓
Ekiden [19]	$\mathcal{O}(m)$	no support for money		✓
FastKitten	$\mathcal{O}(1)$	$\mathcal{O}(n)$	✓	✓

Table 1: Selected solutions for contract execution over Bitcoin and their comparison to Ethereum smart contracts. Above, n denotes the number of parties and m is the number of reactive execution rounds.

all generic n-party contract solutions we are aware of, each party needs to lock $\mathcal{O}(nm)$ coins, which overall results in $\mathcal{O}(n^2 m)$ of locked collateral. In contrast, the total collateral in FastKitten is $\mathcal{O}(n)$, see column "Collateral" in Table 1. It has been shown that for specific applications, concretely, a multi-party lottery, significant improvements in the required collateral are possible when using MPC-based solutions [48]. This however comes at the cost of an inefficient setup phase, communication complexity of order $\mathcal{O}(2^n)$, and $\mathcal{O}(\log n)$ on-chain transactions for the execution phase. Let us stress that the approach used in [48] cannot be applied to generic contracts.

Overall, while MPC-based contracts are an interesting direction for further research, we emphasize that these systems are currently far from providing a truly practical general-purpose platform for contract execution over Bitcoin—which is the main goal of FastKitten.

TEEs for blockchains There has recently been a large body of work on using TEEs to improve certain features of blockchains [10, 43, 59, 63, 64]. A prominent example is Teechain [43], which enables off-chain payment channel systems over Bitcoin. Most of these prior works do not use the TEE for smart contract execution. Some notable exceptions include Hawk [36] and the "Ring of Gyges" [33], who propose privacy preserving off-chain contracts execution, but, as already mentioned, do not work over Bitcoin.

Probably most related to our work is Ekiden [19], which proposes a system for private off-chain smart contract execution using TEEs. While Ekiden focuses on solutions over Ethereum, it does not require a powerful scripting language of the underlying blockchain technology – just like FastKitten. Despite the conceptual similarities of Ekiden and FastKitten, the goals of these systems are orthogonal. Ekiden aims at moving heavy smart contract execution off the chain in order to reduce the cost of executing complex contract functions. In contrast, FastKitten focuses on efficient off-chain execution of multi-round contracts between a set of parties. Importantly, we require our system to natively handle coins of the underlying blockchain. A joint goal of both systems is to provide state privacy of the contracts.

Ekiden considers clients (contract parties) and computing nodes which have a similar task as FastKitten's TEE operator since they also execute contracts inside a TEE. In contrast to FastKitten, Ekiden sends the encryption of the resulting contract state to the blockchain after every function call. If a client requests another function call, a selected computing node takes the state from the blockchain, decrypts it inside its enclave and performs the contract execution. This implies that reactive multi-round contracts are very costly even in the standard case when all participating parties are honest (c.f. column "Minimal # TX" in Table 1).

Ekiden relies on multiple TEEs and guarantees service availability as long as at least one TEE is controlled by an honest computing node. We note in Section 9.2 that fault tolerance can be integrated into FastKitten in a straightforward way. Additionally, Ekiden aims to achieve forward secrecy even if a small fraction of TEEs gets corrupted via, e.g., a side-channel attack. Their strategy is to secret-share a long-term secret key between the TEEs and use it to generate a short-term secret key every "epoch". Hence, an attacker learning the short-term key can only decrypt state from the current epoch. While side-channel attacks are out of scope of this work, note that FastKitten can achieve forward secrecy of states in case of side-channel attacks using the same mechanism as Ekiden. An important part of the FastKitten construction is the fair distribution of coins through the enclave. Ekiden does neither model nor discuss the handling of coins. It is not straightforward to add this feature to their model since the contract state is encrypted and hence the money cannot be unlocked automatically on-chain.

3 Design

FastKitten allows a set of n users $P_1, \ldots, P_n$ to execute an arbitrary complex smart contract over a decentralized cryptocurrency that only supports very simple scripts. Concretely, FastKitten considers cryptocurrencies that, in addition to supporting simple transactions between users, offer so-called *time-locked transactions*. A transaction is time-locked if it is only processed and integrated into the blockchain after a certain amount of time has passed. Moreover, FastKitten requires that transactions contain space for storing arbitrary raw data. We emphasize that these are very mild requirements on the underlying cryptocurrency that, for instance, are satisfied by the most prominent cryptocurrency Bitcoin.[1] FastKitten leverages these properties together with the power of trusted execution environments to provide an efficient general-purpose smart contract execution platform.

As discussed in the introduction, a contract is a program that handles coins according to some—possibly complex—program logic. In this work, we consider n-party contracts, which are run among a group of parties $P_1, \ldots, P_n$ and have the following structure. During the initialization phase, the contract receives coins from the parties and some initial in-

[1] Bitcoin transactions can store up to 97 KB of data [44]; multiple transactions can be used for bigger payloads.

puts. Next, it runs for m reactive rounds, where in each round the contract can receive additional inputs from the parties P_i, and produces an output. Finally, after the m-th round is completed the contract pays out the coins to the parties according to its final state and terminates.

A key feature of FASTKITTEN is very low execution cost and high performance compared to contract execution over cryptocurrencies such as Ethereum. This is achieved by not executing contracts *by all parties maintaining the cryptocurrency* but instead running the contract within a TEE which could, e.g., be owned and operated by a single service provider which we call the *operator Q*. In the standard case when all parties are honest, FASTKITTEN runs the entire contract off-chain within the enclave and only needs to touch the blockchain during contract initialization and finalization. More concretely, during initialization, the parties transfer their coins to the enclave by time-locking coins with *deposit transactions*, while at the end of finalization the enclave produces transactions that transfer coins back to the users according to the results of the contract execution. These transactions are called *output transactions* and can be published by the users of the system to receive their coins.

3.1 Design Challenges of FASTKITTEN

Leveraging TEEs for building a general-purpose contract execution platform requires us to resolve the following main challenges.

Protection against malicious operator. The operator runs the TEE and hence controls its interaction with the environment (e.g., with other parties or the blockchain). Thus, the operator can abort the execution of the TEE, delay and change inputs, or drop any ingoing or outgoing message. To protect honest users from such an operator, the enclave program running inside the TEE must identify such malicious behavior and punish the operator. In particular, we require that even if the TEE execution is aborted, all parties must be able to get their coins refunded eventually. To achieve this, we let the operator create a so-called *penalty transaction*: the penalty transaction time-locks coins of the operator, which in case of misbehavior can be used to refund the users and punish the operator.

Note that designing such a scheme for punishment is highly non-trivial. Consider a situation where party P_i was supposed to send a message x to the contract. From the point of view of the enclave that runs the contract, it is not clear whether the operator was behaving maliciously and did not forward a message to the enclave, or, e.g., party P_i did not send the required message to the operator. To resolve this conflict, we leverage a challenge-response mechanism carried out via the blockchain. We emphasize that this challenge-response mechanism is only required when parties are malicious, and typically will not be executed often due to the high financial costs for an adversary.

Verification of blockchain evidence. To ensure that a malicious operator cannot make up false blockchain evidence, we need to design a secure blockchain validation algorithm which can efficiently be executed inside a TEE. We achieve this by simplifying the verification process typically carried out by full blockchain nodes by using a *checkpoint block* to serve as the initial starting point for verification. This drastically reduces blockchain verification time in comparison to verification starting from the genesis block. To further speed up the transaction verification, we only validate correctness of block headers. Finally, when the TEE needs to verify whether a certain transaction was integrated into a block, we set a minimum number of blocks that must confirm a transaction as part of the security parameter within our protocol. This guarantees that faking a valid-looking chain is computationally infeasible for a malicious operator. Finally, it is computationally infeasible for a malicious operator to load a fake (but valid-looking) chain into the enclave before the penalty transaction is published on the blockchain.

Minimizing blockchain interaction. Since blockchain interactions are expensive, FASTKITTEN only requires interaction with the blockchain in the initialization and finalization phases if all parties follow the protocol. As already discussed above, however, in case of malicious behavior FASTKITTEN may require additional interaction with the blockchain for conflict resolution. This is required to allow the TEE to attribute malicious behavior either to the operator or to some other participant P_i that provides input to the contract. We achieve this through a novel challenge-response protocol, where the TEE will ask the operator to challenge P_i via the blockchain. The operator can then either deliver a proof that he challenged P_i via the blockchain but did not receive a response, in which case P_i will get punished; or the operator receives P_i's input and can continue with the protocol.

Of course, this challenge-response protocol adds to the worst-case execution time of our system, and additionally will result in fees for blockchain interaction. To address the latter, our protocol ensures that both parties involved in the challenge-response mechanism have to split the fees resulting from blockchain interaction equally.[2] This incentivizes honest behavior if parties aim to maximize their personal profits.

Preventing denial of service attacks. Complex smart contracts may take a very long time to complete, and in the worst case not terminate. Hence, a malicious party may carry out a denial-of-service attack against the contract execution platform, where the platform is asked to execute a contract that never halts. It is well known that determining whether a program terminates is undecidable. Hence, general-purpose contract platforms, such as Ethereum, mitigate this risk by letting users pay via fees for every step of the contract execution. This effectively limits the amount of computation that

[2] In the cryptocurrency community, this is often referred to as griefing factor 1 : 1, meaning that for every coin spent by the honest users on fees the adversary is required to also spend one coin.

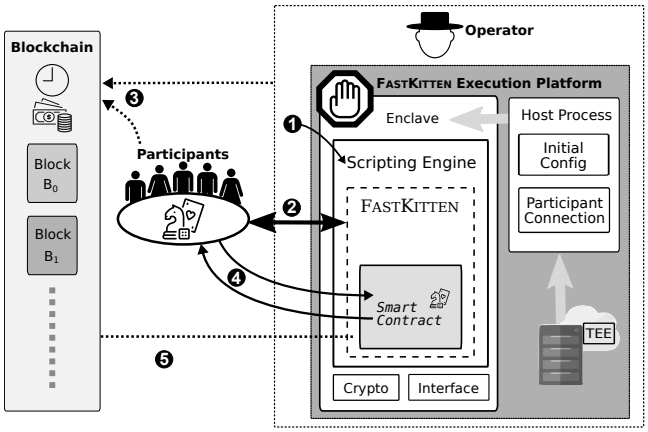


Figure 1: Architecture of the **FastKitten** Smart Contract Execution Platform. Dashed arrows indicate interaction with the blockchain and non-dashed arrows depict communication between parties.

can be carried out by the contract. Since FastKitten allows multiple parties to provide input to the contract in the same round, it might be impossible to decide which party (parties) caused the denial of service and should pay the fee. To this end, FastKitten protects against such denial-of-service attacks using a time-out mechanism. As all users of the system (including the operator) have to agree on the contract to be executed, we assume that this agreement includes a limit on the maximum amount of execution steps that can be performed inside the enclave per one execution round. See Section 6.5 for more details.

3.2 Architecture and Protocol

To enable secure off-chain contract execution, our architecture builds on existing TEEs, which are widely available through commercial off-the-shelf hardware. In particular, our architecture can be implemented using Intel's Software Guard Extensions (SGX) [4, 29, 45] which is a prominent TEE instantiation built into most recent Intel processors. SGX incorporates a set of new instructions to create, control and communicate with enclaves. While enclaves are part of a legacy host process, SGX enforces strict isolation of computation and memory between enclave and host process on the hardware level. Another prominent instantiation of the TEE concept is ARM TrustZone [6], which provides similar functionality for mobile devices. We note that only the operator Q is required to own TEE-enabled hardware.

As depicted in Figure 1, our FastKitten Execution Facility is run by the operator Q and consists of a host process and an enclave. The untrusted host process takes care of setting up the enclave with an *initial config*, handles the *participant connections*, and blockchain communication over the network. While this means that Q has complete control over these parts, the influence of a malicious operator on a running enclave is limited: he can interrupt enclave execution, but not tamper with it. Further, the enclave will sign and hash all code and data as part of its *attestation* towards parties, so they can verify correctness of the setup before placing deposits. To support arbitrary contract functionality, FastKitten includes a *scripting engine* inside the enclave and several helper libraries, such as the *Crypto library* to generate and verify transactions, and an *Interface library* to pass data between host process and enclave. The individual contracts are loaded into the FastKitten enclave during the initialization of our protocol by the underlying host process and participants can verify that contracts are loaded correctly. Our protocol then proceeds in three phases, which we call *setup phase*, *round computation*, and *finalization phase*. Figure 1 depicts the architecture of the FastKitten framework.

During the setup phase (Steps ❶–❸) the contract is loaded into the enclave. Using the TEE's attestation functionality, all parties $P_1, \ldots, P_n$ can verify that this step was completed correctly. Then the operator and all parties block their coins for the contract execution. If any party aborts in this phase, the money is refunded to all parties that deposited money and the protocol stops. Otherwise, all parties receive a time-locked penalty transaction, needed in case Q aborts the protocol.

Afterwards, the round computation phase (Step ❹) starts, in which Q sends the previous round's output to all parties. If a party P_i receives such an output, which is correctly signed by the enclave, it signs and sends the input for the following round to Q. If all parties behave honestly, Q will forward the received round inputs to the enclave, which computes the outputs for the next round. In case that the enclave does not receive an input from party P_i the enclave needs to determine whether P_i failed to send its input or if Q behaved maliciously (e.g., by dropping the message). Therefore, the enclave will punish Q unless it can prove, that it sent the last round output to P_i but did not receive a response. This proof is generated via the blockchain: Q publicly challenges P_i to respond with the input for the next round by posting the output of the previous round to the blockchain. As soon as this challenge transaction is confirmed, P_i needs to respond publicly by spending the coins of the challenge transaction and include its input for the next round. If P_i responds, Q can extract P_i's input and continue with the protocol execution. If P_i did not respond, Q forwards the respective blocks as a transcript to the enclave, to prove that P_i misbehaved.[3] So, while a malicious party (or the operator) can force this on-chain challenge-response procedure without direct punishment, posting these transactions will also act against its own financial interests by extending the time lock of its own coins and leading to transaction fees. Nevertheless, such malicious behavior cannot prevent the fair termination of our protocol.

The last phase of the protocol is the payout phase (Step ❺). In

[3] Alternatively, we could allow the operator to spend the challenge transaction after a timeout has passed. While this would result in easier verification for the TEE, the operator would need to publish an additional transaction, increasing both fees and the overall time for the challenge-response phase.

this phase the enclave returns the output transaction generated by the *Crypto library*. This transaction distributes the coins according to the terminated contract. In case of a protocol abort, the coins initially put by the users will be refunded to all honest parties. If any party was caught cheating, this party will not receive back its coins. This means the money will stay in control of the enclave and will never be spent.

4 Adversary Model

The FASTKITTEN protocol is executed n parties $P_1, \ldots, P_n$ and an operator Q (who owns the TEE) with the goal of executing a smart contract C. FASTKITTEN's design depends on a TEE to ensure its confidentiality and integrity. Our design is TEE-agnostic, even if our implementation is based on Intel SGX. Recent research showed that the security and privacy guarantees of SGX can be affected by memory-corruption vulnerabilities [11], architectural [13] and micro-architectural side-channel attacks [60]. For the operator, we assume that Q has full control over the machine and consequently can execute arbitrary code with supervisor privileges. While memory corruption vulnerabilities can exist in the enclave code, a malicious operator must exploit such vulnerabilities through the standard interface between the host process and the enclave. For the enclave code, we assume a common code-reuse defense such as control-flow integrity (CFI) [3, 15], or fine-grained code randomization [23, 42] to be in place and active. Architectural side-channel attacks, e.g., based on caches, can expose access patterns [13] from SGX enclaves (and therefore our FASTKITTEN prototype). However, this prompted the community to develop a number of software mitigations [12, 18, 27, 56, 57] and new hardware-based solutions [22, 28, 52]. Microarchitectural side-channel attacks like Foreshadow [60] can extract plaintext data and effectively undermine the attestation process FASTKITTEN relies on, leaking secrets and enabling the enclave to run a different application than agreed on by the parties; however, the vulnerability enabling Foreshadow was already patched by Intel [32]. Since existing defenses already target SGX vulnerabilities and since FASTKITTEN's design is TEE agnostic (i.e., it can also be implemented using ARM TrustZone or next-generation TEEs), we consider mitigating side-channel leakage as an orthogonal problem and out of scope for this paper.

For our protocol we consider a *byzantine adversary* [41], which means that corrupted parties can behave arbitrarily. In particular, this includes aborting the execution, dropping messages, and changing their inputs and outputs even if it means that they will lose money. FASTKITTEN is secure even if n parties are corrupt (including the two cases where only the operator is honest, and only one party is honest but the operator is corrupt). We show that no honest party will lose coins, a corrupt party will be penalized and that no adversary can tamper with the result of the contract execution. While we prove security in this very strong adversarial model, we additionally observe that incentive-driven parties (i.e., parties that aim at maximizing their financial profits) will behave honestly, which significantly boosts efficiency of our scheme. We stress that security of FASTKITTEN relies on the security of the underlying blockchain. We require that the underlying blockchain systems satisfies three security properties: *liveness*, *consistency* and *immutability* [26]. *Liveness* means that valid transactions are guaranteed to be included within the next δ blocks. *Consistency* guarantees that eventually all users have the same view on the current state of the blockchain (i.e., the transactions processed and their order). In addition, blockchains also are *immutable*, which means that once transactions end up in the blockchain they cannot be reverted. Most blockchain based cryptocurrencies guarantee consistency and immutability only after some time has passed, where time is measured by so-called *confirmations*. A block b_i is confirmed k-times if there exists a valid chain extending b_i with k further blocks. Once block b_i has been sufficiently often confirmed, we can assume that the transactions in b_i cannot be reverted and all honest parties agree on an order of the chain $(b_0, b_1, b_2, \ldots, b_i)$. For most practical purposes k can be a small constant, i.e., in Bitcoin it is generally believed that for $k = 6$ a block can be assumed final.[4]

5 The FASTKITTEN Protocol

In this section we give a more detailed description of our protocol, which includes the specification of the protocol run by Q and honest parties $P_1, \ldots, P_n$, all transactions and a description of the enclave program FASTKITTEN. The interaction between Q, P_i and the blockchain is depicted in Figure 2. We first describe the interactions with the blockchain and TEE.

5.1 Modeling the Blockchain

We will introduce some basic concepts of cryptocurrencies that are relevant for our work before we describe our high-level design. Cryptocurrencies are built using blockchains—a distributed data structure that is maintained by special parties called *miners*. The blockchain is comprised as a chain of blocks $(b_0, b_1, b_2, \ldots)$ that store the transactions of the system. The miners create new blocks by verifying new transactions and comprising them into new blocks that extend the tail of the chain. New blocks are created within some period of time t, where, for instance, in Bitcoin a new valid block is created every 10 minutes on average.

In cryptocurrencies users are identified by addresses, where an address is represented by a public key. To send coins from one address to another, most cryptocurrencies rely on transactions. If a user A with address pk_A wants to send x coins to user B with address pk_B, she creates a transaction tx which states that x coins from address pk_A are transferred to pk_B. Such a

[4] We notice that in blockchain-based cryptocurrencies there is no guaranteed finality, and even for very large values of k blocks can be reverted in principle. We emphasize however that even for small values of k reverting blocks becomes impossible in practice very quickly.

transaction tx is represented by the following tuple:

$$\mathsf{tx} := (\mathsf{tx.Input}, \mathsf{tx.Output}, \mathsf{tx.Time}, \mathsf{tx.Data}),$$

where tx.Input refers to a previously unspent transaction, tx.Output denotes the address to which tx.Value are going to be transferred to. Note that a transaction tx is unspent if it is not referred to by any other transaction in its Input field. Further, $\mathsf{tx.Time} \in \mathbb{N}$, which denotes the block counter after which this transaction will be included by miners, i.e., tx can be integrated into blocks $b_i, b_{i+1}, ...$, where $i = \mathsf{tx.Time}$. Finally, $\mathsf{tx.Data} \in \{0,1\}^*$ is a data field that can store arbitrary raw data. Similar to [5], we will often represent transactions by tables as shown exemplary in the table below, where the first row of the table gives the name of the transaction.

Transaction tx	
tx.Input:	Coins from unspent input transaction
tx.Output:	Coins to receiver address
tx.Time:	Some timelock (optional)
tx.Data:	Some data (optional)

Notice that a transaction tx only becomes valid if it is signed with the corresponding secret key of the output address from tx.Input. We emphasize that the properties described above are very mild and are for instance achieved by the most prominent cryptocurrency Bitcoin.

In order to model interaction with the cryptocurrency, we use a simplified blockchain functionality BC, which maintains a continuously growing chain of blocks. Internally it stores a block counter c which starts initially with 0 and is increased on average every t minutes. Every time the counter is increased, a new block will be created and all parties are notified. To address the uncertainty of the block creation duration we give the adversary control over the exact time when the counter is increased but it must not deviate more than $\Delta \in [t-1]$ seconds from t. Whenever any party publishes a valid transaction, it is guaranteed to be included in any of the next δ blocks.

Parties can interact with the blockchain functionality BC using the following commands.

- $\mathsf{BC.post}(\mathsf{tx})$: If the transaction tx is valid (i.e., all inputs refer to unspent transactions assigned to creator of tx and the sum of all output coins is not larger than the sum of all input coins) then tx is stored in any of the blocks $\{b_{c+1}, \ldots, b_{c+\delta}\}$.
- $\mathsf{BC.getAll}(i)$: If $i < c$, this function returns the latest block count $c-1$ and a list of blocks that extend b_i: $\mathbf{b} = (b_{i+1}, \ldots, b_c)$
- $\mathsf{BC.getLast}()$: The function getLast can be called by any party of the protocol and returns the last (finished) block and its counter: (c, b_c).

For every cryptocurrency there must exist a validation algorithm for validating consistency of the blocks and transactions therein, which we model using the function Extends. It takes as input, a chain of blocks $\mathbf{b}$ and a checkpoint block b_{cp} and outputs 1 if $\mathbf{b} = (b_{\mathsf{cp}+1}, \ldots, b_{\mathsf{cp}+i})$ is a valid chain of blocks extending b_{cp} and otherwise it outputs 0. In Section 6 we give more details on the validation algorithm, and how this function is implemented for the Bitcoin system. Recall, that we assume an adversary which cannot compute a chain of blocks of length k by itself (c.f. Section 4). This guarantees that he cannot produce a false chain such that this function outputs 1. To make the position of some transaction tx inside a chain of blocks explicit, we write $\ell := \mathsf{Pos}(\mathbf{b}, \mathsf{tx})$ when the transaction is part of the ℓ-th block of $\mathbf{b}$. If the transaction is in none of the blocks, the function returns ∞. For more details on the transaction and block verification we refer the reader to [7, 26, 51].

5.2 Modeling the TEE

In order to model the functionality of a TEE, we follow the work of Pass et. al. [54]. We explain here only briefly the simplified version of the TEE functionality whose formal definition can be found in [54, Fig. 1]. On initialization, the TEE generates a pair of signing keys (mpk, msk) which we call master public key and master secret key of the TEE. The TEE functionality has two enclave operations: install and resume. The operation TEE.install takes as input a program p which is then stored under an enclave identifier *eid*. The program stored inside an enclave can be executed via the second enclave operation TEE.resume which takes as input an enclave identifier *eid*, a function f and the function input *in*. The output of TEE.resume is the output *out* of the program execution and a quote ϱ over the tuple (eid, p, out). In the protocol description we abstract from the details how the users verify the quote that is generated through the enclave attestation. Since we only consider one instance E of the specific program p, we will simplify the resume command $[out, \varrho] := \mathsf{TEE.resume}(eid, \mathsf{f}, in)$ and write[5]:

$$[out, \varrho] := E.\mathsf{f}(in)$$

For every attestable TEE there must exist a function $\mathsf{vrfyQuote}(mpk, p, out, \varrho)$ which on input of a correct quote ϱ outputs 1, if and only if *out* was outputted by an enclave with master public key *mpk* and which indeed loaded p. Again, we assume that the adversary cannot forge a quote such that the function vrfyQuote() outputs 1. For more information on how this verification of the attestation is done in practice we refer the reader to [54].

5.3 Detailed Protocol Description

As explained in Section 3, our protocol $\pi_{\text{FastKitten}}$ proceeds in three phases. During the *setup phase* the contract is installed in the enclave, attested, and all parties deposit their

[5] Since we only need the quote of the first activation of E, we will omit this parameter from there on.

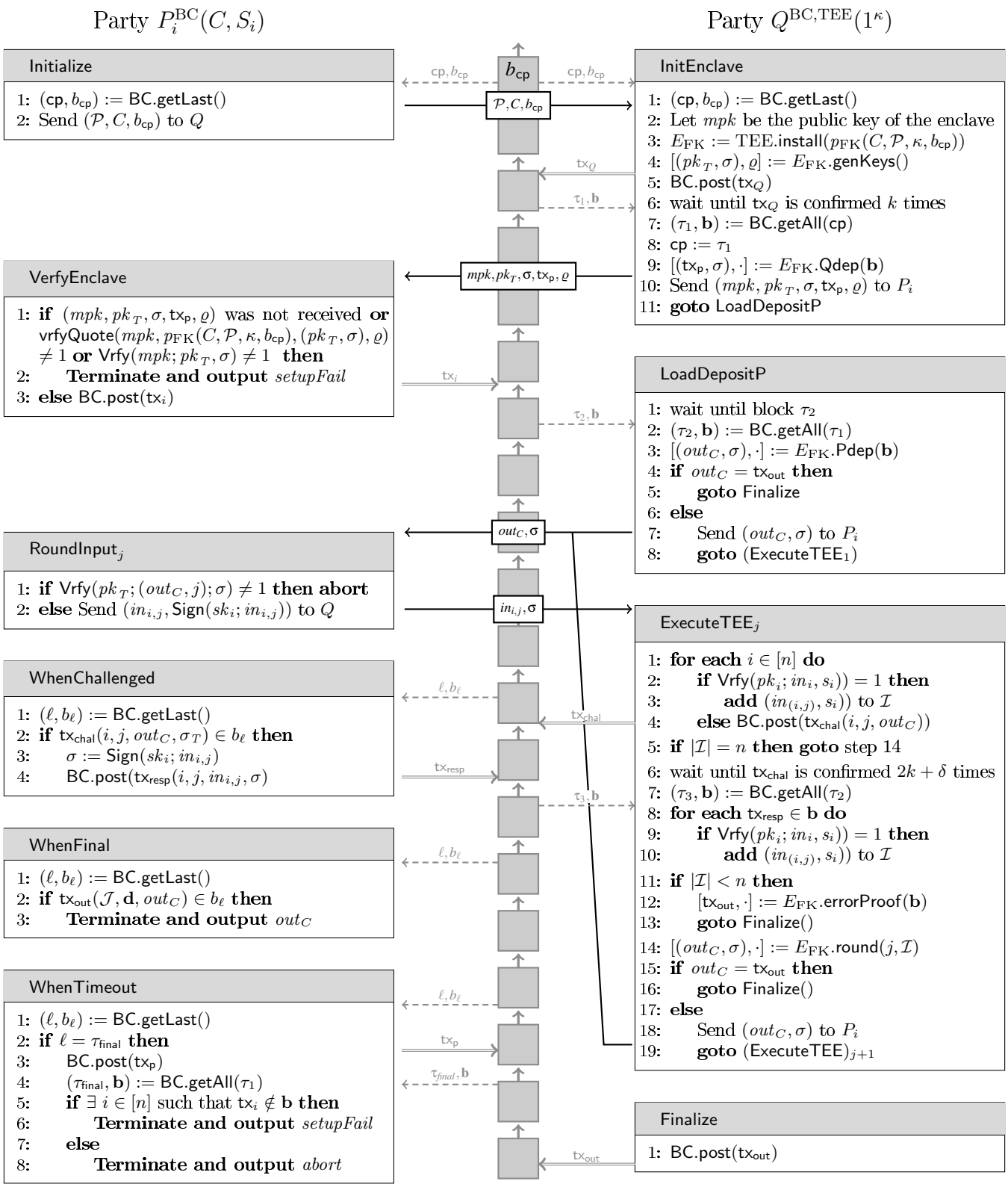


Figure 2: Protocol $\pi_{\text{FASTKITTEN}}$. Direct black arrows indicate communication between the parties and Q, gray dashed arrows indicate reading from the blockchain and gray double arrows posting on the blockchain.

coins. Then the *round execution* follows for all m rounds of the interactive contract. When the contract execution aborts or finishes, the protocol enters the *finalize phase*. We now explain all phases and the detailed protocol steps for all involved parties and the operator Q in depth. The detailed interactions as well as the subprocedure of the parties and the operator are displayed in Figure 2, Figure 3 describes the FASTKITTEN enclave program p_{FK}. Overall the protocol requires six different type of transactions.

Setup phase. In the setup phase, each party P_i first runs Initialize to generate its key pairs and gets the latest block b_{cp} which serves as a genesis block or checkpoint of the protocol. Then P_i sends the set of parties $\mathcal{P}$, the b_{cp} and the contract C to the operator Q. Upon receiving the initial values from all n parties, Q runs the subprocedure InitEnclave to initialize the trusted execution of the enclave program $p_{\mathrm{FK}}(\mathcal{P}, C, \kappa, b_{\mathsf{cp}})$ where κ is the security parameter of the scheme. This security parameter κ also determines the values for the timeout period t and the confirmation constant k. This ensures that all parties and the TEE agree on these fixed values. Once p_{FK} is installed in the enclave, it generates key pairs for the protocol execution and in particular the blockchain public key pk_T[6]. Now, Q can make its deposit transaction tx_Q which assigns q coins to the enclave public key.

Q's Deposit Transaction $\mathsf{tx_Q}$	
tx.Input:	Some unspent tx from Q
tx.Output:	Assign q coins to pk_T

Let block counter τ_1 denote the time when this transaction has been included and confirmed in the blockchain. Q loads all blocks from cp to τ_1 as evidence to the enclave. If this evidence is correct, the execution of p_{FK} function Qdep outputs a penalty transaction tx_{p}, stating that after timeout τ_{final} (after which the protocol must be terminated) the q coins of Q's deposit transaction tx_Q are payed out to the parties $P_1, \ldots, P_n$.

Penalty Transaction $\mathsf{tx_p}$	
tx.Input:	Q's Deposit Transaction tx_Q
	For all $i \in [n]$:
$\mathsf{tx.Output}_i$:	Assign c_i coins to P_i
tx.Time:	Spendable after τ_{final}

Q sends the penalty transaction to all parties $P_1, \ldots, P_n$, who run subprocedure VerfyEnclave. This transaction is used whenever the protocol does not finish before the final timeout τ_{final}, which equals $(3+2m) \times (\delta + k)$ blocks after the protocol start (recall, that we use δ to bound the time until some transaction is guaranteed to be included and it will be confirmed after k blocks).[7] Only if participant P_i received this penalty transaction from Q during the setup and verified that the program $p_{\mathrm{FK}}(\mathcal{P}, C, \kappa, b_0)$ is installed in the enclave, it creates and publishes its deposit transaction.

$\mathbf{P_i}$'s Deposit Transaction $\mathsf{tx_i}$	
tx.Input:	Some unspent tx from P_i
tx.Output:	Assign c_i coins to TEE

After time $\tau_2 < \tau_1$, Q executes LoadDepositP and again provides the block evidence to the enclave execution of p_{FK}. If all parties published the deposit transactions, the first-round execution starts. Otherwise the enclave proceeds to the finalize phase and outputs a refund transaction $\mathsf{tx}_{\mathsf{out}}(T, \vec{c})$ that returns the deposit back to honest users and Q, where $T \subset \mathcal{P}$ is the set of all parties that submitted the deposit transaction until time τ_2. Note, that the internal state of the contract execution is maintained by the p_{FK} program inside the enclave. This guarantees that the contract is not executed on outdated state.

Round computation phase. When the protocol arrives to the round computation phase, Q sends the authenticated output of the enclave to every party P_i and requests input for the next round. Each party P_i runs the round algorithm. Internally it verifies whether the input request came from the enclave by verifying the attached signature. Then it generates and signs its round input and sends it to Q. While P_i waits for the next round, Q verifies all received inputs and their signatures in the ExecuteTEE subprocedure. If all the parties P_i responded with correctly signed round inputs, Q triggers the execution of the contract in the enclave. Let us emphasize that in this simplified description of our protocol we do not focus on the privacy aspect and hence we omit that all round inputs to the contract could be encrypted with the public key of the enclave. In this case the trusted enclave execution needs to decrypt them before it evaluates the contract on them. See Section 9.3 for more details.

Note that the operator Q may be malicious and refrain from requesting a party P_i for the input to a round computation. Instead Q may pretend that it actually did not receive any input from the party P_i. On the other hand, one can imagine a scenario where Q is behaving honestly but the party P_i is dishonest and does not send the correctly signed round input to Q. Note, that the program p_{FK} cannot distinguish between these two cases without additional information. We will next show how an honest Q can generate a proof to attribute the malicious behavior to P_i. First, Q has to publish a challenge transaction $\mathsf{tx}_{\mathsf{chal}}$ which includes the signed output of the previous step. $\mathsf{tx}_{\mathsf{chal}}$ spends a very small amount μ of coins from Q and assign them to party P_i[8].

[6] For simplicity we omit here, that the enclave might need multiple key pairs for signing transactions and messages.

[7] The definition of τ_{final} guarantees that even if the execution is delayed in every round, an honest operator will not be penalized.

[8] Cryptocurrencies like Bitcoin allow transactions with very small denominations (e.g. fractions of cents).

Challenge Transaction $\mathsf{tx}_{\mathsf{chal}}(i, j, out_C, \sigma_T)$	
tx.Data:	Store i, j, out_C, σ_T
tx.Input:	Some unspent tx from Q
tx.Output:	Assign μ coins to P_i

Once $\mathsf{tx}_{\mathsf{chal}}$ is included in the blockchain, party P_i can read the correct output information from the transaction. The party should respond with $\mathsf{tx}_{\mathsf{resp}}$, which includes its signed round input. $\mathsf{tx}_{\mathsf{resp}}$ spends the $\mathsf{tx}_{\mathsf{chal}}$ and assigns the μ coins back to Q. The action of P_i is depicted via the WhenChallenged subprocedure.

Response Transaction $\mathsf{tx}_{\mathsf{resp}}(\mathbf{i}, \mathbf{j}, in, \sigma_{\mathbf{i}})$	
tx.Data:	Store i, j, in, σ_i
tx.Input:	Challenge Transaction $\mathsf{tx}_{\mathsf{chal}}(i, j, state)$
tx.Output:	Assign μ coins to Q

If some party does not send the response after it was challenged, Q can prove this misbehavior to the FASTKITTEN program, by providing the blockchain evidence of the challenge-response transcript. If the enclave program identifies a cheating party, it proceeds to the finalize phase. Otherwise, if all the parties' inputs were received with authentication (possibly after challenge-response phase), Q instructs the enclave to execute the contract on the accumulated input.

The result of the contract execution is the output out_C, the updated state *state*, and a coin distribution denoted by $\mathbf{d}$. If *state* equals $\perp$, the contract execution is finished, and the protocol proceeds to the finalize phase. Otherwise, FASTKITTEN internally stores the state and outputs out_C to Q who sends this output to all parties and waits for next round inputs.

Finalize phase. In the finalize phase, the enclave publishes a final output transaction $\mathsf{tx}_{\mathsf{out}}$ which distributes the coins back to all honest parties. It is parameterized by a set of parties to receive coins $\mathcal{J}$, a final coin distribution $\vec{e}$ and a final state out_C. The transaction $\mathsf{tx}_{\mathsf{out}}(\mathcal{J}, \vec{e}, out_C)$, spends all deposit transactions tx_i for all $i \in \mathcal{J}$ and Q's deposit transaction tx_Q. It includes the out_C in the data field and assigns q coins back to Q and e_i coins to party P_i, for every $i \in \mathcal{J}$. Let us note that $\mathcal{J} = [n]$ implies correct protocol termination. If $\mathcal{J} \neq [n]$, then some party misbehaved and the protocol failed. Either a party did not make a deposit in the setup phase (signaled by $out_C = setupFail$) or some party aborted in the round computation phase (signaled by $out_C = abort$). In both cases all other parties get their initial deposits back. Note, that if a party P_j is caught cheating by the TEE, it will lose its deposit.

Q now has to publish this transaction to get his coins before time τ_{final} and by that also distributes coins and reveals out_C to honest parties. The participants need to constantly monitor the blockchain for transactions which challenge them or indicate final output. When they see a challenge transaction they respond as described above. If they see an output transaction they know the protocol execution ended and output the final contract output according to subroutine WhenFinal.

Output Transaction $\mathsf{tx}_{\mathsf{out}}(\mathcal{J}, \vec{e}, out_C)$	
tx.Data:	Store out_C
tx.Input:	Deposit Transactions $\mathsf{tx}_Q, \{\mathsf{tx}_i\}_{i \in \mathcal{J}}$
$\mathsf{tx.Output}_1$:	q coins to Q
	For all $i \in \mathcal{J}$:
$\mathsf{tx.Output}_{i+1}$:	e_i coins to P_i

6 Execution Facility

As shown in Figure 1, we leverage a TEE for smart contract execution. For our prototype, we implemented FASTKITTEN for the Bitcoin blockchain using Intel SGX as a TEE. We chose Python as our scripting engine because it's memory safe, very well known, and widely available. To interact with the Bitcoin blockchain data in the enclave, we implemented our *Crypto library* using the open-source *breadwallet-core* [14], a simplified payment verification (SPV) library for Bitcoin used by the *Breadwallet* mobile wallet app. To abstract from SGX's peculiarities, and thus simplify smart contract development, we use the Graphene Library OS [17] (referred to as "Graphene" in the rest of the paper) as a basis. Graphene enables running arbitrary native Linux binaries in SGX enclaves while providing compatible library interfaces for networking and other OS services. Note that the design of the FASTKITTEN protocol does not require a trusted time source in the TEE.

6.1 The Enclave Program FASTKITTEN

An execution facility in the sense of FASTKITTEN must provide a set of abstract functionalities like key generation, transaction generation, smart contract execution, and error handling, all executed inside the enclave. This set of procedures is described in detail in Figure 3. We implemented each of the procedures using equivalent Python scripts. It is parameterized by the set of parties $\mathcal{P}$, the contract C which internally specifies the expected deposits $\mathbf{c}$, a security parameter κ and a genesis block b_{cp}. This does not need to be the actual genesis block of the underlying blockchain but it can be a later block which is used as a checkpoint. All parties must verify that this block is indeed a block of the blockchain. The security parameter κ also determines the waiting time k which is needed for the verification of the blocks.

6.2 Blockchain Verification

Blockchain communication is important for the setup and the finalization phase in the protocol. Thanks to the integrity properties of blockchains, a secure connection between the enclave and the blockchain is not needed if verification of received data can be done in the enclave. As it is not practical to download a complete copy of the blockchain to the enclave, we only concentrate on transactions caused by FASTKITTEN

The execution of p_{FK} is initialized with the secret key msk, the set of parties (where every $P_i \in \mathcal{P}$ is identified by its key pk_i), a contract C, a security parameter κ (which also defines the waiting period t and confirm period k) and a checkpoint b_{cp}. Internally it stores the state of the contract *state* and the status flag s initially set to $state = \emptyset$ and $s = \mathsf{genKeys}$.

procedure genKeys()

1: **if** $s \neq \mathsf{genKeys}$ **then** abort
2: $(sk_T, pk_T) := \mathsf{Gen}(1^\kappa)$
3: $s := \mathsf{Qdep}$
4: **return** $pk_T, \mathsf{Sign}(msk; pk_T)$

procedure Qdep($\mathbf{b}$)

1: **if** $s \neq \mathsf{Qdep}$ **or** $\mathsf{Extends}(b_{\mathsf{cp}}, \mathbf{b}) \neq 1$ **or** $\mathsf{Pos}(\mathbf{b}, \mathsf{tx}_Q) > |\mathbf{b}| - k$ **then** abort
2: $s := \mathsf{Pdep}$
3: $b_{\mathsf{cp}} :=$ last block of $\mathbf{b}$
4: **return** tx_{p} ▷ Else, output penalty transaction

procedure Pdep($\mathbf{b}$)

1: **if** $s \neq \mathsf{Pdep}$ **or** $\mathsf{Extends}(b_{\mathsf{cp}}, \mathbf{b}) \neq 1$ **then** abort
2: set $\mathcal{J} := \emptyset$
3: **for** $i \in \mathcal{P}$ **do**
4: $\quad \ell_i := \mathsf{Pos}(\mathbf{b}, \mathsf{tx}_i)$
5: $\quad$ **if** $\ell_i < \delta$ **and** $\ell_i < |\mathbf{b}| - k$ **then** add i to $\mathcal{J}$
6: **if** $\mathcal{J} = [n]$ **then**
7: $\quad s := \mathsf{round}_1$
8: $\quad b_{\mathsf{cp}} := \mathbf{b}.\mathsf{last}$
9: $\quad$ **return** $\emptyset, \mathsf{Sign}(sk_T; \emptyset, b_{\mathsf{cp}})$
10: **else**
11: $\quad s := \mathsf{terminated}$
12: $\quad$ **return** $\mathsf{tx}_{\mathsf{out}}(\mathcal{J}, \mathbf{c}, setupFail)$

procedure round($j, (in_1, \sigma_1) \ldots, (in_n, \sigma_n)$)

1: **if** $s \neq \mathsf{round}_j$ **or** for any $i \in [n] : \mathsf{Vrfy}(pk_i; in_i, s_i) \neq 1$ **then** abort
2: $(out_C, state', \mathbf{d}) := C(state, \vec{in})$
3: **if** $state' \neq \perp$ **then**
4: $\quad s := \mathsf{round}_{j+1}$
5: $\quad state := state'$
6: $\quad$ **return** $out_C, \mathsf{Sign}(sk_T; (out_C, j))$
7: **else**
8: $\quad s := \mathsf{terminated}$
9: $\quad$ **return** $\mathsf{tx}_{\mathsf{out}}([n], \mathbf{d}, out_C)$

procedure errorProof, $(j, \mathbf{b})$

1: **if** $s \neq \mathsf{round}_j$ **or** $\mathsf{Extends}(b_{\mathsf{cp}}, \mathbf{b}) \neq 1$ **then** abort
2: Let $\sigma := \mathsf{Sign}(sk_T; (out_C, j))$
3: $\mathcal{J} := [n]$
4: **for** $i \in \mathcal{P}$ **do**
5: $\quad$ **if** $\mathsf{Pos}(\mathbf{b}, \mathsf{tx}_{\mathsf{chal}}(i, j, out_C, \sigma)) < |\mathbf{b}| - \delta - k$ **then**
6: $\quad\quad$ **if** $\mathsf{Pos}(\mathbf{b}, \mathsf{tx}_{\mathsf{resp}}(i, j, in, \sigma) > |\mathbf{b}| - k$ **then**
7: $\quad\quad\quad$ delete i from $\mathcal{J}$
8: $\quad\quad$ **else if** $\mathsf{Vrfy}(pk_i; in, \sigma) \neq 1$ **then**
9: $\quad\quad\quad$ delete i from $\mathcal{J}$
10: $s = \mathsf{terminated}$
11: **if** $\mathcal{J} \neq [n]$ **then**
12: $\quad$ **return** $\mathsf{tx}_{\mathsf{out}}(\mathcal{J}, \mathbf{c}, abort)$

Figure 3: FASTKITTEN enclave program $p_{\mathrm{FK}}(\mathcal{P}, C, \kappa, b_{\mathsf{cp}})$

protocol invocation. Thus, it is sufficient to verify that these transactions are part of a valid block—without downloading entire blocks, which can be done efficiently using simplified payment verification (SPV). However, SPV libraries can only prove that a transaction *is* part of a block on the blockchain, but they cannot prove that a transaction *is not* part of any block. As required by the challenge-response case, we added an alternative verification mode that fully downloads every block that could potentially contain the transaction and checks whether its present in any of those blocks.

6.3 Participant Communication

To place the deposits and receive them later, as well for sending input, communication between participants (including the Operator Q) is needed in the off-chain phase. We secure this communication using TLS sockets provided by Python. This transparently encrypts participants' communication, and thus ensures input integrity and confidentiality of parties' messages towards the operator.

6.4 Enclave Setup

In the FASTKITTEN prototype, we leverage Intel SGX as a TEE. SGX is a TEE included in recent Intel CPUs which introduces the concept of isolated hardware *enclaves* that can be created and managed using new CPU instructions. SGX enclaves are even shielded from the operating system; only the CPU is trusted. To support smart contract execution in these enclaves we provide a run-time environment based on Graphene, which replaces the Intel SDK in both the enclave and the host process. This allows Graphene to transparently provide services from the untrusted OS (and check the integrity of the results). To protect the enclave application from the host process, a *manifest* has to be provided at enclave initialization. The manifest includes interfaces, services, and respective integrity checksums, e.g., hashes of files the enclave requires. Accesses to these files will be checked against hashes in the manifest to guarantee integrity.

As depicted by Figure 3, the Execution Facility incorporates a set of functionalities. For key derivation (*genKeys*) we leverage the `rdrand` instruction to get high-entropy randomness inside of the enclave. After checking that tx_Q (*Qdep*) is in the blockchain, the derived private key sk_T is used to generate the penalty transaction tx_p using our Crypto library. tx_p is distributed to the other participants over a TLS connection. Other participants can generate their deposit transactions tx_i (*Pdep*) using a regular wallet. This concludes the setup phase, and the smart contract gets executed (*round*).

The Graphene run-time environment enables FASTKITTEN to support arbitrary Linux binaries, thus, can be used to implement smart contracts. However, instead of allowing binaries, we use a scripting engine based on a Python interpreter in our proof-of-concept implementation. First, this makes development easier for contract developers, as they are not always familiar with lower-level programming languages, and second,

this makes smart contracts less prone to memory corruption vulnerabilities. Two use cases we implemented are presented and evaluated in Section 8.

6.5 Denial of Service Protection

The protocol as described in Section 5 assumes instantaneous contract execution meaning that the execution of a contract inside a TEE takes no time. For most practical contracts, this simplifying assumption is reasonable since executing a simple contract function inside a TEE is much faster than the network/blockchain delay. However, this is not true when considering arbitrary contracts which might potentially contain endless loops. Moreover, the halting problem states that it is impossible to predict if a certain algorithm will halt within a certain number of steps. A simple protection against endless loops and denial-of-service attacks, is letting the enclave monitor the execution of the smart contract and terminate execution if the number of execution steps exceeds a predefined limit. If the contract execution is aborted due to an execution timeout, the enclave signs an outputs transaction $\mathsf{tx}_{\mathsf{out}}$ which returns deposited coins back to parties and to the operator.

7 Security

In this section we present the underlying security considerations of FASTKITTEN.

7.1 Protocol Security

Due to limited space, we present our novel model in the extended version of this paper, where we also formally state the security properties, the formal statement of the theorem as well as the proof. Here we will only briefly explain the security properties.

In order to guarantee security for the protocol, we require three security properties: *correctness*, *fairness* and *operator balance security*.

Intuitively, correctness states that in case all parties behave honestly (including the operator), every party $P_i \in \mathcal{P}$ outputs the correct result and earns the amount of coins she is supposed to get according to the correct contract execution. The fairness property guarantees that if at least one party $P_i \in \mathcal{P}$ is honest, then (i) either the protocol correctly completes an execution of the contract or (ii) all honest parties output *setupFail* and stay financially neutral or (iii) all honest parties output *abort*, stay financially neutral, and at least one corrupt party must have been financially punished. Finally, the operator balance security property says that in case the operator behaves honestly, he cannot lose money.

Theorem 1 (Informal statement)**.** *The protocol* $\pi_{\text{FASTKITTEN}}$ *as defined in Section 5 satisfies correctness, fairness and operator balance security property.*

The most challenging part of the proof is the fairness property. We need to show how honest parties reach consensus on the result of the execution and prove that coins are always distributed between parties according to this result (even if malicious parties collude with the operator). In order to prove the operator balance security, we show that an honest operator has always enough time to publish a valid output transaction which pays him back his deposit, before the time-locked penalty transaction can be posted on the blockchain.

Incentive-driven adversary If we consider only incentive-driven adversaries, then statement (iii) of the fairness property is never true. Hence, if the setup phase completes successfully, then the result of the protocol is a correct contract execution. This follows directly from the fact, that when the protocol aborts the misbehaving parties lose coins. By definition of incentive-driven parties, losing coins is against their interest. This is why the only possible outcome of the protocol is correct execution of the contract. Moreover, when we consider fees for positing transaction on the blockchain, parties are additionally incentivized to prevent the challenge-response transactions. These additional incentives enforce fast and protocol compliant behavior of the parties.

7.2 Architecture Security

The main goal of FASTKITTEN is to enable efficient execution of general multi-round smart contracts. Hence, we analyze the security of FASTKITTEN with regards to its system architecture and implementation. Possible adversaries can be malicious participants, a malicious operator, or a combination of both.

We note that participating clients are only required to send and receive transactions from the blockchain (e.g., to enter an execution) and the ability to exchange protocol messages (e.g., to play rounds). Hence, client implementations can be based on a diverse set of entirely different code bases in practice, possibly using memory-safe languages such as Python, Go, or Rust. Malicious participants are further limited to interacting with other parties and the operator through the exchange of messages as specified within our protocol, and hence, we focus on the TEE-based execution facility in the following.

A malicious operator could deny execution, however, he is incentivized to adhere to the protocol or lose money. Thus, we assume that the goal of a malicious operator is to try and exploit the execution facility at runtime. Since the operator already controls the host process, the main target would be the enclave that executes the contract. Enclaves have a well-defined interface with the rest of the system, and any attack has to be launched using this interface. By providing fake data through this interface, the attacker could try to exploit a memory-corruption vulnerability in the low-level enclave code to launch (a) a code-reuse attack, e.g., by manipulating enclave stack memory, or (b) a data-only attack, e.g., to leak information about the game state or manipulate Bitcoin addresses in contracts. As mentioned in Section 4, for (a) we assume a standard code-reuse defense such as control-flow integrity [3, 15, 50, 62, 65] or fine-grained code randomization [21,23,30,42,53,61]. The core functionality of

FastKitten additionally tackles both attack vectors by implementing the main enclave code in Python, which provides memory-safety features such as implicit bounds checking. The only parts that are implemented in unsafe languages are the initialization code of Graphene [17] and the Simple Payment Verification (SPV) library [14]. FastKitten actually has no strong dependency on Graphene in principle, it was mainly used to simplify and speed up prototype implementation. Finally, SPV represents a standard library used by most blockchain clients and an adversary that is able to construct a data-only attack against it would be able to exploit any of those clients connected to the Bitcoin network using the same data-only attack.

8 FastKitten Contracts

In this section we take a look at applications and performance through a number of benchmarks.

8.1 Complexity

The FastKitten protocol consists of *setup*, *round computation* and *finalize* phases. During the *setup* phase, each party P_i deposits a constant amount of coins c_i. The operator needs to deposit an amount $\sum_{i\in[n]} c_i$ which equals the sum of all other deposits from $\mathcal{P}$ together. To post the deposit transactions tx_is and tx_Q, a total of $n+1$ transactions is necessary.

During the *round computation phase*, in the optimistic case FastKitten can operate completely off-chain without any blockchain interaction. Any user can force that challenge response transactions are posted to attribute misbehavior of a party, in any given round. If this (pessimistic) case occurs, it can add 2 to another $2n$ transactions. In the worst case, a challenge response transaction pair needs to be posted on the blockchain for every party P_i at every round $j \in [m]$ leading to $\mathcal{O}(nm)$ blockchain interactions. In *finalize* phase, FastKitten requires one additional payout transaction $\mathsf{tx}_{\mathsf{out}}$ to settle money distribution among parties. Scenarios of missing deposit at the *Setup* phase or an abort by a party at the *round computation* phase are dealt with by posting the refund transaction $\mathsf{tx}_{\mathsf{out}}$ and the penalty transaction tx_{p} respectively.

Setup time In the optimistic case (which we have shown is the standard case when considering incentive-driven parties) the overall execution of the protocol only requires $n+2$ transactions on the blockchain. This also indicates at what speed the protocol can be executed in this case. If all parties agree, the setup phase can be finished in 2 blockchain rounds and from that point on the protocol can be played off-chain. In the next subsection we give some indication how fast this second part can be achieved. Running the protocol as fast as possible is in the interest of every party since it shortens the locking time of the deposits.

8.2 Performance Evaluation

We performed a number of performance measurements to demonstrate the practicality of FastKitten using our lab setup, which consists of three machines: First, an SGX-enabled machine running Ubuntu 16.04.5 LTS with an Intel i7-7700 CPU clocked at 3.60GHz and 8GB RAM, where we installed FastKitten's contract execution facility to play the role of the operator's server. Second, a machine running Ubuntu 14.04.4 LTS on an Intel i7-6700 CPU clocked at 3.40GHz with 32GB RAM, which provides unmodified blockchain nodes in a local test network using Bitcoin Core version 0.16.1. Third, a laptop machine with macOS 10.13.6 on with Intel i7-4850HQ CPU clocked at 2.30GHz and 16GB of RAM, which takes the role of the participants in the protocol. All three machines are connected through a Gigabit Ethernet LAN. For tests involving the real Bitcoin network the individual machines are connected through the Internet using our Internet connection.

Block validation In our experiments, the enclave takes approximately 5 s to validate one block from the Bitcoin main network, thus proving that it is capable of validating real blocks in real time.

Enclave Startup The time to setup an enclave until it is ready is 2 s, proving that instantiating enclaves on the fly is feasible.

End-to-end Time Assuming all parties are incentive-driven and, thus, comply with the protocol, the total time required by FastKitten is the time of 2 blockchain interactions (see Section 8.1), plus the computation time (a few milliseconds in our use cases), plus the time required by the parties to choose the next inputs.

8.3 Applications

FastKitten allows to run complex smart contracts on top of cryptocurrencies that would not natively support such contracts, like Bitcoin. But in contrast to Turing-complete contract execution platforms like Ethereum, a secure off-chain execution such as FastKitten puts some restrictions on the contracts it can run:

- The number of parties interacting with the contract must be known at the start of the protocol.
- It must be possible to estimate an upper bound on the number of rounds and the maximum run time of any round.

All of these restrictions make FastKitten contracts different from smart contracts running on Ethereum itself. The restrictions above come from the fact that the contract can be completely (and repeatedly) executed without blockchain interactions. Other off-chain solutions (like state channels [20,24,49]) come with similar caveats. By allowing additional blockchain interaction we could get around those restrictions but we would lose efficiency in the optimistic case (which is also similar to state channel constructions).

FastKitten has important features which are supported by neither Bitcoin nor Ethereum — FastKitten allows private inputs and batched execution of user inputs. Overall, this leads to cheaper, faster and private contract execution than what

is possible with on-chain contracts in Ethereum. Below, we highlight these efficiency gains by presenting four concrete use-cases in which FASTKITTEN outperforms contracts run over Ethereum or in Ethereum state channels.

Lottery A lottery contract takes coins from every involved party as input, and randomly selects one winner, who gets all the coins. The key challenge for such a contract is to fairly generate randomness to select the winner. In Ethereum or Bitcoin the randomness is computed from user inputs through an expensive commit-reveal scheme [48]. In FASTKITTEN, all parties can immediately send their random inputs to the enclave which will securely determine a winner. Hence, we reduce the round complexity from $\mathcal{O}(\log n)$ [48] to $\mathcal{O}(1)$.

Auctions Another interesting use-case for smart contracts are auctions, where parties place bids on how much they are willing to pay and the contract determines the final price. In a straightforward auction, the bids can be public, but more fair versions, like second bid auctions, require the users not to learn the other bids before they place their own. The privacy features of FASTKITTEN can be used to reduce the round complexity for such auctions which would otherwise require complex cryptographic protocols [25].

Rock-paper-scissors We implemented the popular two-party game rock-paper-scissors to show the feasibility of FASTKITTEN contracts. Again, the privacy features allow one match to be executed in a single round, which would have required at least 3 rounds in Ethereum. The pure execution time in the optimistic case, excluding delays due to human reaction times, is 12ms for one round (averaged over 100 matches). This demonstrates that off-chain protocols, like FASTKITTEN, are highly efficient when the same set of parties wants to run complex contracts (like multiple matches of a game).

Poker We also implemented a Texas Hold'em Poker game, to prove that multi-party contracts which inherently require multiple rounds can also be efficiently executed in FASTKITTEN. In our implementation, each player starts with an equal chip stack and participates in an initial betting round and in additional rounds after the flop, river, and turn have been dealt by the enclave. If more than two players remain in the game after the final bets, the enclave reveals the winner and distributes the chips in the current pot to the winner. The game continues until only one player remains. We measured 50 matches between 10 players resulting in an average time of 45ms per match (multiple betting rounds are included in each match). The run time was measured starting from the moment all deposits are committed to the blockchain.

Real-world Fees We generated examples of the transaction types used in our protocol for a 10-player poker match. In Table 2 we estimate the fees required to commit to the blockchain our transactions, in addition to a typical deposit transaction. Assuming all parties comply with the protocol, each party (including Q) must pay between 0.05 USD and 0.46 USD for the deposit. Additionally, the output transaction $\mathsf{tx}_{\mathsf{out}}$ requires between 0.36 USD and 3.65 USD in fees.

Transaction	Size (Bytes)	Fees (BTC)	Fees (USD)
Deposit (typical)	250	0.000007-0.000073	0.05-0.46
Penalty (tx_{p})	504	0.000015-0.000148	0.09-0.93
Challenge ($\mathsf{tx}_{\mathsf{chal}}$)	293	0.000009-0.000086	0.05-0.54
Response ($\mathsf{tx}_{\mathsf{resp}}$)	266	0.000008-0.000078	0.05-0.49
Output ($\mathsf{tx}_{\mathsf{out}}$)	1986	0.000058-0.000582	0.36-3.65

Table 2: Estimated fees for a typical deposit transaction and the FASTKITTEN transactions, using data from CoinMarketCap [2] and BlockCypher [1] retrieved on Nov. 14, 2018.

Other Well-known Contracts Certain well-known contracts like ERC20 token and CryptoKitties inherently need to be publicly available on the blockchain, since they are accessed frequently by participants which are not previously known. In contrast, contracts resembling our examples above, which rely on private data and where a fixed set of participants sends a large number of transactions, are highly efficient when moved off-chain using a system like FASTKITTEN. The nature of off-chain solutions like FASTKITTEN or state channels requires advance knowledge of the participants. Open contracts like ERC20 and CryptoKitties that require continuous synchronization with the blockchain and are meant to be publicly accessible would eliminate the advantages of off-chain solutions.

9 Discussion and Extensions

In order to explain and analyze the FASTKITTEN protocol, we presented a simplified protocol version which only includes the building blocks required to guarantee security. Depending on the use case one might be interested in further properties. Possible extensions discussed in this section include the option to pay the operator for his service, protect the operator against TEE faults, hide the contract output from through a layer of output encryption and allow cross-currency smart contracts. In the following, we explain how to achieve these features and at what cost they can be added to the simplified protocol.

9.1 Fees for the Operator

The owner of the TEE provides a service to the users who want to run a smart contract and, naturally, he wants to be paid for it. In addition to the costs of buying, maintaining and running the trusted hardware, he also needs to block the security deposit q for the duration of the protocol. While the security of FASTKITTEN ensures that he will never lose this money, he still cannot use it for other purposes. The goal of the operator-fees is to make both investments attractive for Q. We assume that the operator will be paid ξ coins for each protocol round for each party. Since the maximum number of rounds m is fixed at the protocol start, Q will receive $\xi \times$

$n \times m$ coins if the protocol succeeds (even if the contract terminated in less than m rounds). If the operator proves to the TEE in round x that another party did not respond to the round challenge, he will only receive a fee for the passed x number of rounds (namely $\xi \times x \times n$). This pay-per-round model ensures that the operator does not have any incentive to end the protocol too early. If the protocol setup does not succeed or the operator cheats, he will not receive any coins. The extended protocol with operator fees requires each party to lock $c_i + m \times \xi$ coins and the operator needs to level this investment with $qc_i + m \times \xi$ coins.

9.2 Fault Tolerance

In order to ensure that the execution of the smart contract can proceed even in the presence of software or hardware faults, the enclave can save a snapshot of the current state in an encrypted format, e.g., after every round of inputs. This encrypted state would be sent to the operator and stored on redundant storage. If the enclave fails, the operator can instantiate a new enclave which will restart the computation starting from the encrypted snapshot. If the TEE uses SGX, snapshots would leverage SGX's sealing functionality [31] to protect the data from the operator while making it available to future enclave instances.

9.3 Privacy

As mentioned in the introduction, traditional smart contracts cannot preserve privacy of user inputs and thus always leak internal data to the public. In contrast to common smart contract technologies, the FASTKITTEN protocol supports privacy preserving smart contracts as proposed in Hawk [36]. This requires *private contract state* to hide the internal execution of the contract and *input privacy*, which means that no party (including the operator) sees any other parties' round input before sending its own.

It is straightforward to see that FASTKITTEN has a secret state, since it is stored and maintained inside the enclave. Input privacy can easily be achieved by encrypting all inputs with the public key of the enclave. This guarantees that only the FASTKITTEN execution facility and the party itself knows the inputs. If required, FASTKITTEN could also be extended to support privacy of outputs from the contract to the parties, by letting the enclave encrypt the individual outputs with the parties' public keys. But this additional layer should only be used when the contract requires it, since in the worst case this increases the output complexity of the challenge and output transaction.

9.4 Multi-currency Contracts

FASTKITTEN requires from the underlying blockchain technology that transactions can contain additional data and can be timelocked. Any blockchain like Bitcoin, Ethereum, Litecoin and many others which allow these transaction types can be used for the FASTKITTEN protocol. With some minor modifications FASTKITTEN can even support contracts which can be funded via multiple different currencies. This allows parties that own coins in different currencies to still execute a contract (play a game) together. The main modification to the FASTKITTEN protocol is that the operator and the enclave need to simultaneously handle multiple blockchains in parallel. In particular, for each of the considered currencies, Q needs to deposit the sum of all coins that were deposited by parties in that currency. This is in order to guarantee that if the operator cheats, players get back their invested coins in the correct currency. In addition, the operator is obliged to challenge each party via its blockchain. If the execution completes (or the operator proves to the enclave that one of the players cheated), the enclave signs one output transaction for each of the currencies. While this extension adds complexity to the enclave program and leads to more transactions and thus transaction-fees, the overall deposit amount stays identical to the single blockchain use case.[9] A complete design and proof of correctness of a cross-ledger FASTKITTEN is left to future work.

10 Conclusion

In this paper we have shown that efficient smart contracts are possible using only standard transactions by combining blockchain technology with trusted hardware. We present FASTKITTEN, our Bitcoin-based smart contract execution framework that can be executed off-chain. Since FASTKITTEN is the first work that supports efficient multi-round contracts handling coins, for the first time, this enables real-time application scenarios, like interactive online gaming, with millisecond round latencies between participants. We formally prove and thoroughly analyze the security of our general framework, also extensively evaluating its performance in a number of use cases and benchmarks.

Additionally, we discuss multiple extensions to our protocol, such as adding output privacy or operator fees, which enrich the set of features provided by our system.

Acknowledgments

We are grateful to our anonymous reviewers and our shepherd Mihai Christodorescu for their constructive feedback.

This work has been supported by the German Research Foundation (DFG) as part of projects HWSec, P3 and S7 within the CRC 1119 CROSSING and the Emmy Noether Program FA 1320/1-1, by the German Federal Ministry of Education and Research (BMBF) and the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP, by BMBF within the iBlockchain project, by the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS).

[9] This solution assumes that any party can receive coins in any of the considered currencies.

Availability

An extended version of this paper, which includes the bytecode of our sample Bitcoin transactions, will be publicly available at the Cryptology ePrint Archive at https://eprint.iacr.org.

References

[1] BlockCypher, Nov 2018. https://live.blockcypher.com/btc/.

[2] CoinMarketCap, Nov 14 2018. https://coinmarketcap.com.

[3] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information System Security*, 13, 2009.

[4] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. ACM, 2013.

[5] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. Secure multiparty computations on bitcoin. In *2014 IEEE Symposium on Security and Privacy*, 2014.

[6] ARM Limited. Security technology: building a secure system using TrustZone technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2008.

[7] C. Badertscher, U. Maurer, D. Tschudi, and V. Zikas. Bitcoin as a transaction ledger: A composable treatment. In *CRYPTO*, 2017.

[8] J. Barbie. Why smart contracts are not feasible on plasma, Jul 2018. https://ethresear.ch/t/why-smart-contracts-are-not-feasible-on-plasma/2598.

[9] G. Belisle. A glimpse into the future of blockchain, 2018. Available at https://the-blockchain-journal.com/2018/03/29/a-glimpse-into-the-future-of-blockchain/.

[10] I. Bentov, Y. Ji, F. Zhang, Y. Li, X. Zhao, L. Breidenbach, P. Daian, and A. Juels. Tesseract: Real-time cryptocurrency exchange using trusted hardware. *IACR Cryptology ePrint Archive*, 2017.

[11] A. Biondo, M. Conti, L. Davi, T. Frassetto, and A.-R. Sadeghi. The guard's dilemma: Efficient code-reuse attacks against intel sgx. In *Proceedings of the 27th USENIX Conference on Security Symposium*. USENIX Association, 2018.

[12] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, U. Müller, and A. Sadeghi. DR.SGX: hardening SGX enclaves against cache attacks with data location randomization. *CoRR*, abs/1709.09917, 2017.

[13] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software grand exposure: SGX cache attacks are practical. In *USENIX Workshop on Offensive Technologies*, 2017.

[14] Breadwallet. Breadwallet-core - spv bitcoin c library, 2018.

[15] N. Burow, S. A. Carr, S. Brunthaler, M. Payer, J. Nash, P. Larsen, and M. Franz. Control-flow integrity: Precision, security, and performance. *CoRR*, 2016.

[16] V. Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 2014.

[17] C. che Tsai, D. E. Porter, and M. Vij. Graphene-sgx: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference*, 2017.

[18] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. Detecting privileged side-channel attacks in shielded execution with Déjá Vu. In *ACM Symposium on Information, Computer and Communications Security*, 2017.

[19] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contract execution. *arXiv preprint arXiv:1804.05141*, 2018.

[20] J. Coleman, L. Horne, and L. Xuanji. Counterfactual: Generalized state channels, Jun 2018. https://l4.ventures/papers/statechannels.pdf.

[21] M. Conti, S. Crane, T. Frassetto, A. Homescu, G. Koppen, P. Larsen, C. Liebchen, M. Perry, and A.-R. Sadeghi. Selfrando: Securing the tor browser against de-anonymization exploits. *Proceedings on Privacy Enhancing Technologies*, 2016.

[22] V. Costan, I. A. Lebedev, and S. Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security Symposium*, 2016.

[23] L. Davi, A. Dmitrienko, S. Nürnberger, and A. Sadeghi. Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In *8th ACM Symposium on Information, Computer and Communications Security*, ASIACCS, 2013.

[24] S. Dziembowski, S. Faust, and K. Hostáková. General state channel networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, 2018.

[25] H. Galal and A. Youssef. Verifiable sealed-bid auction on the ethereum blockchain. In *International Conference on Financial Cryptography and Data Security, Trusted Smart Contracts Workshop. Springer*, 2018.

[26] J. A. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol with chains of variable difficulty. In *CRYPTO*. Springer, 2017.

[27] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *26th USENIX Security Symposium*, 2017.

[28] M. Hachman. Intel's plan to fix meltdown in silicon raises more questions than answers. https://www.pcworld.com/article/3251171/components-processors/intels-plan-to-fix-meltdown-in-silicon-raises-more-questions-than-answers.html, 2018.

[29] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using Innovative Instructions to Create Trustworthy Software Solutions. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. ACM, 2013.

[30] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. Librando: transparent code randomization for just-in-time compilers. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2013.

[31] Intel. Intel Software Guard Extensions developer guide, 2016. https://download.01.org/intel-sgx/linux-1.7/docs/Intel_SGX_Developer_Guide.pdf.

[32] Intel. Resources and Response to Side Channel L1 Terminal Fault. https://www.intel.com/content/www/us/en/architecture-and-technology/l1tf.html, 2018.

[33] A. Juels, A. E. Kosba, and E. Shi. The ring of gyges: Investigating the future of criminal smart contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, 2016.

[34] H. A. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten. Arbitrum: Scalable, private smart contracts. In *USENIX Security Symposium*, 2018.

[35] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016.

[36] A. E. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *IEEE Symposium on Security and Privacy*, 2016.

[37] R. Kumaresan and I. Bentov. How to use bitcoin to incentivize correct computations. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014.

[38] R. Kumaresan and I. Bentov. Amortizing secure computation with penalties. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[39] R. Kumaresan, T. Moran, and I. Bentov. How to use bitcoin to play decentralized poker. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015.

[40] R. Kumaresan, V. Vaikuntanathan, and P. N. Vasudevan. Improvements to secure computation with penalties. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[41] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1982.

[42] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated software diversity. In *35th IEEE Symposium on Security and Privacy*, S&P, 2014.

[43] J. Lind, O. Naor, I. Eyal, F. Kelbert, P. R. Pietzuch, and E. G. Sirer. Teechain: Reducing storage costs on the blockchain with offline payment channels. In *11th ACM International Systems and Storage Conference*, 2018.

[44] R. Matzutt, J. Hiller, M. Henze, J. H. Ziegeldorf, D. Müllmann, O. Hohlfeld, and K. Wehrle. A quantitative analysis of the impact of arbitrary blockchain content on bitcoin. In *Proceedings of the 22nd International Conference on Financial Cryptography and Data Security (FC). Springer*, 2018.

[45] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. ACM, 2013.

[46] Microsoft. The coco framework, 2018. GIT repository available at `https://github.com/Azure/coco-framework`.

[47] I. Miers, C. Garman, M. Green, and A. D. Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013.

[48] A. Miller and I. Bentov. Zero-collateral lotteries in bitcoin and ethereum. In *Security and Privacy Workshops (EuroS&PW), 2017 IEEE European Symposium on*. IEEE, 2017.

[49] A. Miller, I. Bentov, R. Kumaresan, and P. McCorry. Sprites: Payment channels that go faster than lightning. *CoRR*, abs/1702.05812, 2017.

[50] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz. Opaque control-flow integrity. In *NDSS*, 2015.

[51] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system," http://bitcoin.org/bitcoin.pdf, 2008.

[52] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *22nd USENIX Security symposium*, USENIX Sec, 2013.

[53] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *33rd IEEE Symposium on Security and Privacy*, S&P, 2012.

[54] R. Pass, E. Shi, and F. Tramèr. Formal abstractions for attested execution secure processors. *IACR Cryptology ePrint Archive*, 2016.

[55] J. Poon and V. Buterin. Plasma: Scalable autonomous smart contracts, Aug 2017. Plasma, `https://plasma.io/plasma.pdf/`.

[56] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim. SGX-Shield: Enabling address space layout randomization for SGX programs. In *Annual Network and Distributed System Security Symposium*, 2017.

[57] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Annual Network and Distributed System Security Symposium*, 2017.

[58] J. Teutsch and C. Reitwießner. A scalable verification solution for blockchains, Nov 2017. `https://people.cs.uchicago.edu/~teutsch/papers/truebit.pdf`.

[59] F. Tramèr, F. Zhang, H. Lin, J. Hubaux, A. Juels, and E. Shi. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P*, 2017.

[60] J. Van Bulck, F. Piessens, and R. Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018.

[61] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2012.

[62] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *34th IEEE Symposium on Security and Privacy*, S&P, 2013.

[63] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM, 2016.

[64] F. Zhang, P. Daian, I. Bentov, and A. Juels. Paralysis proofs: Safe access-structure updates for cryptocurrencies and more. *IACR Cryptology ePrint Archive*, 2018.

[65] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *22nd USENIX Security Symposium*, USENIX Sec, 2013.

A Further Related Work

There is a large body of work trying to improve the scalability of blockchains by moving a major part of smart contract executions off the blockchain (for example, via second layer solutions [24, 34, 49, 55] or outsourcing of computation [58]). As discussed in the main body of this paper, all of these solutions run on top of blockchains with sufficiently complex scripting language, e.g., on Ethereum. However, they cannot be integrated into popular legacy cryptocurrencies such as Bitcoin, which is their main difference compared to our work. Recall that one of the main goals of FASTKITTEN is make minimal assumption on the underlying blockchain technology and in particular, to run over the Bitcoin blockchain.

Another motivation for off-chain contract execution might be the goal of protecting privacy. Hawk [36] and the "Ring of Gyges" [33] are examples of works that do keep the state, all inputs and outputs private. It is also true for the scaling solutions mentioned above; These techniques work only over cryptocurrencies with support for complex smart contracts, e.g. over Ethereum.

Below we discuss the differences between these solutions and FASTKITTEN when run on top of Ethereum.

A.1 Second-layer Scaling Solutions

State Channels State channels [20, 24, 49] are a prominent second layer scaling solution. They allow a set of parties

to execute complex smart contracts off-chain. As long as all parties are honest and agree on the state transitions, the blockchain is contacted only during the channel creation, when parties lock funds in the channel, and during channel closure, when the locked funds are distributed back to the parties according to the result of contract execution. However, once parties run into disagreement off-chain, they have to resolve their dispute on-chain and perform the state transition via the blockchain.

While in the optimistic case when all parties are honest, state channels are very efficient, a potentially heavy computation might need to be done on-chain in case of disagreement. This is in contrast to the FASTKITTEN protocol which does not require any computation to be performed on the blockchain even in case of disputes.

Plasma Another promising second-layer scaling solution is Plasma, first introduced by Poon and Buterin [55]. The main idea of Plasma is to build new chains (Plasma chains) on top of the Ethereum blockchain. Each Plasma chain has its own operator that is responsible for validating transactions and regularly posting a short commitment about the current state of the Plasma chain to a smart contract on the Ethereum blockchain. The regular commitments guarantee to the participants of the Plasma chain that in case the operator cheats, his misbehavior can be proven to the Ethereum smart contract and parties can exit the Plasma chain with all their funds.

While the original goal of Plasma [55] was to support arbitrary complex smart contracts, to the best of our knowledge, there is no concrete protocol that would achieve this goal (the existing Plasma designs support only payment transactions). Moreover, the plasma research community currently conjectures that Plasma with general smart contracts might be impossible to construct [8].

A.2 Incentive-driven Verification

Arbitrum The disadvantage of state channels, i.e., the potentially heavy on-chain execution in case of dispute, is being addressed by the work Arbitrum [34]. Every smart contract, which Arbitrum models as a virtual machine (VM), to be executed off-chain has a set of "manager" parties responsible for correct VM execution. As long as managers reach consensus on the VM state transitions, execution progresses off-chain similarly as in state channels. In case of dispute, managers do not perform the VM state transition on-chain as in state channel. Instead, one manager can propose the next VM state which other managers can challenge. If the newly posted state is challenged, the proposer and the challenger run an interactive protocol via the blockchain, so-called "bisection" protocol, in which one disputable computation step is eventually identified and whose correct execution is verified on-chain. Hence, instead of executing the entire state transition on-chain (which might potentially require a lot of time/space), only one computation step of the state transition has to be performed on-chain in addition to the bisection protocol (which might require $\mathcal{O}(\log(s))$ blockchain transactions, where s is the number of computations steps in the state transition). The Arbitrum protocol works under the assumption that at least one manager of the VM is honest and challenges false states if they are posted by other managers. Since the blockchain interaction during the bisection protocol is rather expensive, Arbitrum uses monetary incentives to motivate managers to behave honestly and follow the protocol.

TrueBit Another solution that supports off-chain execution of smart contracts using incentive verification is TrueBit [58]. For each off-chain execution, the TrueBit system selects (using a lottery) one party, called the "Solver", that is responsible for performing the state transition and inform all other parties about the new contract state. The TrueBit system incentives parties to become so called "verifiers" and check the correctness of the computation performed by the Solver. In case they detect misbehavior, they are supposed to challenge the Solver on the blockchain and run the "verification game" which works similarly as the "bisection protocol" of Arbitrum. Similar to Arbitrum, TrueBit relies on the assumption that there is at least one honest verifier which correctly performs all the validations and challenges malicious Solvers. In contrast to Arbitrum, all inputs and the contract state are inherently public even in the optimistic case when everyone is honest.

Apart from the different trust model and lower requirement on the underlying blockchain technology, FASTKITTEN differs from Arbitrum and TrueBit by providing stronger privacy guarantees, meaning that in both the optimistic and the pessimistic case, inputs of honest parties as well as the state of the smart contract remains private.

A.3 TEEs for privacy

None of the solutions discussed above achieves privacy preserving off-chain contract execution. This is tackled by the work Hawk [36] which keeps the state, all inputs and all outputs private. Hawk contracts [35] achieve these properties using Ethereum smart contracts that judge computations done by a third party (a manager), who executes the contract on private inputs and is trusted not to reveal any secrets. First all parties submit their encrypted inputs to the contract, then the manager computes the result and proves its correctness with a zero knowledge proof. If the proof is correct, the contract pays out money accordingly. While the authors of Hawk discuss the possibility to use SGX for instantiating the manager and reducing the trust assumptions in this party, it still leverages the blockchain for every user input, and it only supports single round protocols which is their main difference to FASTKITTEN. A possible extension to multi-round protocols would be difficult to achieve without letting the smart contract verify the correctness of every round individually, and thus create a large blockchain communication overhead.

StrongChain: Transparent and Collaborative Proof-of-Work Consensus

Pawel Szalachowski[1] Daniël Reijsbergen[1] Ivan Homoliak[1] Siwei Sun[2,*]
[1]*Singapore University of Technology and Design (SUTD)*
[2]*Institute of Information Engineering and DCS Center, Chinese Academy of Sciences*

Abstract

Bitcoin is the most successful cryptocurrency so far. This is mainly due to its novel consensus algorithm, which is based on proof-of-work combined with a cryptographically-protected data structure and a rewarding scheme that incentivizes nodes to participate. However, despite its unprecedented success Bitcoin suffers from many inefficiencies. For instance, Bitcoin's consensus mechanism has been proved to be incentive-incompatible, its high reward variance causes centralization, and its hardcoded deflation raises questions about its long-term sustainability.

In this work, we revise the Bitcoin consensus mechanism by proposing StrongChain, a scheme that introduces transparency and incentivizes participants to collaborate rather than to compete. The core design of our protocol is to reflect and utilize the computing power aggregated on the blockchain which is invisible and "wasted" in Bitcoin today. Introducing relatively easy, although important changes to Bitcoin's design enables us to improve many crucial aspects of Bitcoin-like cryptocurrencies making it more secure, efficient, and profitable for participants. We thoroughly analyze our approach and we present an implementation of StrongChain. The obtained results confirm its efficiency, security, and deployability.

1 Introduction

One of the main novelties of Bitcoin [28] is Nakamoto consensus. This mechanism enabled the development of a permissionless, anonymous, and Internet-scale consensus protocol, and combined with incentive mechanisms allowed Bitcoin to emerge as the first decentralized cryptocurrency. Bitcoin is successful beyond all expectations, has inspired many other projects, and has started new research directions. Nakamoto consensus is based on proof-of-work (PoW) [8] in order to mitigate Sybil attacks [6]. To prevent modifications, a cryptographically-protected append-only list [2] is introduced. This list consists of transactions grouped into blocks and is usually referred to as a *blockchain*. Every active protocol participant (called a *miner*) collects transactions sent by users and tries to solve a computationally-hard puzzle in order to be able to write to the blockchain (the process of solving the puzzle is called *mining*). When a valid solution is found, it is disseminated along with the transactions that the miner wishes to append. Other miners verify this data and, if valid, append it to their replicated blockchains. The miner that has found a solution is awarded by a) the system, via a rewarding scheme programmed into the protocol, and b) fees paid by transaction senders. All monetary transfers in Bitcoin are expressed in its native currency (called bitcoin, abbreviated as BTC) whose supply is limited by the protocol.

Bitcoin has started an advent of decentralized cryptocurrency systems and as the first proposed and deployed system in this class is surprisingly robust. However, there are multiple drawbacks of Bitcoin that undermine its security promises and raise questions about its future. Bitcoin has been proved to be incentive-incompatible [9, 11, 39, 47]. Namely, in some circumstances, the miners' best strategy is to not announce their found solutions immediately, but instead withhold them for some time period. Another issue is that the increasing popularity of the system tends towards its centralization. Strong competition between miners resulted in a high reward variance, thus to stabilize their revenue miners started grouping their computing power by forming *mining pools*. Over time, mining pools have come to dominate the computing power of the system, and although they are beneficial for miners, large mining pools are risky for the system as they have multiple ways of abusing the protocol [9, 11, 18, 39]. Recently, researchers rigorously analyzed one of the impacts of Bitcoin's deflation [4, 27, 47]. Their results indicate that Bitcoin may be unsustainable in the long term, mainly due to decreasing miners' rewards that will eventually stop completely. Besides that, unusually for a transaction system, Bitcoin is designed to favor availability over consistency. This choice was motivated by its open and

*This work was done while the author was at SUTD.

permissionless spirit, but in the case of inconsistencies (i.e., *forks* in the blockchain) the system can be slow to converge.

Motivated by these drawbacks, we propose StrongChain, a simple yet powerful revision of the Bitcoin consensus mechanism. Our main intuition is to design a system such that the mining process is more transparent and collaborative, i.e., miners get better knowledge about the mining power of the system and they are incentivized to solve puzzles together rather than compete. In order to achieve it, in the heart of the StrongChain's design we employ *weak solutions*, i.e., puzzle solutions with a PoW that is significant yet insufficient for a standard solution. We design our system, such that a) weak solutions are part of the consensus protocol, b) their finders are rewarded independently, and c) miners have incentives to announce own solutions and append solutions of others immediately. We thoroughly analyze our approach and show that with these changes, the mining process is becoming more transparent, collaborative, secure, efficient, and decentralized. Surprisingly, we also show how our approach can improve the freshness properties offered by Bitcoin. We present an implementation and evaluation of our scheme.

2 Background and Problem Definition

2.1 Nakamoto Consensus and Bitcoin

The Nakamoto consensus protocol allows decentralized and distributed network comprised of mutually distrusting participants to reach an agreement on the state of the global distributed ledger [28]. The distributed ledger can be regarded as a linked list of blocks, referred to as the *blockchain*, which serializes and confirms "transactions". To resolve any *forks* of the blockchain the protocol specifies to always accept the longest chain as the current one. Bitcoin is a peer-to-peer cryptocurrency that deploys Nakamoto consensus as its core mechanism to avoid double-spending. Transactions spending bitcoins are announced to the Bitcoin network, where miners validate, serialize all non-included transactions, and try to create (mine) a block of transactions with a PoW embedded into the block header. A valid block must fulfill the condition that for a cryptographic hash function H, the hash value of the block header is less than the target T.

Brute-forcing the nonce (together with some other changeable data fields) is virtually the only way to produce the PoW, which costs computational resources of the miners. To incentivize miners, the Bitcoin protocol allows the miner who finds a block to insert a special transaction (see below) minting a specified amount of new bitcoins and collecting transaction fees offered by the included transactions, which are transferred to an account chosen by the miner. Currently, every block mints 12.5 new bitcoins. This amount is halved every four years, upper-bounding the number of bitcoins that will be created to a fixed total of 21 million coins. It implies that after around the year 2140, no new coins will be created, and the transaction fees will be the only source of reward for miners. Because of its design, Bitcoin is a deflationary currency.

The overall hash rate of the Bitcoin network and the difficulty of the PoW determine how long it takes to generate a new block for the whole network (the block interval). To stabilize the block interval at about 10 minutes for the constantly changing total mining power, the Bitcoin network adjusts the target T every 2016 blocks (about two weeks, i.e., a *difficulty window*) according to the following formula

$$T_{new} = T_{old} \cdot \frac{\textit{Time of the last 2016 blocks}}{\textit{2016} \cdot \textit{10 minutes}}. \qquad (1)$$

In simple terms, the difficulty increases if the network is finding blocks faster than every 10 minutes, and decrease otherwise. With dynamic difficulty, Nakamoto's longest chain rule was considered as a bug,[1] as it is trivial to produce long chains that have low difficulty. The rule was replaced by the strongest-PoW chain rule where competing chains are measured in terms of PoW they aggregated. As long as there is one chain with the highest PoW, this chain is chosen as the current one.

Bitcoin introduced and uses the *unspent transaction output* model. The validity of a Bitcoin transaction is verified by executing a script proving that the transaction sender is authorized to redeem unspent coins. The only exception is the first transaction in the transaction list of a block, which implements how the newly minted bitcoins and transaction fees are distributed. It is called a *coinbase transaction* and it contains the amount of bitcoins (the sum of newly minted coins and the fees derived from all the transactions) and the beneficiary (typically the creator of the block). Also, the Bitcoin scripting language offers a mechanism (`OP_RETURN`) for recording data on the blockchain, which facilitates third-party applications built-on Bitcoin.

Bitcoin proposes the simplified payment verification (SPV) protocol, that allows resource-limited clients to verify that a transaction is indeed included in a block provided only with the block header and a short transaction's inclusion proof. The key advantage of the protocol is that SPV clients can verify the existence of a transaction without downloading or storing the whole block. SPV clients are provided only with block headers and on-demand request from the network inclusion proofs of the transactions they are interested in.

In the original white paper, Nakamoto heuristically argues that the consensus protocol remains secure as long as a majority ($> 50\%$) of the participants' computing power honestly follow the rule specified by the protocol, which is compatible with their own economic incentives.

[1] `https://goo.gl/thhusi`

2.2 Bitcoin Mining Issues

Despite its popularity, Nakamoto consensus and Bitcoin suffer from multiple issues. Bitcoin mining is not always incentive-compatible. By deviating from the protocol and strategically withholding found blocks, a miner in possession of a proportion α of the total computational power may occupy more than α portion of the blocks on the blockchain, and therefore gain disproportionally higher payoffs with respect to her share [1, 11, 39]. More specifically, an attacker tries to create a private chain by keeping found blocks secret as long as the chain is in an advantageous position with one or more blocks more than the public branch. She releases her private chain only when the public chain has almost caught up, hence invalidating the public branch and all the efforts made by the honest miners. This kind of attack, called *selfish mining*, can be more efficient when a well-connected selfish miner's computational power exceeds a certain threshold (around more than 30%). Thus, selfish mining does not pay off if the mining power is sufficiently decentralized.

Unfortunately, the miners have an impulse to centralize their computing resources due to Bitcoin's rewarding scheme. In Bitcoin, rewarding is a zero-sum game and only the lucky miner who manages to get her block accepted receives the reward, while others who indeed contributed computational resources to produce the PoW are completely invisible and ignored. Increasing mining competition leads to an extremely high variance of the payoffs of a miner with a limited computational power. A solo miner may need to wait months or years to receive any reward at all. As a consequence, miners are motivated to group their resources and form mining pools, that divide work among pool participants and share the rewards according to their contributions. As of November 2018, only five largest pools account for more than 65% of the mining power of the whole Bitcoin network.[2] Such mining pools not only undermine the decentralization property of the system but also raise various in-pool or cross-pool security issues [5, 9, 22, 37].

Another seemingly harmless characteristic of Bitcoin is its finite monetary supply. However, researchers in their recent work [4, 27, 47] investigate the system dynamics when incentives coming from transaction fees are non-negligible compared with block rewards (in one extreme case the incentives come only from fees). They provide analysis and evidence, indicating an undesired system degradation due to the rational and self-interested participants. Firstly, such a system incentivizes large miner coalitions, increasing the system centralization even more. Secondly, it leads to a mining gap where miners would avoid mining when the available fees are insufficient. Even worse, rational miners tend to mine on chains that do not include available transactions (and their fees), rather than following the block selection rule specified by the protocol, resulting in a backlog of transactions. Finally, in the sole transaction fee regime, selfish mining attacks are efficient for miners with arbitrarily low mining power, regardless of their network connection qualities. These results suggest that making the block reward permanent and accepting the monetary inflation may be a wise design choice to ensure the stability of the cryptocurrency in the long run.

Moreover, the chain selection rule (i.e., the strongest chain is accepted), together with the network delay, occasionally lead to forks, where two or more blocks pointing to the same block are created around the same time, causing the participants to have different views of the current system state. Such conflicting views will eventually be resolved since with a high probability one branch will finally beat the others (then the blocks from the "losing" chain become *stale blocks*). The process of fork resolution is quite slow, as blocks have the same PoW weight and they arrive in 10-minutes intervals (on average).

Finally, the freshness properties provided by Bitcoin are questionable. By design, the Bitcoin blockchain preserves the order of blocks and transactions, however, the accurate estimation of time of these events is challenging [43], despite the fact that each block has an associated timestamp. A block's timestamp is accepted if a) it is greater than the median timestamp of the previous eleven blocks, and b) it is less than the network time plus two hours.[3] This gives significant room for manipulation — in theory, a timestamp can differ in hours from the actual time since it is largely determined by a single block creator. In fact, as time cannot be accurately determined from the timestamps, the capabilities of the Bitcoin protocol as a timestamping service are limited, which may lead to severe attacks by itself [3, 17].

2.3 Requirements

For the purpose of revising a consensus protocol of PoW blockchains in a secure, well-incentivized, and seamless way, we define the following respective requirements:

- **Security** – the scheme should improve the security of Nakamoto consensus by mitigating known attack vectors and preventing new ones. In essence, the scheme should be incentive-compatible, such that miners benefit from following the consensus rules and have no gain from violating them.
- **Reward Variance** – another objective is to minimize the variance in rewards. This requirement is crucial for decentralization since a high reward variance is the main motivation of individual miners to join centralized mining pools. Centralization is undesirable as large-enough mining pools can attack the Bitcoin protocol.
- **Chain Quality** – the scheme should provide a high chain quality, which usually is described using the two following properties.

[2] `https://btc.com/stats/pool?pool_mode=month`

[3] `https://en.bitcoin.it/wiki/Block_timestamp`

  - **Mining Power Utilization** – the ratio between the mining power on the main chain and the mining power of the entire blockchain network. This property describes the performance of mining and its ideal value is 1, which denotes that all mining power of the system contributes to the "official" or "canonical" chain. A high mining power utilization implies a low stale block rate.
  - **Fairness** – the protocol should be fair, i.e., a miner should earn rewards proportionally to the resources invested by her in mining. We denote a miner with α of the global mining power as an *α-strong miner*.
- **Efficiency and Practicality** – the scheme should not introduce any significant computational, storage, or bandwidth overheads. This is especially important since Bitcoin works as a replicated state machine, therefore all full nodes replicate data and the validation process. In particular, the block validation time, its size, and overheads of SPV clients should be at least similar as today. Moreover, the protocol should not introduce any assumptions that would be misaligned with Bitcoin's spirit and perceived as unacceptable by the community. In particular, the scheme should not introduce any trusted parties and should not assume strong synchronization of nodes (like global and reliable timestamps).

3 High-level Overview

3.1 Design Rationale

Our first observation is that Bitcoin mining is not transparent. It is difficult to quickly estimate the computing power of the different participants, because the only indicator is the found blocks. After all, blocks arrive with a low frequency, and each block is equal in terms of its implied computational power. Consequently, the only way of resolving forks is to wait for a stronger chain to emerge, which can be a time-consuming process. A related issue is block-withholding-like attacks (e.g., selfish mining) which are based on the observation that sometimes it is profitable for an attacker to deviate from the protocol by postponing the announcement of new solutions. We see transparency as a helpful property also in this context. Ideally, non-visible (hidden) solutions should be penalized, however, in practice it is challenging to detect and prove that a solution was hidden. We observe that an alternative way of mitigating these attacks would be to promote visible solutions, such that with more computing power aggregated around them they get stronger. This would incentivize miners to publish their solutions immediately, since keeping it secret may be too risky as other miners could strengthen a competing potential (future) solution over time. Finally, supported by recent research results [4, 11, 27, 39, 47], we envision that redesigning the Bitcoin reward scheme is unavoidable to keep the system sustainable and more secure. Beside the deflation issues (see Section 2.2), the reward scheme in Bitcoin is a zero-sum game rewarding only lucky miners and ignoring all effort of other participants. That causes fierce competition between miners and a high reward variance, which stimulates miners to collaborate, but within mining pools, introducing more risk to the system. We aim to design a system where miners can benefit from collaboration but without introducing centralization risks.

3.2 Overview

Motivated by these observations, we see weak puzzle solutions, currently invisible and "wasted" in Bitcoin, as a promising direction. Miners exchanging them could make the protocol more transparent as announcing them could reflect the current distribution of computational efforts on the network. Furthermore, if included in consensus rules, they could give blocks a better granularity in terms of PoW, and incentivize miners to collaborate. In our scheme, miners solve a puzzle as today but in addition to publishing solutions, they exchange weak solutions too (i.e., almost-solved puzzles). The lucky miner publishes her solution that embeds gathered weak solutions (pointing to the same previous block) of other miners. Such a published block better reflects the aggregated PoW of a block, which in the case of a fork can indicate that more mining power is focused on a given branch (i.e., actually it proves that more computing power "believes" that the given branch is correct). Another crucial change is to redesign the Bitcoin reward system, such that the finders of weak solutions are also rewarded. Following lessons learned from mining pool attacks, instead of sharing rewards among miners, our scheme rewards weak solutions proportionally to their PoW contributed to a given block and all rewards are independent of other solutions of the block. (Note, that this change requires a Bitcoin *hard fork*.)

There are a few intuitions behind these design choices. First, a selfish miner finding a new block takes a high risk by keeping this block secret. This is because blocks have a better granularity due to honest miners exchanging partial solutions and strengthening their prospective block, which in the case of a fork would be stronger than the older block kept secret (i.e., the block of the selfish miner). Secondly, miners are actually incentivized to collaborate by a) exchanging their weak solutions, and b) by appending weak solutions submitted by other miners. For the former case, miners are rewarded whenever their solutions are appended, hence keeping them secret can be unprofitable for them. For the latter case, a miner appending weak solutions of others only increases the strength of her potential block, and moreover, appending these solutions does not negatively influence the miner's potential reward. Finally, our approach comes with another benefit. Proportional rewarding of weak solutions

decreases the reward variance, thus miners do not have to join large mining pools in order to stabilize their revenue. This could lead to a higher decentralization of mining power on the network.

In the following sections, we describe details of our system, show its analysis, and report on its implementation.

4 StrongChain Details

4.1 Mining

As in Bitcoin, in StrongChain miners authenticate transactions by collecting them into blocks whose headers are protected by a certain amount of PoW. A simplified description of a block mining procedure in StrongChain is presented as the *mineBlock()* function in Algorithm 1. Namely, every miner tries to solve a PoW puzzle by computing the hash function over a newly created header. The header is constantly being changed by modifying its nonce field,[4] until a valid hash value is found. Whenever a miner finds a header hdr whose hash value $h = H(hdr)$ is smaller than the *strong target* T_s, i.e., a h that satisfies the following:

$$h < T_s,$$

then the corresponding block is announced to the network and becomes, with all its transactions and metadata, part of the blockchain. We refer to headers of included blocks as *strong headers*.

One of the main differences with Bitcoin is that our mining protocol handles also headers whose hash values do not meet the strong target T_s, but still are low enough to prove a significant PoW. We call such a header a *weak header* and its hash value h has to satisfy the following:

$$T_s \leq h < T_w, \tag{2}$$

where $T_w > T_s$ and T_w is called the *weak target*.

Whenever a miner finds such a block header, she adds it to her local list of weak headers (i.e., *weakHdrsTmp*) and she propagates the header among all miners. Then every miner that receives this information first validates it (see *onRecvWeakHdr()*) by checking whether

- the header points to the last strong header,
- its other fields are correct (see Section 4.2),
- and Equation 2 is satisfied.

Afterward, miners append the header to their lists of weak headers. We do not limit the number of weak headers appended, although this number is correlated with the T_w/T_s ratio (see Section 5).

Finally, miners continue the mining process in order to find a strong header. In this process, a miner keeps creating candidate headers by computing hash values and checking whether the strong target is met. Every candidate header

[4] In fact, other fields can be modified too if needed.

Algorithm 1: Pseudocode of StrongChain functions.

```
function mineBlock()
    weakHdrsTmp ← ∅;
    for nonce ∈ {0,1,2,...} do
        hdr ← createHeader(nonce);
        /* check if the header meets the strong target */
        h_tmp ← H(hdr);
        if h_tmp < T_s then
            B ← createBlock(hdr, weakHdrsTmp, Txs);
            broadcast(B);
            return; /* signal to mine with the new block */
        /* check if the header meets the weak target */
        if h_tmp < T_w then
            weakHdrsTmp.add(hdr);
            broadcast(hdr);
function onRecvWeakHdr(hdr)
    h_w ← H(hdr);
    assert(T_s ≤ h_w < T_w and validHeader(hdr));
    assert(hdr.PrevHash == H(lastBlock.hdr)) ;
    weakHdrsTmp.add(hdr);
function rewardBlock(B)
    /* reward block finder with R */
    reward(B.hdr.Coinbase, R + B.TxFees);
    w ← γ * T_s/T_w; /* reward weak headers proportionally */
    for hdr ∈ B.weakHdrSet do
        reward(hdr.Coinbase, w * c * R);
function validateBlock(B)
    assert(H(B.hdr) < T_s and validHeader(B.hdr));
    assert(B.hdr.PrevHash == H(lastBlock.hdr)) ;
    assert(validTransactions(B));
    for hdr ∈ B.weakHdrSet do
        assert(T_s ≤ H(hdr) < T_w and validHeader(hdr));
        assert(hdr.PrevHash == H(lastBlock.hdr));
function chainPoW(chain)
    sum ← 0;
    for B ∈ chain do
        /* for each block compute its aggregated PoW */
        T_s ← B.hdr.Target;
        sum ← sum + T_max/T_s;
        for hdr ∈ B.weakHdrSet do
            sum ← sum + T_max/T_w;
    return sum;
function getTimestamp(B)
    sumT ← B.hdr.Timestamp;
    sumW ← 1.0;
    /* average timestamp by the aggregated PoW */
    w ← T_s/T_w;
    for hdr ∈ B.weakHdrSet do
        sumT ← sumT + w * hdr.Timestamp;
        sumW ← sumW + w;
    return sumT/sumW;
```

"protects" all collected weak headers (note that all of these weak headers point to the same previous strong header).

In order to keep the number of found weak headers close to a constant value, StrongChain adjusts the difficulty T_w of weak headers every 2016 blocks immediately following the adjustment of the difficulty T_s of the strong headers according to Equation 1, such that the ratio T_w/T_s is kept at a constant (we discuss its value in Section 5).

4.2 Block Layout and Validation

A block in our scheme consists of transactions, a list of weak headers, and a strong header that authenticates these transactions and weak headers. Strong and weak headers in our system inherit the fields from Bitcoin headers and additionally enrich it by a new field. A block header consists of the following fields:

PrevHash: is a hash of the previous block header,
Target: is the value encoding the current target defining the difficulty of finding new blocks,
Nonce: is a nonce, used to generate PoW,
Timestamp: is a Unix timestamp,
TxRoot: is the root of the Merkle tree [24] aggregating all transactions of the block, and
Coinbase: represents an address of the miner that will receive a reward.

As our protocol rewards finders of weak headers (see details in Section 4.4), every weak header has to be accompanied with the information necessary to identify its finder. Otherwise, a finder of a strong block could maliciously claim that some (or all) weak headers were found by her and get rewards for them. For this purpose and for efficiency, we introduced a new 20B-long header field named *Coinbase*. With the introduction of this field, StrongChain headers are 100B long. But on the other hand, there is no longer any need for Bitcoin coinbase transactions (see Section 2.1), as all rewards are determined from headers.

In our scheme, weak headers are exchanged among nodes as part of a block, hence it is necessary to protect the integrity of all weak headers associated with the block. To realize it, we introduce a special transaction, called a *binding transaction*, which contains a hash value computed over the weak headers. This transaction is the first transaction of each block and it protects the collected weak headers. Whenever a strong header is found, it is announced together with all its transactions and collected weak headers, therefore, this field protects all associated weak headers. To encode this field we utilize the `OP_RETURN` operation as follows:

$$\texttt{OP_RETURN} \quad H(hdr_0 \| hdr_1 \| ... \| hdr_n), \tag{3}$$

where hdr_i is a weak header pointing to the previous strong header. Since weak headers have redundant fields (the *PrevHash*, *Target*, and *Version* fields have the same values as the strong header), we propose to save bandwidth and storage by not including these fields into the data of a block. This modification reduces the size of a weak header from 100B to 60B only, which is especially important for SPV clients who keep downloading new block headers.

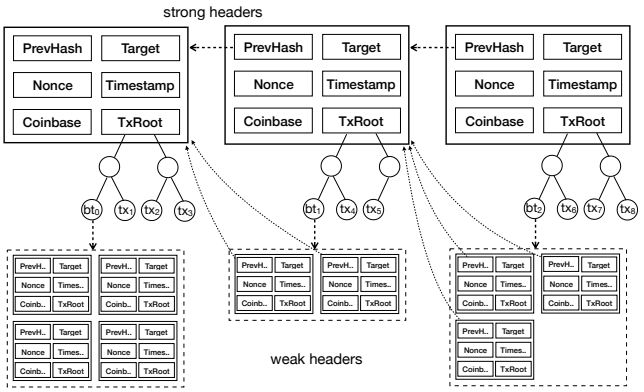


Figure 1: An example of a blockchain fragment with strong headers, weak headers, and binding and regular transactions.

With our approach, a newly mined and announced block can encompass multiple weak headers. Weak headers, in contrast to strong headers, are not used to authenticate transactions, and they are even stored and exchanged *without* their corresponding transactions. Instead, the main purpose of including weak headers it to contribute and reflect the aggregated mining power concentrated on a given branch of the blockchain. We present a fragment of a blockchain of StrongChain in Figure 1. As depicted in the figure, each block contains a single strong header, transactions, and a set of weak headers aggregated via a binding transaction.

On receiving a new block, miners validate the block by checking the following (see *validateBlock()* in Algorithm 1):

1. The strong header is protected by the PoW and points to the previous strong header.
2. Header fields have correct values (i.e., the version, target, and timestamp are set correctly).
3. All included transactions are correct and protected by the strong header. This check also includes checking that all weak headers collected are protected by a binding transaction included in the block.
4. All included weak headers are correct: a) they meet the targets as specified in Equation 2, b) their *PrevHash* fields point to the previous strong header, and c) their version, targets, and timestamps have correct values.

If the validation is successful, the block is accepted as part of the blockchain.

4.3 Forks

One of the main advantages of our approach is that blocks reflect their aggregated mining power more precisely. Each block beside its strong header contains multiple weak head-

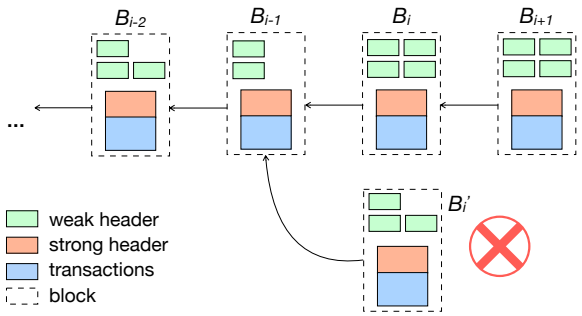


Figure 2: An example of a forked blockchain in StrongChain.

ers that contribute to the block's PoW. In the case of a fork, our scheme relies on the strongest chain rule, however, the PoW is computed differently than in Bitcoin. For every chain its PoW is calculated as presented by the *chainPoW()* procedure in Algorithm 1. Every chain is parsed and for each of its blocks the PoW is calculated by adding:

1. the PoW of the strong header, computed as T_{max}/T_s, where T_{max} is the maximum target value, and
2. the accumulated PoW of all associated weak headers, counting each weak header equally as T_{max}/T_w.

Then the chain's PoW is expressed as just the sum of all its blocks' PoW. Such an aggregated chain's PoW is compared with the competing chain(s). The chain with the largest aggregated PoW is determined as the current one. As difficulty in our protocol changes over time, the strong target T_s and PoW of weak headers are relative to the maximum target value T_{max}. We assume that nodes of the network check whether every difficulty window is computed correctly (we skipped this check in our algorithms for easy description).

Including and empowering weak headers in our protocol moves away from Bitcoin's "binary" granularity and gives blocks better expression of the PoW they convey. An example is presented in Figure 2. For instance, nodes having the blocks B_i and B_i' can immediately decide to follow the block B_i as it has more weak headers associated, thus it has accumulated more PoW than the block B_i'.

An exception to this rule is when miners solve conflicts. Namely, on receiving a new block, miners run the algorithm as presented, however, they also take into consideration PoW contributions of known weak headers that point to the last blocks. For instance, for a one-block-long fork within the same difficulty window, if a block B includes l weak headers and a miner knows of k weak headers pointing to B, then that miner will select B over any competing block B' that includes l' weak and has k' known weak headers pointing to it if $l+k > l'+k'$. Note that this rule incentivizes miners to propagate their solutions as quickly as possible as competing blocks become "stronger" over time.

4.4 Rewarding Scheme

The rewards distribution is another crucial aspect of StrongChain and it is presented by the *rewardBlock()* procedure from Algorithm 1. The miner that found the strong header receives the full reward R. Moreover, in contrast to Bitcoin, where only the "lucky" miner is paid the full reward, in our scheme all miners that have contributed to the block's PoW (i.e., whose weak headers are included) are paid by commensurate rewards to the provided PoW. A weak header finder receive a fraction of R, i.e., $\gamma * c * R * T_s/T_w$, as a reward for its corresponding solution contributing to the total PoW of a particular branch, where the γ parameter influences the relative impact of weak header rewards and c is just a scaling constant (we discuss their potential values and implications in Section 5). Moreover, we do not limit weak header rewards and miners can get multiple rewards for their weak headers within a single block. Similar reward mechanisms are present in today's mining pools (see Section 8), but unlike them, weak header rewards in StrongChain are independent of each other. Therefore, the reward scheme is not a zero-sum game and miners cannot increase their own rewards by dropping weak headers of others (actually, as we discuss in Section 5, they can only lose since their potential solutions would have less PoW without others' weak headers). Furthermore, weak header rewards decrease significantly the mining variance as miners can get steady revenue, making the system more decentralized and collaborative.

As mentioned before, the number of weak headers of a block is unlimited, they are rewarded independently (i.e., do not share any reward), and all block rewards in our system are proportional to the PoW contributed. In such a setting, a mechanism incentivizing miners to terminate a block creation is needed (without such a mechanism, miners could keep creating huge blocks with weak headers only). In order to achieve this, StrongChain always attributes block transaction fees ($B.TxFees$) to the finder of the strong header (who also receives the full reward R).

Note that in our rewarding scheme, the amount of newly minted coins is always at least R, and consequently, unlike Bitcoin or Ethereum [48], the total supply of the currency in our protocol is not upper-bounded. This design decision is made in accordance with recent results on the long-term instability of deflationary cryptocurrencies [4, 27, 47].

4.5 Timestamps

In StrongChain, we follow the Bitcoin rules on constraining timestamps (see Section 2.1), however, we redefine how block timestamps are interpreted. Instead of solely relying on a timestamp put by the miner who mined the block, block timestamps in our system are derived from the strong header and all weak headers included in the corresponding block. The algorithm to derive a block's timestamp is presented as

getTimestamp() in Algorithm 1. A block's timestamp is determined as a weighted average timestamp over the strong header's timestamp and all timestamps of the weak headers included in the block. The strong header's timestamp has a weight of 1, while weights of weak header timestamps are determined as their PoW contributed (namely, a weak header's timestamp has a weight of the ratio between the strong target and the weak target). Therefore, the timestamp value is adjusted proportionally to the mining power associated with a given block. That change reflects an average time of the block creation and mitigates miners that intentionally or misconfigured put incorrect timestamps into the blockchain. We show the effectiveness of this approach in Section 5.5.

4.6 SPV Clients

Our protocol supports light SPV clients. With every new block, an SPV client is updated with the following information:

$$hdr, hdr_0, hdr_1, ..., hdr_n, BTproof, \tag{4}$$

where *hdr* is a strong header, hdr_i are associated weak headers, and *BTproof* is an inclusion proof of a binding transaction that contains a hash over the weak headers (see Equation 3). Note that headers contain redundant fields, thus as described in Section 4.2, they can be provided to SPV clients efficiently.

With this data, the client verifies fields of all headers, computes the PoW of the block (analogous, as in *chainPoW()* from Algorithm 1), and validates the *BTproof* proof to check whether all weak headers are correct, and whether the transaction is part of the blockchain (the proof is validated against *TxRoot* of *hdr*). Afterward, the client saves the strong header *hdr* and its computed PoW, while other messages (the weak headers and the proof) can be dropped.

5 Analysis

In this section, we evaluate the requirements discussed in Section 2.3. We start with analyzing StrongChain's efficiency and practicality. Next, we study how our design helps with reward variance, chain quality, and security.

5.1 Efficiency and Practicality

For the efficiency, it is important to consider the main source of additional load on the bandwidth, storage, and processing power of the nodes: the weak headers. Hence, in the following section we analyze the probability distribution of the number of weak headers. Next, we discuss the value of the impact of the parametrization on the average block rewards.

5.1.1 Number of Weak Headers

In Bitcoin, we assume that hashes are drawn randomly between 0 and $T_{max} = 2^{256} - 1$. Hence, a single hash being smaller than T_w is a *Bernoulli trial* with parameter $p_w = T_w/2^{256}$. The number of hashes tried until a weak header is found is therefore *geometrically* distributed, and the time in seconds between two weak headers is approximately *exponentially* distributed with rate ηp_w, where η is the total hash rate per second and p_w is chosen such that $\eta p_w \approx 1/600$. When a weak header is found, it is also a strong block with probability p_s/p_w (where $p_s = T_s/2^{256}$), which is again a Bernoulli trial. Hence, the probability distribution of the number of weak headers found between two strong blocks is that of the number of trials before the first successful trial — as such, it also follows a geometric distribution, but with mean $p_w/p_s - 1$.[5] For example, for $T_w/T_s = 2^{10}$ this means that the average number of weak headers per block equals 1023. With 60 bytes per weak header (see Section 4.2) and 1MB per Bitcoin block, this would mean that the load increases by little over 6% on average with a small computational overhead introduced (see details in Section 7). The probability of having more than 16667 headers (or 1MB) in a block would equal.[6]

$$\left(1 - \frac{p_s}{p_w}\right)^{16668} = \left(1 - 2^{-10}\right)^{16668} \approx 8.4603 \cdot 10^{-8}.$$

Since around 51,000 Bitcoin blocks are found per year, this is expected to happen roughly once every 230 years.

5.1.2 Total Rewards

To ease the comparison to the Bitcoin protocol, we can enforce the same average mining reward per block (currently 12.5 BTC). Let R denote Bitcoin's mining reward. Since we reward weak headers as well as strong blocks, we need to scale all mining rewards by a constant c to ensure that the total reward remains unchanged — this is done in the *rewardBlock* function in Algorithm 1. As argued previously, we reward all weak headers equally by $\gamma R T_s/T_w$. Since the average number of weak headers per strong block is $T_w/T_s - 1$, this means that the expected total reward per block (i.e., strong block and weak header rewards) equals $cR + cR\gamma T_s/T_w \cdot (T_w/T_s - 1)$. Hence, we find that

$$c = \frac{1}{1 + \gamma(T_w/T_s - 1)T_s/T_w},$$

[5] Another way to reach this conclusion is as follows: the number of weak headers found in a fixed time interval is Poisson distributed, and it can be shown that the number of Poisson arrivals in an interval with exponentially distributed length is geometrically distributed.

[6] For an actual block implementation, we advice to introduce separate spaces for weak headers and transactions. With such a design, miners do not have incentives and trade-offs between including more transactions instead of weak headers.

which for large values of T_w/T_s is close to $1/(1+\gamma)$. This means that if $\gamma = 1$, the strong block and weak header rewards contribute almost equally to a miner's total reward.

5.2 Reward Variance of Solo Mining

The tendency towards centralization in Bitcoin caused by powerful mining pools can largely be attributed to the high reward variance of solo mining [15, 37]. Therefore, keeping the reward variance of a solo miner at a low level is a central design goal.

Let $\mathbf{R}^{BC}$ and $\mathbf{R}^{SC}$ be the random variables representing the per-block rewards for an α-strong solo miner in Bitcoin and in StrongChain, respectively. For any given strong block in both protocols, we define the random variable $\mathbf{I}$ as follows:

$$\mathbf{I} = \begin{cases} 1 & \text{the block is mined by the solo miner,} \\ 0 & \text{otherwise.} \end{cases}$$

By definition, $\mathbf{I}$ has a Bernoulli distribution, which means that $\mathbb{E}(\mathbf{I}) = \alpha$ and $\mathrm{Var}(\mathbf{I}) = \alpha(1-\alpha)$, where $\mathbb{E}$ and Var are the mean and variance of a random variable respectively. The following technical lemma will aid our analysis of the reward variances of solo miners:

Lemma 1. *Let $X_1, X_2, \ldots$ be independent and identically distributed random variables. Let N be defined on $\{0,1,\ldots\}$ and independent of $X_1, X_2, \ldots$. Let N and all X_i have finite mean and variance. Then*

$$\mathrm{Var}\left(\sum_{i=1}^{N} X_i\right) = \mathbb{E}(N)\mathrm{Var}(X) + \mathrm{Var}(N)(\mathbb{E}(X))^2.$$

Proof. See [7]. □

Reward Variance of Solo Mining in Bitcoin. Bitcoin rewards the miner of a block creator with the fixed block reward R and the variable (total) mining fees, which we denote by the random variable $\mathbf{F}$. Therefore, we have

$$\mathbf{R}^{BC} = \mathbf{I}(R + \mathbf{F}),$$

which implies that

$$\mathrm{Var}(\mathbf{R}^{BC}) = R^2\mathrm{Var}(\mathbf{I}) + \mathrm{Var}(\mathbf{IF}). \tag{5}$$

Since $\mathbf{IF} = \sum_{i=0}^{\mathbf{I}} \mathbf{F}$, we can use Lemma 1 (substituting $\mathbf{I}$ for N and $\mathbf{F}$ for X) to obtain

$$\mathrm{Var}(\mathbf{IF}) = \mathbb{E}(\mathbf{I})\mathrm{Var}(\mathbf{F}) + \mathrm{Var}(\mathbf{I})\mathbb{E}^2(\mathbf{F}). \tag{6}$$

Combining (5) and (6) gives

$$\begin{aligned} \mathrm{Var}(\mathbf{R}^{BC}) &= \mathbb{E}(\mathbf{I})\mathrm{Var}(\mathbf{F}) + \mathrm{Var}(\mathbf{I})\left(\mathbb{E}^2(\mathbf{F}) + R^2\right) \\ &= \alpha\mathrm{Var}(\mathbf{F}) + \alpha(1-\alpha)\left(\mathbb{E}^2(\mathbf{F}) + R^2\right). \end{aligned} \tag{7}$$

When the fees are small compared to the mining reward, this simplifies to $\alpha(1-\alpha)R^2$. By comparison, in [37] the variance of the block rewards (without fees) earned by a solo miner across a time period of t seconds is studied, and found to equal $\alpha R^2 t/600$.[7] The same quantity can be obtained by using (7), Lemma 1, and the total number of strong blocks found (by any miner) after t seconds of mining (which has a Poisson distribution with mean $t/600$).

Reward Variance of Solo Mining in StrongChain. For $\mathbf{R}^{SC}$, we assume that the solo miner has $\mathbf{N}$ weak headers included in the strong block, and that she obtains $c\gamma RT_s/T_w$ reward per weak header. Then the variance equals

$$\mathbf{R}^{SC} = \mathbf{I}(cR + \mathbf{F}) + c\gamma RT_s/T_w\mathbf{N},$$

where c is the scaling constant derived in Section 5.1.2. Hence, by applying Lemma 1, we compute the variance of $\mathbf{R}^{SC}$ as

$$\begin{aligned} \mathrm{Var}(\mathbf{R}^{SC}) = (cR)^2\mathrm{Var}(\mathbf{I}) + \mathrm{Var}(\mathbf{IF}) \\ + (c\gamma RT_s/T_w)^2\mathrm{Var}(\mathbf{N}). \end{aligned} \tag{8}$$

The first term, which represents the variance of the strong block rewards, is similar to Bitcoin but multiplied by c^2. If we choose $T_w/T_s = 1024$ and $\gamma = 10$ (this choice is motivated later in this section), c^2 roughly equals 0.0083, which is quite small. Hence, the strong block rewards have a much smaller impact on the reward variance in our setting than in Bitcoin. The second term, which represents the variance of the fees, is precisely the same as for Bitcoin. The third term represents the variance of the weak header rewards, which in turn completely depends on $\mathrm{Var}(\mathbf{N})$.

To evaluate $\mathrm{Var}(\mathbf{N})$, we again use Lemma 1: let, for any weak header, $\mathbf{J}$ equal 1 if it is found by the solo miner, and 0 otherwise. Also, let $\mathbf{L}$ be the total number of weak headers found in the block, so including those not found by the solo miner. Then $\mathbf{N}$ is the sum of $\mathbf{L}$ instances of $\mathbf{J}$, where $\mathbf{J}$ has a Bernoulli distribution with success probability α (and therefore $\mathbb{E}(\mathbf{J}) = \alpha$ and $\mathrm{Var}(\mathbf{J}) = \alpha(1-\alpha)$), and $\mathbf{L}$ has a geometric distribution with success probability T_s/T_w (and therefore $\mathbb{E}(\mathbf{L}) = T_w/T_s - 1$ and $\mathrm{Var}(\mathbf{L}) = (T_w/T_s)^2 - T_w/T_s$. By substituting this into (8), we obtain:

$$\begin{aligned} \mathrm{Var}(\mathbf{N}) &= \mathbb{E}(\mathbf{L})\mathrm{Var}(\mathbf{J}) + \mathrm{Var}(\mathbf{L})(\mathbb{E}(\mathbf{J}))^2 \\ &= (T_w/T_s - 1)\alpha(1-\alpha) \\ &\quad + ((T_w/T_s)^2 - T_w/T_s)\alpha^2 \end{aligned} \tag{9}$$

Substituting (9) for $\mathrm{Var}(\mathbf{N})$ and $\alpha(1-\alpha)$ for $\mathrm{Var}(\mathbf{I})$ into (8) then yields an expression that can be evaluated for different values of T_w/T_s, γ, and α, as we discuss in the following.

[7] In particular, it is found to be $htR^2/(2^{32}D)$, where $h = \alpha\eta$ and $\eta/(2^{32}D) \approx 1/600$.

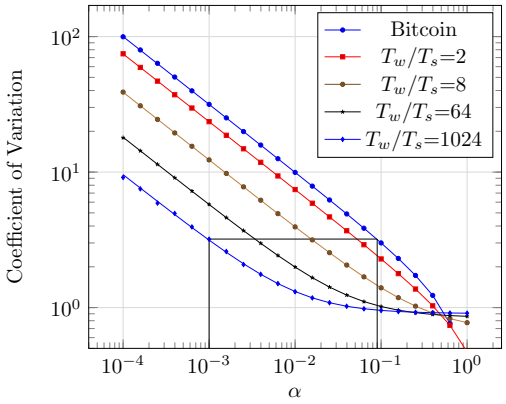


Figure 3: *Coefficients of variation* for the total rewards of α-strong miners for different strong/weak header difficulty ratios ($T_w/T_s = 1$ corresponds to Bitcoin). The lines indicate the exact results obtained using our analysis, whereas the markers indicate simulation results. We used $\gamma = \log_2(T_w/T_s)$. The black lines indicate that for $T_w/T_s = 1024$, a 0.1%-strong miner has a coefficient of variation that is comparable to a 9%-strong miner's in Bitcoin.

Comparison The difference between between (7) and (8) in practice is illustrated in Figure 3. This is done by comparing for a range of different values of α the block rewards' *coefficient of variation*, which is the ratio of the square root of the variance to the mean.

To empirically validate the results, we have also implemented a simulator in Java that can evaluate Bitcoin as well as StrongChain. We use two nodes, one of which controls a share α of the hash rate, and another controls a share $1-\alpha$. The nodes can broadcast information about blocks, although we abstract away from most of the other network behavior. We do not consider transactions (i.e., we mine empty blocks), and we use a simplified model for the propagation delays: delays are drawn from a Weibull distribution with shape parameter 0.6 [31], although for Figure 3 the mean was chosen to be negligible (more realistic values are chosen for Table 1).

The black lines in Figure 3 demonstrate that when $T_w/T_s = 1024$, a miner with share 0.1% of the mining power has the same coefficient of reward variation as a miner with stake 9% in Bitcoin. Also note that for $T_w/T_s = 1024$ and $\alpha \geq 1\%$, the coefficient of variation does not substantially decrease anymore, because nearly all of the reward variance is due to the number of weak headers. Hence, there would be fewer reasons for miners in our system to join large and cooperative mining pools, which has a positive effect on the decentralization of the system.

5.3 Chain Quality

One measure for the 'quality' of a blockchain is the stale rate of blocks [16], i.e., the percentage of blocks that appear during forks and do not make it onto the main chain. This is closely related to the notion of mining power utilization [10], which is the fraction of mining power used for non-stale blocks. In StrongChain, the stale rate of strong blocks may increase due to high latency. After all, while a new block is being propagated through the network, weak headers that strengthen the previous block that are found will be included by miners in their PoW calculation. As a result, some miners may refuse to switch to the new block when it arrives. However, the probability of this happening is very low: because each weak header only contributes T_s/T_w to the difficulty of a block, it would take on average 10 minutes to find enough weak headers to outweigh a block. As we can see in Table 1, the effect on the stale rate is negligible even for very high network latencies (i.e., 53 seconds). We also emphasize that the strong block stale rate is less important in our setting, as the losing miner still would benefit from her weak headers appended to the winning block.

Regarding the fairness, defined as the ratio between the observed share of the rewards (we simulate using one 10%-strong miner and a 90%-strong one) and the share of the mining power, we see that StrongChain does slightly worse than Bitcoin for high network latencies. The most likely cause is that due to the delay in the network, the 10%-strong miner keeps mining on a chain that has already been extended for longer than necessary. This gives the miner a slight disadvantage compared to the 90%-strong miner.

5.4 Security

One of the main advantages of StrongChain is the added robustness to selfish mining strategies akin to those discussed in [11] and [39]. In selfish mining, attackers aim to increase their share of the earned rewards by tricking other nodes into mining on top of a block that is unlikely to make it onto the main chain, thus wasting their mining power. This may come at a short-term cost, as the chance of the attacker's blocks going stale is increased — however, the difficulty rescale that occurs every 2016 blocks means that if the losses to the honest nodes are structural, the difficulty will go down and the gains of the attacker will increase.

In the following, we will consider the selfish mining strategy of [11],[8] described as follows:

- The attacker does not propagate a newly found block until she finds at least a second block on top of it, and then only if the difference in difficulty between her chain and the strongest known alternative chain is between zero and R.
- The attacker adopts the strongest known alternative chain if its difficulty is at least greater than her own by R.

[8] The 'stubborn mining' strategy of [39] offers mild improvements over [11] for powerful miners, but the comparison with StrongChain is similar. We have also modeled StrongChain using a Markov decision process, in a way that is similar to the recently proposed framework of [51]. Due to the state space explosion problem, we could only investigate the protocol with a small number of expected weak headers, but we have not found any strategies noticeably that are better than those presented.

	Latency	Bitcoin	StrongChain						
			$T_w/T_s = 2$	$T_w/T_s = 64$			$T_w/T_s = 1024$		
			$\gamma = 1$	$\gamma = 1$	$\gamma = 7$	$\gamma = 63$	$\gamma = 1$	$\gamma = 10$	$\gamma = 1023$
strong stale rate	low	.0023	.0025	.0021	.0026	.0028	.0023	.0025	.0019
	medium	.0073	.0082	.0087	.0077	.0078	.0084	.0067	.0081
	high	.0243	.0297	.0242	.0263	.0247	.0274	.0249	.0263
weak stale rate	low	—	.0043	.0047	.0049	.0046	.0049	.0047	.0047
	medium	—	.0142	.0151	.0154	.0149	.0145	.0147	.0149
	high	—	.0400	.0459	.0474	.0452	.0469	.0455	.0463
fairness	low	.9966	.9814	.9749	.9747	.9838	.9645	.9809	.9812
	medium	.9276	.9384	.9570	.9360	.9364	.9329	.9400	.9385
	high	.7951	.7640	.7978	.7820	.7757	.7756	.7766	.7775

Table 1: For several different protocols, the strong block stale rate, weak header rate, and the 'fairness' for an α-strong honest miner with $\alpha = 0.1$. Here, fairness is defined as the ratio between the observed share of the reward and the 'fair' share of the rewards (i.e, 0.1). 'Low', 'medium', and 'high' latencies refer to the mean of the delay distribution in the simulator; these are roughly 0.53 seconds, 5.3 seconds, and 53 seconds respectively. The simulations are based on a time period corresponding to roughly 20 000 blocks.

In Figure 4a, we have depicted the profitability of this selfish mining strategy for different choices of T_w/T_s. As we can see, for $T_w/T_s = 1024$ the probability of being 'ahead' after two strong blocks is so low that the strategy only begins to pay off when the attackers' mining power share is close to 45% — this is an improvement over Bitcoin, where the threshold is closer to 33%.

StrongChain does introduce new adversarial strategies based on the mining of new weak headers. Some examples include not broadcasting any newly found weak blocks ("reclusive" mining), refusing to include the weak headers of other miners ("spiteful" mining), and postponing the publication of a new strong block and wasting the weak headers found by other miners in the meantime. In the former case, the attacker risks losing their weak blocks, whereas in both of the latter two cases, the attacker risks their strong block going stale as other blocks and weak headers are found. Hence, these are not cost-free strategies. Furthermore, because the number of weak headers does not affect the difficulty rescale, the attacker's motive for increasing the stale rate of other miners' weak headers is less obvious (although in the long run, an adversarial miner could push other miners out of the market entirely, thus affecting the difficulty rescale).

In Figure 4b, we have displayed the relative payout (with respect to the total rewards) of a reclusive α-strong miner — this strategy does not pay for any $\alpha < 0.5$. In Figure 4c, we have depicted the relative payoff of a spiteful mine who does not include other miners' weak blocks unless necessary (i.e., unless others' weak blocks together contribute more than R to the difficulty, which would mean that any single block found by the spiteful miner would always go stale). For low latencies (the graphs were generated with an average latency of 0.53 seconds), the strategy is almost risk-free, and the attacker does manage to hurt other miners more than herself, leading to an increased relative payout. However, as displayed in Figure 4d, there are no absolute gains, even mild

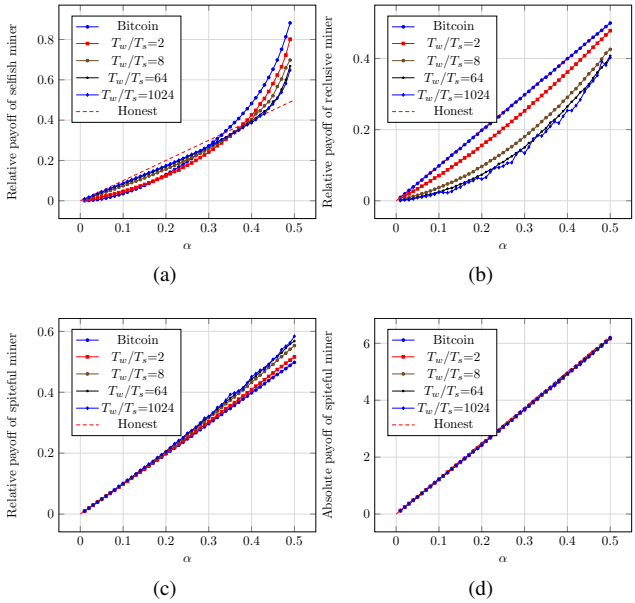
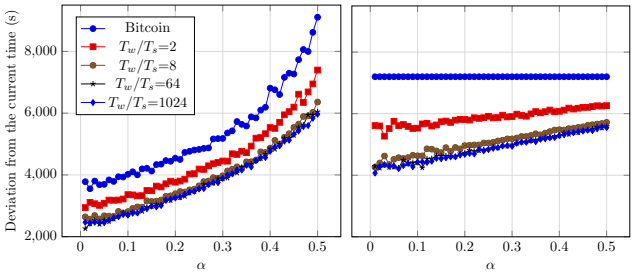


Figure 4: Payoffs of an α-strong adversarial miner for different strategies. Figure (a): relative payoff of a *selfish* miner following the strategy of [11], compared to an $(1-\alpha)$-strong honest miner. Figure (b): relative payoff of a *reclusive* miner who does not broadcast her weak blocks. Figure (c): *relative* payoff (with respect to the rewards of all miners combined) of a *spiteful* miner, who does not include other miners' weak blocks unless necessary. Figure (d): *absolute* payoff of a *spiteful* miner, with 12.5 BTC on average awarded per block. We consider Bitcoin and StrongChain with different choices of T_w/T_s, with $\gamma = \log_2(T_w/T_s)$.

losses. As mentioned earlier, the weak headers do not affect the difficulty rescale so there is no short-term incentive to engage in this behavior — additionally there is little gain in computational overhead as the attacker still needs to process her own weak headers. In Section 6.1 we will discuss protocol updates that can mitigate these strategies regardless.



(a) (b)

Figure 5: The deviation from the network time that an α-strong adversary can introduce for its mined blocks by slowing (the left graph) and accelerating (the right graph) timestamps.

5.5 More Reliable Timestamps

Finally, we conducted a series of simulations to investigate how the introduced redefinition of timestamps interpretation (see *getTimestamp()* in Algorithm 1 and Section 4.5) influences the timestamp reliability in an adversarial setting. We assume that an adversary wants to deviate blockchain timestamps by as much as possible. There are two strategies for such an attack, i.e., an adversary can either "slow down" timestamps or "accelerate" them. In the former attack, the best adversary's strategy is to use the minimum acceptable timestamp in every header created by the adversary. Namely, the adversary sets its timestamps to the median value of the last eleven blocks (a header with a lower timestamp would not be accepted by the network – see Section 2.2). As for the latter attack, the adversary can analogously bias timestamps towards the future by putting the maximum acceptable value in all her created headers. The maximum timestamp value accepted by network nodes is two hours in the future with respect to the nodes' internal clocks (any header with a higher timestamp would be rejected).

In our study, we assume that honest nodes maintain the network time which the adversary tries to deviate from. We consider the worst-case scenario, which is when the adversary, who also biases all her header timestamps, mines the strong block. We measure (over 10000 runs) how such a malicious timestamp can be mitigated by our redefinition of the block timestamps interpretation. We present the obtained results in Figure 5, and as shown in the slow-down case our protocol achieves much more precise timestamps than Bitcoin (the difference is around 2000 seconds). Similarly, when the adversary accelerates timestamps, our protocol can mitigate it effectively, adjusting the adversarial timestamps by 2000-3500 seconds towards the correct time. This effect is achieved due to the block's timestamp calculation as a weighted average of all block headers. The adversary could try to remove honest participants' weak headers in order to give a stronger weight to its malicious timestamps, but in Section 6.1 we discuss ways to mitigate it.

6 Discussion

6.1 Impact of the Parameter Choice

The results presented in Section 5 required several parameters to be fixed. First of all, we had to choose γ, which determines the relative contribution of the weak headers to the total mining rewards. Second, there is the contribution of the weak blocks to the chain difficulty, which in the *chainPoW()* function in Algorithm 1 was set to be only T_{max}/T_w. This means that the PoW of a weak header relative to a strong block's PoW — we call this the *difficulty factor* — is fixed to be T_s/T_w. In the following, we first discuss the relevant trade-offs and then motivate our choice.

When both γ and the difficulty factor are low, the impact on the reward variance of the miners (as per Figure 3) will be mild as the strong block rewards still constitute about 50% of the mining rewards. This reliance on the block rewards also means that 'spiteful' mining as discussed in Section 5.4 is disincentivized as the risk of strong blocks going stale still has a considerable impact on total rewards. However, selfish mining as proposed in [11] relies on several blocks in a row being mined in secret, and even for a low difficulty factor it becomes much harder for the attacker's chain to stay 'ahead' of the honest chain, as the latter accumulates strength from the weak headers at a faster rate. Hence, in this setting we only gain protection against selfish mining.

When γ is high but the difficulty factor is not (which is the setting of Section 5), then in addition to disincentivizing selfish mining, the reward variances become much less dependent on the irregular strong block rewards. This benefits small miners and reduces centralization, as we also discuss in Section 6.2. However, spiteful mining will have more of an impact as the possible downside (i.e., a latency-dependent increase in the strong block stale rate) will have less of an effect on the total rewards.

When both γ and the difficulty factor are high, the impact of spiteful mining is mitigated. The reason is that blocks quickly accumulate enough weak headers to outweigh a strong block, and in this case spiteful miners need to adopt the other weak blocks or risk their strong block becoming stale with certainty. The downside in this setting is that the system-wide block stale rate is increased. For example, if each weak header contributes $\gamma T_s/T_w$ to the difficulty and $\gamma = 10$, then after (on average) one minute enough weak headers are found to outweigh a strong block, and if propagation of the block takes longer than one minute then some miners will not adopt the block, increasing the likelihood of a fork.

In this paper, we have chosen the second of the three approaches — a moderately high γ, yet a minor difficulty factor. The reason is that the only downside (spiteful mining) was considered less of a concern than the other downsides (namely a low impact on reward variances and a higher block

stale rate respectively) for two reasons: a) because spiteful mining does not lead to clear gains for the attacker, and b) because it only has a large impact on other miners' profits if the attacker controls a large share of the mining power, whereas the emergence of large mining pools is exactly what StrongChain discourages. The specific value of $\gamma = 10$ for $T_w/T_s = 1024$ (or $\gamma = \log_2(T_w/T_s)$ in general) was chosen to sufficiently reduce mining reward variances, yet leaving some incentive to discourage spiteful mining.

The protocol can be further extended to disincentivize spiteful mining, e.g., by additionally awarding strong block finders a reward that is proportional to the number of weak headers included. This would make StrongChain more similar to Ethereum, where stale block ('uncle') rewards are paid both to the miner of a stale block and the miner of the successful block that included it (see Section 8 for additional discussion of Ethereum's protocol). However, we leave such modifications and their consequences as future work.

6.2 StrongChain and Centralized Mining

Decentralized mining pools aim to reduce variance while providing benefits for the system (i.e., trust minimization for pools, and a higher number of validating nodes). However, mining in Bitcoin is in fact dominated by centralized mining pools whose value proposition, over decentralized pools, is an easy setup and participation. Therefore, rational miners motivated by their own benefit, instead of joining decentralized pools prefer centralized "plug-and-play" mining. It is still debatable whether centralized mining pools are beneficial or harmful to the system. However, it has been proved multiple times, that the concentration of significant computing power caused by centralized mining is risky and should be avoided, as such a strong pool has multiple ways of misbehaving and becomes a single point of failure in the system. One example is the pool GHash.IO, which in 2014 achieved more than 51% of the mining power. This undermined trust in the Bitcoin network to the extent that the pool was forced to actively ask miners to join other pools [12].

In order to follow incentives of rational miners, StrongChain does not require any radical changes from them and is compatible with centralized mining pools; however, it is specifically designed to mitigate their main security risk (i.e., power centralization). In StrongChain such pools could be much smaller than in Bitcoin (due to minimized variance) and to support this argument we conducted a study. We listed the largest Bitcoin mining pools and their shares in the global mining power (according to `https://www.blockchain.com/en/pools` as for the time of writing). Then for each pool, we calculated what would be the pool size in StrongChain to offer the miner the same payout variance experience, and the variance reduction factor in that case. As shown in Table 2, for the Bitcoin largest mining pool with 18.1% of the global hash rate, an equivalent pool in StrongChain (to provide miners the same reward experience) could be as small as 0.245% of the hash rate – around 74 times smaller. Even better reduction factors are achieved for smaller pools. Therefore, our study indicates that StrongChain makes the size of a pool almost an irrelevant factor for miners' benefits (i.e., there is no objective advantage of joining a large pool over a medium or a small one). Therefore we envision that with StrongChain, centralized mining pools will naturally be much more distributed.

Mining Pool	Pool Size		Size
	Bitcoin	StrongChain	Reduction
BTC.com	18.1%	0.245%	74×
F2Pool	14.1%	0.172%	82×
AntPool	11.7%	0.135%	87×
SlushPool	9.1%	0.099%	92×
ViaBTC	7.5%	0.079%	95×
BTC.TOP	7.1%	0.074%	96×
BitClub	3.1%	0.030%	103×
DPOOL	2.6%	0.025%	104×
Bitcoin.com	1.9%	0.018%	106×
BitFury	1.7%	0.016%	106×

Table 2: Largest Bitcoin mining pools and the corresponding pool sizes in StrongChain offering the same relative reward variance ($T_w/T_s = 1024$ and $\gamma = 10$).

Limitations

As discussed, it is beneficial for the system if as many participants as possible independently run full nodes; however, miners join large centralized pools not only due to high reward variance. Other potential reasons include the minimization of operational expenses as running a full node is a large overhead, higher efficiency since large pools may use high-performance hardware and network, better ability to earn extra income from merge mining [29], better protection against various attacks, anonymity benefits, etc. This work focuses on removing the reward variance reason. Although we believe that StrongChain would produce a larger number of small pools in a natural way, it does not eliminate the other reasons, so some large centralized pools may still remain. Luckily, our system is orthogonal to multiple concurrent solutions. For instance, StrongChain could be easily combined with non-outsourceable puzzle schemes (see Section 8) to increase the number of full nodes by explicitly disincentivizing miners from outsourcing their computing power. We leave such a combination as interesting future work.

7 Realization in Practice

We implemented our system in order to investigate its feasibility and confirm the stated properties. We implemented a StrongChain full node with interactive client in Python, and

our implementation includes the complete logic from Algorithm 1 and all functionalities required to have a fully operational system (communication modules, message types, validation logic, etc...).[9] As described before, the main changes in our implementation to the Bitcoin's block layout are:

- a new (20B-long) *Coinbase* header field,
- a new binding transaction protecting all weak headers of the block,
- removed original coinbase transaction,

where a binding transaction has a single (32B-long) output as presented in Equation 3.[10]

Weak headers introduced by our system impact the bandwidth and storage overhead (when compared with Bitcoin). Due to compressing them (see Section 4.2), the size of a single weak header in a block is 60B. For example, with an average number of weak headers equal 1024, the storage and bandwidth overhead increases by about 61.5KB per block (e.g., with 64 weak headers, the overhead is only 3.8KB). Taking into account the average Bitcoin block size of about 1MB (the average between 15 Oct and 15 Nov 2018[11]), 1024 weak headers constitute around 6.1% of today's blocks, while 64 headers only 0.4%. The same overhead is introduced to SPV clients, that besides a strong header need to obtain weak headers and a proof for their corresponding binding transaction. Thus, an SPV update (every 10 minutes) would be 61.5KB or 3.8KB on average for 1024 or 64 weak headers, respectively. However, since only strong headers authenticate transactions, SPV clients do not need to store weak headers and after they are validated, they can remove them (they need to just calculate and associate their aggregated PoW with the strong header). Such an approach would not introduce any noticeable storage overhead on SPV clients.

Nodes validate all incoming weak headers; however, this overhead is a single hash computation and simple sanity checks per header. Even with our unoptimized implementation running on a commodity PC the total validation of a single weak header takes around 50μs on average (i.e., 51ms per 1024 headers on a single core). Given that we do not believe this overhead can lead to more serious denial-of-service attacks than ones already known and existing in Bitcoin (e.g., spamming with large invalid blocks). Additionally, StrongChain can adopt prevention techniques present in Bitcoin, like blacklisting misbehaving peers.

[9] Our implementation is available at `https://github.com/ivan-homoliak-sutd/strongchain-demo/`.

[10] An alternative choice is to store a hash of weak headers in a header itself. Although simpler, that option would incur a higher overhead if the number of weak headers is greater than several.

[11] `https://www.blockchain.com/en/charts/avg-block-size`

8 Related work

Employing weak solutions (and their variations) in Bitcoin is an idea [36,38] circulating on Bitcoin forums for many years. Initial proposals leverage weak solutions (i.e., *weak blocks*) for faster transaction confirmations [45,46], for signaling the current working branch of particular miners [13,14,30]. Unfortunately, most of these proposals come without necessary details or lack rigorous analysis. Below, we discuss the most related attempts that have been made to utilize weak or stale blocks in PoW-based decentralized consensus protocols. We compare these systems in Table 3 according to their reward and PoW calculation schemes.

Subchains. Rizun proposes Subchains [35], where a chain of weak blocks (a so-called subchain) bridging each pair of subsequent strong blocks is created. The design of Subchain puts a special focus on increasing the transaction throughput and the double-spend security for unconfirmed transactions. Rizun argues that since the (weak) block interval of subchains is much smaller than the strong block interval, it allows for faster (weak) transaction confirmations. Another claimed advantage of such an approach is that during the process of building subchains, the miners can detect forks earlier, and take actions accordingly to avoid wasting computational power. However, the design of Subchain sidesteps a concrete security analysis at the subchain level. In detail, by using a chaining data structure where one weak header referencing the previous weak header in a subchain, it introduces high stale rate on a subchain. More importantly, due to applying a Bitcoin-like subchain selection policy in case of conflicts, this approach is vulnerable to the selfish mining attack launched on a subchain.

Flux. Based on similar ideas as Subchain, Zamyatin et al. propose Flux [49]. In contrast to Subchain, Flux shares rewards (from newly minted coins and transaction fees) evenly among the finders of weak and strong blocks according to the computational resources they invested. This approach reduces the reward variance of miners, and therefore mitigates the need for large mining pools, which is beneficial for the system's decentralization. In addition, simulation experiments show that Flux renders selfish mining on the main chain less profitable. However, alike Subchains, Flux employs a chain structure for weak blocks, which inevitably introduces race conditions, increasing the stale rate of weak blocks and making it more susceptible to selfish mining attacks at the subchain level. The designers of Flux let both of these issues open and discuss the potential application of GHOST [41] to subchains. Another limitation of this work is that the authors do not analyze the requirements on space consumption when putting possibly a high number of overlapping transactions into Flux subchains, which could negatively influence network, storage, and processing resources.

Remarks on Subchain and Flux. One important difference between our approach and the above two designs is that we

	Bitcoin v0.1	Bitcoin	Fruitchains	Flux	StrongChain
Reward (strong)	$R+F$	$R+F$	0	$(R+F)/(E+1)$	$cR+F$
Reward (weak)	0	0	$(R+F)/E$	$(R+F)/(E+1)$	$c\gamma RT_s/T_w$
Chain weight contrib. (strong)	1	T_{max}/T_s	T_{max}/T_s	T_{max}/T_s	T_{max}/T_s
Chain weight contrib. (weak)	0	0	0	0	T_{max}/T_w

Table 3: The comparison of reward and PoW computation schemes of StrongChain and the related systems. (F: block transaction fees, E: expected number of weak headers per block. The entries for Flux are approximations for the PPLNS scheme in P2Pool, on which it is based.)

adopt a flat hierarchy for the weak blocks, which not only eliminates the possibility of selfish mining in a set of weak solutions, but also avoids the issue of stale rate of weak blocks. In contrast, both Subchain and Flux employ a chain structure for weak blocks, inevitably making them more susceptible to selfish mining attacks at the subchain level. Moreover, in our approach rewards are not shared, therefore miners can only benefit from appending received weak solutions. In addition, none of Subchain and Flux provide a concrete implementation demonstrating their applicability.

FruitChains. Another approach to address the mining variance and selfish mining issues is the FruitChains protocol proposed by Pass and Shi [32]. In FruitChains, instead of directly storing the records inside blocks, the records or transactions are put inside "fruits" with relatively low mining difficulties. Fruits then are appended to a blockchain via blocks which are mined with a higher difficulty. Mined fruits and blocks yield rewards, hence, miners can be paid more often and there is no need to form a mining pool.

However, some practical and technical details of FruitChains lead to undesired side effects. First, the scheme allows fruits with small difficulties to be announced and accepted by other miners. With too small difficulty it could render high transmission overheads or even potential denial-of-service attacks and its effects on the network are not investigated. On the other hand, too high fruit difficulty could result in a low transaction throughput and a high reward variance. Second, duplicate fruits are discarded, even though they might be found by different miners – this naturally implies some stale fruit rate (uninvestigated in the paper). Similarly, it is unclear would a block finder have an incentive to treat all fruits equally and to not prioritize her mined fruits (especially when fruits are associated with transaction fees). Moreover, fruits that are not appended to the blockchain quickly enough have to be mined and broadcast again, rendering additional overheads. Finally, the description of FruitChains lacks important details (e.g., the size of the fruits or the overheads introduced) as well as an actual implementation.

GHOST and Ethereum. An alternative approach for decreasing a high reward variance of miners is to shorten the block creation rate to the extent that does not hurt the overall system security – such an approach increases transaction throughput as well. The Greedy Heaviest-Observed Sub-Tree (GHOST) chain selection rule [41] makes use of stale blocks to increase the weight of their ancestors, which achieves a 600 fold speedup for the block generation compared to Bitcoin, while preserving its security strength. Despite the inclusion of stale blocks in the blockchain, only the miners of the main chain get rewards for the inclusion of the stale blocks.

In contrast to the original GHOST, the white and yellow papers of Ethereum [44, 48] propose to reward also miners of stale blocks in order to further increase the security – not wasting with the consumed resources for mining of stale blocks. However, Ritz and Zugenmaier shows that rewarding so called "uncle blocks" lowers the threshold at which selfish mining is profitable [34] – a selfish miner can built-up the "heaviest" chain, as she can reference blocks previously not broadcast to the honest network. Likewise, the inclusive blockchain protocol [20], which increases the transaction throughput, but leaves the selfish mining issue unsolved.

DAG-based Protocols. SPECTRE [40] is an example of the protocols family that leverages a directed acyclic graph (DAG). This family proposed more radical design changes motivated by the observation that one essential throughput limitation of Nakamoto consensus is the data structure leveraged which can be maintained only sequentially. SPECTRE generalizes the Nakamoto's blockchain into a DAG of blocks, while allowing miners to add blocks concurrently with a high frequency, just basing on their individual current views of the DAG. Such a design provides multiple advantages over chain-based protocols including StrongChain. Frequently published blocks increase transaction throughput and provide fast confirmation times while relaxed consistency requirements allow to tolerate propagation delays. Like StrongChain, SPECTRE also aims to decrease mining reward variance, but achieves it again via frequent blocks. However, frequent blocks have a side effect of transaction redundancy which negatively impacts the storage and transmission overheads, and this aspect was not investigated. Moreover, SPECTRE provides a weaker property than chain-based consensus protocols as simultaneously added transactions cannot be ordered. This and schemes following a similar design are payments oriented and to support order-specific applications, like smart contracts, they need to be enhanced with an additional ordering logic.

More recently, Sompolinsky and Zohar [42] proposed two DAG-based protocols improving the prior work. PHANTOM introduces and uses a greedy algorithm (called the

GHOSTDAG protocol) to determine the order of transactions. This eliminates the applicability issues of SPECTRE, but for the cost of slowing down transaction confirmation times. Combining advantages of PHANTOM and SPECTRE into a full system was left by the authors as a future work.

Decentralization-oriented Schemes. Mining decentralization was a goal of multiple previous proposals. One direction is to design mining such that miners do not have incentive to outsource resources or forming coalitions. Permacoin [25] is an early attempt to achieve it where miners instead of proving their work prove that their store (fragments of) a globally-agreed file. Permacoin is designed such that: a) payment private keys are bound to puzzle solutions – outsourcing private keys is risky for miners, b) sequential and random storage access is critical for the mining efficiency, thus it disincentives miners from outsourcing data. If the file is valuable, then a side-effect of Permacoin is its usefulness, as miners replicate the file.

The notion of non-outsourceable mining was further extended and other schemes were proposed [26, 50]. Miller et al. [26] introduces "strongly non-outsourceable puzzles" that aim to disincentivize pool creation by requiring all pool participants to remain honest. In short, with these puzzles any pool participant can steal the pool reward without revealing its identity. The scheme relies on zero knowledge proofs requiring a trusted setup and introducing significant computational overheads. The scheme is orthogonal to StrongChain and could be integrated with easily integrated with StrongChain, however, after a few years of their introduction no system of this class was actually deployed at scale; thus, we do not have any empirical results confirming their promised benefits.

SmartPool is a different approach that was proposed by Luu et al. [23]. In SmartPool, the functionality of mining pools was implemented as a smart contract. Such an approach runs natively only on smart-contract platforms but it allows to eliminate actual mining pools and their managers (note that SmartPool still imposes fees for running smart contracts), while preserving most benefits of pool mining.

Rewarding Schemes for Mining Pools. Mining pools divide the block reward (together with the transaction fees) in such a way that each miner joining the pool is paid his fair share in proportion to his contribution. Typically, the contribution of an individual miner in the pool is witnessed by showing weak solutions called *shares*.

There are various rewarding schemes that mining pools employ. The simplest and most natural method is the proportional scheme where the reward of a strong block is divided in proportion to the numbers of shares submitted by the miners. However, this scheme leads to pool hopping attacks [33]. To avoid this security issue, many other rewarding systems are developed, including the Pay-per-last-N-shares (PPLNS) scheme and its variants. We refer the reader to [37] for a systematic analysis of different pool rewarding systems.

The reward mechanisms in StrongChain can be seen conceptually as a mining pool built-in into the protocol. However, there are important differences between our design and the mining pools. The most contrasting one is that in StrongChain rewarding is not a zero-sum game and miners do not share rewards. In mining pools, all rewards are shared and this characteristic causes multiple in- and cross-pool attacks that cannot be launched against our scheme. Furthermore, the miner collaboration within Bitcoin mining pools is a "necessary evil", while in StrongChain the collaboration is beneficial for miners and the overall system. We discuss StrongChain and mining pools further in Section 6.2.

9 Conclusions

In this paper, we proposed a transparent and collaborative proof-of-work protocol. Our approach is based on Nakamoto consensus and Bitcoin, however, we modified their core designs. In particular, in contrast to them, we take advantage of weak solutions, which although they do not finalize a block creation positively contribute to the blockchain properties. We also proposed a rewarding scheme such that miners can benefit from exchanging and appending weak solutions. These modifications lead to a more secure, fair, and efficient system. Surprisingly, we show that our approach helps with seemingly unrelated issues like the freshness property. Finally, our implementation indicates the efficiency and deployability of our approach.

Incentives-oriented analysis of consensus protocols is a relatively new topic and in the future we would like to extend our work by modeling our protocol with novel frameworks and tools. Another topic worth investigating in future is how to combine StrongChain with systems solving other drawbacks of Nakamoto consensus [10, 19, 21], or how to mimic the protocol in the proof-of-stake setting.

Acknowledgment

We thank the anonymous reviewers and our shepherd Joseph Bonneau for their valuable comments and suggestions.

This work was supported in part by the National Research Foundation (NRF), Prime Minister's Office, Singapore, under its National Cybersecurity R&D Programme (Award No. NRF2016NCR-NCR002-028) and administered by the National Cybersecurity R&D Directorate, and by ST Electronics and NRF under Corporate Laboratory @ University Scheme (Programme Title: STEE Infosec - SUTD Corporate Laboratory).

References

[1] L. Bahack. Theoretical Bitcoin attacks with less than half of the computational power (draft). *arXiv preprint*

arXiv:1312.7013, 2013.

[2] D. Bayer, S. Haber, and W. S. Stornetta. Improving the efficiency and reliability of digital time-stamping. In *Sequences II*. Springer, 1993.

[3] A. Boverman. Timejacking & Bitcoin. `https://culubas.blogspot.sg/2011/05/timejacking-bitcoin_802.html`, 2011.

[4] M. Carlsten, H. A. Kalodner, S. M. Weinberg, and A. Narayanan. On the instability of Bitcoin without the block reward. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[5] N. T. Courtois and L. Bahack. On subversive miner strategies and block withholding attack in Bitcoin digital currency. *arXiv preprint arXiv:1402.1718*, 2014.

[6] J. R. Douceur. The Sybil attack. In *International workshop on peer-to-peer systems*. Springer, 2002.

[7] S. Dunbar. Random sums of random variables. `http://www.math.unl.edu/~sdunbar1/ProbabilityTheory/Lessons/Conditionals/RandomSums/randsum.shtml`.

[8] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *Annual International Cryptology Conference*. Springer, 1992.

[9] I. Eyal. The miner's dilemma. In *2015 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2015.

[10] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse. Bitcoin-NG: A scalable blockchain protocol. In *Proceedings of NSDI*, 2016.

[11] I. Eyal and E. G. Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *International conference on financial cryptography and data security*. Springer, 2014.

[12] C. Farivar. Bitcoin pool GHash.io commits to 40% hashrate limit after its 51% breach. `https://arstechnica.com/information-technology/2014/07/bitcoin-pool-ghash-io-commits-to-40-hashrate-limit-after-its-51-breach/`, 2014.

[13] Gavin Andresen. Faster blocks vs bigger blocks. `https://bitcointalk.org/index.php?topic=673415.msg7658481#msg7658481`, 2014.

[14] Gavin Andresen. Weak block thoughts. `https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2015-September/011157.html`, 2015.

[15] A. E. Gencer, S. Basu, I. Eyal, R. van Renesse, and E. G. Sirer. Decentralization in Bitcoin and Ethereum networks. *arXiv preprint arXiv:1801.03998*, 2018.

[16] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun. On the security and performance of proof of work blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016.

[17] A. Gervais, H. Ritzdorf, G. O. Karame, and S. Capkun. Tampering with the delivery of blocks and transactions in Bitcoin. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015.

[18] G. O. Karame, E. Androulaki, and S. Capkun. Double-spending fast payments in Bitcoin. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012.

[19] E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[20] Y. Lewenberg, Y. Sompolinsky, and A. Zohar. Inclusive block chain protocols. In *International Conference on Financial Cryptography and Data Security*. Springer, 2015.

[21] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016.

[22] L. Luu, R. Saha, I. Parameshwaran, P. Saxena, and A. Hobor. On power splitting games in distributed computation: The case of Bitcoin pooled mining. In *Computer Security Foundations Symposium (CSF), 2015 IEEE 28th*. IEEE, 2015.

[23] L. Luu, Y. Velner, J. Teutsch, and P. Saxena. Smartpool: Practical decentralized pooled mining. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, 2017.

[24] R. C. Merkle. A digital signature based on a conventional encryption function. In *Proceedings of Advances in Cryptology*, 1988.

[25] A. Miller, A. Juels, E. Shi, B. Parno, and J. Katz. Permacoin: Repurposing Bitcoin work for data preservation. In *2014 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2014.

[26] A. Miller, A. Kosba, J. Katz, and E. Shi. Nonoutsourceable scratch-off puzzles to discourage Bitcoin mining coalitions. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015.

[27] M. Möser and R. Böhme. Trends, tips, tolls: A longitudinal study of bitcoin transaction fees. In *Financial Cryptography Workshops*, 2015.

[28] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[29] A. Narayanan, J. Bonneau, E. Felten, A. Miller, and S. Goldfeder. *Bitcoin and cryptocurrency technologies: A comprehensive introduction*. Princeton University Press, 2016.

[30] T. Nolan. Distributing low POW headers. `https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2013-July/002976.html`, 2013.

[31] K. Papagiannaki, S. Moon, C. Fraleigh, P. Thiran, and C. Diot. Measurement and analysis of single-hop delay on an IP backbone network. *IEEE Journal on Selected Areas in Communications*, 21(6), 2003.

[32] R. Pass and E. Shi. Fruitchains: A fair blockchain. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*. ACM, 2017.

[33] Raulo. Optimal pool abuse strategy. `http://bitcoin.atspace.com/poolcheating.pdf`, 2011.

[34] F. Ritz and A. Zugenmaier. The impact of uncle rewards on selfish mining in Ethereum. *arXiv preprint arXiv:1805.08832*, 2018.

[35] P. R. Rizun. Subchains: A technique to scale Bitcoin and improve the user experience. *Ledger*, 1, 2016.

[36] K. Rosenbaum. Weak Blocks – The Good And The Bad. `http://popeller.io/index.php/2016/01/19/weak-blocks-the-good-and-the-bad/`, 2016.

[37] M. Rosenfeld. Analysis of Bitcoin pooled mining reward systems. *arXiv preprint arXiv:1112.4980*, 2011.

[38] R. Russell. Weak block simulator for Bitcoin. `https://bitcointalk.org/index.php?topic=179598.0`, 2017.

[39] A. Sapirshtein, Y. Sompolinsky, and A. Zohar. Optimal selfish mining strategies in Bitcoin. In *International Conference on Financial Cryptography and Data Security*. Springer, 2016.

[40] Y. Sompolinsky, Y. Lewenberg, and A. Zohar. SPECTRE: Serialization of proof-of-work events: confirming transactions via recursive elections, 2016.

[41] Y. Sompolinsky and A. Zohar. Accelerating Bitcoin's transaction processing. *Fast Money Grows on Trees, Not Chains*, 2013.

[42] Y. Sompolinsky and A. Zohar. PHANTOM, GHOSTDAG: Two scalable BlockDAG protocols. Cryptology ePrint Archive, Report 2018/104, 2018. `https://eprint.iacr.org/2018/104`.

[43] P. Szalachowski. (short paper) towards more reliable Bitcoin timestamps. In *Proceedings of the Crypto Valley Conference on Blockchain Technology (CVCBT)*, 2018.

[44] E. team. A Next-Generation Smart Contract and Decentralized Application Platform. `https://github.com/ethereum/wiki/wiki/White-Paper#modified-ghost-implementation`, 2018.

[45] TierNolan (Pseudonymous). Decoupling Transactions and PoW. `https://bitcointalk.org/index.php?topic=179598.0`, 2013.

[46] P. Todd. Near-block broadcasts for proof of tx propagation. `https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2013-September/003275.html`, 2013.

[47] I. Tsabary and I. Eyal. The gap game. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018.

[48] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151, 2014.

[49] A. Zamyatin, N. Stifter, P. Schindler, E. Weippl, and W. J. Knottenbelt. Flux: Revisiting near blocks for proof-of-work blockchains. Cryptology ePrint Archive, Report 2018/415, 2018. `https://eprint.iacr.org/2018/415/20180529:172206`.

[50] G. Zeng, S. M. Yiu, J. Zhang, H. Kuzuno, and M. H. Au. A nonoutsourceable puzzle under GHOST rule. In *2017 15th Annual Conference on Privacy, Security and Trust (PST)*. IEEE, 2017.

[51] R. Zhang and B. Preneel. Lay down the common metrics: Evaluating proof-of-work consensus protocols' security. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019.

Tracing Transactions Across Cryptocurrency Ledgers

Haaroon Yousaf, George Kappos, and Sarah Meiklejohn
University College London
{h.yousaf,g.kappos,s.meiklejohn}@ucl.ac.uk

Abstract

One of the defining features of a cryptocurrency is that its ledger, containing all transactions that have ever taken place, is globally visible. As one consequence of this degree of transparency, a long line of recent research has demonstrated that — even in cryptocurrencies that are specifically designed to improve anonymity — it is often possible to track money as it changes hands, and in some cases to de-anonymize users entirely. With the recent proliferation of alternative cryptocurrencies, however, it becomes relevant to ask not only whether or not money can be traced as it moves within the ledger of a single cryptocurrency, but if it can in fact be traced as it moves *across* ledgers. This is especially pertinent given the rise in popularity of automated trading platforms such as ShapeShift, which make it effortless to carry out such cross-currency trades. In this paper, we use data scraped from ShapeShift over a thirteen-month period and the data from eight different blockchains to explore this question. Beyond developing new heuristics and creating new types of links across cryptocurrency ledgers, we also identify various patterns of cross-currency trades and of the general usage of these platforms, with the ultimate goal of understanding whether they serve a criminal or a profit-driven agenda.

1 Introduction

For the past decade, cryptocurrencies such as Bitcoin have been touted for their transformative potential, both as a new form of electronic cash and as a platform to "re-decentralize" aspects of the Internet and computing in general. In terms of their role as cash, however, it has been well established by now that the usage of pseudonyms in Bitcoin does not achieve meaningful levels of anonymity [1, 11, 17, 18, 21], which casts doubt on its role as a payment mechanism. Furthermore, the ability to track flows of coins is not limited to Bitcoin: it extends even to so-called "privacy coins" like Dash [10, 12], Monero [4, 7, 13, 24], and Zcash [6, 16] that incorporate features explicitly designed to improve on Bitcoin's anonymity guarantees.

Traditionally, criminals attempting to cash out illicit funds would have to use exchanges; indeed, most tracking techniques rely on identifying the addresses associated with these exchanges as a way to observe when these deposits happen [11]. Nowadays, however, exchanges typically implement strict Know Your Customer/Anti-Money Laundering (KYC/AML) policies to comply with regulatory requirements, meaning criminals (and indeed all users) risk revealing their real identities when using them. Users also run risks when storing their coins in accounts at custodial exchanges, as exchanges may be hacked or their coins may otherwise become inaccessible [9, 19]. As an alternative, there have emerged in the past few years frictionless trading platforms such as ShapeShift[1] and Changelly,[2] in which users are able to trade between cryptocurrencies without having to store their coins with the platform provider. Furthermore, while ShapeShift now requires users to have verified accounts [22], this was not the case before October 2018.

Part of the reason for these trading platforms to exist is the sheer rise in the number of different cryptocurrencies: according to the popular cryptocurrency data tracker CoinMarketCap there were 36 cryptocurrencies in September 2013, only 7 of which had a stated market capitalization of over 1 million USD,[3] whereas in January 2019 there were 2117 cryptocurrencies, of which the top 10 had a market capitalization of over 100 million USD. Given this proliferation of new cryptocurrencies and platforms that make it easy to transact across them, it becomes important to consider not just whether or not flows of coins can be tracked within the transaction ledger of a given currency, but also if they can be tracked as coins move across their respective ledgers as well. This is especially important given that there are documented cases of criminals attempting to use these cross-currency trades to obscure the flow of their coins: the WannaCry ransomware operators, for example, were observed using ShapeShift to convert their ransomed bitcoins into Monero [3]. More generally, these

[1] https://shapeshift.io
[2] https://changelly.com
[3] https://coinmarketcap.com/historical/20130721/

services have the potential to offer an insight into the broader cryptocurrency ecosystem and the thousands of currencies it now contains.

In this paper, we initiate an exploration of the usage of these cross-currency trading platforms, and the potential they offer in terms of the ability to track flows of coins as they move across different transaction ledgers. Here we rely on three distinct sources of data: the cryptocurrency blockchains, the data collected via our own interactions with these trading platforms, and — as we describe in Section 4 — the information offered by the platforms themselves via their public APIs.

We begin in Section 5 by identifying the specific on-chain transactions associated with an advertised ShapeShift transaction, which we are able to do with a relatively high degree of success (identifying both the deposit and withdrawal transactions 81.91% of the time, on average). We then describe in Section 6 the different transactional patterns that can be traced by identifying the relevant on-chain transactions, focusing specifically on patterns that may be indicative of trading or money laundering, and on the ability to link addresses across different currency ledgers. We then move in Section 7 to consider both old and new heuristics for clustering together addresses associated with ShapeShift, with particular attention paid to our new heuristic concerning the common social relationships revealed by the usage of ShapeShift. Finally, we bring all the analysis together by applying it to several case studies in Section 8. Again, our particular focus in this last section is on the phenomenon of trading and other profit-driven activity, and the extent to which usage of the ShapeShift platform seems to be motivated by criminal activity or a more general desire for anonymity.

2 Related Work

We are not aware of any other research exploring these cross-currency trading platforms, but consider as related all research that explores the level of anonymity achieved by cryptocurrencies. This work is complementary to our own, as the techniques it develops can be combined with ours to track the entire flow of cryptocurrencies as they move both within and across different ledgers.

Much of the earlier research in this vein focused on Bitcoin [1, 11, 17, 18, 21], and operates by adopting the so-called "multi-input" heuristic, which says that all input addresses in a transaction belong to the same entity (be it an individual or a service such as an exchange). While the accuracy of this heuristic has been somewhat eroded by privacy-enhancing techniques like CoinJoin [8], new techniques have been developed to avoid such false positives [12], and as such it has now been accepted as standard and incorporated into many tools for Bitcoin blockchain analytics.[4][5] Once addresses are clustered together in this manner, the entity can then further be identified using hand-collected tags that form a ground-truth dataset. We adopt both of these techniques in order to analyze the clusters formed by ShapeShift and Changelly in a variety of cryptocurrency blockchains, although as described in Section 7 we find them to be relatively unsuccessful in this setting.

In response to the rise of newer "privacy coins", a recent line of research has also worked to demonstrate that the deployed versions of these cryptocurrencies have various properties that diminish the level of anonymity they achieve in practice. This includes work targeting Dash [10, 12], Monero [4, 7, 13, 24], and Zcash [6, 16].

In terms of Dash, its main privacy feature is similar to CoinJoin, in which different senders join forces to create a single transaction representing their transfer to a diverse set of recipients. Despite the intention for this to hide which recipient addresses belong to which senders, research has demonstrated that such links can in fact be created based on the value being transacted [10, 12]. Monero, which allows senders to hide which input belongs to them by using "mix-ins" consisting of the keys of other users, is vulnerable to de-anonymization attacks exploiting the (now-obsolete) case in which some users chose not to use mix-ins, or exploiting inferences about the age of the coins used as mix-ins [4, 7, 13, 24]. Finally, Zcash is similar to Bitcoin, but with the addition of a privacy feature called the shielded pool, which can be used to hide the values and addresses of the senders and recipients involved in a transaction. Recent research has shown that it is possible to significantly reduce the anonymity set provided by the shielded pool, by developing simple heuristics for identifying links between hidden and partly obscured transactions [6, 16].

3 Background

3.1 Cryptocurrencies

The first decentralized cryptocurrency, Bitcoin, was created by Satoshi Nakamoto in 2008 [14] and deployed in January 2009. At the most basic level, bitcoins are digital assets that can be traded between sets of users without the need for any trusted intermediary. Bitcoins can be thought of as being stored in a public key, which is controlled by the entity in possession of the associated private key. A single user can store their assets across many public keys, which act as pseudonyms with no inherent link to the user's identity. In order to spend them, a user can form and cryptographically sign a transaction that acts to send the bitcoins to a recipient of their choice. Beyond Bitcoin, other platforms now offer more robust functionality. For example, Ethereum allows users to deploy *smart contracts* onto the blockchain, which act as stateful programs that can be triggered by transactions providing inputs to their functions.

In order to prevent double-spending, many cryptocurrencies are *UTXO-based*, meaning coins are associated not with

[4] https://www.chainalysis.com/
[5] https://www.elliptic.co/

an address but with a uniquely identifiable UTXO (unspent transaction output) that is created for all outputs in a given transaction. This means that one address could be associated with potentially many UTXOs (corresponding to each time it has received coins), and that inputs to transactions are also UTXOs rather than addresses. Checking for double-spending is then just a matter of checking if an input is in the current UTXO'set, and removing it from the set once it spends it contents.

3.2 Digital asset trading platforms

In contrast to a traditional (custodial) exchange, a digital asset trading platform allows users to move between different cryptocurrencies without storing any money in an account with the service; in other words, users keep their own money in their own accounts and the platform has it only at the time that a trade is being executed. To initiate such a trade, a user approaches the service and selects a supported input currency curIn (i.e., the currency from which they would like to move money) and a supported output currency curOut (the currency that they would like to obtain). A user additionally specifies a destination address $\mathsf{addr_u}$ in the curOut blockchain, which is the address to which the output currency will be sent. The service then presents the user with an exchange rate rate and an address $\mathsf{addr_s}$ in the curIn blockchain to which to send money, as well as a miner fee fee that accounts for the transaction it must form in the curOut blockchain. The user then sends to this address $\mathsf{addr_s}$ the amount amt in curIn they wish to convert, and after some delay the service sends the appropriate amount of the output currency to the specified destination address $\mathsf{addr_u}$. This means that an interaction with these services results in two transactions: one on the curIn blockchain sending amt to $\mathsf{addr_s}$, and one on the curOut blockchain sending (roughly) $\mathsf{rate} \cdot \mathsf{amt} - \mathsf{fee}$ to $\mathsf{addr_u}$.

This describes an interaction with an abstracted platform. Today, the two best-known examples are ShapeShift and Changelly. Whereas Changelly has always required account creation, ShapeShift introduced this requirement only in October 2018. Each platform supports dozens of cryptocurrencies, ranging from better-known ones such as Bitcoin and Ethereum to lesser-known ones such as FirstBlood and Clams. In Section 4, we describe in more depth the operations of these specific platforms and our own interactions with them.

4 Data Collection and Statistics

In this section, we describe our data sources, as well as some preliminary statistics about the collected data. We begin in Section 4.1 by describing our own interactions with Changelly, a trading platform with a limited personal API. We then describe in Section 4.2 both our own interactions with ShapeShift, and the data we were able to scrape from their public API, which provided us with significant insight into their overall set of transactions. Finally, we describe in Section 4.3 our collection of the data backing eight different cryptocurrencies.

4.1 Changelly

Changelly offers a simple API[6] that allows registered users to carry out transactions with the service. Using this API, we engaged in 22 transactions, using the most popular ShapeShift currencies (Table 1) to guide our choices for curIn and curOut.

While doing these transactions, we observed that they would sometimes take up to an hour to complete. This is because Changelly attempts to minimize double-spending risk by requiring users to wait for a set number of confirmations (shown to the user at the time of their transaction) in the curIn blockchain before executing the transfer on the curOut blockchain. We used this observation to guide our choice of parameters in our identification of on-chain transactions in Section 5.

4.2 ShapeShift

ShapeShift's API[7] allows users to execute their own transactions, of which we did 18 in total. As with Changelly, we were able to gain some valuable insights about the operation of the platform via these personal interactions. Whereas ShapeShift did not disclose the number of confirmations they waited for on the curIn blockchain, we again observed long delays, indicating that they were also waiting for a sufficient number.

Beyond these personal interactions, the API provides information on the operation of the service as a whole. Most notably, it provides three separate pieces of information: (1) the current trading rate between any pair of cryptocurrencies, (2) a list of up to 50 of the most recent transactions that have taken place (across all users), and (3) full details of a specific ShapeShift transaction given the address $\mathsf{addr_s}$ in the curIn blockchain (i.e., the address to which the user sent their coins).

For the trading rates, ShapeShift provides the following information for all cryptocurrency pairs $(\mathsf{curIn}, \mathsf{curOut})$: the rate, the limit (i.e., the maximum that can be exchanged), the minimum that can be exchanged, and the miner fee (denominated in curOut). For the 50 most recent transactions, information is provided in the form: $(\mathsf{curIn}, \mathsf{curOut}, \mathsf{amt}, t, \mathsf{id})$, where the first three of these are as discussed in Section 3.2, t is a UNIX timestamp, and id is an internal identifier for this transaction. For the transaction information, when provided with a specific $\mathsf{addr_s}$ ShapeShift provides the tuple $(\mathsf{status}, \mathsf{address}, \mathsf{withdraw}, \mathsf{inCoin}, \mathsf{inType}, \mathsf{outCoin}, \mathsf{outType}, \mathsf{tx}, \mathsf{txURL}, \mathsf{error})$. The status field is a flag that is

[6] https://api-docs.changelly.com/
[7] https://info.shapeshift.io/api

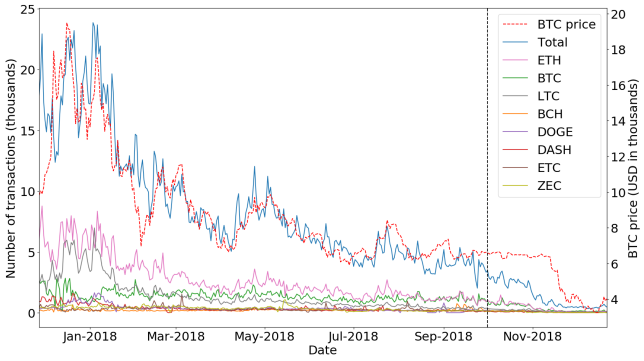


Figure 1: The total number of transactions per day reported via ShapeShift's API, and the numbers broken down by cryptocurrency (where a transaction is attributed to a coin if it is used as either curIn or curOut). The dotted red line indicates the BTC-USD exchange rate, and the horizontal dotted black line indicates when KYC was introduced into ShapeShift.

either `complete`, to mean the transaction was successful; `error`, to mean an issue occurred with the transaction or the queried address was not a ShapeShift address; or `no_deposits`, to mean a user initiated a transaction but did not send any coins. The error field appears when an error is returned and gives a reason for the error. The address field is the same address $addr_s$ used by ShapeShift, and withdraw is the address $addr_u$ (i.e., the user's recipient address in the curOut blockchain). inType and outType are the respective curIn and curOut currencies and inCoin is the amt received. outCoin is the amount sent in the curOut blockchain. Finally, tx is the transaction hash in the curOut blockchain and txURL is a link to this transaction in an online explorer.

Using a simple Web scraper, we downloaded the transactions and rates every five seconds for close to thirteen months: from November 27 2017 until December 23 2018. This resulted in a set of 2,843,238 distinct transactions. Interestingly, we noticed that several earlier test transactions we did with the platform did not show up in their list of recent transactions, which suggests that their published transactions may in fact underestimate their overall activity.

4.2.1 ShapeShift currencies

In terms of the different cryptocurrencies used in ShapeShift transactions, their popularity was distributed as seen in Figure 1. As this figure depicts, the overall activity of ShapeShift is (perhaps unsurprisingly) correlated with the price of Bitcoin in the same time period. At the same time, there is a decline in the number of transactions after KYC was introduced that is not clearly correlated with the price of Bitcoin (which is largely steady and declines only several months later).

ShapeShift supports dozens of cryptocurrencies, and in our data we observed the use of 65 different ones. The most commonly used coins are shown in Table 1. It is clear that Bitcoin and Ethereum are the most heavily used currencies, which is perhaps not surprising given the relative ease with which they can be exchanged with fiat currencies on more traditional exchanges, and their rank in terms of market capitalization.

Currency	Abbr.	Total	curIn	curOut
Ethereum	ETH	1,385,509	892,971	492,538
Bitcoin	BTC	1,286,772	456,703	830,069
Litecoin	LTC	720,047	459,042	261,005
Bitcoin Cash	BCH	284,514	75,774	208,740
Dogecoin	DOGE	245,255	119,532	125,723
Dash	DASH	187,869	113,272	74,597
Ethereum Classic	ETC	179,998	103,177	76,821
Zcash	ZEC	154,142	111,041	43,101

Table 1: The eight most popular coins used on ShapeShift, in terms of the total units traded, and the respective units traded with that coin as curIn and curOut.

4.3 Blockchain data

For the cryptocurrencies we were interested in exploring further, it was also necessary to download and parse the respective blockchains, in order to identify the on-chain transactional behavior of ShapeShift and Changelly. It was not feasible to do this for all 65 currencies used on ShapeShift (not to mention that given the low volume of transactions for many of them, it would likely not yield additional insights anyway), so we chose to focus instead on just the top 8, as seen in Table 1. Together, these account for 95.7% of all ShapeShift transactions if only one of curIn and/or curOut is one of the eight, and 60.5% if both are.

For each of these currencies, we ran a full node in order to download the entire blockchain. For the ones supported by the BlockSci tool [5] (Bitcoin, Dash and Zcash), we used it to parse and analyze their blockchains. BlockSci does not, however, support the remaining five currencies. For these we thus parsed the blockchains using Python scripts, stored the data as Apache Spark parquet files, and analyzed them using custom scripts. In total, we ended up working with 654 GB of raw blockchain data and 434 GB of parsed blockchain data.

5 Identifying Blockchain Transactions

In order to gain deeper insights about the way these trading platforms are used, it is necessary to identify not just their internal transactions but also the transactions that appear on the blockchains of the traded currencies. This section presents heuristics for identifying these on-chain transactions, and the next section explores the additional insights these transactions can offer.

Recall from Section 3.2 that an interaction with ShapeShift results in the deposit of coins from the user to the service on the curIn blockchain (which we refer to as "Phase 1"), and

the withdrawal of coins from the service to the user on the curOut blockchain ("Phase 2"). To start with Phase 1, we thus seek to identify the deposit transaction on the input (curIn) blockchain. Similarly to Portnoff et al. [15], we consider two main requirements for identifying the correct on-chain transaction: (1) that it occurred reasonably close in time to the point at which it was advertised via the API, and (2) that the value it carried was identical to the advertised amount.

For this first requirement, we look for candidate transactions as follows. Given a ShapeShift transaction with timestamp t, we first find the block b (at some height h) on the curIn blockchain that was mined at the time closest to t. We then look at the transactions in all blocks with height in the range $[h-\delta_b, h+\delta_a]$, where δ_b and δ_a are parameters specific to curIn. We looked at both earlier and later blocks based on the observation in our own interactions that the timestamp published by ShapeShift would sometimes be earlier and sometimes be later than the on-chain transaction.

For each of our eight currencies, we ran this heuristic for every ShapeShift transaction using curIn as the currency in question, with every possible combination of δ_b and δ_a ranging from 0 to 30. This resulted in a set of candidate transactions with zero hits (meaning no matching transactions were found), a single hit, or multiple hits. To rule out false positives, we initially considered as successful only ShapeShift transactions with a single candidate on-chain transaction, although we describe below an augmented heuristic that is able to tolerate multiple hits. We then used the values of δ_b and δ_a that maximized the number of single-hit transactions for each currency. As seen in Table 2, the optimal choice of these parameters varies significantly across currencies, according to their different block rates; typically we needed to look further before or after for currencies in which blocks were produced more frequently.

In order to validate the results of our heuristic for Phase 1, we use the additional capability of the ShapeShift API described in Section 4.2. In particular, we queried the API on the recipient address of every transaction identified by our heuristic for Phase 1. If the response of the API was affirmative, we flagged the recipient address as belonging to ShapeShift and we identified the transaction in which it received coins as the curIn transaction. This also provided a way to identify the corresponding Phase 2 transaction on the curOut blockchain, as it is just the tx field returned by the API. As we proceed only in the case that the API returns a valid result, we gain ground-truth data in both Phase 1 and Phase 2. In other words, this method serves to not only validate our results in Phase 1 but also provides a way to identify Phase 2 transactions.

The heuristic described above is able to handle only single hits; i.e., the case in which there is only a single candidate transaction. Luckily, it is easy to augment this heuristic by again using the API. For example, assume we examine a BTC-ETH ShapeShift transaction and we find three candidate transactions in the Bitcoin blockchain after applying the basic heuristic described above. To identify which of these transactions is the right one, we simply query the API on all three recipient addresses and check that the status field is affirmative (meaning ShapeShift recognizes this address) and that the outType field is ETH. In the vast majority of cases this uniquely identifies the correct transaction out of the candidate set, meaning we can use the API to both validate our results (i.e., we use it to eliminate potential false positives, as described above) and to augment the heuristic by being able to tolerate multiple candidate transactions. The augmented results for Phase 1 can be found in the last column of Table 2 and clearly demonstrate the benefit of this extra usage of the API. In the most dramatic example, we were able to go from identifying the on-chain transactions for ShapeShift transactions involving Bitcoin 65.75% of the time with the basic heuristic to identifying them 76.86% of the time with the augmented heuristic.

Currency	Parameters		Basic %	Augmented %
	δ_b	δ_a		
BTC	0	1	65.76	76.86
BCH	9	4	76.96	80.23
DASH	5	5	84.77	88.65
DOGE	1	4	76.94	81.69
ETH	5	0	72.15	81.63
ETC	5	0	76.61	78.67
LTC	1	2	71.61	76.97
ZEC	1	3	86.94	90.54

Table 2: For the selected (optimal) parameters and for a given currency used as curIn, the percentage of ShapeShift transactions for which we found matching on-chain transactions for both the basic (time- and value-based) and the augmented (API-based) Phase 1 heuristic. The augmented heuristic uses the API and thus also represents our success in identifying Phase 2 transactions.

5.1 Accuracy of our heuristics

False negatives can occur for both of our heuristics when there are either too many or too few matching transactions in the searched block interval. These are more common for the basic heuristic, as described above and seen in Table 2, because it is conservative in identifying an on-chain transaction only when there is one candidate. This rate could be improved by increasing the searched block radius, at the expense of adding more computation and potentially increasing the false positive rate.

False positives can occur for both of our heuristics if someone sends the same amount as the ShapeShift transaction at roughly the same time, but this transaction falls within our searched interval whereas the ShapeShift one doesn't. In theory, this should not be an issue for our augmented heuristic, since the API will make it clear that the candidate transaction

is not in fact associated with ShapeShift. In a small number of cases (fewer than 1% of all ShapeShift transactions), however, the API returned details of a transaction with different characteristics than the one we were attempting to identify; e.g., it had a different pair of currencies or a different value being sent. This happened because ShapeShift allows users to re-use an existing deposit address, and the API returns only the latest transaction using a given address.

If we blindly took the results of the API, then this would lead to false positives in our augmented heuristic for both Phase 1 and Phase 2. We thus ensured that the transaction returned by the API had three things in common with the ShapeShift transaction: (1) the pair of currencies, (2) the amount being sent, and (3) the timing, within the interval specified in Table 2. If there was any mismatch, we discarded the transaction. For example, given a ShapeShift transaction indicating an ETH-BTC shift carrying 1 ETH and occurring at time t, we looked for all addresses that received 1 ETH at time t or up to 5 blocks earlier. We then queried the API on these addresses and kept only those transactions which reported shifting 1 ETH to BTC. While our augmented heuristic still might produce false positives in the case that a user quickly makes two different transactions using the same currency pair, value, and deposit address, we view this as unlikely, especially given the relatively long wait times we observed ourselves when using the service (as mentioned in Section 4.2).

5.2 Alternative Phase 2 identification

Given that our heuristic for Phase 2 involved just querying the API for the corresponding Phase 1 transaction, it is natural to wonder what would be possible without this feature of the API, or indeed if there are any alternative strategies for identifying Phase 2 transactions. Indeed, it is possible to use a similar heuristic for identifying Phase 1 transactions, by first looking for transactions in blocks that were mined close to the advertised transaction time, and then looking for ones in which the amount was close to the expected amount. Here the amount must be estimated according to the advertised amt, rate, and fee. In theory, the amount sent should be $\mathsf{amt} \cdot \mathsf{rate} - \mathsf{fee}$, although in practice the rate can fluctuate so it is important to look for transactions carrying a total value within a reasonable error rate of this amount.

When we implemented and applied this heuristic, we found that our accuracy in identifying Phase 2 transactions decreased significantly, due to the larger set of transactions that carried an amount within a wider range (as opposed to an exact amount, as in Phase 1) and the inability of this type of heuristic to handle multiple candidate transactions. More importantly, this approach provides no ground-truth information at all: by choosing conservative parameters it is possible to limit the number of false positives, but this is at the expense of the false negative rate (as, again, we observed in our own application of this heuristic) and in general it is not guaranteed that the final set of transactions really are associated with ShapeShift. As this is the exact guarantee we can get by using the API, we continue in the rest of the paper with the results we obtained there, but nevertheless mention this alternative approach in case this feature of the API is discontinued or otherwise made unavailable.

6 Tracking Cross-Currency Activity

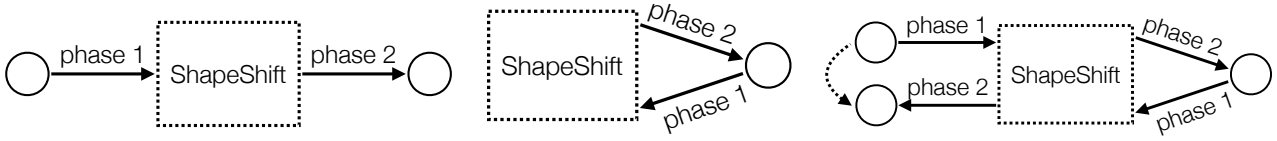
In the previous section, we saw that it was possible in many cases to identify the on-chain transactions, in both the curIn and curOut blockchains, associated with the transactions advertised by ShapeShift. In this section, we take this a step further and show how linking these transactions can be used to identify more complex patterns of behavior.

As shown in Figure 2, we consider these for three main types of transactions. In particular, we look at (1) *pass-through* transactions, which represent the full flow of money as it moves from one currency to the other via the deposit and withdrawal transactions; (2) *U-turns*, in which a user who has shifted into one currency immediately shifts back; and (3) *round-trip* transactions, which are essentially a combination of the first two and follow a user's flow of money as it moves from one currency to another and then back to the original one. Our interest in these particular patterns of behavior is largely based on the role they play in tracking money as it moves across the ledgers of different cryptocurrencies. In particular, our goal is to test the validity of the implicit assumption made by criminal usage of the platform — such as we examine further in Section 8 — that ShapeShift provides additional anonymity beyond simply transacting in a given currency.

In more detail, identifying pass-through transactions allows us to create a link between the input address(es) in the deposit on the curIn blockchain and the output address(es) in the withdrawal on the curOut blockchain.

Identifying U-turns allows us to see when a user has interacted with ShapeShift not because they are interested in holding units of the curOut cryptocurrency, but because they see other benefits in shifting coins back and forth. There are several possible motivations for this: for example, traders may quickly shift back and forth between two different cryptocurrencies in order to profit from differences in their price. We investigate this possibility in Section 8.3. Similarly, people performing money laundering or otherwise holding "dirty" money may engage in such behavior under the belief that once the coins are moved back into the curIn blockchain, they are "clean" after moving through ShapeShift regardless of what happened with the coins in the curOut blockchain.

Finally, identifying round-trip transactions allows us to create a link between the input address(es) in the deposit on the curIn blockchain with the output address(es) in the later withdrawal on the curIn blockchain. Again, there are many reasons why users might engage in such behavior, including



(a) Pass-through (b) U-turn (c) Round-trip

Figure 2: The different transactional patterns, according to how they interact with ShapeShift and which phases are required to identify them.

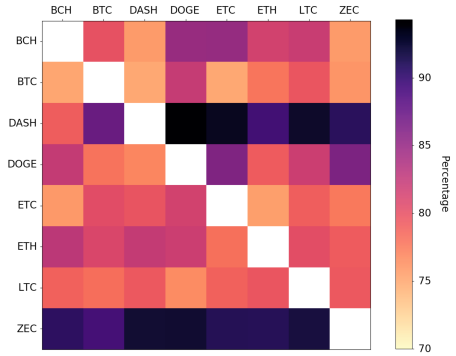


Figure 3: For each pair of currencies, the number of transactions we identified as being a pass-through from one to the other, as a percentage of the total number of transactions between those two currencies.

the trading and money laundering examples given above. As another example, if a curIn user wanted to make an anonymous payment to another curIn user, they might attempt to do so via a round-trip transaction (using the address of the other user in the second pass-through transaction), under the same assumption that ShapeShift would sever the link between their two addresses.

6.1 Pass-through transactions

Given a ShapeShift transaction from curIn to curOut, the methods from Section 5 already provide a way to identify pass-through transactions, as depicted in Figure 2a. In particular, running the augmented heuristic for Phase 1 transactions identifies not only the deposit transaction in the curIn blockchain but also the Phase 2 transaction (i.e., the withdrawal transaction in the curOut blockchain), as this is exactly what is returned by the API. As discussed above, this has the effect on anonymity of tracing the flow of funds across this ShapeShift transaction and linking its two endpoints; i.e., the input address(es) in the curIn blockchain with the output address(es) in the curOut blockchain. The results, in terms of the percentages of all possible transactions between a pair $(\mathsf{curIn}, \mathsf{curOut})$ for which we found the corresponding on-chain transactions, are in Figure 3.

The figure demonstrates that our success in identifying these types of transactions varied somewhat, and depended — not unsurprisingly — on our success in identifying transactions in the curIn blockchain. This means that we were typically least successful with curIn blockchains with higher transaction volumes, such as Bitcoin, because we frequently ended up with multiple hits (although here we were still able to identify more than 74% of transactions). In contrast, the dark stripes for Dash and Zcash demonstrate our high level of success in identifying pass-through transactions with those currencies as curIn, due to our high level of success in their Phase 1 analysis in general (89% and 91% respectively). In total, across all eight currencies we were able to identify 1,383,666 pass-through transactions.

6.2 U-turns

As depicted in Figure 2b, we consider a U-turn to be a pattern in which a user has just sent money from curIn to curOut, only to turn around and go immediately back to curIn. This means linking two transactions: the Phase 2 transaction used to send money to curOut and the Phase 1 transaction used to send money back to curIn. In terms of timing and amount, we require that the second transaction happens within 30 minutes of the first, and that it carries within 1% of the value that was generated by the first Phase 2 transaction. This value is returned by the ShapeShift API in the outCoin field.

While the close timing and amount already give some indication that these two transactions are linked, it is of course possible that this is a coincidence and they were in fact carried out by different users. In order to gain additional confidence that it was the same user, we have two options. In UTXO-based cryptocurrencies (see Section 3.1), we could see if the input is the same UTXO that was created in the Phase 2 transaction, and thus see if a user is spending the coin immediately. In cryptocurrencies based instead on accounts, such as Ethereum, we have no choice but to look just at the addresses. Here we thus define a U-turn as seeing if the address that was used as the output in the Phase 2 transaction is used as the input in the later Phase 1 transaction.

Once we identified such candidate pairs of transactions $(\mathsf{tx}_1, \mathsf{tx}_2)$, we then ran the augmented heuristic from Section 5 to identify the relevant output address in the curOut blockchain, according to tx_1. We then ran the same heuristic to identify the relevant input address in the curOut blockchain, this time according to tx_2.

In fact though, what we really identified in Phase 2 was not just an address but, as described above, a newly created

Currency	# (basic)	# (addr)	# (utxo)
BTC	36,666	565	314
BCH	2864	196	81
DASH	3234	2091	184
DOGE	546	75	75
ETH	53,518	5248	-
ETC	1397	543	-
LTC	8270	1429	244
ZEC	772	419	222

Table 3: The number of U-turns identified for each cryptocurrency, according to our basic heuristic concerning timing and value, and both the address-based and UTXO-based heuristics concerning identical ownership. Since Ethereum and Ethereum Classic are account-based, the UTXO heuristic cannot be applied to them.

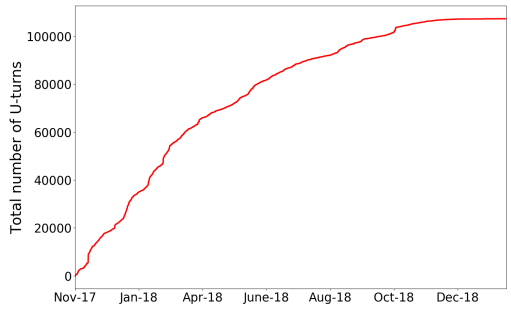


Figure 4: The total number of U-turns over time, as identified by our basic heuristic.

UTXO. If the input used in tx_2 was this same UTXO, then we found a U-turn according to the first heuristic. If instead it corresponded just to the same address, then we found a U-turn according to the second heuristic. The results of both of these heuristics, in addition to the basic identification of U-turns according to the timing and amount, can be found in Table 3, and plots showing their cumulative number over time can be found in Figures 4 and 5. In total, we identified 107,267 U-turns according to our basic heuristic, 10,566 U-turns according to our address-based heuristic, and 1,120 U-turns according to our UTXO-based heuristic.

While the dominance of both Bitcoin and Ethereum should be expected given their overall trading dominance, we also observe that both Dash and Zcash have been used extensively as "mixer coins" in U-turns, and are in fact more popular for this purpose than they are overall. Despite this indication that users may prefer to use privacy coins as the mixing intermediary, Zcash has the highest percentage of identified UTXO-based U-turn transactions. Thus, these users not only do not gain extra anonymity by using it, but in fact are easily identifiable given that they did not change the address used in 419 out of 772 (54.24%) cases, or — even worse — immediately shifted back the exact same coin they received in 222 (28.75%) cases. In the case of Dash, the results suggest

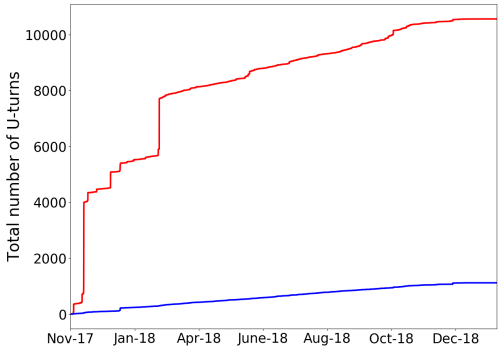


Figure 5: The total number of U-turns over time, as identified by our address-based (in red) and UTXO-based (in blue) heuristics.

something a bit different. Once more, the usage of a privacy coin was not very successful since in 2091 out of the 3234 cases the address that received the fresh coins was the same as the one that shifted it back. It was the exact same coin in only 184 cases, however, which suggests that although the user is the same, there is a local Dash transaction between the two ShapeShift transactions. We defer a further discussion of this asymmetry to Section 8.4, where we also discuss more generally the use of anonymity features in both Zcash and Dash.

Looking at Figure 5, we can see a steep rise in the number of U-turns that used the same address in December 2017, which is not true of the ones that used the same UTXO or in the overall number of U-turns in Figure 4. Looking into this further, we observed that the number of U-turns was particularly elevated during this period for four specific pairs of currencies: DASH-ETH, DASH-LTC, ETH-DASH, and LTC-ETH. This thus affected primarily the address-based heuristic due to the fact that (1) Ethereum is account-based so the UTXO-based heuristic does not apply, and (2) Dash has a high percentage of U-turns using the same address, but a much smaller percentage using the same UTXO. The amount of money shifted in these U-turns varied significantly in terms of the units of the input currency, but all carried between 115K and 138K in USD. Although the ShapeShift transactions that were involved in these U-turns had hundreds of different addresses in the curIn blockchain, they used only a small number of addresses in the curOut blockchain: 4 addresses in Ethereum, 13 in Dash, and 9 in Litecoin. As we discuss further in Section 7.2, the re-use of addresses and the fact that the total amount of money (in USD) carried by the transactions was roughly the same indicates that perhaps a small group of people was responsible for creating this spike in the graph.

6.3 Round-trip transactions

As depicted in Figure 2c, a round-trip transaction requires performing two ShapeShift transactions: one out of the initial

currency and one back into it. To identify round-trip transactions, we effectively combine the results of the pass-through and U-turn transactions; i.e., we tagged something as a round-trip transaction if the output of a pass-through transaction from X to Y was identified as being involved in a U-turn transaction, which was itself linked to a later pass-through transaction from Y to X (of roughly the same amount). As described at the beginning of the section, this has the powerful effect of creating a link between the sender and recipient within a single currency, despite the fact that money flowed into a different currency in between.

In more detail, we looked for consecutive ShapeShift transactions where for a given pair of cryptocurrencies X and Y: (1) the first transaction was of the form X-Y; (2) the second transaction was of the form Y-X; (3) the second transaction happened relatively soon after the first one; and (4) the value carried by the two transaction was approximately the same. For the third property, we required that the second transaction happened within 30 minutes of the first. For the fourth property, we required that if the first transaction carried x units of curIn then the second transaction carried within 0.5% of the value in the (on-chain) Phase 2 transaction, according to the outCoin field provided by the API.

As with U-turns, we considered an additional restriction to capture the case in which the user in the curIn blockchain stayed the same, meaning money clearly did not change hands. Unlike with U-turns, however, this restriction is less to provide accuracy for the basic heuristic and more to isolate the behavior of people engaged in day trading or money laundering (as opposed to those meaningfully sending money to other users). For this pattern, we identify the input addresses used in Phase 1 for the first transaction, which represent the user who initiated the round-trip transaction in the curIn blockchain. We then identify the output addresses used in Phase 2 for the second transaction, which represent the user who was the final recipient of the funds. If the address was the same, then it is clear that money has not changed hands. Otherwise, the round-trip transaction acts as a heuristic for linking together the input and output addresses.

The results of running this heuristic (with and without the extra restriction) are in Table 4. In total, we identified 95,547 round-trip transactions according to our regular heuristic, and identified 10,490 transactions where the input and output addresses were the same. Across different currencies, however, there was a high level of variance in the results. While this could be a result of the different levels of accuracy in Phase 1 for different currencies, the more likely explanation is that users indeed engage in different patterns of behavior with different currencies. For Bitcoin, for example, there was a very small percentage (1.2%) of round-trip transactions that used the same address. This suggests that either users are aware of the general lack of anonymity in the basic Bitcoin protocol and use ShapeShift to make anonymous payments, or that if they do use round-trip transactions as a form of money laundering they are at least careful enough to change their addresses. More simply, it may just be the case that generating new addresses is more of a default in Bitcoin than it is in other currencies.

Currency	# (regular)	# (same addr)
BTC	35,019	437
BCH	1780	84
DASH	3253	2353
DOGE	378	0
ETH	45,611	4085
ETC	1122	626
LTC	6912	2733
ZEC	472	172

Table 4: The number of regular round-trip transactions identified for each cryptocurrency, and the number that use the same initial and final address.

In other currencies, however, such as Dash, Ethereum Classic, Litecoin, and Zcash, there were relatively high percentages of round-trip transactions that used the same input and output address: 72%, 56%, 40%, and 36% respectively. In Ethereum Classic, this may be explained by the account-based nature of the currency, which means that it is common for one entity to use only one address, although the percentage for Ethereum is much lower (9%). In Dash and Zcash, as we have already seen in Section 6.2 and explore further in Section 8.4, it may simply be the case that users assume they achieve anonymity just through the use of a privacy coin, so do not take extra measures to hide their identity.

7 Clustering Analysis

7.1 Shared ownership heuristic

As described in Sections 4.1 and 4.2, we engaged in transactions with both ShapeShift and Changelly, which provided us with some ground-truth evidence of addresses that were owned by them. We also collected three sets of tagging data (i.e., tags associated with addresses that describe their real-world owner): for Bitcoin we used the data available from WalletExplorer,[8] which covers a wide variety of different Bitcoin-based services; for Zcash we used hand-collected data from Kappos et al. [6], which covers only exchanges; and for Ethereum we used the data available from Etherscan,[9] which covers a variety of services and contracts.

In order to understand the behavior of these trading platforms and the interaction they had with other blockchain-based services such as exchanges, our first instinct was to combine these tags with the now-standard "multi-input" clus-

[8] https://www.walletexplorer.com/
[9] https://etherscan.io/

tering heuristic for cryptocurrencies [11, 17], which states that in a transaction with multiple input addresses, all inputs belong to the same entity. When we applied this clustering heuristic to an earlier version of our dataset [23], however, the results were fairly uneven. For Dogecoin, for example, the three ShapeShift transactions we performed revealed only three addresses, which each had done a very small number of transactions. The three Changelly transactions we performed, in contrast, revealed 24,893 addresses, which in total had received over 67 trillion DOGE. These results suggest that the trading platforms operate a number of different clusters in each cryptocurrency, and perhaps even change their behavior depending on the currency, which in turns makes it clear that we did not capture a comprehensive view of the activity of either.

More worrying, in one of our Changelly transactions, we received coins from a Ethereum address that had been tagged as belonging to HitBTC, a prominent exchange. This suggests that Changelly may occasionally operate using exchange accounts, which would completely invalidate the results of the clustering heuristic, as their individually operated addresses would end up in the same cluster as all of the ones operated by HitBTC. We thus decided not to use this type of clustering, and to instead focus on a new clustering heuristic geared at identifying common social relationships.

7.2 Common relationship heuristic

As it was clear that the multi-input heuristic would not yield meaningful information about shared ownership, we chose to switch our focus away from the interactions ShapeShift had on the blockchain and look instead at the relationships between individual ShapeShift users. In particular, we defined the following heuristic:

Heuristic 7.1. *If two or more addresses send coins to the same address in the* curOut *blockchain, or if two or more addresses receive coins from the same address in the* curIn *blockchain, then these addresses have some common social relationship.*

The definition of a common social relationship is (intentionally) vague, and the implications of this heuristic are indeed less clear-cut than those of heuristics around shared ownership. Nevertheless, we consider what it means for two different addresses, in potentially two different blockchains, to have sent coins to the same address; we refer to these addresses as belonging in the *input* cluster of the output address (and analogously refer to the *output* cluster for an address sending to multiple other addresses). In the case in which the addresses are most closely linked, it could represent the same user consolidating money held across different currencies into a single one. It could also represent different users interacting with a common service, such as an exchange. Finally, it could simply be two users who do not know each other directly but happen to be sending money to the same individual. What cannot be the case, however, is that the addresses are not related in any way.

To implement this heuristic, we parsed transactions into a graph where we defined a node as an address and a directed edge (u,v) as existing when one address u initiated a ShapeShift transaction sending coins to v, which we identified using the results of our pass-through analysis from Section 5. (This means that the inputs in our graph are restricted to those for which we ran Phase 1 to find the address, and thus that our input clusters contain only the top 8 currencies. In the other direction, however, we obtain the address directly from the API, which means output clusters can contain all currencies.) Edges are further weighted by the number of transactions sent from u to v. For each node, the cluster centered on that address was then defined as all nodes adjacent to it (i.e., pointing towards it).

Performing this clustering generated a graph with 2,895,445 nodes (distinct addresses) and 2,244,459 edges. Sorting the clusters by in-degree reveals the entities that received the highest number of ShapeShift transactions (from the top 8 currencies, per our caveat above). The largest cluster had 12,868 addresses — many of them belonging to Ethereum, Litecoin, and Dash — and was centered on a Bitcoin address belonging to CoinPayments.net, a multi-coin payment processing gateway. Of the ten largest clusters, three others (one associated with Ripple and two with Bitcoin addresses) are also connected with CoinPayments, which suggests that ShapeShift is a popular platform amongst its users.

Sorting the individual clusters by out-degree reveals instead the users who initiated the highest number of ShapeShift transactions. Here the largest cluster (consisting of 2314 addresses) was centered on a Litecoin address, and the second largest cluster was centered on an Ethereum address that belonged to Binance (a popular exchange). Of the ten largest clusters, two others were centered on Binance-tagged addresses, and three were centered on other exchanges (Freewallet, Gemini, and Bittrex). While it makes sense that exchanges typically dominate on-chain activity in many cryptocurrencies, it is somewhat surprising to also observe that dominance here, given that these exchanges already allow users to shift between many different cryptocurrencies. Aside from the potential for better rates or the perception of increased anonymity, it is thus unclear why a user wanting to shift from one currency to another would do so using ShapeShift as opposed to using the same service with which they have already stored their coins.

Beyond these basic statistics, we apply this heuristic to several of the case studies we investigate in the next section. We also revisit here the large spike in the number of U-turns that we observed in Section 6.2. Our hypothesis then was that this spike was caused by a small number of parties, due to the similar USD value carried by the transactions and by the re-use of a small number of addresses across Dash, Ethereum, and Lite-

coin. Here we briefly investigate this further by examining the clusters centered on these addresses.

Of the 13 Dash addresses, all but one of them formed small input and output clusters that were comprised of addresses solely from Litecoin and Ethereum. Of the 9 Litecoin addresses, 6 had input clusters consisting solely of Dash and Ethereum addresses, with two of them consisting solely of Dash addresses. Finally, of the 4 Ethereum addresses, all of them had input clusters consisting solely of Dash and Litecoin addresses. One of them, however, had a diverse set of addresses in its output cluster, belonging to Bitcoin, Bitcoin Cash, and a number of Ethereum-based tokens. These results thus still suggest a small number of parties, due to the tight connection between the three currencies in the clusters, although of course further investigation would be needed to get a more complete picture.

8 Patterns of ShapeShift Usage

In this section, we examine potential applications of the analysis developed in previous sections, in terms of identifying specific usages of ShapeShift. As before, our focus is on anonymity, and the potential that such platforms may offer for money laundering or other illicit purposes, as well as for trading. To this end, we begin by looking at two case studies associated with explicitly criminal activity and examine the interactions these criminals had with the ShapeShift platform. We then switch in Section 8.3 to look at non-criminal activity, by attempting to identify trading bots that use ShapeShift and the patterns they may create. Finally, in Section 8.4 we look at the role that privacy coins (Monero, Zcash, and Dash) play, in order to identify the extent to which the usage of these coins in ShapeShift is motivated by a desire for anonymity.

8.1 Starscape Capital

In January 2018, an investment firm called Starscape Capital raised over 2,000 ETH (worth 2.2M USD at the time) during their Initial Coin Offering, after promising users a 50% return in exchange for investing in their cryptocurrency arbitrage fund. Shortly afterwards, all of their social media accounts disappeared, and it was reported that an amount of ETH worth 517,000 USD was sent from their wallet to ShapeShift, where it was shifted into Monero [20].

We confirmed this for ourselves by observing that the address known to be owned by Starscape Capital participated in 192 Ethereum transactions across a three-day span (January 19-21), during which it received and sent 2,038 ETH; in total it sent money in 133 transactions. We found that 109 of these transactions sent money to ShapeShift, and of these 103 were shifts to Monero conducted on January 21 (the remaining 6 were shifts to Ethereum). The total amount shifted into Monero was 465.61 ETH (1388.39 XMR), and all of the money was shifted into only three different Monero addresses, of which one received 70% of the resulting XMR. Using the clusters defined in Section 7.2, we did not find evidence of any other addresses (in any other currencies) interacting with either the ETH or XMR addresses associated with Starscape Capital.

8.2 Ethereum-based scams

EtherScamDB[10] is a website that, based on user reports that are manually investigated by its operators, collects and lists Ethereum addresses that have been involved in scams. As of January 30 2019, they had a total of 6374 scams listed, with 1973 associated addresses. We found that 194 of these addresses (9% of those listed) had been involved in 853 transactions to ShapeShift, of which 688 had a `status` field of `complete`. Across these successful transactions, 1797 ETH was shifted to other currencies: 74% to Bitcoin, 19% to Monero, 3% to Bitcoin Cash, and 1% to Zcash.

The scams which successfully shifted the highest volumes belonged to so-called trust-trading and MyEtherWallet scams. Trust-trading is a scam based on the premise that users who send coins prove the legitimacy of their addresses, after which the traders "trust" their address and send back higher amounts (whereas in fact most users send money and simply receive nothing in return). This type of scam shifted over 918 ETH, the majority of which was converted to Bitcoin (691 ETH, or 75%). A MyEtherWallet scam is a phishing/typosquatting scam where scammers operate a service with a similar name to the popular online wallet MyEtherWallet,[11] in order to trick users into giving them their account details. These scammers shifted the majority of the stolen ETH to Bitcoin (207 ETH) and to Monero (151 ETH).

Given that the majority of the overall stolen coins was shifted to Bitcoin, we next investigated whether or not these stolen coins could be tracked further using our analysis. In particular, we looked to see if they performed a U-turn or a round-trip transaction, as discussed in Section 6. We identified one address, associated with a trust-trading scam, that participated in 34 distinct round-trip transactions, all coming back to a different address from the original one. All these transactions used Bitcoin as curOut and used the same address in Bitcoin to both receive and send coins; i.e., we identified the U-turns in Bitcoin according to our address-based heuristic. In total, more than 70 ETH were circulated across these round-trip transactions.

8.3 Trading bots

ShapeShift, like any other cryptocurrency exchange, can be used by traders who wish to take advantage of the volatility in cryptocurrency prices. The potential advantages of doing this via ShapeShift, as compared with other platforms that

[10] https://etherscamdb.info/

[11] https://www.myetherwallet.com/

focus more on the exchange between cryptocurrencies and fiat currencies, are that (1) ShapeShift transactions can be easily automated via their API, and (2) a single ShapeShift transaction acts to both purchase desired coins and dump unwanted ones. Such trading usually requires large volumes of transactions and high precision on their the timing, due to the constant fluctuation in cryptocurrency prices. We thus looked for activity that involved large numbers of similar transactions in a small time period, on the theory that it would be associated primarily with trading bots.

We started by searching for sets of consecutive ShapeShift transactions that carried approximately the same value in curIn (with an error rate of 1%) and involved the same currencies. When we did this, however, we found thousands of such sets. We thus added the extra conditions that there must be at least 15 transactions in the set that took place in a span of five minutes; i.e., that within a five-minute block of ShapeShift transactions there were at least 15 involving the same currencies and carrying the same approximate USD value. This resulted in 107 such sets.

After obtaining our 107 trading clusters, we removed transactions that we believed were false positives in that they happened to have a similar value but were clearly the odd one out. For example, in a cluster of 20 transactions with 19 ETH-BTC transactions and one LTC-ZEC transaction, we removed the latter. We were thus left with clusters of either a particular pair (e.g., ETH-BTC) or two pairs where the curOut or the curIn was the same (e.g., ETH-BTC and ZEC-BTC), which suggests either the purchase of a rising coin or the dump of a declining one. We sought to further validate these clusters by using our heuristic from Section 7.2 to see if the clusters shared common addresses. While we typically did not find this in UTXO-based currencies (as most entities operate using many addresses), in account-based currencies we found that in almost every case there was one particular address that was involved in the trading cluster.

We summarize our results in Figure 6, in terms of the most common pairs of currencies and the total money exchanged by trading clusters using those currencies. It is clear that the most common interactions are performed between the most popular currencies overall, with the exception of Monero (XMR) and SALT. In particular, we found six clusters consisting of 17-20 transactions that exchanged BTC for XMR, and 13 clusters that exchanged BTC for SALT, an Ethereum-based token. The sizes of each trading cluster varied between 16 and 33 transactions and in total comprise 258 transactions, each of which shifted exactly 0.1 BTC. In total they originated from 514 different Bitcoin addresses, which may make it appear as though different people carried out these transactions. After applying our pass-through heuristic, however, we found that across all the transactions there were only two distinct SALT addresses used to receive the output. It is thus instead likely that this represents trading activity involving one or two entities.

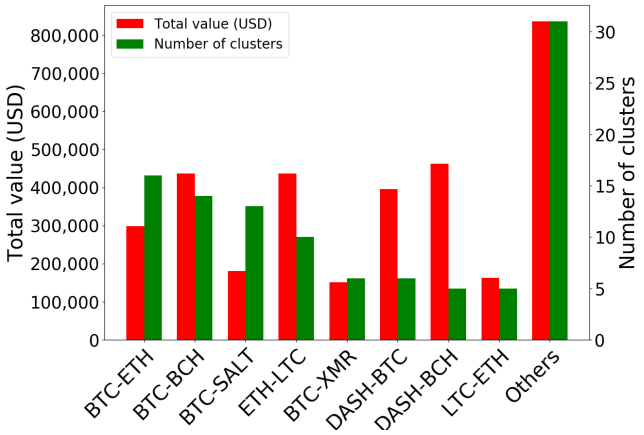


Figure 6: Our 107 clusters of likely trading bots, categorized by the pair of currencies they trade between and the total amount transacted by those clusters (in USD).

8.4 Usage of anonymity tools

Given the potential usage of ShapeShift for money laundering or other criminal activities, we sought to understand the extent to which its users seemed motivated to hide the source of their funds. While using ShapeShift is already one attempt at doing this, we focus here on the combination of using ShapeShift and so-called "privacy coins" (Dash, Monero, and Zcash) that are designed to offer improved anonymity guarantees.

In terms of the effect of the introduction of KYC into ShapeShift, the number of transactions using Zcash as curIn averaged 164 per day the month before, and averaged 116 per day the month after. We also saw a small decline with Zcash as curOut: 69 per day before and 43 per day after. Monero and Dash, however, saw much higher declines, and in fact saw the largest declines across all eight cryptocurrencies. The daily average the month before was 136 using Monero as curIn, whereas it was 47 after. Similarly, the daily average using it as curOut was 316 before and 62 after. For Dash, the daily average as curIn was 128 before and 81 after, and the daily average as curOut was 103 before and 42 after.

In terms of the blockchain data we had (according to the most popular currencies), our analysis in what follows is restricted to Dash and Zcash, although we leave an exploration of Monero as interesting future work.

8.4.1 Zcash

The main anonymity feature in Zcash is known as the *shielded pool*. Briefly, transparent Zcash transactions behave just like Bitcoin transactions in that they reveal in the clear the sender and recipient (according to so-called *t-addresses*), as well as the value being sent. This information is hidden to various degrees, however, when interacting with the pool. In particular, when putting money into the pool the recipient is specified using a so-called *z-address*, which hides the recipient but still

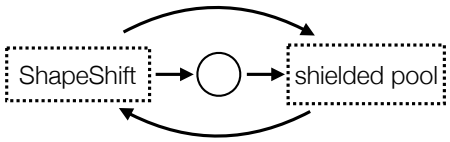


Figure 7: The three types of interactions we investigated between ShapeShift and the shielded pool in Zcash.

reveals the sender, and taking money out of the pool hides the sender (through the use of zero-knowledge proofs [2]) but reveals the recipient. Finally, Zcash is designed to provide privacy mainly in the case in which users transact *within* the shielded pool, which hides the sender, recipient, and the value being sent.

We considered three possible interactions between ShapeShift and the shielded pool, as depicted in Figure 7: (1) a user shifts coins directly from ShapeShift into the shielded pool, (2) a user shifts to a t-address but then uses that t-address to put money into the pool, and (3) a user sends money directly from the pool to ShapeShift.

For the first type of interaction, we found 29,003 transactions that used ZEC as curOut. Of these, 758 had a z-address as the output address, meaning coins were sent directly to the shielded pool. The total value put into the pool in these transactions was 6,707.86 ZEC, which is 4.3% of all the ZEC received in pass-through transactions. When attempting to use z-addresses in our own interactions with ShapeShift, however, we consistently encountered errors or were told to contact customer service. It is thus not clear if usage of this feature is supported at the time of writing.

For the second type of interaction, there were 1309 where the next transaction (i.e., the transaction in which this UTXO spent its contents) involved putting money into the pool. The total value put into the pool in these transactions was 12,534 ZEC, which is 8.2% of all the ZEC received in pass-through transactions.

For the third type of interaction, we found 111,041 pass-through transactions that used ZEC as curIn. Of these, 3808 came directly from the pool, with a total value of 22,490 ZEC (14% of all the ZEC sent in pass-through transactions).

Thus, while the usage of the anonymity features in Zcash was not necessarily a large fraction of the overall usage of Zcash in ShapeShift, there is clear potential to move large amounts of Zcash (representing over 10 million USD at the time it was transacted) by combining ShapeShift with the shielded pool.

8.4.2 Dash

As in Zcash, the "standard" transaction in Dash is similar to a Bitcoin transaction in terms of the information it reveals. Its main anonymity feature — *PrivateSend* transactions — are a type of CoinJoin [8]. A CoinJoin is specifically designed to invalidate the multi-input clustering heuristic described in Section 7, as it allows multiple users to come together and send coins to different sets of recipients in a single transaction. If each sender sends the same number of coins to their recipient, then it is difficult to determine which input address corresponds to which output address, thus severing the link between an individual sender and recipient.

In a traditional CoinJoin, users must find each other in some offline manner (e.g., an IRC channel) and form the transaction together over several rounds of communication. This can be a cumbersome process, so Dash aims to simplify it for users by automatically finding other users for them and chaining multiple mixes together. In order to ensure that users cannot accidentally de-anonymize themselves by sending uniquely identifiable values, these PrivateSend transactions are restricted to specific denominations: 0.01, 0.1, 1, and 10 DASH. As observed by Kalodner et al. [5], however, the CoinJoin denominations often contain a fee of 0.0000001 DASH, which must be factored in when searching for these transactions. Our parameters for identifying a CoinJoin were thus that (1) the transaction must have at least three inputs, (2) the outputs must consist solely of values from the list of possible denominations (modulo the fees), and (3) and all output values must be the same. In fact, given how Dash operates there is always one output with a non-standard value, so it was further necessary to relax the second and third requirements to allow there to be at most one address that does not carry the specified value.

We first looked to see how often the DASH sent to ShapeShift had originated from a CoinJoin, which meant identifying if the inputs of a Phase 1 transaction were outputs from a CoinJoin. Out of 100,410 candidate transactions, we found 2,068 that came from a CoinJoin, carrying a total of 11,929 DASH in value (6.5% of the total value across transactions with Dash as curIn). Next, we looked at whether or not users performed a CoinJoin after receiving coins from ShapeShift, which meant identifying if the outputs of a Phase 2 transaction had been spent in a CoinJoin. Out of 50,545 candidate transactions, we found only 33 CoinJoin transactions, carrying a total of 187 DASH in value (0.1% of the total value across transactions using Dash as curOut).

If we revisit our results concerning the use of U-turns in Dash from Section 6.2, we recall that there was a large asymmetry in terms of the results of our two heuristics: only 5.6% of the U-turns used the same UTXO, but 64.6% of U-turns used the same address. This suggests that some additional on-chain transaction took place between the two ShapeShift transactions, and indeed upon further inspection we identified many cases where this transaction was a CoinJoin. There thus appears to have been a genuine attempt to take advantage of the privacy that Dash offers, but this was completely ineffective due to the use of the same address that both sent and received the mixed coins.

9 Conclusions

In this study, we presented a characterization of the usage of the ShapeShift trading platform over a thirteen-month period, focusing on the ability to link together the ledgers of multiple different cryptocurrencies. To accomplish this task, we looked at these trading platforms from several different perspectives, ranging from the correlations between the transactions they produce in the cryptocurrency ledgers to the relationships they reveal between seemingly distinct users. The techniques we develop demonstrate that it is possible to capture complex transactional behaviors and trace their activity even as it moves across ledgers, which has implications for any criminals attempting to use these platforms to obscure their flow of money.

Acknowledgments

We would like to thank Bernhard Haslhofer and Rainer Stütz for performing the Bitcoin multi-input clustering using the GraphSense tool, and Zooko Wilcox, the anonymous reviewers, and our shepherd Matthew Green for their feedback. All authors are supported by the EU H2020 TITANIUM project under grant agreement number 740558.

References

[1] E. Androulaki, G. Karame, M. Roeschlin, T. Scherer, and S. Capkun. Evaluating user privacy in Bitcoin. In A.-R. Sadeghi, editor, *FC 2013*, volume 7859 of *LNCS*, pages 34–51, Okinawa, Japan, Apr. 1–5, 2013. Springer, Heidelberg, Germany.

[2] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, Berkeley, CA, USA, May 18–21, 2014. IEEE Computer Society Press.

[3] J. Dunietz. The Imperfect Crime: How the WannaCry Hackers Could Get Nabbed, Aug. 2017. https://www.scientificamerican.com/article/the-imperfect-crime-how-the-wannacry-hackers-could-get-nabbed/.

[4] A. Hinteregger and B. Haslhofer. Short paper: An empirical analysis of Monero cross-chain traceability. In *Proceedings of the 23rd International Conference on Financial Cryptography and Data Security (FC)*, 2019.

[5] H. Kalodner, S. Goldfeder, A. Chator, M. Möser, and A. Narayanan. Blocksci: Design and applications of a blockchain analysis platform, 2017. https://arxiv.org/pdf/1709.02489.pdf.

[6] G. Kappos, H. Yousaf, M. Maller, and S. Meiklejohn. An empirical analysis of anonymity in Zcash. In *Proceedings of the USENIX Security Symposium*, 2018.

[7] A. Kumar, C. Fischer, S. Tople, and P. Saxena. A traceability analysis of monero's blockchain. In S. N. Foley, D. Gollmann, and E. Snekkenes, editors, *ESORICS 2017, Part II*, volume 10493 of *LNCS*, pages 153–173, Oslo, Norway, Sept. 11–15, 2017. Springer, Heidelberg, Germany.

[8] G. Maxwell. Coinjoin: Bitcoin privacy for the real world. In *Post on Bitcoin forum*, 2013.

[9] R. McMillan. The Inside Story of Mt. Gox, Bitcoin's $460 Million Disaster, Mar. 2014. https://www.wired.com/2014/03/bitcoin-exchange/.

[10] S. Meiklejohn and C. Orlandi. Privacy-enhancing overlays in bitcoin. In M. Brenner, N. Christin, B. Johnson, and K. Rohloff, editors, *FC 2015 Workshops*, volume 8976 of *LNCS*, pages 127–141, San Juan, Puerto Rico, Jan. 30, 2015. Springer, Heidelberg, Germany.

[11] S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, and S. Savage. A fistful of bitcoins: characterizing payments among men with no names. In *Proceedings of the 2013 Internet Measurement Conference*, pages 127–140. ACM, 2013.

[12] M. Möser and R. Böhme. Anonymous alone? measuring Bitcoin's second-generation anonymization techniques. In *IEEE Security & Privacy on the Blockchain (IEEE S&B)*, 2017.

[13] M. Möser, K. Soska, E. Heilman, K. Lee, H. Heffan, S. Srivastava, K. Hogan, J. Hennessey, A. Miller, A. Narayanan, and N. Christin. An empirical analysis of linkability in the Monero blockchain. *Proceedings on Privacy Enhancing Technologies*, pages 143–163, 2018.

[14] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008. bitcoin.org/bitcoin.pdf.

[15] R. S. Portnoff, D. Y. Huang, P. Doerfler, S. Afroz, and D. McCoy. Backpage and Bitcoin: uncovering human traffickers. In *Proceedings of the ACM SIGKDD Conference*, 2017.

[16] J. Quesnelle. On the linkability of Zcash transactions. arXiv:1712.01210, 2017. https://arxiv.org/pdf/1712.01210.pdf.

[17] F. Reid and M. Harrigan. An analysis of anonymity in the Bitcoin system. In *Security and privacy in social networks*, pages 197–223. Springer, 2013.

[18] D. Ron and A. Shamir. Quantitative analysis of the full Bitcoin transaction graph. In A.-R. Sadeghi, editor, *FC 2013*, volume 7859 of *LNCS*, pages 6–24, Okinawa, Japan, Apr. 1–5, 2013. Springer, Heidelberg, Germany.

[19] D. Rushe. Cryptocurrency investors locked out of $190m after exchange founder dies, Feb. 2019. https://www.theguardian.com/technology/2019/feb/04/quadrigacx-canada-cryptocurrency-exchange-locked-gerald-cotten.

[20] J. Scheck and S. Shifflett. How dirty money disappears into the black hole of cryptocurrency, Sept. 2018. https://www.wsj.com/articles/how-dirty-money-disappears-into-the-black-hole-of-cryptocurrency-1538149743.

[21] M. Spagnuolo, F. Maggi, and S. Zanero. BitIodine: Extracting intelligence from the bitcoin network. In N. Christin and R. Safavi-Naini, editors, *FC 2014*, volume 8437 of *LNCS*, pages 457–468, Christ Church, Barbados, Mar. 3–7, 2014. Springer, Heidelberg, Germany.

[22] E. Voorhees. Announcing ShapeShift membership, Sept. 2018. https://info.shapeshift.io/blog/2018/09/04/introducing-shapeshift-membership/.

[23] H. Yousaf, G. Kappos, and S. Meiklejohn. Tracing transactions across cryptocurrency ledgers, Oct. 2018. https://arxiv.org/abs/1810.12786v1.

[24] Z. Yu, M. H. Au, J. Yu, R. Yang, Q. Xu, and W. F. Lau. New empirical traceability analysis of CryptoNote-style blockchains. In *Proceedings of the 23rd International Conference on Financial Cryptography and Data Security (FC)*, 2019.

Reading the Tea Leaves: A Comparative Analysis of Threat Intelligence

Vector Guo Li[1], *Matthew Dunn*[2], *Paul Pearce*[4], *Damon McCoy*[3],
Geoffrey M. Voelker[1], *Stefan Savage*[1], *Kirill Levchenko*[5]

[1] *University of California, San Diego* [2] *Northeastern University* [3] *New York University*
[4] *Georgia Institute of Technology* [5] *University of Illinois Urbana-Champaign*

Abstract

The term "threat intelligence" has swiftly become a staple buzzword in the computer security industry. The entirely reasonable premise is that, by compiling up-to-date information about known threats (i.e., IP addresses, domain names, file hashes, etc.), recipients of such information may be able to better defend their systems from future attacks. Thus, today a wide array of public and commercial sources distribute threat intelligence data feeds to support this purpose. However, our understanding of this data, its characterization and the extent to which it can meaningfully support its intended uses, is still quite limited. In this paper, we address these gaps by formally defining a set of metrics for characterizing threat intelligence data feeds and using these measures to systematically characterize a broad range of public and commercial sources. Further, we ground our quantitative assessments using external measurements to qualitatively investigate issues of coverage and accuracy. Unfortunately, our measurement results suggest that there are significant limitations and challenges in using existing threat intelligence data for its purported goals.

1 Introduction

Computer security is an inherently adversarial discipline in which each "side" seeks to exploit the assumptions and limitations of the other. Attackers rely on exploiting knowledge of vulnerabilities, configuration errors or operational lapses in order to penetrate targeted systems, while defenders in turn seek to improve their resistance to such attacks by better understanding the nature of contemporary threats and the technical fingerprints left by attacker's craft. Invariably, this means that attackers are driven to innovate and diversify while defenders, in response, must continually monitor for such changes and update their operational security practices accordingly. This dynamic is present in virtually every aspect of the operational security landscape, from anti-virus signatures to the configuration of firewalls and intrusion detection systems to incident response and triage. Common to all such reifications, however, is the process of monitoring for new data on attacker behavior and using that data to update defenses and security practices. Indeed, the extent to which a defender is able to gather and analyze such data effectively defines a de facto window of vulnerability—the time during which an organization is less effective in addressing attacks due to ignorance of current attacker behaviors.

This abstract problem has given rise to a concrete demand for contemporary threat data sources that are frequently collectively referred to as *threat intelligence* (**TI**). By far the most common form of such data are so-called *indicators of compromise:* simple observable behaviors that signal that a host or network may be compromised. These include both network indicators such as IP addresses (e.g., addresses known to launch particular attacks or host command-and-control sites, etc.) and file hashes (e.g., indicating a file or executable known to be associated with a particular variety of malware). The presence of such indicators is a symptom that alerts an organization to a problem, and part of an organization's defenses might reasonably include monitoring its assets for such indicators to detect and mitigate potential compromises as they occur.

While each organization naturally collects a certain amount of threat intelligence data on its own (e.g., the attacks they repel, the e-mail spam they filter, etc.) any single entity has a limited footprint and few are instrumented to carefully segregate crisp signals of attacks from the range of ambiguity found in normal production network and system logs. Thus, it is now commonly accepted that threat intelligence data procurement is a specialized activity whereby third-party firms, and/or collections of public groups, employ a range of monitoring techniques to aggregate, filter and curate quality information about current threats. Indeed, the promised operational value of threat intelligence has created a thriving (multi-billion dollar) market [43]. Established security companies with roots in anti-virus software or network intrusion detection now offer threat intelligence for sale, while some vendors specialize in threat intelligence exclusively, often promising coverage of more sophisticated threats than conventional sources.

Unfortunately, in spite of this tremendous promise, there has been little empirical assessment of threat intelligence data

or even a consensus about what such an evaluation would entail. Thus, consumers of TI products have limited means to compare offerings or to factor the cost of such products into any model of the benefit to operational security that might be offered.

This issue motivates our work to provide a grounded, empirical footing for addressing such questions. In particular, this paper makes the following contributions:

- We introduce a set of basic *threat intelligence metrics* and describe a methodology for measuring them, notably: **Volume**, **Differential Contribution**, **Exclusive Contribution**, **Latency**, **Coverage** and **Accuracy**.
- We analyze 47 distinct IP address TI sources covering six categories of threats and 8 distinct malware file hash TI sources, and report their metrics.
- We demonstrate techniques to evaluate the accuracy and coverage of certain categories of TI sources.
- We conduct the analyses in two different time periods two years apart, and demonstrate the strong consistency between the findings.

From our analysis, we find that while a few TI data sources show significant overlap, most do not. This result is consistent with the hypothesis advanced by [42] that different kinds of monitoring infrastructure will capture different kinds of attacks, but we have demonstrated it in a much broader context. We also reveal that underlying this issue are broader limitations of TI sources in terms of coverage (most indicators are unique) and accuracy (false positives may limit how such data can be used operationally). Finally, we present a longitudinal analysis suggesting that these findings are consistent over time.

2 Overview

The threat intelligence data collected for our study was obtained by subscribing to and pulling from numerous public and private intelligence sources. These sources ranged from simple blacklists of bad IPs/domains and file hashes, to rich threat intelligence exchanges with well labeled and structured data. We call each item (e.g., IP address or file hash) an *indicator* (after *indicator of compromise*, the industry term for such data items).

In this section we enumerate our threat intelligence sources, describe each source's structure and how we collected it, and then define our measurement metrics for empirically measuring these sources. When the source of the data is public, or when we have an explicit agreement to identify the provider, we have done so. However, in other cases, the data was provided on the condition of anonymity and we restrict ourself to describing the nature of the provider, but not their identity. All of our private data providers were appraised of the nature of our research, its goals and the methodology that we planned to employ.

2.1 Data Set and Collection

We use several sources of TI data for our analysis:

Facebook ThreatExchange (FB) [17]. This is a closed-community platform that allows hundreds of companies and organizations to share and interact with various types of labeled threat data. As part of an agreement with Facebook, we collected all its data that it shared broadly. In subsequent analyses, sources with prefix "FB" indicate a unique contributor on the Facebook ThreatExchange.

Paid Feed Aggregator (PA). This is a commercial paid threat intelligence data aggregation platform. It contains data collected from over a hundred other threat intelligence sources, public or private, together with its own threat data. In subsequent analyses all data sources with prefix "PA" are from unique data sources originating from this aggregator.

Paid IP Reputation Service. This commercial service provides an hourly-updated blacklist of known bad IP addresses across different attack categories.

Public Blacklists and Reputation Feeds. We collected indicators from public blacklists and reputation data sources, including well-known sources such as AlienVault [3], Badips [5], Abuse.ch [1] and Packetmail [28].

Threat Intelligence indicators include different types of data, such as IP address, malicious file hash, Domain, URL, etc. In this paper, we focus our analysis on sources that provide IP addresses and file hashes, as they are the most prevalent data types in our collection.

We collect data from all sources on an hourly basis. However, both the Facebook ThreatExchange and the Paid Feed Aggregator change their members and contributions over time, creating irregular collection periods for several of the sub-data sources. Similarly, public threat feeds had varying degrees of reliability, resulting in collection gaps. In this paper, we use the time window from **December 1, 2017** to **July 20, 2018** for most of the analyses, as we have the largest number of active sources during this period. We eliminated duplicates sources (e.g., sources we collected individually and also found in the Paid Aggregator) and sub-sources (a source that is a branch of another source). We further break IP sources into separate categories and treat them as individual feeds, as shown in Section 3. This filtering leaves us with 47 IP feeds and 8 malware file hash feeds.

The ways each TI source collects data varies, and in some cases the methodology is unknown. For example, Packetmail IPs and Paid IP Reputation collect threat data themselves via honeypots, analyzing malware, etc. Other sources, such as Badips or the Facebook ThreatExchange, collect their indicators from general users or organizations—e.g., entities may be attacked and submit the indicators to these threat intelligence services. These services then aggregate the data and report it to their subscribers. Through this level of aggregation the precise collection methodologies and data providence can be lost.

2.2 Data Source Structure

TI sources in our corpus structure and present data in different ways. Part of the challenge in producing cross-dataset metrics is normalizing both the structure of the data as well as its *meaning*. A major structural difference that influences our analysis occurs between data sources that provide data in *snapshots* and data sources that provide *events*.

Snapshot. Snapshot feeds provide periodic snapshots of a set of indicators. More formally, a snapshot is a set of indicators that is a function of time. It defines, for a given point in time, the set of indicators that are members of the data source. Snapshot feeds imply *state*: at any given time, there is a set of indicators that are *in* the feed. A typical snapshot source is a published list of IPs periodically updated by its maintainer. For example, a list of command-and-control IP addresses for a botnet may be published as a snapshot feed subject to periodic updates.

All feeds of file hashes are snapshots and are *monotonic* in the sense that indicators are only added, not removed, from the feed. Hashes are a proxy for the file content, which does not change (malicious file content will not change to benign in the future).

Event. In contrast, event feeds report newly discovered indicators. More formally, an event source is a set of indicators that is a function of a time *interval.* For a given time interval, the source provides a set of indicators that were seen or discovered in that time interval. Subscribers of these feeds query data by asking for new indicators added in a recent time window. For example, a user might, once a day, request the set of indicators that appeared in the last 24 hours.

This structural difference is a major challenge when evaluating feeds comparatively. We need to normalize the difference to make a fair comparison, especially for IP feeds. From a TI consumer's perspective, an *event* feed does not indicate when an indicator will expire, so it is up to the consumer to act on the age of indicators. Put another way, the expiration dates of indicators are decided by how users query the feed: if a user asks for the indicators seen in the last 30 days when quering data, then there is an implicit 30-day valid time window for these indicators.

In this paper, we choose a 30-day valid period for all the indicators we collected from event feeds—the same valid period used in several snapshot feeds, and also a common query window option offered by event feeds. We then convert these event feeds into snapshot feeds and evaluate all of them in a unified fashion.

2.3 Threat Intelligence Metrics

The aim of this work is to develop *threat intelligence metrics* that allow a TI consumer to compare threat intelligence sources and reason about their fitness for a particular purpose. To this end, we propose six concrete metrics: *Volume*, *Differential contribution*, *Exclusive contribution*, *Latency*, *Accuracy* and *Coverage*.

✧ **Volume.** We define the *volume* of a feed to be the total number of indicators appearing in a feed over the measurement interval. Volume is the simplest TI metric and has an established history in prior work [21,23,24,30,35,36,42]. It is also useful to study the daily *rate* of a feed, which quantifies the amount of data appearing in a feed on a daily basis.

Rationale: To a first approximation, volume captures how much information a feed provides to the consumer. For a feed without false positives (see *accuracy* below), and if every indicator has equal value to the consumer, we would prefer a feed of greater volume to a feed of lesser volume. Of course, indicators do not all have the same value to consumers: knowing the IP address of a host probing the entire Internet for decades-old vulnerabilities is less useful than the address of a scanner targeting organizations in your sector looking to exploit zero-day vulnerabilities.

✧ **Differential contribution.** The *differential contribution* of one feed with respect to another is the number of indicators in the first that are not in the second during the same measurement period. We define differential contribution relative to the size of the first feed, so that the differential contribution of feed A with respect to feed B is $\text{Diff}_{A,B} = |A \setminus B|/|A|$. Thus, $\text{Diff}_{A,B} = 1$ indicates that the two feeds have no elements in common, and $\text{Diff}_{A,B} = 0$ indicates that every indicator in A also appears in B. It is sometimes useful to consider the complement of differential contribution, namely the normalized *intersection* of A in B, given by $\text{Int}_{A,B} = |A \cap B|/|A| = 1 - \text{Diff}_{A,B}$.

Rationale: For a consumer, it is often useful to know how many *additional* indicators a feed offers relative to one or more feeds that the consumer has already. Thus, if a consumer already has feed A and is considering paying for feed B, then $\text{Diff}_{A,B}$ indicates how many new indicators feed A will provide.

✧ **Exclusive contribution.** The *exclusive contribution* of a feed with respect to a set of other feeds is the proportion of indicators unique to a feed, that is, the proportion of indicators that occur in the feed but no others. Formally, the exclusive contribution of feed A is defined as $\text{Uniq}_{A,B} = |A \setminus \bigcup_{B \neq A} B|/|A|$. Thus, $\text{Uniq}_{A,B} = 0$ means that every element of feed A appears in some other feeds, while $\text{Uniq}_{A,B} = 1$ means no element of A appears in any other feed.

Rationale: Like differential contribution, exclusive contribution tells a TI consumer how much of a feed is different. However, exclusive contribution compares a feed to all other feeds available for comparison, while differential contribution compares a feed to just another feed. From a TI consumer's perspective, exclusive contribution is a general measure of a feed's unique value.

✧ **Latency.** For an indicator that occurs in two or more feeds, its *latency* in a feed is the elapsed time between its first appearance in any feed and its appearance in the feed in question. In the feed where an indicator first appeared, its latency is zero. For all other feeds, the latency indicates how much later the

same indicators appears in those feeds. Taster's Choice [30] referred to latency as *relative first appearance time*. (We find the term *latency* to be more succinct without loss of clarity.) Since latency is defined for one indicator, for a feed it makes sense to consider statistics of the distribution of indicator latencies, such as the median indicator latency.

Rationale: Latency characterizes how quickly a feed includes new threats: the sooner a feed includes a threat, the more effective it is at helping consumers protect their systems. Indeed, several studies report on the impact of feed latency on its effectiveness at thwarting spam [10, 32].

The metrics above are defined without regard for the *meaning* of the indicators in a feed. We can calculate the volume of a single feed or the differential contribution of one feed with respect to another regardless of what the feed purports to contain. While these metrics are easy to compute, they do little to tell us about the fitness of a feed for a particular purpose. For this, we need to consider the meaning or purpose of the feed data, as advertised by the feed provider. We define the following two metrics.

✧ **Accuracy.** The *accuracy* of a feed is the proportion of indicators in a feed that are correctly included in the feed. Feed accuracy is analogous to *precision* in Information Retrieval. This metric presumes that the description of the feed is well-defined and describes a set of elements that should be in the feed given perfect knowledge. In practice, we have neither perfect knowledge nor a perfect description of what a feed should contain. In some cases, however, we can construct a set A^- of elements that should definitely not be in a feed A. Then $\text{Acc}_A \leq |A \setminus A^-|/|A|$.

Rationale: The accuracy metric tells a TI consumer how many false positives to expect when using a feed, and, therefore, dictates how a feed can be used. For example, if a consumer automatically blocks all traffic to IP addresses appearing in a feed, then false positives may cause disruption in an enterprise by blocking traffic to legitimate sites. On the other hand, consumers may tolerate some false positives if a feed is only used to gain additional insight during an investigation.

✧ **Coverage.** The *coverage* of a feed is the proportion of the intended indicators contained in a feed. Feed coverage is analogous to *recall* in Information Retrieval. Like accuracy, coverage presumes that the description of the feed is sufficient to determine which elements should be in a feed, given perfect knowledge. In some cases, it is possible to construct a set A^+ of elements that should be in a feed. We can then upper-bound the coverage $\text{Cov}_A \leq |A|/|A^+|$.

Rationale: For a feed consumer who aims to obtain complete protection from a specific kind of threat, coverage is a measure of how much protection a feed will provide. For example, an organization that wants to protect itself from a particular botnet will want to maximize its coverage of that botnet's command-and-control servers or infection vectors.

In the following two sections, we use these metrics to evaluate two types of TI: IP address feeds and file hash feeds.

3 IP Threat Intelligence

One of the most common forms of TI are feeds of IP addresses considered malicious, suspicious, or otherwise untrustworthy. This type of threat intelligence dates back at least to the early spam and intrusion detection blacklists, many of which are still active today such as SpamhausSBL [40], CBL [8] and SORBS [39]. Here, we apply the metrics described above to quantify the differences between 47 different IP address TI feeds.

3.1 Feed Categorization

IP address TI feeds have different meanings, and, therefore, purposes. To meaningfully compare feeds to each other, we first group feeds into *categories* of feeds whose indicators have the same intended meaning. Unfortunately, there is no standard or widely accepted taxonomy of IP TI feeds. To group feeds into semantic categories, we use metadata associated with the feed as well as descriptions of the feed provided by the producer, as described below.

Metadata. Some feeds provide category information with each indicator as metadata. More specifically, all of the Paid Aggregator feeds, Alienvault IP Reputation and Paid IP Reputation include this category metadata. In this case, we use its pre-assigned category in the feed. Facebook ThreatExchange feeds do not include category information in the metadata, but instead provide a descriptive phrase with each indicator. We then derive its category based on the description.

Feed description. For feeds without metadata, we rely on online descriptions of each feed, where available, to determine its semantic category. For example, the website of feed Nothink SSH [27] describes that the feed reports brute-force login attempts on its corresponding honeypot, which indicates the feed belongs to brute-force category.

We grouped our IP feeds into categories derived from the information above. In this work, we analyze six of the most prominent categories:

- **Scan:** Hosts doing port or vulnerability scans.
- **Brute-force:** Hosts making brute force login attempts.
- **Malware:** Malware C&C and distribution servers.
- **Exploit:** Hosts trying to remotely exploit vulnerabilities.
- **Botnet:** Compromised hosts belonging to a botnet.
- **Spam:** Hosts that sent spam or should not originate email.

Table 1 lists the feeds, grouped by category, used in the rest of this section. The symbols ○ and △ before the feed name indicate whether the feed is a snapshot feed or an event feed, respectively (see Section 2.2). All data was collected during our measurement period, **December 1st, 2017** to **July 20th, 2018**. Note that a few feeds, like Paid IP Reputation, appear in multiple categories. In these feeds, indicators are associated with different categories via attached metadata. We split these feeds into multiple virtual feeds each containing indicators belonging to the same category.

3.2 Volume

Volume is one of the oldest and simplest TI metrics representing how informative each data source is. Table 1 shows the total number of unique IP addresses collected from each feed during the measurement period, under column *Volume*. Feeds are listed in order of decreasing volume, grouped by category. The numbers we show are after the removal of invalid entries identified by the sources themselves. Column *Avg. Rate* shows the average number of new IPs we received per day, and *Avg. Size* lists the average daily working set size of each feed, that is, the average size of the snapshot.

✦ **Finding:** Feeds vary dramatically in volume. Within every category, big feeds can contain orders of magnitude more data than small feeds. For example, in the scan category, we saw over 361,004 unique IP addresses in DShield IPs but only 1,572 unique addresses in PA Analyst in the same time period. Clearly, volume is a major differentiator for feeds.

Average daily rate represents the amount of new indicators collected from a feed each day. Some feeds may have large volume but low daily rates, like Feodo IP Blacklist in the malware category. This means most indicators we get from that feed are old data present in the feed before our measurement started. On the other hand, the average rate of a feed could be greater than the volume would suggest, like Nothink SSH in the brute-force category. This is due to the fact that indicators can be added and removed multiple times in a feed. In general, IP indicators tend to be added in a feed only once: 37 among 47 IP feeds have over 80% of their indicators appearing only once, and 30 of them have this rate over 90%. One reason is that some snapshot feeds maintain a valid period for each indicator, as we found in all *PA* feeds where the expiration date of each indicator is explicitly recorded. When the same indicator is discovered again by a feed before its expiration time, the feed will just extend its expiration date, so this occurrence will not be captured if we simply subtract the old data from the newly collected data to derive what is added on a day. For event feeds and snapshot feeds in *PA* where we can precisely track every occurrence of each indicator, we further examed data occurrence frequency and still found that the vast majority of IPs in feeds only occurred once—an observation that relates to the dynamics of cyber threats themselves.

Nothink SSH, as we mentioned above, is a notable exception. It has over 64% of its indicators appearing 7 times in our data set. After investigating, we found that this feed posts all its previous data at the end of every month, behavior very likely due to the feed provider instead of the underlying threats.

The working set size defines the daily average amount of indicators users need to store in their system to use a feed (the storage cost of using a feed). The average working set size is largely decided by the valid period length of the indicators, controlled either by the feed (snapshot feeds) or the user (event feeds). The longer the valid period is, the larger the working set will be. Different snapshoot feeds have different choices for this valid period: PA AlienVault IPs in the scan category sets a 90-day valid period for every indicator added to the feed, while PA Abuse.ch Ransomware uses a 30-day period. Although we do not know the data expiration mechanism used by snapshot feeds other than *PA* feeds, as there is no related information recorded, we can still roughly estimate this by checking the *durations* of their indicators—the time

[1]This feed is aggregated by *PA* from Alienvault OTX, the Alienvault IP Reputation is the public reputation feed we collected from AlienVault directly. They are different feeds.

Table 1. IP TI feeds used in the study. A ○ denotes a *snapshot feed* and △ indicates an *event feed* (Section 2.2). *Volume* is the total number of IPs collected during our measurement period. *Exclusive* is the exclusive contribution of each feed (Section 3.4). *Avg. Rate* is the number of average daily new IPs added in the feed (Section 3.6), and *Avg. Size* is the average working set size of each feed (Section 3.2).

Feed	*Volume*	*Exclusive*	*Avg. Rate*	*Avg. Size*
Scan Feeds				
○ PA AlienVault IPs[1]	425,967	48.6%	1,359	128,821
△ DShield IPs	361,004	31.1%	1,556	69,526
○ PA Packetmail ramnode	258,719	62.0%	870	78,974
△ Packetmail IPs	246,920	48.6%	942	29,751
○ Paid IP Reputation	204,491	75.6%	1,362	8,756
○ PA Lab Scan	169,078	63.1%	869	9,775
○ PA Snort BlockList	19,085	96.3%	56	4,000
△ FB Aggregator$_1$	6,066	71.3%	24	693
○ PA Analyst	1,572	34.5%	6.3	462
Botnet Feeds				
○ PA Analyst	180,034	99.0%	697	54,800
○ PA CI Army	103,281	97.1%	332	30,388
○ Paid IP Reputation	77,600	99.9%	567	4,278
○ PA Botscout IPs	23,805	93.8%	81	7,180
○ PA VoIP Blacklist	10,712	88.0%	40	3,633
○ PA Compromised IPs	7,679	87.0%	21	2,392
○ PA Blocklist Bots	4,179	80.7%	16	1,160
○ PA Project Honeypot	2,600	86.5%	8.5	812
Brute-force Feeds				
△ Badips SSH	542,167	84.1%	2,379	86,677
△ Badips Badbots	91,553	70.8%	559	17,577
○ Paid IP Reputation	89,671	52.8%	483	3,705
○ PA Brute-Force	41,394	92.1%	138	14,540
△ Badips Username Notfound	37,198	54.2%	179	3662.8
△ Haley SSH	31,115	43.6%	40	1,224
△ FB Aggregator$_2$	22,398	77.3%	74	2,086
△ Nothink SSH	20,325	62.7%	224	12,577
△ Dangerrulez Brute	10,142	4.88%	37	1,102
Malware Feeds				
○ Paid IP Reputation	234,470	99.1%	1,113	22,569
△ FB Malicious IPs	30,728	99.9%	129	3,873
○ Feodo IP Blacklist	1,440	47.7%	1.3	1,159
○ PA Lab Malware	1,184	84.6%	3.5	366
△ Malc0de IP Blacklist	865	61.0%	2.9	86.6
○ PA Bambenek C2 IPs	785	92.1%	3.4	97.9
○ PA SSL Malware IPs	676	53.9%	2.9	84.0
○ PA Analyst	492	79.8%	2.1	149
○ PA Abuse.ch Ransomware	256	7.03%	1.6	117
○ PA Mal-Traffic-Anal	251	60.5%	0.9	72
○ Zeus IP Blacklist	185	49.1%	0.5	101
Exploit Feeds				
△ Badips HTTP	305,020	97.6%	1,592	22,644
△ Badips FTP	285,329	97.5%	1,313	27,601
△ Badips DNS	46,813	99.3%	231	4,758
△ Badips RFI	3,642	91.4%	16	104
△ Badips SQL	737	79.5%	4.4	99.2
Spam Feeds				
○ Paid IP Reputation	543,583	99.9%	3,280	6,551
△ Badips Postfix	328,258	90.5%	842	27,951
△ Badips Spam	302,105	89.3%	1,454	30,197
○ PA Botscout IPs	14,514	89.3%	49	4,390
○ Alienvault IP Reputation	11,292	96.6%	48	1,328

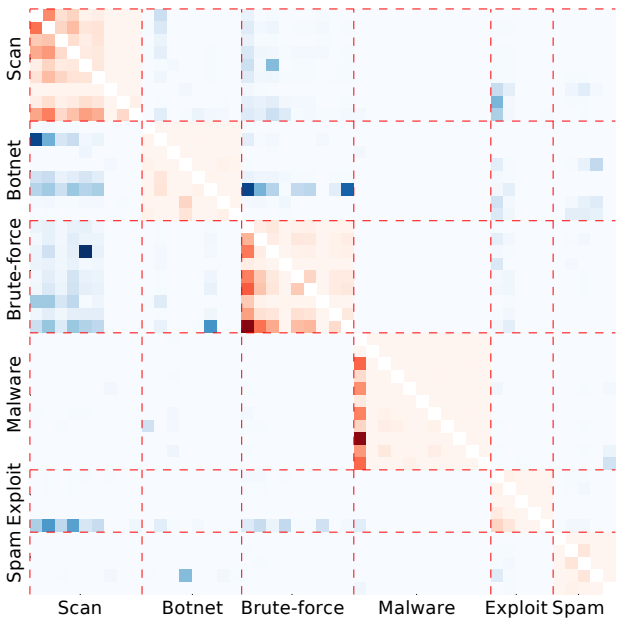


Figure 1. Feed intersection for all IP feeds. Each row/column represents a feed, shown in the same order as Table 1. Darker (more saturated) colors indicate greater intersection.

between an indicator being added and being removed. Four Paid IP Reputation feeds have more than 85% of durations shorter than 10 days, while the one in the malware category has more than 40% that span longer than 20 days. Feodo IP Blacklist has over 99% of its indicators valid for our entire measurement period, while over 70% of durations in the Zeus IP Blacklist are less than 6 days. We did not observe a clear pattern regarding how each snapshot feed handles the expiration of indicators.

3.3 Differential Contribution and Intersection

The differential contribution metric measures the number of indicators in one feed that are not in another. Equivalently, we can consider the intersection of two feeds, which is the number of elements in one feed that are present in the other, normalized by the size of the first: $|A \cap B|/|A|$. Figure 1 shows the intersection relationship of all feeds in the study. Each cell in the matrix represents the number of elements in both feeds, normalized by the size of the feed spanning the rows on the table. That is, A, in the expression above, ranges over rows, and B over columns of the matrix. Darker (more saturated) colors indicate greater intersection. Comparisons of feeds within a category are shaded red and comparisons of feeds between different categories are shaded blue. Note that the matrix is asymmetric, because, in general, $|A \cap B|/|A| \neq |A \cap B|/|B|$. Elements of the matrix are in the same order as in Table 1.

✦ **Finding:** Feeds in scan and brute-force categories have higher pairwise intersections: Half of the pairwise intersection rates in two categories are greater than 5%. The scan category has 29 out of 72 pairs (excluding self comparisons) with an intersection rate larger than 10%, and the same case occurred in 19 out of 72 pairs in the brute-force category.

On the other side, feeds in the botnet, exploit, malware and spam category do not share much data between each other: all 4 categories have more than three-quarters of pairwise intersection rates less than 1%. A few big feeds in these categories can share a significant amount of data with some small feeds in the same category—a characteristic that appears as a dark vertical line within its category in Figure 1. Paid IP Reputation in the malware category, for example, shares over 30% of 6 other malware feeds. But the intersections among the vast majority of feeds in these 4 categories are low. This finding is consistent with prior work [26, 42], but we provide a more comprehensive view regarding different categories.

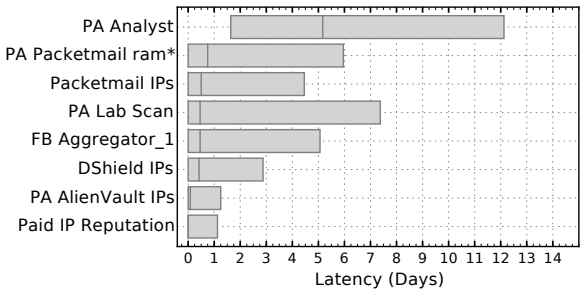
Figure 1 also shows the relation between feeds across different categories. We can clearly see a relation between scan and brute-force feeds: multiple scan feeds have non-trivial intersection with feeds in the brute-force category. In fact, 23.1% of all 760,263 brute-force IPs we collected are also included by scan feeds in our dataset. There are also three botnet feeds—PA CI Army, PA VoIP Blacklist and PA Compromised IPs—that have over 10% of its data shared with multiple feeds in the scan category.

3.4 Exclusive Contribution

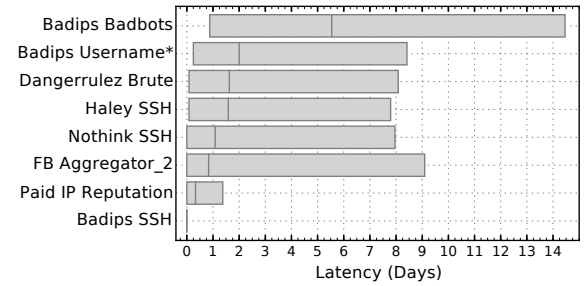
Exclusive contribution represents the number of indicators in a feed that are in no other feeds. We calculate each feed's exclusive contribution among all the feeds in the same category, emphasizing their uniqueness regarding the scope of data they claim to report. Each feed's exclusive contribution is presented in Table 1 in column *Exclusive*, calculated based on its volume.

✦ **Finding:** As we already observed in Section 3.3, botnet, exploit and spam feeds have relatively low pairwise intersections. Consequently, the feeds in these four categories have high exclusive contribution rates in general: the median exclusive contribution rates of these four categories are 90.9%, 97.5% and 90.5%, respectively. The malware category has a low median exclusive rate, since multiple small feeds have non-trivial intersection with the largest feed Paid IP Reputation, but the two largest feeds in malware both have a exclusive rate over 99%. Scan and brute-force feeds have more intersection within its category, and their exclusive rates are lower: 62.0% median rate in scan and 62.7% in brute-force, and the top two largest feeds in both categories have an exclusive rate below 85%.

If we assume a process where a feed is more likely to have popular elements, then smaller feeds would be subsumed by larger feeds. Yet, for some small feeds like Malc0de IP Blacklist in the malware and PA Project Honeypot in the botnet categories, even though they are several orders of magnitude smaller than the largest feeds in their categories, a significant



(a) Latency distribution in scan feeds



(b) Latency distribution in brute-force feeds

Figure 2. Distribution of indicators' latency in scan and brute-force feeds. Each box shows the latency distribution of shared IPs in the feed calculated in hours from 25 percentile to 75 percentile, with the middle line indicating the median. ("Badips Username*" here is the abbreviation for feed name Badips Username Notfound; "PA Packetmail Ram*" for PA Packetmail Ramnode)

proportion of their indicators is still unique to the feed. When we aggregate the data in each category, 73% of all scan feed indicators are unique to a single feed and 88% of brute force feed indicators are unique to one feed. For other categories, over 97% of elements in the category are unique to a single feed. This result agrees with previous work that most data in threat intelligence feeds is unique [26, 42].

3.5 Latency

Feed latency measures how quickly a feed reports new threat indicators. The sooner a feed can report potential threats, the more valuable it is for consumers. The absolute latency of an indicator in a feed is the time from the beginning of the corresponding event until when the indicator shows up in the feed. However, it is difficult to know the actual time when an event begins from the threat intelligence data. Instead, we measure the *relative latency*, which is the delay of an indicator in one feed to be the time between its appearance in that feed and the first seen among all the feeds.

Relative latency can only be calculated for indicators that occur in at least two feeds. As discussed in Section 3.4, the number of common indicators in the botnet, malware, exploit and spam feeds is very low (fewer than 3% of elements occur in more than one feed). Relative latency calculated for these feeds is less meaningful. For this analysis, therefore, we focus on scan and brute-force feeds.

Another issue is the time sensitivity of IP threats. An event that originated from an IP address, like scanning activity or a brute-force attack, will not last forever. If one scan feed reports an IP address today and another feed reports the same IP three months later, it would make little sense to consider them as one scanning event and label the second occurrence as being three months late. Unfortunately, there is no easy way we can clearly distinguish events from each other. Here we use a one-month window to restrict an event, assuming that the same attack from one source will not last for more than 30 days; although arbitrary, it provides a reasonably conservative threshold, and experimenting with other thresholds produced similar overall results. More specifically, we calculate relative latency by tracking the first occurrence of IPs in all feeds in a category, then recording the latency of the following occurrences while excluding ones that occur after 30 days. By just using the first appearance of each IP as the base, we avoid the uncertainty caused by multiple occurrence of indicators and different valid periods used among feeds.

Figures 2a and 2b show the relative latency distribution among feeds in the scan and brute-force categories, in hours. We focus on just those feeds that have over 10% of their data shared with others to ensure the analysis can represent the latency distribution of the overall feed. There is one feed in each category (PA Snort BlockList in scan and PA Brute-Force in brute-force) that is excluded from the figure.

✦ **Finding:** From the distribution boxes we can see that Paid IP Reputation in scan and Badips SSH in brute-force are the fastest feeds in their category, as they have the lowest median and 75th percentile latencies. On the other hand, PA Analyst in scan and Badips Badbots in brute-force are the slowest feeds. Figure 2a shows that all scan feeds except one have their 25th percentile latency equal to 0, indicating these feeds, across different sizes, all reported a significant portion of their shared data first. A similar case also happens in the brute-force category.

One may reasonably ask whether large feeds report data sooner than small feeds. The result shows that this is not always the case. FB Aggregator$_1$ is the second smallest feed in our scan category, yet it is no slower than several other feeds which have over 10 times of its daily rate. Badips Badbots, on the other hand, has the second largest rate in brute-force category, but it is slower than all the other feeds in the brute-force category. Feeds that are small in volume can still report a lot of their data first.

Another factor that could affect latency is whether feeds copy data from each other. For example, 93% of Dangerrulez Brute also appears in Badips SSH. If this is the case, we expect Dangerrulez Brute will be faster than Badips SSH on reporting their shared data. However, we compared the relative latency between just two feeds and found Badips SSH reported 88% of their shared indicators first. We further conducted this pairwise latency comparison between all feeds

Table 2. IP TI feeds accuracy overview. *Unrt* is fraction of unroutable addresses in each feed (Section 3.6). *Alexa Top* is the number of IPs intersected with top Alexa domain IP addresses, and *CDNs* is the number of IPs intersected with top CDN provider IP addresses.

Feed	*Added*	*Unrt*	*Alexa*	*CDNs*
Scan Feeds				
PA AlienVault IPs	313,175	0.0%	1	0
DShield IPs	339,805	0.03%	68	62
PA Packetmail ramnode	200,568	<0.01%	0	0
Packetmail IPs	211,081	0.0%	0	0
Paid IP Reputation	200,915	1.65%	6	21
PA Lab Scan	169,037	<0.01%	0	0
PA Snort BlockList	12,957	0.42%	1	0
FB Aggregator$_1$	5,601	0.0%	0	0
PA Analyst	1,451	0.41%	0	0
Botnet Feeds				
PA Analyst	180,034	<0.01%	0	0
PA CI Army	76,125	<0.01%	0	0
Paid IP Reputation	73,710	1.66%	6	74
PA Botscout IPs	18,638	0.09%	1	0
PA VoIP Blacklist	9,290	0.32%	0	0
PA Compromised IPs	4,883	0.0%	0	0
PA Blocklist Bots	3,594	0.0%	0	0
PA Project Honeypot	1,947	0.0%	0	0
Brute-force Feeds				
Badips SSH	456,605	0.19%	217	1
Badips Badbots	91,553	1.04%	46	1,251
Paid IP Reputation	87,524	0.03%	0	10
PA Brute-Force	31,555	0.0%	0	0
Badips Username Notfound	37,198	0.53%	4	0
Haley SSH	8,784	0.03%	0	0
FB Aggregator$_2$	17,779	0.0%	0	0
Nothink SSH	20,325	1.51%	2	0
Dangerrulez Brute	8,247	0.0%	0	0
Malware Feeds				
Paid IP Reputation	217,073	0.13%	291	3,489
FB Malicious IPs	29,840	2.14%	2	0
Feodo IP Blacklist	296	0.0%	0	0
PA Lab Malware	806	2.85%	0	0
Malc0de IP Blacklist	668	0.0%	8	11
PA Bambenek C2 IPs	777	9.13%	0	0
PA SSL Malware IPs	674	0.0%	0	0
PA Analyst	486	0.0%	0	0
PA Abuse.ch Ransomware	256	3.12%	0	0
PA Mal-Traffic-Anal	193	0.51%	0	0
Zeus IP Blacklist	67	0.0%	1	0
Exploit Feeds				
Badips HTTP	305,020	0.67%	16	2,590
Badips FTP	285,329	1.33%	14	2
Badips DNS	46,813	0.50%	119	244
Badips RFI	3,642	2.22%	0	0
Badips SQL	737	1.89%	0	1
Spam Feeds				
Paid IP Reputation	543,546	78.7%	1	0
Badips Spam	302,105	0.02%	19	0
Badips Postfix	193,674	1.29%	18	1
PA Botscout IPs	11,358	0.06%	0	0
Alienvault IP Reputation	10,414	0.07%	63	1,040

in scan, brute-force and malware (since Paid IP Reputation shares non-trivial amount of data with a few small feeds in the malware category), and did not see a clear latency advantage between any two feeds. Note that this observation does *not* prove there is no data copying, since the shared data between two feeds might partially come from copying and partially from the feeds' own data collection. Furthermore, our latency analysis is at a one-hour granularity.

3.6 Accuracy

Accuracy measures the rate of false positives in a feed. A false positive is an indicator that data is labeled with a category to which it does not belong. For example, an IP address found in a scan feed that has not conducted any Internet scanning is one such false positive. As well, even if a given IP is in fact associated with malicious activity, if it is not unambiguously actionable (e.g., Google's DNS at 8.8.8.8 is used by malicious and benign software alike) then for many use cases it must also be treated as a false positive. False positives are problematic for a variety of reasons, but particularly because they can have adverse operational consequences. For example, one might reasonably desire to block all new network connections to and from IP addresses reported as hosting malicious activity (indeed, this use is one of the promises of threat intelligence). False positives in such feeds, though, could lead to blocking legitimate connections as well. Thus, the degree of accuracy for a feed may preclude certain use cases.

Unfortunately, determining which IPs belong in a feed and which do not can be extremely challenging. In fact, at any reasonable scale, we are unaware of any method for unambiguously and comprehensively establishing "ground truth" on this matter. Instead, in this section we report on a proxy for accuracy that provides a conservative assessment of this question. To wit, we assemble a *whitelist* of IP addresses that either should not reasonably be included in a feed, or that, if included, would cause significant disruption. We argue that the presence of such IPs in a feed are clearly false positives and thus define an upper bound on a feed's accuracy. We populate our list from three sources: unroutable IPs, IPs associated with top Alexa domains, and IPs of major content distribution networks (CDNs).

Unroutable IPs. Unroutable IPs are IP addresses that were not BGP-routable *when they first appeared* in a feed, as established by contemporaneous data in the RouteViews service [44]. While such IPs could have appeared in the source address field of a packet (i.e., due to address spoofing), it would not be possible to complete a TCP handshake. Feeds that imply that such an interaction took place should not include such IPs. For example, feeds in the Brute-force category imply that the IPs they contain were involved in brute-force login attempts, but this could not have taken place if the IPs are not routable. While including unroutable addresses in a feed is not, in itself, a problem, their inclusion suggests a quality control issue with the feed, casting shade on the validity of other indicators in the feed.

To allow for some delays in the feed, we check if an IP was routable at any time in the seven days prior to its first appearance in a feed, and if it had, we do not count it as unroutable. Table 2, column *Unrt*, shows the fraction of IP indicators that were not routable at any time in the seven days prior to appearing in the feed. This analysis is only conducted for the IPs that are added after our measurement started. The number of such IPs is shown in column *Added*,

and the unroutable fraction shown in *Unrt* is with respect to this number.

Alexa. Blocking access to popular Internet sites or triggering alarms any time such sites are accessed would be disruptive to an enterprise. For our analysis, we periodically collected the Alexa top 25 thousand domains (3–4 times a month) over the course of the measurement period [2]. To address the challenge that such lists can have significant churn [33], we restrict our whitelist to hold the *intersection* of all these top 25K lists (i.e., domains that were in the top 25K every time we polled Alexa over our 8-month measurement period), which left us with 12,009 domains. We then queried DNS for the A records, NS records and MX records of each domain, and collected the corresponding IP addresses. In total, we collected 42,436 IP addresses associated with these domains. We compute the intersection of these IPs with TI feeds and show the results in column *Alexa* in Table 2.

CDNs. CDN providers serve hundreds of thousands of sites. Although these CDN services can (and are) abused to conduct malicious activities [9], their IP addresses are not actionable. Because these are fundamentally shared services, blocking such IP addresses will also disrupt access to benign sites served by these IPs. We collected the IP ranges used by 5 popular CDN providers: AWS CloudFron [12], Cloudflare [11], Fastly [18], EdgeCast [16] and MaxCDN [25]. We then check how many IPs in TI feeds fall into these ranges. Column *CDNs* in Table 2 shows the result.

✦ **Finding:** Among the 47 feeds in the table, 33 feeds have at least one unroutable IP, and for 13 of them, over 1% of the addresses they contain are unrouteable. Notably, the Paid IP Reputation feed in the spam category has an unroutable rate over 78%. Although it is not documented, a likely explanation is that this feed may include unroutable IPs intentionally, as this is a known practice among certain spam feeds. For example, the Spamhaus DROP List [41] includes IP address ranges known to be owned or operated by malicious actors, whether currently advertised or not. Thus, for feeds that explicitly do include unroutable IPs, their presence in the feeds should not necessarily be interpreted as a problem with quality control.

We further checked feeds for the presence of any "reserved IPs" which, as documented in RFC 8190, are not globally routable (e.g., private address ranges, test networks, loopback and multicast). Indeed, 12 feeds reported at least one reserved IP, including four of the Paid IP Reputation feeds (excepting the spam category), six of the Badips feeds, and the FB Malicious IPs and DShield IPs feeds. Worse, the Paid IP Reputation feeds together reported over 100 reserved IPs. Since such addresses should never appear on a public network, reporting such IPs indicates that a feed provider fails to incorporate some basic sanity checks on its data.

There are 21 feeds that include IPs from top Alexa domains, as shown in column *Alexa* in Table 2. Among these IPs there are 533 A records, 333 IPs of MX records and 63 IPs of NS records. The overlapped IPs include multiple instances from notable domains. For example, the IP addresses of www.github.com are included by Malc0de IP Blacklist. Paid IP Reputation in the malware category contains the IP address for www.dropbox.com. Alienvault IP Reputation contains the MX record of groupon.com, and Badips SSH also contains the IP addresses of popular websites such as www.bing.com.

Most of the feeds we evaluated do not contain IPs in CDN ranges, yet there are a few (including multiple Paid IP Reputation feeds, Badips feeds and Alienvault IP Reputation) that have significant intersection with CDN IPs. Alienvault IP Reputation and Badips feeds primarily intersect with Cloudflare CDN, while most of the overlap in the Paid IP Reputation malware category overlaps with AWS CloudFront.

Overall, the rate of false positives in a feed is not strongly correlated with its volume. Moreover, certain classes of false positives (e.g., the presence of Top Alex IPs or CDN IPs) seem to be byproducts of how distinct feeds are collected (e.g., Badips feeds tend to contain such IPs, irrespective of volume). Unsurprisingly, we also could find not correlation between a feed's latency and its accuracy.

3.7 Coverage

The coverage metric provides a quantitative measure of how well a feed captures the intended threat. A feed with perfect coverage would include all indicators that belong in a category. Unfortunately, as discussed above, there is no systematic way for evaluating the exact accuracy or coverage of a feed since it is unrealistic to obtain ground truth of all threat activities on the Internet.

However, there are some large-scale threat activities that are well-collected and well-studied. One example is Internet scanning. Researchers have long been using "Internet telescopes" to observe and measure network scanning activities [6, 15, 29]. With a large telescope and well-defined scan filtering logic, one can obtain a comprehensive view of global scanning activities on the Internet.

To this end, we collected three months of traffic (from January 1st to March 31st 2018) using the UCSD network telescope [38], which monitors a largely quiescent /8 network comprising over 16 million IP addresses. We then used the default parameters of the Bro IDS [7] to identify likely scanning traffic, namely flows in which the same source IP address is used to contact 25 unique destination IP addresses on the same destination port/protocol within 5 minutes. Given the large number of addressed being monitored, any indiscriminate scanner observed by TI feeds will likely also be seen in our data. Indeed, by intersecting against this telescope data we are able to partially quantify the coverage of each TI scanning feed.

The scanners we collected from the telescope consist of 20,674,149 IP addresses. The total number of IPs in all the scan feeds during this period is 425,286, which covers only 1.7% (363,799 shared IPs) of all the telescope scan IPs. On the

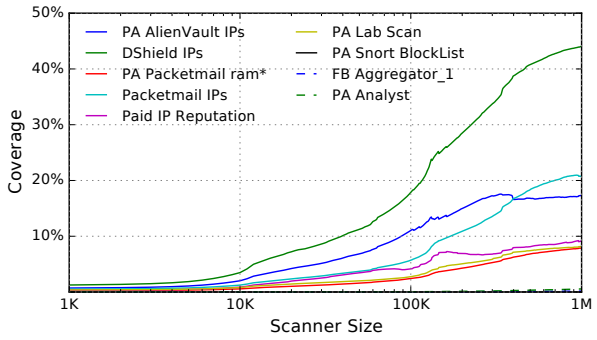


Figure 3. The coverage of each feed on different sizes of scanners. **Y** axis is the proportion of scanners of a given size or larger that are covered by each feed.

other hand, telescope scanners intersect with 85% of all IPs in scan feeds. When looking at each feed, PA AlienVault IPs, DShield IPs Packetmail IPs, PA Lab Scan and PA Packetmail ramnode all have over 85% of their data intersected with telescope scanners; the other four, though, have less than 65% of their data shared (and the rate for PA Snort BlockList is only 8%).

To further understand how well each scan feed detects scanning activities, we measure how different sizes of scanners in the telescope are covered by each feed. Here, *scanner size* means how many IPs a scanner has scanned in the telescope within a day. Figure 3 shows the coverage rate of each feed over different sizes of scanners, ranging from 1,000 to 1 million. (There are 7,212,218 scanners from the telescope whose sizes are over 1K, 271,888 that are over 100K and 17,579 are over 1 million.)

✦ **Finding:** The union of all the scan IPs in the feeds covers less than 2% of the scanners collected by the telescope. Even if we only look at the scanners with sizes larger than 10,000, the overall coverage is still around 10%, suggesting the coverage capability of scan feeds is very limited. The graph shows that, as the scanner size increases, the coverage of each feed over the datasets also increases, and large feeds cover more percent of telescope scanners than small feeds. This trend aligns with the intuition that scan feeds tend to capture more extensive scanners.

It is surprising that the small scan feeds in our collection have a smaller percentage of their IPs shared with telescope scanners. This contradicts the idea that small feeds would contain a larger percentage of extensive scanners (that would most likely also be observed by the telescope).

4 File Hash Threat Intelligence

File hashes in a threat intelligence feed are indicators for malicious files. It is one of the most lightweight ways to mark files as suspicious. One can incorporate this data to block malicious downloads, malicious email attachments, and malware. Likewise, file hashes can be used to whitelist applications and these feeds can be used to ensure malicious files do not appear in a customer's whitelist. In this section we present our analysis on eight file hash feeds, also collected from December 1st, 2017 to July 20th, 2018. We use the same metrics defined in Section 2.3.

The file hash feeds we collected use a range of different hash functions to specify malicious files, including MD5, SHA1, SHA256 and SHA512 (and some feeds provided values for multiple different hash functions to support interoperability). Since most indicators in our dataset are MD5s, we have normalized to this representation by using other feeds and the VirusTotal service to identify hash aliases for known malicious files (i.e., which MD5 corresponds to a particualr SHA256 value).

4.1 Volume

File hashes, unlike IP threat data, are not transient—a file does not change from malicious to benign—and thus a far simpler volume analysis is appropriate. We report volume as the number of new hashes that are added to each feed during our measurement period.

As seen in Table 3, we examine each feed's volume and average daily rate. Like IP feeds, file hash feeds also vary dramatically in volume. The majority of the hashes are concentrated in three feeds: FB Malware, PA Malware Indicators, and PA Analyst, which also exhibit the highest daily rates. The other feeds are multiple order of magnitude smaller comparatively.

4.2 Intersection and Exclusive Contribution

As we mentioned earlier, to conduct intersection and exclusive analysis of file hash feeds, we need to convert indicators into the same hash type. Here we convert non-MD5 hashes into MD5s, using either metadata in the indicator itself (i.e., if it reports values for multiple hash functions) or by querying the source hash from VirusTotal [45] which reports the full suite of hashes for all files in its dataset. However, for a small fraction of hashes we are unable to find aliases to conver them to the MD5 representation and must exclude them from the analysis in this section. This filtering is reflected in Table 3, in which the Volume column represents the number of unique hashes found in each feed and the Converted column is the subset that we have been able to normalize to a MD5 representation.

✦ **Finding:** The intersections between hash feeds are minimal, even among the feeds that have multiple orders of magnitude differences in size. Across all feeds, only PA Analyst has relatively high intersections: PA Analyst shares 27% of PA OSINT's MD5s and 13% of PA Twitter Emotet's MD5s. PA Malware Indicators has a small intersection also with these two feeds. All other intersections are around or less than 1%. Consequently, the vast majority of MD5s are unique to one feed, as recorded in column *Exclusive* in Table 3. The "lowest" exclusivity belongs to PA Twitter Emotet and PA OSINT (still

Table 3. File hash feeds overview. The second column group presents feed volume, average daily rate, the number of converted MD5s (Section 4.2) and exclusive proportion. *Not in VT* is fraction of hashes that are not found in VirusTotal, *Not det.* the fraction of hashes that are found in VirusTotal but are not labeled as malicious by any products, and *Detected* the fraction that are found in VirusTotal and are labeled malicious by at least one product. Column *Not in SD* shows the fraction of hashes in a feed that are not in Shadowserver Bin Check. *In NSRL* and *In AppInfo* show the absolute number of hashes found in Shadowserver (Section 4.3). *Exclusive* is based on the MD5-normalized hashes counted under *Converted*. All the other percentages in the table are based on *Volume*.

Feed	Volume	Avg. Rate	Converted	Exclusive	Not in VT	Not det.	Detected	Not in SD	In NSRL	In AppInfo
FB Malware	944,257	4,070	944,257	>99.99%	37.41%	50.50%	12.09%	99.89%	442	706
PA Malware Indicators	39,702	171	39,702	98.73%	0.02%	0.04%	99.94%	>99.99%	2	0
PA Analyst	38,586	166	37,665	97.97%	4.26%	2.82%	92.92%	99.95%	8	19
PA Twitter Emotet	1,031	4.44	960	77.29%	11.74%	0.78%	87.49%	99.81%	0	2
PA OSINT	829	3.57	783	71.65%	19.06%	0.84%	80.10%	99.88%	1	0
PA Sandbox	298	1.28	115	95.65%	72.81%	0.34%	26.85%	100%	0	0
PA Abuse.ch	267	1.15	3	100%	98.88%	0.75%	0.37%	100%	0	0
PA Zeus Tracker	17	0.07	17	100%	88.24%	5.88%	5.88%	100%	0	0

77.29% and 71.65%, respectively). All other feeds showcase an over 95% exclusive percentage, demonstrating that most file hash feeds are distinct from each other.

Due to the different sources of malware between feeds, a low intersection is to be expected in some cases. For example, PA Twitter Emotet and PA Zeus Tracker should have no intersection, since they are tracking different malware strains. The other, more general feeds could expect some overlap, but mostly exhibit little to no intersection. Considering the sheer volume of the FB Malware feed, one might expect it would encapsulate many of the smaller feeds or at least parts of them. This is not the case, however, as FB Malware has a negligible intersection with all other feeds.

Due to the lack of intersection among the feeds, we omit the latency analysis of the hash feeds, as there is simply not enough intersecting data to conclude which feeds perform better with regards to latency.

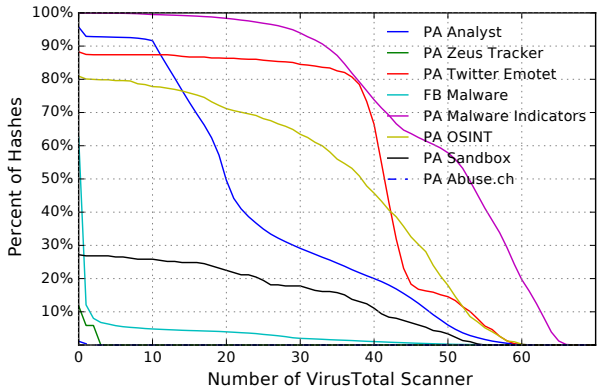


Figure 4. VirusTotal detection distribution. Each point means the proportion of indicators (Y value) in a feed that is detected by *over* X number of AV scanners in VirusTotal.

4.3 Accuracy

Assessing the accuracy of file hash feeds presents a problem: there is no universal ground truth to determine if a file is malicious or benign. Thus, to gauge the accuracy of the feeds, we use two metrics: a check for malicious hashes against VirusTotal, and a check for benign hashes against Shadowserver's Bin Check service. Note that all the percentages discussed below are based on the *Volume* of each feed.

4.3.1 VirusTotal

VirusTotal is a service that is often used when analyzing malware to get a base of information about a suspected file. Anyone can upload a file to be scanned. Upon submission, these files will be scanned by more than 70 antivirus scanners, which creates a report on how many antivirus scanners mark it malicious, among other information. In this analysis, we query VirusTotal for the hashes in each file hash feed and then inspect the percent of hashes that are marked as malicious and how many AV scanners have recorded them. Due to the high volume of the FB Malware feed and the query rate limit of VirusTotal, we randomly sampled 80,000 hashes from the feed for this analysis.

Table 3 shows a breakdown of the base detection rates for each feed from VirusTotal. As the PA feeds decrease in volume, the rates at which they are found in VirusTotal also decreases. The larger PA feeds have a much higher detection rate than their smaller counterparts. On the other hand, FB Malware only has 37% of its data detected by antivirus scanners and 50% in VirusTotal with no detection despite being the largest feed. This could indicate that FB Malware focuses on threats that specifically target Facebook and that are not as relevant to most VirusTotal users, such as malicious browser extensions [14, 20, 22]. This might undermine the limited coverage of VirusTotal as an oracle to detect targeted threats that are not of broader interest.

To further understand how the scanners in VirusTotal report the feed's data, we plot a graph of what percentage of hashes in each feed are detected by how many VirusTotal scanners. As seen in Figure 4, four feeds have more than 50% of their samples detected by over 20 scanners. PA Malware Indicators and PA Twitter Emotet did not experience a large detection drop before 35 scanners, indicating that most indicators in the

two feeds are popular malicious files recognized by many AV vendors. While PA Sandbox has a large percent of its hashes not presented in VirusTotal, over 70% of its samples that are detected are marked by over 20 AV scanners, showcasing a high confidence detection.

4.3.2 Shadowserver

To more fully gauge the accuracy of the file hash feeds, we also examined how each feed measured against Shadowserver's Bin Check Service [34]. The service checks file hashes against NIST's National Software Registry List (NSRL) in addition to Shadowserver's own repository of known software. Table 3 details how each feed compares with Shadowserver's Bin Check service.

It might be expected that there would be no hash found with Shadowserver's Bin Check service, but it is not the case. Some of the samples from the feeds that appear in Shadowserver are well known binaries such as versions of Microsoft Office products, Window's Service Packs, calc.exe, etc. In the event malware injects itself into a running process, it remains plausible that some of these well-known binaries find their way into TI feeds from users wrongly attributing maliciousness. While FB Malware has over one thousand hashes in Shadowserver, this is not a widespread issue, as all feeds have <1% of their hashes contained within Shadowserver's Bin Check service. This showcases that while there are a few exceptions, the feeds mostly do not contain well-known, benign files.

✦ **Finding:** Each PA feed has a negligible rate of occurrence within Shadowserver regardless of their VirusTotal detection, showing they do not contain generic false positives. Larger feeds exhibit high VirusTotal detection rates except for FB Malware, while small feeds have relatively low detection rates. This suggests that small hash feeds might focus more on specific malicious files that are not widely known. FB Malware has a low VirusTotal occurrence despite its size and has over one thousand hashes in Shadowserver, but its overall low percentage of hashes within Shadowserver indicates that it does not contain many known files and might have threats not typically recognized by VirusTotal's scanners.

5 Longitudinal Comparison

In addition to the measurement period considered so far (December 1, 2017 to July 20, 2018), we also analyzed data from the same IP feeds from January 1, 2016 to August 31, 2016. These two measurement periods, 23 months apart, allow us to measure how these IP feeds have changed in two years. Table 4 summarizes the differences between these two measurement periods. In the table, *2018* represents the current measurement period and *2016* the period January 1, 2016 to August 31, 2016.

Volume. As shown in Table 4, feed volume has definitely changed after two years. Among 43 IP feeds that overlap both time periods, 21 have a higher daily rate compared with 2 years ago, 15 feeds have a lower rate, and 7 feeds do not change substantially (the difference is below 20%). Volume can change dramatically over time, such as PA AlienVault IPs in the scan category which is 13 times larger than before. On the other hand, a feed like PA Blocklist Bots is now over 90% smaller.

Intersection and Exclusive Contribution. Despite the volume differences, the intersection statistics between feeds are largely the same across two years, with feeds in scan and brute-force having high pairwise intersections and feeds in other categories being mostly unique. Certain specific pairwise relations also did not change. For example, Badips SSH still shared over 90% of data in Dangerrulez Brute back in

Table 4. Data changes in IP feeds compared against the ones in 2016, *Avg. Rate* shows the percentage of daily rate changed over the old feeds. The two columns under *Unrt* show the unroutable rates of feeds in 2016 and 2018 separately. The two columns under *CDN* present the number of IPs fall in CDN IP ranges in old and new data.

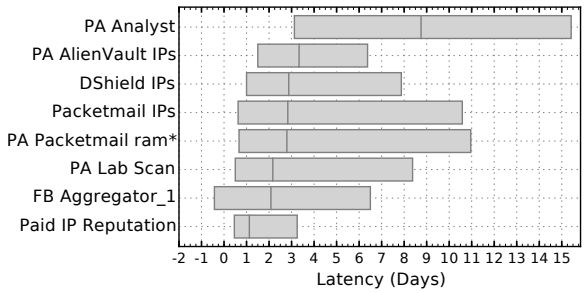
		Unroutable		*CDN*	
Feed	*Avg. Rate*	2016	2018	2016	2018
Scan Feeds					
PA AlienVault IPs	+1,347%	0.0%	0.0%	0	0
PA Packetmail ram*	+733%	<0.01%	<0.01%	0	0
Packetmail IPs	+135%	0.0%	0.0%	0	0
Paid IP Reputation	−57%	8.73%	1.65%	910	21
PA Lab Scan	−1%	0.0%	<0.01%	0	0
PA Snort BlockList	−97%	<0.01%	0.42%	1	0
FB Aggregator$_1$	+332%	0.0%	0.0%	6	0
PA Analyst	−44%	0.0%	0.41%	0	0
Botnet Feeds					
PA CI Army	+114%	<0.01%	<0.01%	0	0
Paid IP Reputation	−39%	0.63%	1.66%	15	74
PA Botscout IPs	+1%	0.01%	0.09%	1	0
PA VoIP Blacklist	+252%	0.0%	0.32%	0	0
PA Compromised IPs	−36%	0.10%	0.0%	0	0
PA Blocklist Bots	−95%	0.0%	0.0%	0	0
PA Project Honeypot	+63%	0.0%	0.0%	0	0
Brute-force Feeds					
Badips SSH	+30%	0.07%	0.19%	0	1
Badips Badbots	+1,732%	0.0%	1.04%	187	1,251
Paid IP Reputation	−62%	6.55%	0.03%	335	10
PA Brute-Force	−72%	0.0%	0.0%	0	0
Badips Username*	+3,040%	0.0%	0.53%	0	0
Haley SSH	+428%	0.04%	0.03%	0	0
FB Aggregator$_2$	+387%	0.12%	0.0%	0	0
Nothink SSH	+886%	0.56%	1.51%	0	0
Dangerrulez Brute	+0%	0.0%	0.0%	1	0
Malware Feeds					
Paid IP Reputation	−36%	0.18%	0.13%	15265	3,489
FB Malicious IPs	−77%	6.81%	2.14%	264	0
Feodo IP Blacklist	+0%	0.0%	0.0%	0	0
Malc0de IP Blacklist	−9%	0.0%	0.0%	132	11
PA Bambenek C2 IPs	+79%	0.0%	9.13%	0	0
PA SSL Malware IPs	−34%	0.0%	0.0%	0	0
PA Analyst	−93%	0.34%	0.0%	0	0
PA Abuse.ch*	−99%	0.49%	3.12%	0	0
PA Mal-Traffic-Anal	−53%	0.0%	0.51%	0	0
Zeus IP Blacklist	−66%	0.0%	0.0%	6	0
Exploit Feeds					
Badips HTTP	+326%	0.30%	0.67%	436	2,590
Badips FTP	+556%	0.01%	1.33%	0	2
Badips DNS	+9,525%	0.17%	0.50%	7	244
Badips RFI	+226%	0.0%	2.22%	0	0
Spam Feeds					
Paid IP Reputation	+133%	59.3%	78.7%	0	0
Badips Spam	+12,767%	0.0%	0.02%	0	0
Badips Postfix	−53%	<0.01%	1.29%	0	1
PA Botscout IPs	+18%	0.0%	0.06%	0	0
AlienVault IP Rep	+8%	0.57%	0.07%	479	1,040

2016, and Paid IP Reputation in malware was still the only feed that has a non-trivial intersection with multiple small feeds. Again, most data was exclusive to each feed two years ago: Across all six categories more than 90% of the indicators are not shared between feeds.

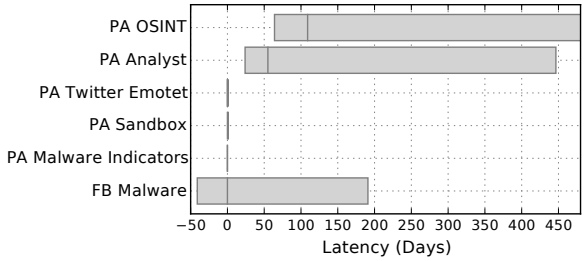
Latency. The latency relationship between feeds was also similar: timely feeds today were also timely two years ago, and the same with tardy feeds.

Accuracy. Feeds have more unroutable IPs now than before as shown in Table 4: In 2016, 22 of the 43 IP feeds had at least 1 unroutable IP; four feeds had unroutable rates over 1%. When checking the intersection with popular CDNs, the feeds that contain IPs in CDN ranges two years ago are also the ones that have these IPs today.

Shared indicators 2016–2018. We compared the data we collected from each feed in the two time periods, and found that 30 out of 43 feeds in 2018 intersect with their data from two years ago, and 9 feeds have an intersection rate over 10%. Three feeds in malware category, namely Feodo IP Blacklist, PA Abuse.ch Ransomware and Zeus IP Blacklist, have over 40% of their data shared with the past feed, meaning a large percent of C&C indicators two years ago are still identified by the feeds as threats today. Feeds in the botnet category, however, are very distinct from the past, with all feeds having no intersection with the past except Paid IP Reputation.



(a) Latency distribution in scan feeds relative to the Internet telescope



(b) Latency distribution in file hash feeds relative to VirusTotal

Figure 5. Distribution of indicators' latency in scan and file hash feeds. Note that the scan feeds' distribution are calculated in hour granularity while the file hash feeds' distribution are calculated in day granularity.

6 Absolute Latency

We defined our latency metric in this paper as relative latency between TI sources, since it is easy to compute and allows consumers to compare feeds to each other on this aspect. However, it is also critical to know about the absolute latency distribution of indicators. Absolute latency represents how fast a feed can actually report a threat, which directly decides the effectiveness of the data when used in a pro-active way. As we already discussed in Section 3.5, absolute latency is hard to measure, as we do not have ground truth of the underlying threat.

In Section 3.7, we used an Internet telescope as our approximation for ground truth to measure the coverage of scan feeds. In Section 4.3, we used VirusTotal as an oracle to measure the accuracy of file hash feeds. Although these sources are not real ground truth and it is unclear how far away they are, these large and well-managed sources can help us, to a certain extent, profile the performance of TI feeds. In this section, we use these two sources again to approximate the absolute latency of indicators in scan IP feeds and malicious file hash feeds.

More specifically, we measure the latency of IPs in scan feeds relative to the first occurrence time of the same IP in the scanners collected from the telescope. Considering the massive size of the telescope, it should presumably detect scanners much sooner after the scanning event actually happened. We measure latency of file hashes relative to the `first_seen` timestamps queried from VirusTotal. The `first_seen` timestamp represents the time when the corresponding file is first uploaded to VirusTotal. VirusTotal is a very popular service and it is a convention for many security experts to upload new malware samples to VirusTotal once they discovered them. Therefore, this timestamp roughly entails when the security community first noticed the malicious file and can be a good approximation for absolute latency.

Figure 5 show the latency distribution of each feed, using the same plotting convention as in Section 3.5. Some feeds are not shown in the figure as there are too little data points in those feeds to reason about distribution.

✦ **Finding:** Comparing Figure 5a to Figure 2a, we can see that the median latency of feeds are all larger. This is consistent with our assumption that a large sensor tends to receive indiscriminate scanners sooner. Scan feeds' median lantecy are one to three days relative to the Internet telescope, except PA Analyst, whose median latency is almost nine days. The order of median latency between feeds changed compared with Figure 2a, but since the original relative median latencies among scan feeds are very close, the new order here is more likely to be statistics variances. Also, note that although the PA AlienVault IPs seems much slower than it is in Figure 2a, its 75 percentile latency is still the second smallest one.

On the other hand, the latency distributions of hash feeds vary more dramatically. PA Malware Indicators, PA Sandbox and PA Twitter Emotet are almost as fast as VirusTotal: all three feeds have 25 percentile and median latency equal to

zero. PA OSINT and PA Analyst are comparatively much slower, and PA OSINT even has a 75 percentile latency of 1680 days. This might be because of the heterogeneous nature of malware feeds. The figure also shows that feed volumes do not imply their latency, as PA Analyst and FB Malware are much slower than the small hash feeds.

Figure 5 demonstrates that the Internet telescope and VirusTotal are indeed good approximations for absolute latency measurement, as most indicators in **TI** feeds are observed relatively later. However, every scan feed has over 2% of its indicators detected earlier than the telescope did. FB Aggregator$_1$ and DShield IPs even have over 10% of their indicators observed earlier. There is also a similar case in file hash feeds. This aligns with our observation in Section 3.5 that small feeds can still report a non-trivial amount of their data first. Another interesting observation is that both Facebook feeds, FB Aggregator$_1$ and FB Malware, have a large percent of their data observed earlier than the telescope or VirusTotal. This again suggests that Facebook (and its threat intelligence partners) might face more targeted threats, so those threats will be first observed by Facebook.

7 Discussion

7.1 Metrics Usage

Threat intelligence has many different potential uses. For example, analysts may consume threat data interactively during manual incident investigations, or may use it to automate the detection of suspicious activity and/or blacklisting. When not itself determinative, such information may also be used to *enrich* other data sources, informing investigations or aiding in automatic algorithmic interventions. We have introduced a set of basic threat intelligence metrics—volume, intersection, unique contribution, latency, coverage and accuracy—that can inform and quantify each of those uses. Depending on a number of factors, such as the intended use case and the cost of false positives and negatives, some of these metrics will become more or less important when evaluating a **TI** source. For example, a feed with poor accuracy but high coverage might be ideal when an analyst is using a **TI** source interactively during manually incident investigations (since in this case, the analyst, as a domain expert, can provide additional filtering of false positives). Similarly, latency might not be a critical metric in a retrospective use case (e.g., post-discovery breach investigation). However, if an organization is looking for a **TI** source where the IPs are intended to be added to a firewall's blacklist then accuracy and latency should likely be weighted over coverage, assuming that blocking benign activity is more costly.

Another common real-world scenario is that a company has a limited budget to purchase **TI** sources and has a specific set of threats (i.e., botnet, brute-force) they are focused on mitigating. In such cases, the metrics we have described can be used directly in evaluating **TI** options, biasing towards sources that maximize coverage of the most relevant threats while limiting intersection.

7.2 Data Labeling

Threat intelligence IP data carries different meanings. To properly use this data, it is critical to know what the indicators actually mean: whether they are Internet scanners, members of a botnet or malicious actors who had attacked other places before. We have attempted to group feeds by their intended meaning in our analysis.

However, this category information, which primarily comes from **TI** sources themselves, is not always available. Feeds such as Alienvault IP Reputation and Facebook Threat Exchange sources contain a significant number of indicators labeled "Malicious" or "Suspicious." The meanings of these indicators are unclear, making it difficult for consumers to decide how to use the data and the possible consequences.

For feeds that provide category information, it is sometimes too broad to be meaningful. For example, multiple feeds in our collection simply label their indicators as "Scanner." Network scanning can represent port scanning (by sending SYN packets), or a vulnerability scan (by probing host for known vulnerabilities). The ambiguity here, as a result of ad-hoc data labeling, again poses challenges for security experts when using **TI** data.

Recently, standard **TI** formats have been proposed and developed, notably IODEF [19], CybOX [13] and STIX [37], that try to standardize the threat intelligence presentation and sharing. But these standards focus largely on the data format. There is room to improve these standards by designing a standard *semantics* for threat intelligence data.

7.3 Limitations

There are several questions that our study does not address. We attempted to collect data from a diverse set of sources, including public feeds, commercial feeds and industrial exchange feeds, but it is inherently not comprehensive. There are some prohibitively expensive or publication-restricted data sources that are not available to us. More specialized measurement work should be done in the future to further analyze the performance of these expensive and exclusive data sources.

A second limitation is our visibility into how different companies use threat intelligence operationally. For a company, perhaps the most useful kind of metric measures how a threat intelligence source affects its main performance indicators as well as its exposure to risk. Such metrics would require a deep integration into security workflows at enterprises to measure the operation effect of decisions made using threat intelligence. This would allow CIOs and CSOs to better understand exactly what a particular threat intelligence product contributes to a company. As researchers, we do not use **TI** operationally. A better understanding of operational needs would help refine our metrics to maximize their utility for operations-driven consumers.

The third limitation is the lack of ground truth, a limitation shared by all the similar measurement work. It is simply very difficult to obtain the full picture of a certain category of threat, making it very challenging to precisely determine accuracy and coverage of feeds. In this study, we used data from an Internet telescope and VirusTotal as a close approximation. There are also a handful of cases where a security incident has been comprehensively studied by researchers, such as the Mirai study [4], and such efforts can be used to evaluate certain types of TI data. But such studies are few in number. One alternative is to try to establish the ground truth for a specific network. For example, a company can record all the network traffic going in and out of its own network, and identify security incidents either through its IDS system or manual forensic analysis. Then it can evaluate the accuracy and coverage of a TI feed under the context of its own network. This can provide a customized view of TI feeds.

8 Related Work

Several studies have examined the effectiveness of blacklist-based threat intelligence [23, 31, 32, 35, 36]. Ramachandran *et al.* [32] showed that spam blacklists are both incomplete (missing 35% of the source IPs of spam emails captured in two spam traps), and slow in responding (20% of the spammers remain unlisted after 30 days). Sinha *et al.* [36] further confirmed this result by showing that four major spam blacklists have very high false negative rates, and analyzed the possible causes of the low coverage. Sheng *et al.* [35] studied the effectiveness of phishing blacklists, showing the lists are slow in reacting to highly transient phishing campaigns. These studies focused on specific types of threat intelligence sources, and only evaluated their operational performance rather than producing empirical evaluation metrics for threat intelligence data sources.

Other studies have analyzed the general attributes of threat intelligence data. Pitsillidis *et al.* [30] studied the characteristics of spam domain feeds, showing different perspectives of spam feeds, and demonstrated that different feeds are suitable for answering different questions. Thomas *et al.* [42] constructed their own threat intelligence by aggregating the abuse traffic received from six Google services, showing a lack of intersection and correlation among these different sources. While focusing on broader threat intelligence uses, these studies did not focus on generalizable threat metrics that can be extended beyond the work.

Little work exists that defines a general measurement methodology to examine threat intelligence across a broad set of types and categories. Metcalf *et al.* [26] collected and measured IP and domain blacklists from multiple sources, but only focused on volume and intersection analysis. In contrast, we formally define a set of threat intelligence metrics and conduct a broad and comprehensive study over a rich variety of threat intelligence data. We conducted our measurement from the perspective of consumers of TI data to offer guidance on choosing between different sources. Our study also demonstrated the limitation of threat intelligence more thoroughly, providing comprehensive characteristics of cyber threat intelligence that no work had addressed previously.

9 Conclusion

This paper has focused on the simplest, yet fundamental, metrics about threat intelligence data. Using the proposed metrics, we measured a broad set of TI sources, and reported the characteristics and limitations of TI data. In addition to the individual findings mentioned in each section, here we highlight the high-level lessons we learned from our study:

- TI feeds, far from containing homogeneous samples of some underlying truth, vary tremendously in the kinds of data they capture based on the particularities of their collection approach. Unfortunately, few TI vendors explain the mechanism and methodology by which their data are collected and thus TI consumers must make do with simple labels such as "scan" or "botnet", coupled with inferences about the likely mode of collection. Worse, a significant amount of data does not even have a clear definition of category, and is only labelled as "malicious" or "suspicious", leaving the ambiguity to consumers to decide what action should be taken based on the data.

- There is little evidence that larger feeds contain better data, or even that there are crisp quality distinctions between feeds across different categories or metrics (i.e., that a TI provider whose feed performs well on one metric will perform well on another, or that these rankings will hold across threat categories). How data is collected also does not necessarily imply the feeds' attributes. For example, crowdsourcing-based feeds (e.g., Badips feeds), are not always slower in reporting data than the self-collecting feeds (like Paid IP Reputation).

- Most IP-based TI data sources are collections of singletons (i.e., that each IP address appears in at most one source) and even the higher-correlating data sources frequently have intersection rates of only 10%. Moreover, when comparing with broad sensor data in known categories with broad effect (e.g., random scanning) fewer than 2% of observed scanner addresses appear in most of the data sources we analyzed; indeed, even when focused on the largest and most prolific scanners, coverage is still limited to 10%. There are similar results for file hash-based sources with little overlap among them.

The low intersection and coverage of TI feeds could be the result of several non-exclusive possibilities. First is that the underlying space of indicators (both IP addresses and malicious file hashes) is large and each individual data source can at best sample a small fraction thereof. It is almost certain that this is true to some extent. Second, different collection

methodologies—even for the same threat category—will select for different sub distributions of the underlying ground truth data. Third, this last effect is likely exacerbated by the fact that not all threats are experienced uniformly across the Internet and, thus, different methodologies will skew to either favor or disfavor targeted attacks.

Based on our experience analyzing **TI** data, we try to provide several recommendations for the security community on this topic moving forward:

- The threat intelligence community should standardize data labeling, with a clear definition of what the data means and how the data is collected. Security experts can then assess whether the data fit their need and the type of action should be taken on this data.
- There are few rules of thumb in selecting among **TI** feeds, as there is not a clear correlation between different feed properties. Consumers need empirical metrics, such as those we describe, to meaningfully differentiate data sources, and to prioritize certain metrics based on their specific need.
- Blindly using **TI** data—even if one could afford to acquire many such sources—is unlikely to provide better coverage and is also prone to collateral damage caused by false positives. Customers need to be always aware of these issues when deciding what action should be taken on this data.
- Besides focusing on the **TI** data itself, future work should investigate the operational uses of threat intelligence in industry, as the true value of **TI** data can only be understood in operational scenarios. Moreover, the community should explore more potential ways of using the data, which will extend our understanding of threat intelligence and also influence how vendors are curating the data and providing the services.

There are many ways we can use threat intelligence data. It can be used to *enrich* other information (e.g., for investigating potential explanations of a security incident), as a probabilistic canary (i.e., identifying an overall site vulnerability via a single matching indicator may have value even if other attacks of the same kind are not detected) or in providing a useful source of ground truth data for supervised machine learning systems. However, even given such diverse purposes, organizations still need some way to prioritize which **TI** sources to invest in. Our metrics provide some direction for such choices. For example, an analyst who expects to use **TI** interactively during incident response would be better served by feeds with higher coverage, but can accommodate poor accuracy, while an organization trying to automatically label malicious instances for training purposes (e.g., brute force attacks) will be better served by the converse. Thus, if there is hope for demonstrating that threat intelligence can materially impact operational security practices, we believe it will be found in these more complex uses cases and that is where future research will be most productive.

10 Acknowledgment

We would like to thank our commercial threat providers who made their data available to us and made this research possible. In particular, we would like to thank Nektarios Leontiadis and the Facebook ThreatExchange for providing the threat data that helped facilitate our study. We are also very grateful to Alberto Dainotti and Alistair King for sharing the UCSD telescope data and helping us with the analysis, Gautam Akiwate for helping us query the domain data, and Matt Jonkman. We are also grateful to Martina Lindorfer, our shepherd, and our anonymous reviewers for their insightful feedback and suggestions. This research is a joint work from multiple institutions, sponsored in part by DHS/AFRL award FA8750-18-2-0087, NSF grants CNS-1237265, CNS-1406041, CNS-1629973, CNS-1705050, and CNS-1717062.

References

[1] Abuse.ch. https://abuse.ch/.

[2] Top Alexa domains. https://www.alexa.com/topsites/.

[3] Alienvault IP reputation. http://reputation.alienvault.com/reputation.data.

[4] ANTONAKAKIS, M., APRIL, T., BAILEY, M., BERNHARD, M., BURSZTEIN, E., COCHRAN, J., DURUMERIC, Z., HALDERMAN, J. A., INVERNIZZI, L., KALLITSIS, M., ET AL. Understanding the mirai botnet. In *USENIX Security Symposium* (2017).

[5] Badips. https://www.badips.com/.

[6] BENSON, K., DAINOTTI, A., SNOEREN, A. C., KALLITSIS, M., ET AL. Leveraging internet background radiation for opportunistic network analysis. In *Proceedings of the 2015 Internet Measurement Conference* (2015), ACM.

[7] The Bro network security monitor. https://www.bro.org/index.html.

[8] Composite Blocking List. https://www.abuseat.org/.

[9] Spreading the disease and selling the cure. https://krebsonsecurity.com/2015/01/spreading-the-disease-and-selling-the-cure/.

[10] CHACHRA, N., MCCOY, D., SAVAGE, S., AND VOELKER, G. M. Empirically Characterizing Domain Abuse and the Revenue Impact of Blacklisting. In *Proceedings of the Workshop on the Economics of Information Security (WEIS)* (State College, PA, 2014).

[11] Cloudflare, fast, global content delivery network. https://www.cloudflare.com/cdn/.

[12] AWS CloudFront, fast, highly secure and programmable content delivery network. https://aws.amazon.com/cloudfront/.

[13] Cyber Observable eXpression. http://cyboxproject.github.io/documentation/.

[14] DeKoven, L. F., Savage, S., Voelker, G. M., and Leontiadis, N. Malicious browser extensions at scale: Bridging the observability gap between web site and browser. In *10th USENIX Workshop on Cyber Security Experimentation and Test (CSET 17)* (2017), USENIX.

[15] Durumeric, Z., Bailey, M., and Halderman, J. A. An internet-wide view of internet-wide scanning. In *USENIX Security Symposium* (2014).

[16] Edgecast CDN, Verizon digital and media services. https://www.verizondigitalmedia.com/platform/edgecast-cdn/.

[17] Facebook threat exchange. https://developers.facebook.com/programs/threatexchange.

[18] Fastly managed CDN. https://www.fastly.com/products/fastly-managed-cdn.

[19] Incident Object Description Exchange Format. https://tools.ietf.org/html/rfc5070.

[20] Jagpal, N., Dingle, E., Gravel, J.-P., Mavrommatis, P., Provos, N., Rajab, M. A., and Thomas, K. Trends and lessons from three years fighting malicious extensions. In *USENIX Security Symposium* (2015).

[21] Jung, J., and Sit, E. An empirical study of spam traffic and the use of dns black lists. In *Proceedings of the ACM Conference on Internet Measurement* (2004).

[22] Kapravelos, A., Grier, C., Chachra, N., Kruegel, C., Vigna, G., and Paxson, V. Hulk: Eliciting malicious behavior in browser extensions. In *USENIX Security Symposium* (2014), San Diego, CA.

[23] Kührer, M., Rossow, C., and Holz, T. Paint it black: Evaluating the effectiveness of malware blacklists. In *International Workshop on Recent Advances in Intrusion Detection* (2014), Springer.

[24] Levchenko, K., Pitsillidis, A., Chachra, N., Enright, B., Félegyházi, M., Grier, C., Halvorson, T., Kanich, C., Kreibich, C., Liu, H., McCoy, D., Weaver, N., Paxson, V., Voelker, G. M., and Savage, S. Click Trajectories: End-to-End Analysis of the Spam Value Chain. In *Proceedings of the IEEE Symposium and Security and Privacy* (2011).

[25] MaxCDN. https://www.maxcdn.com/one/.

[26] Metcalf, L., and Spring, J. M. Blacklist ecosystem analysis: Spanning jan 2012 to jun 2014. In *Proceedings of the 2nd ACM Workshop on Information Sharing and Collaborative Security* (2015), ACM.

[27] Nothink honeypot SSH. http://www.nothink.org/honeypot_ssh.php.

[28] Packetmail.net. https://www.packetmail.net/.

[29] Pang, R., Yegneswaran, V., Barford, P., Paxson, V., and Peterson, L. Characteristics of internet background radiation. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement* (2004), ACM.

[30] Pitsillidis, A., Kanich, C., Voelker, G. M., Levchenko, K., and Savage, S. Taster's Choice: A Comparative Analysis of Spam Feeds. In *Proceedings of the ACM Internet Measurement Conference* (Boston, MA, Nov. 2012), pp. 427–440.

[31] Ramachandran, A., Feamster, N., Dagon, D., et al. Revealing botnet membership using dnsbl counter-intelligence. *SRUTI 6* (2006).

[32] Ramachandran, A., Feamster, N., and Vempala, S. Filtering spam with behavioral blacklisting. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)* (2007).

[33] Scheitle, Q., Hohlfeld, O., Gamba, J., Jelten, J., Zimmermann, T., Strowes, S. D., and Vallina-Rodriguez, N. A long way to the top: Significance, structure, and stability of internet top lists. In *Proceedings of the Internet Measurement Conference* (2018), ACM.

[34] Shadowserver. https://www.shadowserver.org/.

[35] Sheng, S., Wardman, B., Warner, G., Cranor, L. F., Hong, J., and Zhang, C. An empirical analysis of phishing blacklists. In *Proceedings of the Conference on Email and Anti-Spam (CEAS)* (2009).

[36] Sinha, S., Bailey, M., and Jahanian, F. Shades of grey: On the effectiveness of reputation-based "blacklists". In *2008 3rd International Conference on Malicious and Unwanted Software (MALWARE)*, IEEE.

[37] Structured Threat Information eXpression. https://stixproject.github.io/.

[38] UCSD network telescope. https://www.caida.org/projects/network_telescope/.

[39] The spam and open relay blocking system. http://www.sorbs.net/.

[40] The Spamhaus block list. https://www.spamhaus.org/sbl/.

[41] The Spamhaus Don't Route Or Peer Lists. https://www.spamhaus.org/drop/.

[42] Thomas, K., Amira, R., Ben-Yoash, A., Folger, O., Hardon, A., Berger, A., Bursztein, E., and Bailey, M. The abuse sharing economy: Understanding the limits of threat exchanges. In *International Symposium on Research in Attacks, Intrusions, and Defenses* (2016), Springer.

[43] Threat intelligence market analysis by solution, by services, by deployment, by application and segment forecast, 2018 - 2025. https://www.grandviewresearch.com/industry-analysis/threat-intelligence-market.

[44] University of Oregon route views project. http://www.routeviews.org/routeviews/.

[45] VirusTotal. https://www.virustotal.com/#/home/upload.

Towards the Detection of Inconsistencies in Public Security Vulnerability Reports

Ying Dong[1,2]*, Wenbo Guo[2,4], Yueqi Chen[2,4],
Xinyu Xing[2,4], Yuqing Zhang[1], and Gang Wang[3]

[1]*School of Computer Science and Technology, University of Chinese Academy of Sciences, China*
[2]*College of Information Sciences and Technology, The Pennsylvania State University, USA*
[3]*Department of Computer Science, Virginia Tech, USA*
[4]*JD Security Research Center, USA*
dongying115@mails.ucas.ac.cn, {wzg13, yxc431, xxing}@ist.psu.edu
zhangyq@ucas.ac.cn, gangwang@vt.edu

Abstract

Public vulnerability databases such as the Common Vulnerabilities and Exposures (CVE) and the National Vulnerability Database (NVD) have achieved great success in promoting vulnerability disclosure and mitigation. While these databases have accumulated massive data, there is a growing concern for their information quality and consistency.

In this paper, we propose an automated system `VIEM` to detect inconsistent information between the fully *standardized* NVD database and the *unstructured* CVE descriptions and their referenced vulnerability reports. `VIEM` allows us, for the first time, to quantify the information consistency at a massive scale, and provides the needed tool for the community to keep the CVE/NVD databases up-to-date. `VIEM` is developed to extract vulnerable software names and vulnerable versions from unstructured text. We introduce customized designs to deep-learning-based named entity recognition (NER) and relation extraction (RE) so that `VIEM` can recognize previous unseen software names and versions based on sentence structure and contexts. Ground-truth evaluation shows the system is highly accurate (0.941 precision and 0.993 recall). Using `VIEM`, we examine the information consistency using a large dataset of 78,296 CVE IDs and 70,569 vulnerability reports in the past 20 years. Our result suggests that inconsistent vulnerable software versions are highly prevalent. Only 59.82% of the vulnerability reports/CVE summaries strictly match the standardized NVD entries, and the inconsistency level increases over time. Case studies confirm the erroneous information of NVD that either overclaims or underclaims the vulnerable software versions.

1 Introduction

Security vulnerabilities in computer and networked systems are posing a serious threat to users, organizations, and nations at large. Unmatched vulnerabilities often lead to real-world attacks with examples ranging from WannaCry ransomware that shut down hundreds of thousands of machines in hospitals and schools [20] to the Equifax data breach that affected half of America's population [21].

To these ends, a strong *community effort* has been established to find and patch vulnerabilities before they are exploited by attackers. The Common Vulnerabilities and Exposures (CVE) program [4] and the National Vulnerability Database (NVD) [11] are among the most influential forces. CVE is a global list/database that indexes publicly known vulnerabilities by harnessing the "the power of the crowd". Anyone on the Internet (security vendors, developers and researchers) can share the vulnerabilities they found on CVE. NVD is a more standardized database established by the U.S. government (*i.e.,* NIST). NVD receives data feeds from the CVE website and perform analysis to assign common vulnerability severity scores (CVSS) and other pertinent metadata [18]. More importantly, NVD standardizes the data format so that algorithms can directly process their data [12]. Both CVE and NVD play an important role in guiding the vulnerability mitigation. So far, over 100,000 vulnerabilities were indexed, and the CVE/NVD data stream has been integrated with hundreds of security vendors all over the world [10].

While the vulnerability databases are accumulating massive data, there is also a growing concern about the *information quality* [28, 42, 44]. More specifically, the information listed on CVE/NVD can be incomplete or outdated, making it challenging for researchers to reproduce the vulnerability [42]. Even worse, certain CVE entries contain erroneous information which may cause major delays in developing and deploying patches. In practice, industrial systems often use legacy software for a long time due to the high cost of an update. When a relevant vulnerability is disclosed, system administrators usually look up to vulnerability databases to determine whether their software (and which versions) need to be patched. In addition, CVE/NVD are serving as a key information source for security companies to assess the secu-

*This work was done when Ying Dong studied at the Pennsylvania State University.

rity level of their customers. Misinformation on CVE/NVD could have left critical systems unpatched.

In this paper, we propose a novel system to automatically detect inconsistent information between the fully *standardized* NVD database and the *unstructured* CVE descriptions and their referenced vulnerability reports. Our system VIEM allows us, for the first time, to quantify the information consistency at a massive scale. Our study focuses on *vulnerable software versions*, which is one of the most important pieces of information for vulnerability reproduction and vulnerability patching. We face three main technical challenges to build VIEM. First, due to the high diversity of software names and versions, it is difficult to build dictionaries or regular expressions [30, 48, 59] to achieve high precision and recall. Second, the unstructured text of vulnerability reports and summaries often contain code, and the unique writing styles are difficult to handle by traditional natural language processing tools [22, 49, 50]. Third, we need to extract "*vulnerable*" software names and their versions, and effectively exclude the distracting items (*i.e.*, non-vulnerable versions).

Our System. To address these challenges, we build VIEM (short for Vulnerability Information Extraction Model) with a Named Entity Recognition (NER) model and a Relation Extraction (RE) model. The goal is to learn the patterns and indicators from the sentence structures to recognize the vulnerable software names/versions. Using "contexts" information, our model can capture previously unseen software names and is generally applicable to different vulnerability types. More specifically, the NER model is a recurrent deep neural network [36, 56] which pinpoints the relevant entities. It utilizes word and character embeddings to encode the text and then leverages a sequence-to-sequence bidirectional GRU (Gated Recurrent Unit) to locate the names and versions of the vulnerable software. The RE model is trained to analyze the relationships between the extracted entities to pair the vulnerable software names and their versions together. Finally, to generalize our model to handle different types of vulnerabilities, we introduce a transfer learning step to minimize the manual annotation efforts.

Evaluation and Measurement. We collect a large dataset from the CVE and NVD databases and 5 highly popular vulnerability reporting websites. The dataset covers 78,296 CVE IDs and 70,569 vulnerability reports across all 13 vulnerability categories in the past 20 years. For evaluation, we manually annotated a sample of 5,193 CVE IDs as the ground-truth. We show that our system is highly accurate with a precision of 0.941 and a recall of 0.993. In addition, our model is generalizable to all 13 vulnerability categories.

To detect inconsistencies in practice, we apply VIEM to the full dataset. We use NVD as the standard (since it is the last hop of the information flow), and examine the inconsistencies between NVD entries and the CVE entries/external sources. We have a number of key findings. First, the inconsistency level is very high. Only 59.82% of the external reports/CVE summaries have exactly the same vulnerable software versions as those of the standardized NVD entries. It's almost equally common for an NVD entry to "overclaim" or "underclaim" the vulnerable versions. Second, we measure the consistency level between NVD and other sources and discover the inconsistency level increased over the past 20 years (but started to decrease since 2016). Finally, we select a small set of CVE IDs with highly inconsistent information to manually verify the vulnerabilities (including 185 software versions). We confirm real cases where the official NVD/CVE entries and/or the external reports falsely included non-vulnerable versions and missed truly vulnerable versions. Such information could affect systems that depend on NVD/CVE to make critical decisions.

Applications. Detecting information inconsistency is the *first step* to updating the outdated entries and mitigating errors in the NVD and CVE databases. There was no existing tool that could automatically extract vulnerable software names and versions from unstructured reports before. A key contribution of VIEM is to enable the possibility to continuously monitor different vulnerability reporting websites and periodically generate a "diff" from the CVE/NVD entries. This can benefit the community in various ways. For employees of CVE/NVD, VIEM can notify them whenever a new vulnerable version is discovered for an existing vulnerability (to accelerate the testing and entry updates). For security companies, VIEM can help to pinpoint the potentially vulnerable versions of their customers' software to drive more proactive testing and patching. For software users and system administrators, the "diff" will help them to make more informed decisions on software updating. To facilitate future research and application development, we released our labeled dataset and the source code of VIEM[1].

In summary, our paper makes three key contributions.

- *First*, we design and develop a novel system VIEM to extract vulnerable software names and versions from unstructured vulnerability reports.
- *Second*, using a large ground-truth dataset, we show that our system is highly accurate and generalizes well to different vulnerability types.
- *Third*, by applying VIEM, we perform the first large-scale measurement of the information consistency for CVE and NVD. The generated "diff" is helpful to drive more proactive vulnerability testing and information curation.

2 Background and Challenges

We first introduce the background of security vulnerability reporting, and describe the technical challenges of our work.

[1] https://github.com/pinkymm/inconsistency_detection

2.1 Vulnerability Reporting

CVE. When people identify a new vulnerability, they can request a unique CVE-ID number from one of the CVE Numbering Authorities (CNAs) [5]. The *MITRE Corporation* is the editor and the primary CNA [19]. CNA will then do research on the vulnerability to determine the details and check if the vulnerability has been previously reported. If the vulnerability is indeed new, then a CVE ID will be assigned and the corresponding vulnerability information will be publicly released through the CVE list [4, 9].

The CVE list [4] is maintained by MITRE as a website on which the CVE team publishes a summary for each of the reported vulnerabilities. As specified in [8], when writing a CVE summary, the CVE team will analyze (public) third-party vulnerability reports and then include details in their description such as the name of the affected software, the vulnerable software versions, the vulnerability type, and the conditions/requirements to exploit the vulnerability.

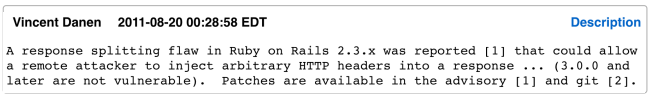
In addition to the summary, each CVE entry contains a list of external references. The external references are links to third-party technical reports or blog/forum posts that provide the needed information for the CVE team to craft the official vulnerability description [1]. The information on CVE can help software vendors and system administrators to pinpoint the versions of the vulnerable software, assess their risk level, and perform remediation accordingly.

NVD. NVD (National Vulnerability Database) is maintained by a different organization (*i.e.,* NIST) from that of CVE [3]. NVD is built fully synchronized with the CVE list. The goal is that any updates to CVE will appear immediately in NVD. After a new CVE ID appears on the CVE list, the NIST NVD team will first perform analysis to add enhanced information such as the severity score before creating the NVD entries [18].

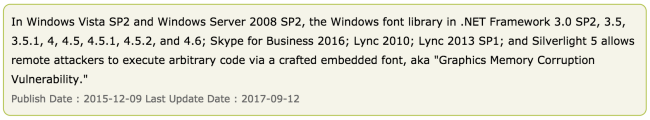
Compared with CVE, NVD provides two additional features. First, NVD data entries are structured. The NIST NVD team would convert the unstructured CVE information into structured JSON or XML, where information fields such as vulnerable software names and versions are formatted and standardized based on the Common Weakness Enumeration Specification (CWE) [12]. Second, data entries are continuously updated. The information in NVD may be updated (manually) after the initial vulnerability reporting. For example, as time goes by, new vulnerable software versions may be discovered by NIST employees or outsiders, which will be added to the existing NVD entries [17].

2.2 Technical Challenges

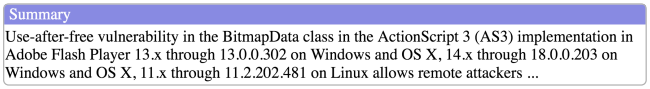
CVE and NVD databases are primarily maintained by *manual efforts*, which leads to a number of important questions. First, given that a vulnerability may be reported and discussed in many different places, how complete is the information (*e.g.*, vulnerable software names and their versions) in the CVE/NVD database? Second, considering the continuous community effort to study a reported vulnerability, how effective is the current manual approach to keep the CVE/NVD entries up-to-date?



(a) An Openwall report containing the vulnerable versions of software (`2.3.x`) and non-vulnerable versions (`3.0.0 and later`).



(b) A CVE summary enclosing multiple entities pertaining to vulnerable software (the vulnerable component: `Windows font library`; vulnerable software: `.NET Framework`, `Skype for Business`, `Lync`, `Silverlight`; dependent software: `Windows`; software versions tied to these entities).



(c) A CVE summary in which the name and versions of vulnerable software are not adjacent.

Figure 1: Examples of vulnerability descriptions and reports.

Our goal is to thoroughly understand the inconsistencies between external vulnerability reporting websites and the CVE/NVD data entries. According to the statistics from [6], the CVE list has archived more than 100,000 distinct CVEs (although certain CVE IDs were merged or withdrawn). Each CVE ID also has 5∼30 external third-party reports. It is infeasible to extract such information manually. The main challenge is to automatically and accurately extract relevant information items from the unstructured reports.

Many existing NLP tools aim to extract relevant information from text (*e.g.*, [22, 49, 50, 53]). However, the unique characteristics of vulnerability reports impose significant challenges, making existing techniques inadequate. ❶ Previously unseen software emerges: the CVE list introduces new vulnerable software frequently, making it difficult to use a pre-defined dictionary to identify the names of all vulnerable software. As such, dictionary-based method is not suitable for this problem (*e.g.*, [30, 48]). ❷ Reports are unstructured: most CVE summaries and vulnerability reports are highly unstructured and thus simple regular-expression-based techniques (*e.g.*, [28, 59]) can be barely effective. ❸ Non-interested entities are prevalent: a vulnerability report usually encloses information about both vulnerable and non-vulnerable versions of software (see Figure 1a). Our goal is to extract "vulnerable" software names and versions while excluding information items related to non-vulnerable software. Techniques that rely on pre-defined rules would hardly

work here (*e.g.*, [28, 55, 59]). ❹ Multiple interested entities exist: the vulnerable software mentioned in a report usually refers to multiple entities (see Figure 1b) and the relationships of these entities are determined by the context of the report. This requirement eliminates techniques that lack the capability of handling multiple entities (e.g., [28, 32, 59]). ❺ Vulnerability types are diverse: CVE covers a variety of vulnerability types, and each has its own characteristics in the descriptions. As a result, we cannot simply use techniques designed for certain vulnerability types. For example, a tool used by [59] is designed specifically for kernel memory corruption vulnerabilities. We tested it against our ground-truth dataset and did not receive satisfying results (the recall is below 40%).

3 The Design of VIEM

To tackle the challenges mentioned above, we develop an automated tool VIEM by combining and customizing a set of state-of-the-art natural language processing (NLP) techniques. In this section, we briefly describe the design of VIEM and discuss the reasons behind our design. Then, we elaborate on the NLP techniques that VIEM adopts.

3.1 Overview

To pinpoint and pair the entities of our interest, we design VIEM to complete three individual tasks.

Named Entity Recognition Model. First, VIEM utilizes a state-of-the-art Named Entity Recognition (NER) model [36, 56] to identify the entities of our interest, *i.e.,* the name and versions of the vulnerable software, those of vulnerable components and those of underlying software systems that vulnerable software depends upon (see Figure 1b).

The reasons behind this design are twofold. First, an NER model pinpoints entities based on the structure and semantics of input text, which provides us with the ability to track down software names that have never been observed in the training data. Second, an NER model can learn and distinguish the contexts pertaining to vulnerable and non-vulnerable versions of software, which naturally allows us to eliminate non-vulnerable versions of software and pinpoint only the entities of our interest.

Relation Extraction Model. With the extracted entities, the next task of VIEM is to pair identified entities accordingly. As is shown in Figure 2, it is common that software name and version jointly occur in a report. Therefore, one instinctive reaction is to group software name and version nearby, and then deem them as the vulnerable software and version pairs. However, this straightforward approach is not suitable for our problem. As is depicted in Figure 1c, the vulnerable software name is not closely tied to all the vulnerable versions. Merely applying the approach above, we might inevitably miss the versions of the vulnerable software.

To address this issue, VIEM first goes through all the possible combinations between versions and software names. Then, it utilizes a Relation Extraction (RE) model [38, 62] to determine the most possible combinations and deems them as the correct pairs of entities. The rationale behind this design is as follows. The original design of an RE model is not for finding correct pairs among entities. Rather, it is responsible for determining the property of a pair of entities. For example, assume that an RE model is trained to assign a pair of entities one of the following three properties – *"born in"*, *"employed by"* and *"capital of"*. Given two pairs of entities $P_1 =$ (*"Steve Jobs"*, *"Apple"*) and $P_2 =$ (*"Steve Jobs"*, "California") in the text *"Steve Jobs was born in California, and was the CEO of Apple."*, an RE model would assign the *"employed by"* to P_1 and *"born in"* properties to P_2.

In our model, each of the possible version-and-software combinations can be treated as an individual pair of entities. Using the idea of the relation extraction model, VIEM assigns each pair a property, indicating the truthfulness of the relationship of the corresponding entities. Then, it takes the corresponding property as the true pairing. Take the case in Figure 2 for example, there are 4 entities indicated by 2 software (`Microsoft VBScript` and `Internet Explorer`) and 2 ranges of versions (`5.7 and 5.8` and `9 through 11`). They can be combined in 4 different ways. By treating the combinations as 4 different pairs of entities, we can use an RE model to assign a binary property to each of the combinations. Assuming that the binary property assigned indicates whether the corresponding pair of the entities should be grouped as software and its vulnerable versions, VIEM can use the property assignment as the indicator to determine the entity pairing. It should be noted that we represent paired entities in the Common Platform Enumeration (CPE) format [2]. For example, `cpe:/a:google:chrome:3.0.193.2:beta`, where `google` denotes the vendor, `chrome` denotes the product, `3.0.193.2` denotes the version number, and `beta` denotes the software update.

Transfer Learning. Recall that we need to measure vulnerability reports across various vulnerability types. As mentioned in §2.2, the reports of different vulnerability types do not necessarily share the same data distribution. Therefore, it is not feasible to use a single machine learning model to deal with all vulnerability reports, unless we could construct and manually annotate a large training dataset that covers all kinds of vulnerabilities. Unfortunately, there is no such labeled dataset available, and labeling a large dataset involves tremendous human efforts. To address this problem, VIEM takes the strategy of transfer learning, which learns the aforementioned NER and RE models using vulnerability reports in one primary category and then transfers their capability into other vulnerability categories. In this way, we can re-

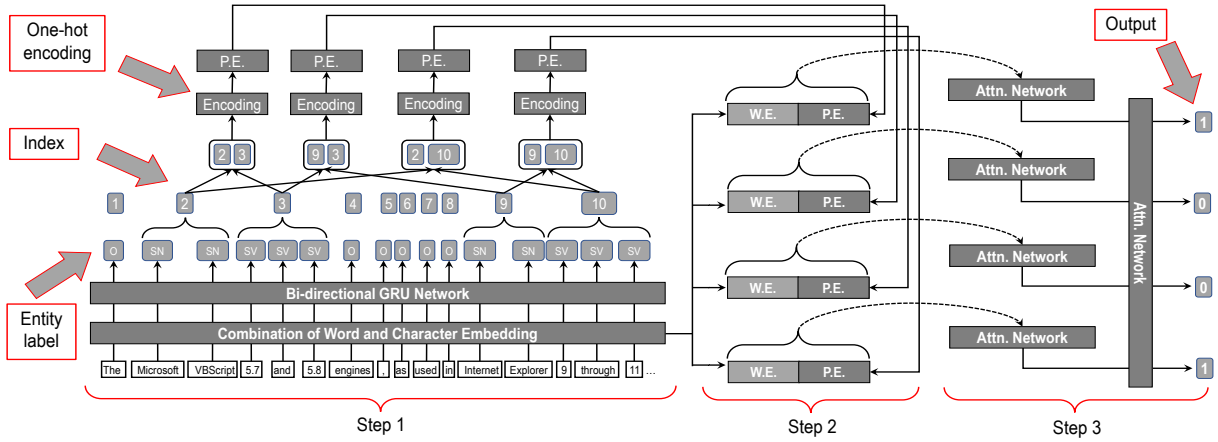


Figure 2: The RE model that pairs the entities of our interests through a three-step procedure. Here, the entities of our interest include software (`Microsoft VBScript`; `Internet Explorer`) as well as version range (`5.7 and 5.8`; `9 through 11`). W.E. and P.E denote word and position embeddings, respectively.

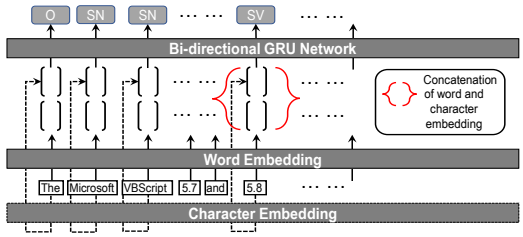


Figure 3: The NER model which utilizes both word and character embeddings to encode a target text and then a Bi-GRU network to assign entity labels to each word.

duce the efforts involved in data labeling, making `VIEM` effective for arbitrary kinds of vulnerability reports. More details of the transfer learning are discussed in §5.

3.2 Named Entity Recognition Model

We start by extracting named entities (vulnerabile software names and versions) from the text. We develop our system based on a recent NER model [36,56]. On top of that, we integrate a gazetteer to improve its accuracy of extracting vulnerable software names. At the high level, the NER model first encodes a text sequence into a sequence of word vectors using the concatenation of word and character embeddings. This embedding process is necessary since a deep neural network cannot process text directly. Then taking the sequence of the word vectors as the input, the model predicts a label for each of the words in the sequence using a bi-directional Recurrent Neural Network. Below, we introduce the key technical details.

For word and character embeddings, the NER model first utilizes a standard word embedding approach [40] to encode each word as a vector representation. Then, it utilizes a Bi-directional Gated Recurrent United (Bi-GRU) network to perform text encoding at the character level. As shown in Figure 3, the NER model concatenates these two embeddings as a single sequence of vectors and then takes it as the input for another Bi-GRU network.

Different from the aforementioned Bi-GRU network used for text embedding, the second Bi-GRU network is responsible for assigning labels to words. Recall that our task is to identify the entities of our interest which pertain to vulnerable software. As a result, we use the Bi-GRU network to pinpoint the words pertaining to this information. More specifically, we train this Bi-GRU network to assign each word with one of the following labels – ❶ `SV` (software version), ❷ `SN` (software name) and ❸ `O` (others) – indicating the property of that word. It should be noted that we assign the same `SN` label to vulnerable software, vulnerable component and the underlying software that vulnerable software is dependent upon. This is because this work measures version inconsistencies of all software pertaining to a vulnerability[2].

Considering that the NER model may not perfectly track down the name of the vulnerable software, we further construct a gazetteer (*i.e.,* a dictionary consisting of 81,551 software mentioned in [10]) to improve the recognition performance of the NER model. To be specific, we design a heuristic approach to rectify the information that the NER model fails to identify or mistakenly tracks down. First, we perform a dictionary lookup on each of the vulnerability reports. Then, we mark dictionary words in that report as the software name, if the NER model has already identified at least one dictionary word as a software name. In this way, we can rectify some labels incorrectly identified. For example, in Figure 2, assume the NER model assigns the `SN` labels to words `Microsoft VBScript` and `Explorer` indicating the

[2]Vulnerable software names may include the names of vulnerable libraries or the affected operating systems. The names of vulnerable libraries and the affected OSes are also extracted by our tool.

software pertaining to the vulnerability. Through dictionary lookup, we track down software `Internet Explorer` in the gazetteer. Since the gazetteer indicates neither `Internet` nor `Explorer` has ever occurred individually as a software name, and they are the closest to the dictionary word `Internet Explorer`, we extend our label and mark the entire dictionary word `Internet Explorer` with an `SN` label and treat it as single software.

3.3 Relation Extraction Model

The relation extraction model was originally used to extract the relationships of two entities [41]. Over the past decade, researchers proposed various technical approaches to build highly accurate and computationally efficient RE model. Of all the techniques proposed, hierarchical attention neural networks [57] demonstrate better performance in many natural language processing tasks. For our system, we modify an existing hierarchical attention neural network to *pair* the extracted entities (*i.e.*, pairing vulnerable software names and their versions). More specifically, we implement a new *combined* word-level and sentence-level attention network in the RE model to improve the performance. In the following, we briefly introduce this model and discuss how we apply it to our problem. For more details about the RE model in general, readers could refer to these research papers [38, 57, 62].

As is depicted in Figure 2, the RE model pinpoints the right relationship between software name and version through a three-step procedure. In the first step, it encodes the occurrence of the software names as well as that of the version information, and then yields a group of position embeddings representing the relative distances from current word to the two named entities (*i.e.,* software name and version) in the same sentence [38]. To be specific, the RE model first indexes the sequence of the entity labels generated by the aforementioned NER model. Then, it runs through all the software names and versions in every possible combination, and encodes the combinations based on the indexes of the entity labels using one-hot encoding. With the completion of one-hot encoding, the RE model further employs a word embedding approach to convert the one-hot encoding into two individual vectors indicating the embeddings of the positions (see Figure 2).

In the second step, similar to the NER model, the RE model uses the same technique to encode text and transfers a text sequence into a sequence of vectors. Then, right behind the word sequence vectors, the RE model appends each group of the position embeddings individually. For example, in Figure 2, the NER model pinpoints two software names and two versions which form four distinct combinations (all the possible ways of pairing software names and versions). For each combination, the RE model appends the position embedding vector to the word embedding vector to form the input for the last step for performing classifications. In this example, four vectors are produced as the input, and each represents a possible software name-version pair.

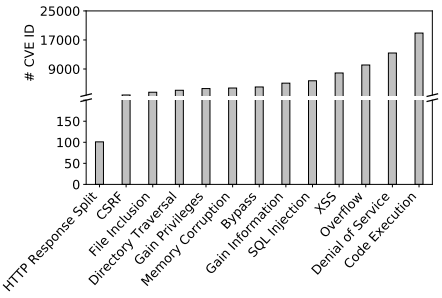


Figure 4: # of CVE IDs per vulnerability category.

In the last step, the RE model takes each sequence of vectors as the input for an attention-based neural network and outputs a vector indicating the new representation of the sequence. Then, as is illustrated in Figure 2, the RE model takes the output vectors as the input for another attention network, through which the RE model predicts which name-version pairing is most likely to capture the relationship between software name and its corresponding versions. Continue the example in Figure 2. The `seq 1` and `4` are associated with a positive output, which indicates the legitimate pairing relationships of `Microsoft VBScript 5.7 and 5.8` and `Internet Explorer 9 through 11`.

4 Dataset

To evaluate our system and detect real-world inconsistencies, we collected a large number of public vulnerability reports and CVE and NVD entries from the past 20 years. We sample a subset of these vulnerability reports for manual labeling (ground-truth) and use the labeled data to evaluate the performance of `VIEM`.

CVE IDs. We first obtain a list of CVE IDs from `cvedetails.com`, which divides the security vulnerabilities indexed in CVE/NVD database into 13 categories. To collect a representative dataset of publicly-reported vulnerabilities, we crawled the CVE IDs from January 1999 to March 2018 (over 20 years) of each vulnerability category. A CVE ID is the unique identifier for a publicly disclosed vulnerability. Even though the CVE website claimed that they have over 105,000 CVE IDs [6], many of the CVE IDs are either not publicly available yet, or have been merged or withdrawn. In total, we obtain 78,296 CVE IDs covering all 13 categories as shown in Figure 4. Each CVE ID corresponds to a short *summary* of the vulnerability as shown in Table 1.

Vulnerability Reports. The webpage of each CVE ID also contains a list of external references pointing to external reports. Our study focuses on 5 representative source websites to obtain the vulnerability reports referenced by the CVE, including ExploitDB [7], SecurityFocus [14], SecurityTracker [16], Openwall [13], and SecurityFocus Fo-

Dataset	CVE IDs	CVE Summaries (Unstructured)	NVD Entries (Structured)	Vulnerability Reports	Structured Reports		Unstructured Reports		
					SecTracker	SecFocus	ExploitDB	Openwall	SecF Forum
All	78,296	78,296	78,296	70,569	7,320	38,492	9,239	5,324	10,194
G-truth	5,193	5,193	0	1,974	0	0	785	520	669

Table 1: Dataset statistics.

rum [15]. Note that we treat SecurityFocus and SecurityFocus Forum as two different websites. SecurityFocus site only displays the "structured" information (*e.g.*, affected OS, software versions). SecurityFocus Forum (also called Bugtraq Mailing List) mainly contains "unstructured" articles and discussion threads between vulnerability reporters and software developers. Regarding the other three websites, SecurityTracker also contains well-structured information, while Openwall and ExploitDB[3] contain unstructured information. In total, we obtain 70,569 vulnerability reports associated with 56,642 CVE IDs. These CVE IDs cover 72.34% of all 78,296 public CVE IDs. This means 72.34% of the CVE IDs have a vulnerability report from one of the 5 source websites, confirming their popularity. There are 45,812 structured reports from SecurityTracker and SecurityFocus, and 24,757 unstructured reports from ExploitDB, Openwall, and SecurityFocus Forum.

NVD Entries. For each CVE ID, we also parse the JSON version of the NVD entries, which contains the structured data fields such as vulnerable software names and their versions. We obtain 78,296 NVD entries in total.

Data Extraction and Preprocessing. For *structured* reports, we directly parse the vulnerable software name and version information following the fixed format. For *unstructured* vulnerability reports and CVE summaries, we extract the text information, remove all the web links, and tokenize the sentences using the NLTK toolkit [24]. Note that we did not remove any stop words or symbols from the unstructured text, considering that they are often parts of the software names and versions.

Ground-Truth Dataset. To evaluate `VIEM`, we construct a "ground-truth" dataset by manually annotating the vulnerable software names and versions. As shown in Table 1, the ground-truth dataset contains only unstructured reports, covering 5,193 CVE IDs (the short summaries) and 1,974 unstructured reports. Some reports are referenced by multiple CVE IDs. We choose to label our own ground-truth dataset instead of using the structured data as the ground-truth, for two reasons. First, the structured data is not necessarily correct. Second, the NER model needs labels at the sentence level and the word level. The labels for the RE model represent the relationship between the extracted entities. The structured data cannot provide such labels.

[3]ExploitDB has some structured information such as the affected OS, but the vulnerable software version often appears in the titles of the posts and the code comments.

The 5,193 CVE IDs are not evenly sampled from different vulnerability categories. Instead, we sampled a large number of CVE IDs from one primary category and a smaller number of CVE IDs from the other 12 categories to evaluate model transferability. We choose memory corruption as the primary category for its severity and real-world impact (*e.g.*, Heartbleed, WannaCry). An analysis of the severity scores (CVSS) also shows that memory corruption vulnerabilities have the highest average severity score (8.46) among all the 13 categories. We intend to build a tool that at least performs well on memory corruption cases. Considering the high costs of manual annotation, we decide to label a large amount of data (3,448 CVE IDs) in one category (memory corruption), and only label a small amount of data (145 CVE IDs) for each of the other 12 categories. With this dataset, we can still apply transfer learning to achieve a good training result.

Given a document, we perform annotation in two steps. First, we manually label the *vulnerable software names* and *vulnerable software versions*. This step produces a ground-truth dataset to evaluate our NER model. Second, we manually *pair* the vulnerable software names with the versions. This step produces a ground-truth mapping to evaluate the RE model. We invited 6 lab-mates to perform the annotation. All the annotators have a bachelor or higher degree in Computer Science and an in-depth knowledge of Software Engineering and Security. Figure 2 shows an example. For the NER dataset, we label each word in a sentence by assigning one of the three labels: vulnerable software name (*SN*), vulnerable software version (*SV*)[4], or others (*O*). Note that entities that are related to non-vulnerable software will be labeled as *O*. For the RE dataset, we pair *SN* and *SV* entities by examining all the possible pairs *within the same sentence*. Through our manual annotation, we never observe a vulnerable software name and its versions located in completely different sentences throughout the entire document.

5 Evaluation

In this section, we use the ground-truth dataset to evaluate the performance of `VIEM`. First, we use the dataset of memory corruption vulnerabilities to assess the system performance and fine-tune the parameters. Then, we use the data of the other 12 categories to examine the model transferability.

[4]For software versions, we treat keywords related to compatibility pack, service pack, and release candidate (*e.g.*, business, express edition) as part of the version. For versions that are described as a "range", we include the conjunctions in the version labels.

Metric		w/o Gazetteer	w/ Gazetteer
Software Version	Precision	0.9880	0.9880
	Recall	0.9923	0.9923
Software Name	Precision	0.9773	0.9782
	Recall	0.9916	0.9941
Overall	Accuracy	0.9969	0.9970

Table 2: NER performance on "memory corruption" dataset.

5.1 Evaluating the NER and RE Model

To evaluate the NER and RE models, we use the *memory corruption* vulnerability reports and their CVE summaries (3,448 CVE IDs).

NER Model. Given a document, the NER model extracts the vulnerable software names and vulnerable versions. The extraction process is first at the *word level*, and then the consecutive words with the *SN* or *SV* label will be grouped into software names or software versions. We use three evaluation metrics on the word-level extraction: (1) *Precision* represents the fraction of the relevant entities over the extracted entities; (2) *Recall* represents the fraction of the relevant entities that are extracted over the total number of relevant entities; (3) *Overall accuracy* represents the fraction of the correct predictions over all the predictions. We compute the precision and recall for software name extraction and version extraction separately.

We split the ground-truth dataset with a ratio of 8:1:1 for training, validation, and testing. We use a set of default parameters, and later we show that our performance is not sensitive to the parameters. Here, the dimension of the pre-trained word embeddings is 300 (50 for character embeddings). To align our input sequences, we only consider the first 200 words per sentence. Empirically, we observe that the vast majority of sentences are shorter than 200 words. All the layers in the NER model are trained jointly except the word-level embedding weight W (using the `FastText` method). The default batch size is 50 and the number of epochs is 20. We use an advanced stochastic gradient descent approach `Adam` as the optimizer, which can adaptively adjust the learning rate to reduce the convergence time. We also adopt `dropout` to prevent overfitting.

We repeat the experiments 10 times by randomly splitting the dataset for training, validation, and testing. We show the average precision, recall, and accuracy in Table 2. Our NER model is highly accurate even without applying the gazetteer (*i.e.*, the dictionary). Both vulnerable software names and versions can be extracted with a precision of 0.978 and a recall of 0.991. In addition, we show that gazetteer can improve the performance of software name extraction as expected. After applying the gazetteer, the overall accuracy is as high as 0.9969. This high accuracy of NER is desirable because any errors could propagate to the later RE model.

We also observe that our NER model indeed extracts software names (and versions) that *never appear* in the training dataset. For example, after applying NER to the testing dataset, we extracted 205 unique software names, and 47 (22.9%) of them never appear in the training dataset. This confirms that the NER model has learned generalizable patterns and indicators to extract relevant entities, which allows the model to extract *previously unseen* software names.

Metric	Ground-truth Software Name/Version as Input	NER Model's Result as Input	
		w/o Gazetteer	w/ Gazetteer
Precision	0.9955	0.9248	0.9411
Recall	0.9825	0.9931	0.9932
Accuracy	0.9916	0.9704	0.9764

Table 3: RE performance on "memory corruption" dataset.

RE Model. We then run experiments to first examine the performance of the RE model itself, and then evaluate the end-to-end performance by combining NER and RE. Similar as before, we split the ground-truth dataset with an 8:1:1 ratio for training, validation, and testing. Here, we set the dimension of the pre-trained word embeddings to 50. The dimension of position embeddings is 10. The default batch size is 80 and the number of epochs is 200. We set the number of bi-directional layers as 2. Like the NER model, our RE model also uses the pre-trained word embedding weight W. The position embedding weights (*i.e.,* W_s and W_v) are randomly initialized and trained together with other parameters in the model.

First, we perform an experiment to evaluate the RE model alone. More specifically, we assume the named entities are already correctly extracted, and we only test the "pairing" process using RE. This assumes the early NER model has a perfect performance. As shown in Table 3, the RE model is also highly accurate. The model has a precision of 0.9955 and a recall of 0.9825.

Second, we evaluate the end-to-end performance, and use the NER's output as the input of the RE model. In this way, NER's errors may affect the performance of RE. As shown in Table 3, the accuracy has decreased from 0.9916 to 0.9704 (without gazetteer) and 0.9764 (with gazetteer). The degradation mainly happens in precision. Further inspection shows that the NER model has falsely extracted a few entities that are not software names, which then become the false input for RE and hurt the classification precision. In addition, the benefit of gazetteer also shows up after NER and RE are combined, bumping the precision from 0.9248 to 0.9411 (without hurting the recall). The results confirm that our model is capable of accurately extracting vulnerable software names and the corresponding versions from unstructured text.

Baseline Comparisons. To compare the performance of `VIEM` against other baselines, we apply other methods to the same dataset. First, for the NER model, we tested SemFuzz [59]. SemFuzz uses hand-built regular expressions to extract vulnerable software versions from vulnerability re-

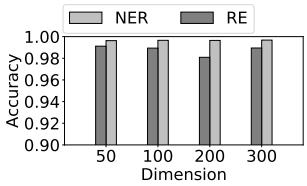


Figure 5: Word embedding dimension vs. accuracy.

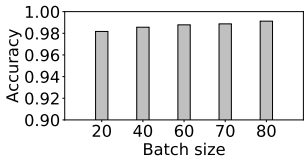


Figure 6: Batch size vs. RE model accuracy.



Figure 7: # network layers vs. RE model accuracy.

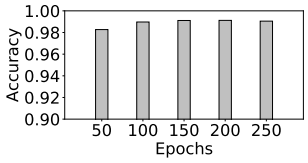


Figure 8: # Epochs vs. RE model accuracy.

ports. We find that SemFuzz achieves a reasonable precision (0.8225) but a very low recall (0.371). As a comparison, our precision and recall are both above 0.978. Second, for the end-to-end evaluation, we implemented a baseline system using off-the-shelf NLP toolkits. More specifically, we use the Conditional Random Field sequence model for extracting named entities. The model uses Stanford Part-Of-Speech tags [52] and other syntactic features. Then we feed the extracted entities to a baseline RE model that is trained with features from Stanford Neural Dependency Parsing [29]. The end-to-end evaluation returns a precision of 0.8436 and a recall of 0.8851. The results confirm the better performance of `VIEM`, at least for our application purposes (*i.e.*, processing vulnerability reports).

Model Parameters. The above results are based on a set of default parameters that have been fine-tuned based on the validation dataset. To justify our parameter choices, we change one parameter at a time and see its impact on the model. We perform this test on all parameters (*e.g.,* word embedding dimensions, batch sizes, network layers, epochs). The takeaway is that our model is not very sensitive to the parameter settings.

We pre-trained our own word embedding layer on the corpus built from all the unstructured reports in our dataset. We have tested two state-of-the-art methods `Word2vec` [39], and `FastText` [25]. We choose `FastText` since it gives a slightly higher accuracy (by 1%). Figure 5 shows the overall accuracy of `FastText` embeddings under different embedding dimensions. The results show that the performance is not sensitive to this parameter (as long as it is configured within a reasonable range). When training RE and NER, we need to set the batch size, the number of epochs, the number of bi-directional layers and the dimension of position embeddings in the neural networks. Again, we swap the parameters for RE and NER separately. For brevity, we only show the results for the RE model in Figure 6–Figure 8. The results for the NER model are similar, which are shown in Appendix-A.

Metric		Before Transfer		After Transfer	
		w/o Gaze	w/ Gaze	w/o Gaze	w/ Gaze
Software Version	Precision	0.8428	0.8467	0.9382	0.9414
	Recall	0.9407	0.9400	0.9410	0.9403
Software Name	Precision	0.8278	0.8925	0.9184	0.9557
	Recall	0.8489	0.9185	0.9286	0.9536
Overall	Accuracy	0.9873	0.9899	0.9942	0.9952

Table 4: Transfer learning result of NER model (average over 12 vulnerability categories).

5.2 Evaluating Transfer Learning

Finally, we examine the generalizability of the model to other vulnerability categories. First, to establish a baseline, we *directly* apply the model trained with memory corruption dataset to other vulnerability categories. Second, we apply transfer learning, and retrain a dedicated model for each of the other vulnerability categories. We refer the two experiments as "before transfer" and "after transfer" respectively.

Transfer learning is helpful when there is not enough labeled data for each of the vulnerability categories. In this case, we can use the memory corruption classifier as the teacher model. By fine-tuning *the last layer* of the teacher model with the data of each category, we can train a series of category-specific classifiers. More specifically, we train a teacher model using all the ground-truth data from the memory corruption category (3,448 CVE IDs). Then we use the teacher model to train a new model for each of the 12 vulnerability categories. Given a target category (*e.g.*, SQL Injection), we split its ground-truth data with a ratio of 1:1 for training (T_{train}) and testing (T_{test}). The training data (T_{train}) will be applied to the pre-trained teacher model to fine tune the final hidden layer. In this way, a new model that is specially tuned for "SQL Injection" reports is constructed. Finally, we apply this new model to the testing data (T_{test}) to evaluate its performance.

The results are presented in Table 4 and Table 5. We find that the NER model is already highly generalizable before transfer learning. Transfer learning only introduces a small improvement in accuracy (from 0.987 to 0.994). Second, the RE model (trained on memory corruption data) has a clear degradation in performance when it is directly applied to other categories. The overall accuracy is only 0.876. After transfer learning, accuracy can be improved to 0.9044.

To confirm that transfer learning is necessary, we run an additional experiment by using all the ground-truth dataset from 13 categories to train a single model. We find that the end-to-end accuracy is only 0.883 which is lower than the transfer learning accuracy. The accuracy of certain categories clearly drops (*e.g.*, 0.789 for CSRF). This shows that the vulnerability reports of different categories indeed have different characteristics, and deserve their own models.

Metric	Before Transfer			After Transfer		
	G-truth as Input	NER Result as Input		G-truth as Input	NER Result as Input	
		w/o Gaze	w/ Gaze		w/o Gaze	w/ Gaze
Precis.	0.9559	0.7129	0.8105	0.9781	0.8062	0.8584
Recall	0.9521	0.9767	0.9724	0.9937	0.9964	0.9964
Accur.	0.9516	0.8215	0.8760	0.9834	0.8698	0.9044

Table 5: Transfer learning result of RE model (average over 12 vulnerability categories.)

Match	Memory Corruption			Avg. over 12 Other Categories		
	Matching Rate		Deviat.	Matching rate		Deviat.
	VIEM	G-truth		VIEM	G-truth	
Loose	0.8725	0.8528	0.0194	0.9325	0.9371	-0.0046
Strict	0.4585	0.4627	-0.0042	0.6100	0.6195	-0.0095

Table 6: Strict matching and loose matching results on the ground-truth dataset.

6 Measuring Information Inconsistency

In this section, we apply VIEM to the full dataset to examine the information consistency. In particular, we seek to examine how well the structured NVD entries are matched up with the CVE entries and the referenced vulnerability reports. In the following, we first define the metrics to quantify consistency. Then we perform a preliminary measurement on the ground-truth dataset to estimate the measurement errors introduced by VIEM. Finally, we apply our model to the full dataset to examine how the consistency level differs across different vulnerability types and over time. In the next section (§7), we will perform case studies on the detected inconsistent reports, and examine the causes of the inconsistency.

6.1 Measurement Methodology

NVD database, with its fully *structured* and *standardized* data entries, makes it possible for automated information processing and intelligent mining. However, given that NVD entries are created and updated by manual efforts [18], we are concerned about its data quality, in particular, its ability to keep up with the most recent discoveries of vulnerable software and versions. To this end, we seek to measure the consistency of NVD entries with other information sources (including CVE summaries and external reports).

Matching Software Names. Given a CVE ID, we first match the vulnerable software names listed in the NVD database, and those from unstructured text. More specifically, let $C = \{(N_1, V_1), (N_2, V_2), ..., (N_n, V_n)\}$ be the vulnerable software name-version tuples extracted from the NVD, and $C' = \{(N_1, V_1'), (N_2, V_2'), ..., (N_m, V_m')\}$ be the name-version tuples extracted from the external text. In our dataset, about 20% of the CVE IDs are associated with multiple software names. In this paper, we only focus on the *matched software names* between the NVD and external reports. Our matching method has the flexibility to handle the slightly different format of the same software name. We consider two names as a match if the number of matched words is higher or equal to the number of unmatched words. For example, "Microsoft Internet Explorer" and "Internet Explorer" are matched because there are more matched words than the unmatched one.

Measuring Version Consistency. Given a software name N_1, we seek to measure the consistency of the reported versions V_1 and V_1'. We examine two types of matching. First, *strict matching* means V_1 and V_1' exactly match each other ($V_1 = V_1'$). Second, *loose matching* means one version is the other version's superset ($V_1 \subseteq V_1'$ or $V_1 \supseteq V_1'$). Note that the loosely matched cases contain those that are strictly matched. Beyond loose matching, it means V_1 and V_1' each contains some vulnerable versions that are not reported by the other (*i.e.*, conflicting information).

To perform the above matching procedure, we need to convert the text format of V_1 and V_1' to a comparable format. In unstructured text, the software version is either described as a set of discrete values (*e.g.*, "version 1.1 and 1.4") or a continuous range (*e.g.*, "version 1.4 and earlier"). For descriptions like "version 1.4 and earlier", we first convert the text representation into a mathematical range. The conversion is based on a dictionary that we prepared beforehand. For example, we convert "... and earlier" into "$\leq$". This means "1.4 and earlier" will be converted into "≤ 1.4". Then, for "range" based version descriptions, we look up the CPE directory maintained by NIST [2] to obtain a list of all the available versions for a given software. This allows us to convert the "range" description ("≤ 1.4") into a set of discrete values $\{1.0, 1.1, 1.2, 1.3, 1.4\}$. After the conversion, we can determine if V_1 and V_1' match or not.

If a CVE ID has more than one software names ($k > 1$), we take a conservative approach to calculate the matching result. Only if all the k software version pairs are qualified as strict matching will we consider the report as a "strict match". Similarly, only if all the pairs are qualified as loose matching will we label the report as "loose matching" report.

6.2 Ground-truth Measurement

Following the above methodology, we first use the ground-truth dataset to estimate the measurement error. More specifically, given all the ground-truth vulnerability reports and the CVE summaries, we use our best-performing model (with gazetteer and transfer learning) to extract the vulnerable software name-version tuples. Then we perform the strict and loose matching on the extracted entries and compare the matching rate with the ground-truth matching rate. The results are shown in Table 6.

We show that VIEM introduced a small deviation to the actual matching rate. For memory corruption vulnerabilities, our model indicates that 87.3% of the reports loosely match the NVD entries, and only 45.9% of the reports strictly

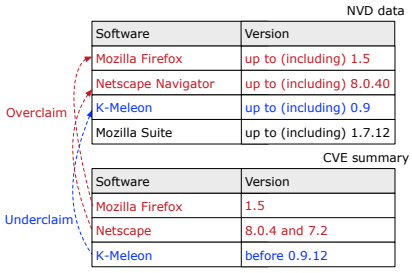


Figure 9: Example of underclaimed and overclaimed versions.

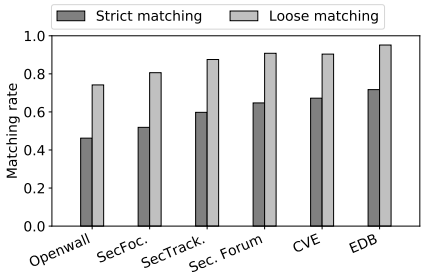


Figure 10: Matching rate for different information sources.

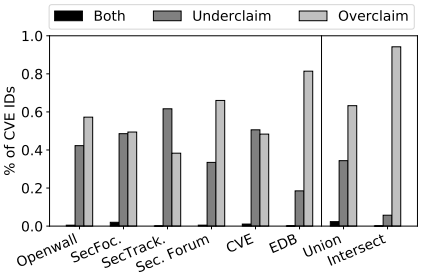


Figure 11: Breakdown of underclaimed and overclaimed cases.

match. The results are very close to the ground-truth where the matching rates are 85.3% and 46.3% respectively. For the rest of the 12 vulnerability categories, our model indicates that the loose matching rate is 93.3% and the strict matching rate is 61%. Again, the results are quite similar to the ground-truth (93.7% and 62%). The deviation from ground-truth ($Rate_{estimated} - Rate_{groundtruth}$) is bounded within $\pm 1.9\%$. The results confirm that our model is accurate enough for the measurement.

6.3 Large-Scale Empirical Measurements

After the validation above, we then use the full ground-truth dataset to train `VIEM` and apply the model to the rest of the unlabeled and unstructured text (vulnerability reports and CVE summaries). Then we calculate the matching rate between the versions of NVD and those from external information sources (the CVE website and 5 external websites).

Result Overview. Across all 78,296 CVE IDs, we extract in total 18,764 unique vulnerable software names. These vulnerable software names correspond to 154,569 software name-version pairs from the CVE summaries, 235,350 name-version pairs from the external vulnerability reports, and 165,822 name-version pairs from NVD database. After matching the software names between NVD and other sources, there are 389,476 pairs left to check consistency.

At the name-version pair level, we find 305,037 strictly matching pairs (78.32%). This means about 22% of the name-version pairs from NVD do not match the external information sources. If we relax the matching condition, we find 361,005 loosely matched pairs (93.49%).

We then aggregate the matching results at the *report level*. Although the loose matching rate is still high (90.05%), the strict matching rate clearly decreases. Only 59.82% of the vulnerability reports/CVE summaries strictly match the NVD entries. This is because strictly matched reports require all the extracted versions to match those of NVD.

In order to understand how the level of consistency varies across different aspects, we next break down the results for more in-depth analyses.

Information Source Websites. Figure 10 shows the matching rates between the NVD entries and the 5 information websites and the CVE website. CVE has a relatively high matching rate (about 70% strict matching rate). This is not too surprising given that NVD is claimed to be synchronized with the CVE feed. More interestingly, we find that ExploitDB has an even higher matching rate with NVD. We further examine the *posting dates* of the NVD entries and the corresponding reports in other websites. We find that the vast majority (95.8%) of the ExploitDB reports were posted after the NVD entries were created. However, 81% of the ExploitDB reports were posted earlier than the reports in the other 4 websites, which might have helped to catch the attention of the NVD team to make an update.

Overclaims vs. Underclaims. For the loosely matched versions, the NVD entries may have overclaimed or underclaimed the vulnerable software versions with respect to the external reports. An example is shown in Figure 9 for `CVE-2005-4134`. Compared with CVE summary, NVD has *overclaimed* the vulnerable version for *Mozilla Firefox* and *Netscape Navigator*, given that NVD listed more vulnerable versions than CVE. On the contrary, for *K-Meleon*, NVD has *underclaimed* the vulnerable software version range.

Figure 11 shows the percentage of overclaimed and underclaimed NVD entries within the loosely matched pairs. "Strict-matched" pairs are not included in this analysis. We are not surprised to find NVD entries might overclaim. Given that NVD is supposed to search for different sources to keep the entries update-to-date, it is reasonable for the NVD entries to cover more vulnerable versions. Even if we take the union of the vulnerable versions across 5 websites and CVE, NVD is still covering more versions. The more interesting observation is that NVD has underclaimed entries compared to each of the external information sources. This suggests that NVD is either suffering from delays to update the entries or fails to keep track of the external information. Only a tiny portion of NVD entries contain both overclaimed and underclaimed versions (see the example in Figure 9).

Examples of Conflicting Cases. We then examined the conflicted pairs, and observed that a common reason for mis-

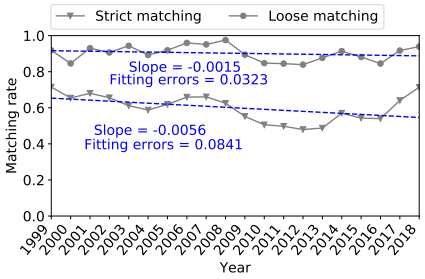


Figure 12: Matching rate over time: NVD vs. (CVE + 5 websites).

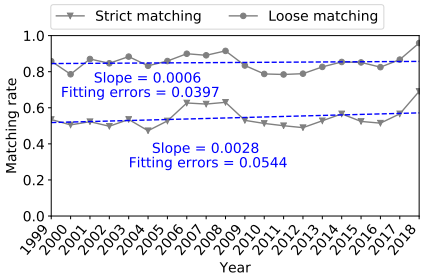


Figure 13: Matching rate over time: CVE vs. 5 websites.

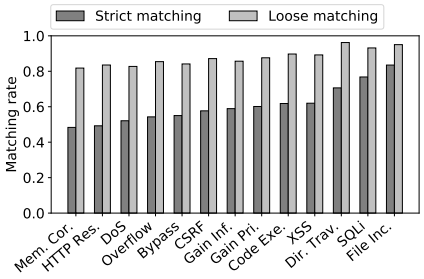


Figure 14: Matching rate for different vulnerability categories.

matching is *typos*. For example, under `CVE-2008-1862`, the versions listed on CVE and ExploitDB are both `0.22 and earlier` for software *ExBB Italia*. However, the NVD version is slightly different "`up to (including) 0.2.2`". Another example is `CVE-2010-0364` where the software versions from NVD and CVE summary are both `0.8.6` for *Videolan VLC media player*. However, the information on SecurityFocus has a clear typo as `0.6.8`.

In other cases, it is not clear whose information is correct. For example, sometimes the referenced vulnerable reports provide more detailed information than the CVE summary. For example, under `CVE-2012-1173`, the vulnerable version for *libtiff* is listed as `3.9.4` on NVD but SecurityTracker claims the vulnerable version should be `3.9.5`. Under `CVE-2000-0773`, software *Bajie HTTP web server* is vulnerable for version `1.0` according to NVD. However, CVE lists version `0.30a` and SecurityFocus lists `0.90, 0.92, 0.93`. There is no way to determine the correctness of the contradicting information at the pure text level, but we argue that the value of such measurement results is to point out the cases that need validation and correction.

Consistency Over Time. Figure 12 shows the consistency level between NVD and the other 6 information sources (CVE and the 5 report websites) is decreasing over time. The strict matching rate has some fluctuation over time but still shows a decreasing trend. We perform a linear regression for both matching rates and find both have a negative slope (-0.0015 and -0.0056 respectively). The result suggests the overall consistency drops over time in the past 20 years. However, if we take a closer look at the recent three years (2016 to 2018), the consistency level is starting to increase, which is a good sign.

Figure 13 shows a different trend when we compare the consistency between CVE and the 5 external websites. The consistency level between CVE and external sites is relatively stable with a slight upward trend. We perform a linear regression for both matching rates which returns a positive slope (0.0006 and 0.0028 respectively). This suggests that CVE websites are getting better at summarizing the vulnerability versions.

Types of Vulnerabilities. As shown in Figure 14, we break down the results based on the vulnerability categories. While the loose matching rates are still similar (around 90%), there are clear differences in their strict matching rates. For example, "SQL Injection" and "File Inclusion" have the highest strict matching rate (over 75%), but categories such as "Memory Corruption" have a much lower strict matching rate (48%). Further manual examination suggests that memory corruption vulnerabilities are typically more complex than those under File Inclusion or SQL Injection, and thus require a longer time to reproduce and validate. As a result, it is not uncommon for NVD to miss newly discovered vulnerable versions over time.

Inferring the Causes of Inconsistencies. Finally, we attempt to infer the causes of inconsistencies by analyzing the NVD entry creation/update time with respect to the posting time of the external reports. More specifically, NVD maintains a "change history" for each CVE ID, which allows us to extract the entry creation time, and the time when new software versions are added/removed. Then we can compare it with the posting time of corresponding reports at the 5 websites. For this analysis, we randomly select 5,000 CVE IDs whose vulnerable versions in NVD are inconsistent with those of the 5 websites.

We find that 66.3% of the NVD entries have never been updated since they were created for the first time. This includes 5.8% NVD entries that were created before any of the 5 websites posted their reports. For example, for `CVE-2006-6516`, NVD claimed *KDPics* `1.16` was vulnerable in 2006. Later in 2010, SecurityFocus reported that both version `1.11` and version `1.16` were vulnerable. NVD has not added the new version `1.11` until today. For the much bigger portion (60.5%) of NVD entries, they were created when at least one of the external reports were already available. An example is `CVE-2016-6855` as ExploitDB claimed *Eye of Gnome* `3.10.2` was vulnerable in August 2016. A month later, the NVD entry was created which did not include version `3.10.2`. No update has been made since then.

For the rest of the 33.7% of the NVD entries, they have made at least one update to the vulnerable versions after the entry creation. For them, we compare the latest update time

CVE ID	Software	Vul. Versions Claimed by Different Sources	Majority Vote	Union	Ground truth	Versions Manually Tested	# Newly Detected Versions by Us (12)	# Overclaimed Reports (15)
CVE-2004-2167	latex2rtf	**NVD:** 1.9.15 (1) **CVE:** 1.9.15 and possibly others (40) **SecurityFocus, SecurityTracker:** 1.9.15 (1) **IBM Security:** $\leq$ 1.9.15 (14)	1.9.15 (1)	1.9.15 and possibly others (40)	1.9.15 (1)	1.8aa - 2.3.17 (40)	0	1
CVE-2008-2950	poppler	**NVD, CVE, SecurityTracker, Security Forum, OCERT, CXsecurity, IBM Security:** $\leq$ 0.8.4 (34) **SecurityFocus, ExploitDB:** 0.8.4 (1) **RedHat:** $<$ 0.6.2 (22) **Gentoo:** $<$ 0.6.3 (23)	$\leq$ 0.8.4 (34)	$\leq$ 0.8.4 (34)	0.5.9 - 0.8.4 (16)	0.1 - 0.8.7 (37)	0	7
CVE-2009-5018	gif2png	**NVD:** 0.99 - 2.5.3 (36) **CVE, Openwall, IBM Security, Bugzilla:** $\leq$ 2.5.3 (36) **SecurityFocus:** 2.5.2 (1) **Gentoo:** $<$ 2.5.1 (33) **Fedora:** 2.5.1 (1)	$\leq$ 2.5.3 (36)	$\leq$ 2.5.3 (36)	2.4.2 - 2.5.6 (13)	0.7 - 2.5.8 (41)	2.5.4 - 2.5.6 (3)	4
CVE-2015-7805	libsndfile	**NVD, CVE, Openwall, Fedora, nemux, Packet Storm:** 1.0.25 (1) **ExploitDB:** $\leq$ 1.0.25 (30) **Gentoo:** $<$ 1.0.26 (30)	1.0.25 (1)	$\leq$ 1.0.25 (30)	1.0.15 - 1.0.25 (11)	0.0.8 - 1.0.26 (31)	0	2
CVE-2016-7445	openjpeg	**NVD, Gentoo:** $\leq$ 2.1.1 (16) **SecurityFocus, Openwall:** 2.1.1 (1) **CVE:** $<$ 2.1.2 (16)	2.1.1 (1)	$<$ 2.1.2 (16)	1.5 - 2.1.1 (7)	$\leq$ 2.2.0 (18)	0	1
CVE-2016-8676	libav	**NVD:** $\leq$ 11.8 (47) **CVE:** 11.9 (1) **SecurityFocus:** 11.3, 11.4, 11.5, 11.7 (4) **Openwall:** 11.8 (1) **agostino's blog**: 11.3 - 11.7 (5)	11.3, 11.4, 11.5, 11.7 (4)	11.3, 11.4, 11.5, 11.7, 11.8, 11.9 (6)	11.0 - 11.8 (9)	11.0 - 11.9 (10)	11.0, 11.1, 11.2, 11.6 (4)	0
CVE-2016-9556	ImageMagick	**NVD, CVE:** 7.0.3.8 (1) **SecurityFocus, Openwall, agostino's blog:** 7.0.3.6 (1)	7.0.3.6	7.0.3.6, 7.0.3.8 (2)	7.0.3.1 - 7.0.3.7 (7)	7.0.3.1 - 7.0.3.8 (8)	7.0.3.1 - 7.0.3.5 (5)	0

Table 7: The summary of case study results. The number in parentheses denotes the total number of software versions.

and the posting time of the external reports at the 5 websites. We find all of the NVD entries made the latest update *after* the posting time of some of the external reports. Overall, these results suggest that the NVD team did not effectively include the vulnerable versions from the external reports, despite that the reports were already available at the time of the entry creation/update. The results in turn reflect the need of automatically monitoring different online sources and extracting vulnerable versions for more proactive version testing and entry updating.

7 Case Study

To demonstrate the real-world implications of our inconsistency measurement, we perform case studies. We randomly select 7 real-world vulnerabilities from the *mismatched cases* in our dataset. Then we attempt to manually reproduce the vulnerabilities of the related software under different versions. These vulnerabilities are associated with 7 distinct CVE IDs, covering 47 vulnerability reports in total. Note that for the case study, we not only included CVE summaries and the reports from the 5 websites, but considered all other source websites in the reference lists of these CVE IDs. For the software mentioned in these reports, we exhaustively gathered all the versions of these software programs and obtained *185* versions of software in total. We list the number of unique versions for each software in Table 7.

With the collected software, we examine the vulnerabilities in each version. We form a team of 3 security researchers to manually analyze the source code of these software programs, and dynamically verify the reported vulnerabilities by manually crafting the PoC (proof-of-concept) input. The 185 software versions took us 4 months to fully verify.

The truly vulnerable versions are listed in the "ground-truth" column in Table 7. In total, out of the 185 software versions, we confirm that 64 versions are vulnerable. *12 of the truly vulnerable versions are discovered by us for the first time*, which have never been mentioned in existing vulnerability reports, or CVE/NVD entries.

Observation 1. Erroneous Information Confirmed. By comparing with the ground-truth vulnerable versions, we confirmed that most information sources including CVE and NVD have either missed real vulnerable versions or falsely included non-vulnerable versions. There are widespread and routine overclaims and underclaims. Given that many system administrators heavily rely on the information in vulnerability reports to assess the risk of their system and determine whether they need to upgrade their software, it is a big concern that the "underclaiming" problem could leave vulnerable software systems unpatched. The overclaims, on the other hand, could have wasted significant manual efforts from security analysts in performing risk assessments.

Observation 2. Benefits and Limits of Majority Voting. Given a vulnerability, if we take a majority voting among different information sources, we can diminish the "overclaiming" issue, which, however, amplifies the "underclaiming" issue. The result suggests that system administrators

and security analysts cannot simply utilize a majority voting mechanism to determine the risk of their software systems.

Observation 3. Benefits and Limits of Union. If we take a union set of all the claimed vulnerable versions, we can see the resulting vulnerable versions are having better coverage of the truly vulnerable versions. This indicates the benefit of broadly searching different online information sources and automatically extracting newly discovered vulnerable versions. While acknowledging the benefit, we also observe that the *union* approach is not perfect. First, the union set easily introduced overclaimed versions. Across all the vulnerabilities in Table 7, we find 15 external reports (not including NVD/CVE entries) where the claimed vulnerable versions turned out to be not vulnerable based on our tests (*i.e.,* overclaimed reports). Second, the union set sometimes fails to cover truly vulnerable versions. As shown in Table 7, we confirm 12 new vulnerable versions for `CVE-2009-5018`, `CVE-2016-8676`, and `CVE-2016-9556`. These vulnerable versions are discovered for the first time by us, by exhaustively testing all the available versions of the given software. In practice, our approach (testing all versions) is not scalable given the significant manual efforts required. To fully automate the vulnerability verification process is still an open challenge. Overall, the union approach at least helps to narrow down the testing space and improve the coverage of the truly vulnerable versions.

8 Discussion

Key Insights. The most important takeaway is NVD contains highly inconsistent information from external information sources and even the CVE list. The inconsistency involves both overclaiming and underclaiming problems. The implication is that system administers or security analysts cannot simply rely on the NVD/CVE information to determine the vulnerable versions of the affected software. At the very least, browsing external vulnerability reports can help to better cover the *potentially vulnerable* versions.

Our system `VIEM` makes it possible to *automatically* keep track of the information of different sources to generate a "diff" from the NVD/CVE entries periodically. This allows employees of NIST NVD and MITRE CVE to get notified when new vulnerable versions are reported in external websites, and helps them to focus on the most inconsistent or outdated entries, which potentially accelerates vulnerability testing, entry updating, and software patching. This was not possible without `VIEM`. Although vulnerability testing and verification are still largely manual efforts (automatically verifying the correctness of the vulnerability information in reports is not yet possible, which is an open problem), our main contribution is that we enable the automation for the information collection and standardization process.

Limitations. Our study has a few limitations. First, we only focus on the 5 most popular source websites in order to make the data collection process manageable (each website needs its own crawler and content parser). We argue that the 5 websites are referenced by 72.34% of all CVE IDs, and the results are representative. Future work will seek to expand the scope of the measurements. Second, our definition of "vulnerable software" is relatively broad, including all different parts that are related to (or affected by) the vulnerability (*e.g.*, dependent libraries, OSes, components, functions). One way to improve our system is to further classify the different types of "software names" (*e.g.*, differentiating vulnerable applications and the affected OSes). Finally, the scale of our case study is still limited. The 185 software versions already cost months to manually verify, and it is difficult to increase the scale further.

9 Related Work

NLP for Security. Natural Language Processing (NLP) has been applied to address different security problems. For example, researchers apply NLP to extract malware detection features from researcher papers [63] or systematically collect cyber threat intelligence from technical blogs [27,37]. Another line of work applies text analysis to mobile apps to study permission abuse and user input sanitization [33, 35, 43, 45, 46]. Finally, NLP has been used to analyze API documentation and infer security policies [55, 61].

A more relevant line of works has employed NLP techniques to facilitate the identification and assessment of software bugs [47, 51, 54, 58]. A recent work [59] proposed a method to extract relevant information from CVE to facilitate vulnerability reproduction, with a specific focus on kernel vulnerabilities. Our work is the first to build customized NLP models to extract vulnerable software names and versions from CVE and vulnerability reports. We apply the model to systematically measure information consistency.

Security Vulnerability Reports. CVE and vulnerability reports have been studied in various contexts. Breu et al. [26] showed that the interaction between software developers and bug reporters can facilitate the remediation of software bugs. Guo et al. [34] found that vulnerabilities reported by professionals with high reputations usually get fixed quicker. Bettenburg et al. [23] discovered that additional information provided by duplicated vulnerability reports can help to resolve the problem more timely. Mu et al. [42] showed that the missing information in vulnerability reports could reduce the success rate of vulnerability reproduction. The authors of [42] only tested if one of the versions is vulnerable, and we need to test all the versions for a given software to obtain the ground-truth (§7). Zhang et al. [60] used NVD data for security risk assessment. Christey et al. [31] pointed out the biased statistics in CVE. Nappa et al. [44] found missing and

extraneous vulnerable versions in NVD data (for 1500 vulnerabilities) by comparing the NVD version with those of the product security advisories. Our work differs from previous works by focusing on the information consistency between *unstructured* information sources and CVE/NVD entries for a large number of vulnerabilities.

10 Conclusion

In this paper, we design and develop an automated tool VIEM to conduct a large-scale measurement of the information consistency between the structured NVD database and unstructured CVE summaries and external vulnerability reports. Our results demonstrate that inconsistent information is highly prevalent. In-depth case studies confirm that the NVD/CVE database and third-party reports have either missed truly vulnerable software versions or falsely included non-vulnerable versions. The erroneous information could leave vulnerable software unpatched, or increase the manual efforts of security analysts for risk assessment. We believe there is an emerging need for the community to systematically rectify inaccurate claims in vulnerability reports.

Acknowledgments

We would like to thank our shepherd Giancarlo Pellegrino and the anonymous reviewers for their helpful feedback. We also want to thank Dongliang Mu and Jing Wang for their insightful suggestions. This project was supported in part by CSC scholarship, National Key Research and Development Program of China (2016YFB0800703), Chinese National Natural Science Foundation (U1836210), and NSF grants (CNS-1750101, CNS-1717028, CNS-1718459). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any funding agencies.

References

[1] Are there references available for cve entries? https://cve.mitre.org/about/faqs.html#cve_entry_references.

[2] CPE dictionary. https://nvd.nist.gov/products/cpe.

[3] CVE and NVD Relationship. https://cve.mitre.org/about/cve_and_nvd_relationship.html.

[4] CVE List. https://cve.mitre.org/cve/.

[5] Cve numbering authorities. https://cve.mitre.org/cve/cna.html.

[6] CVE Website. https://cve.mitre.org/.

[7] Exploitdb. https://www.exploit-db.com/.

[8] How are the cve entry descriptions created or compiled? https://cve.mitre.org/about/faqs.html#cve_entry_descriptions_created.

[9] How does a vulnerability or exposure become a cve entry? https://cve.mitre.org/about/faqs.html#cve_list_vulnerability_exposure.

[10] List of products. https://www.cvedetails.com/product-list.php.

[11] Nvd. https://nvd.nist.gov/.

[12] Nvd data feeds. https://nvd.nist.gov/vuln/data-feeds.

[13] Openwall. http://www.openwall.com/.

[14] Securityfocus. https://www.securityfocus.com/vulnerabilities.

[15] Securityfocus forum. https://www.securityfocus.com/archive/1.

[16] Securitytracker. https://securitytracker.com/.

[17] Vulnerability change record for cve-2014-9662. https://nvd.nist.gov/vuln/detail/CVE-2014-9662/change-record?changeRecordedOn=6%2f30%2f2016+1%3a55%3a54+PM&type=new#0.

[18] What happens after a vulnerability is identified? https://nvd.nist.gov/general/faq#1dc13c24-565f-46eb-90da-c5cac28a1e17.

[19] What is mitre's role in cve? https://cve.mitre.org/about/faqs.html#MITRE_role_in_cve.

[20] A cyberattack the world isn't ready for. The New York Times, 2017. https://www.nytimes.com/2017/06/22/technology/ransomware-attack-nsa-cyberweapons.html.

[21] Giant equifax data breach: 143 million people could be affected. CNN, 2017. https://money.cnn.com/2017/09/07/technology/business/equifax-data-breach/index.html.

[22] ANGELI, G., PREMKUMAR, M. J. J., AND MANNING, C. D. Leveraging linguistic structure for open domain information extraction. In *Proc. of ACL* (2015).

[23] BETTENBURG, N., PREMRAJ, R., ZIMMERMANN, T., AND KIM, S. Duplicate bug reports considered harmful... really? In *Proc. of ICSM* (2008).

[24] BIRD, S., AND LOPER, E. Nltk: the natural language toolkit. In *Proc. of ACL* (2004).

[25] BOJANOWSKI, P., GRAVE, E., JOULIN, A., AND MIKOLOV, T. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics* (2017).

[26] BREU, S., PREMRAJ, R., SILLITO, J., AND ZIMMERMANN, T. Information needs in bug reports: improving cooperation between developers and users. In *Proc. of CSCW* (2010).

[27] CATAKOGLU, O., BALDUZZI, M., AND BALZAROTTI, D. Automatic extraction of indicators of compromise for web applications. In *Proc. of WWW* (2016).

[28] CHAPARRO, O., LU, J., ZAMPETTI, F., MORENO, L., DI PENTA, M., MARCUS, A., BAVOTA, G., AND NG, V. Detecting missing information in bug descriptions. In *Proc. of FSE* (2017).

[29] CHEN, D., AND MANNING, C. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)* (2014), pp. 740–750.

[30] CHITICARIU, L., LI, Y., AND REISS, F. R. Rule-based information extraction is dead! long live rule-based information extraction systems! In *Proc. of EMNLP* (2013).

[31] CHRISTEY, S., AND MARTIN, B. Buying into the bias: Why vulnerability statistics suck. *BlackHat, Las Vegas, USA, Tech. Rep* (2013).

[32] DAVIES, S., AND ROPER, M. What's in a bug report? In *Proc. of ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ISESE)* (2014).

[33] GORLA, A., TAVECCHIA, I., GROSS, F., AND ZELLER, A. Checking app behavior against app descriptions. In *Proc. of ICSE* (2014).

[34] GUO, P. J., ZIMMERMANN, T., NAGAPPAN, N., AND MURPHY, B. Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In *Proc. of ICSE* (2010).

[35] HUANG, J., LI, Z., XIAO, X., WU, Z., LU, K., ZHANG, X., AND JIANG, G. Supor: Precise and scalable sensitive user input detection for android apps. In *Proc. of USENIX Security* (2015).

[36] LAMPLE, G., BALLESTEROS, M., SUBRAMANIAN, S., KAWAKAMI, K., AND DYER, C. Neural architectures for named entity recognition. In *Proc. of NAACL-HLT* (2016).

[37] LIAO, X., YUAN, K., WANG, X., LI, Z., XING, L., AND BEYAH, R. Acing the ioc game: Toward automatic discovery and analysis of open-source cyber threat intelligence. In *Proc. of CCS* (2016).

[38] LIN, Y., SHEN, S., LIU, Z., LUAN, H., AND SUN, M. Neural relation extraction with selective attention over instances. In *Proc. of ACL* (2016).

[39] MIKOLOV, T., CHEN, K., CORRADO, G., AND DEAN, J. Efficient estimation of word representations in vector space. *arXiv:1301.3781* (2013).

[40] MIKOLOV, T., SUTSKEVER, I., CHEN, K., CORRADO, G. S., AND DEAN, J. Distributed representations of words and phrases and their compositionality. In *Proc. of NIPS* (2013).

[41] MINTZ, M., BILLS, S., SNOW, R., AND JURAFSKY, D. Distant supervision for relation extraction without labeled data. In *Proc. of ACL* (2009).

[42] MU, D., CUEVAS, A., YANG, L., HU, H., XING, X., MAO, B., AND WANG, G. Understanding the reproducibility of crowd-reported security vulnerabilities. In *Proc. of USENIX Security* (2018).

[43] NAN, Y., YANG, M., YANG, Z., ZHOU, S., GU, G., AND WANG, X. Uipicker: User-input privacy identification in mobile applications. In *Proc. of USENIX Security* (2015).

[44] NAPPA, A., JOHNSON, R., BILGE, L., CABALLERO, J., AND DUMITRAS, T. The attack of the clones: A study of the impact of shared code on vulnerability patching. In *Proc. of S&P* (2015).

[45] PANDITA, R., XIAO, X., YANG, W., ENCK, W., AND XIE, T. Whyper: Towards automating risk assessment of mobile applications. In *Proc. of USENIX Security* (2013).

[46] QU, Z., RASTOGI, V., ZHANG, X., CHEN, Y., ZHU, T., AND CHEN, Z. Autocog: Measuring the description-to-permission fidelity in android applications. In *Proc. of CCS* (2014).

[47] SAHA, R. K., LEASE, M., KHURSHID, S., AND PERRY, D. E. Improving bug localization using structured information retrieval. In *Proc. of ASE* (2013).

[48] SMITH, A., AND OSBORNE, M. Using gazetteers in discriminative information extraction. In *Proc. of Computational Natural Language Learning* (2006).

[49] STANOVSKY, G., MICHAEL, J., ZETTLEMOYER, L., AND DAGAN, I. Supervised open information extraction. In *Proc. of ACL* (2018).

[50] SUTSKEVER, I., VINYALS, O., AND LE, Q. V. Sequence to sequence learning with neural networks. In *Proc. of NIPS* (2014).

[51] TAN, L., YUAN, D., KRISHNA, G., AND ZHOU, Y. /* icomment: Bugs or bad comments?*. In *ACM SIGOPS Operating Systems Review* (2007).

[52] TOUTANOVA, K., KLEIN, D., MANNING, C. D., AND SINGER, Y. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1* (2003), Association for Computational Linguistics, pp. 173–180.

[53] WANG, H., WANG, C., ZHAI, C., AND HAN, J. Learning online discussion structures by conditional random fields. In *Proc. of SIGIR* (2011).

[54] WONG, C.-P., XIONG, Y., ZHANG, H., HAO, D., ZHANG, L., AND MEI, H. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *Proc. of International Conference on Software Maintenance and Evolution (ICSME)* (2014).

[55] XIAO, X., PARADKAR, A., THUMMALAPENTA, S., AND XIE, T. Automated extraction of security policies from natural-language software documents. In *Proc. of SIGSOFT* (2012).

[56] YANG, Z., SALAKHUTDINOV, R., AND COHEN, W. W. Transfer learning for sequence tagging with hierarchical recurrent networks. *arXiv preprint arXiv:1703.06345* (2017).

[57] YANG, Z., YANG, D., DYER, C., HE, X., SMOLA, A., AND HOVY, E. Hierarchical attention networks for document classification. In *Proc. of NAACL-HLT* (2016).

[58] YE, X., BUNESCU, R., AND LIU, C. Learning to rank relevant files for bug reports using domain knowledge. In *Proc. of SIGSOFT* (2014).

[59] YOU, W., ZONG, P., CHEN, K., WANG, X., LIAO, X., BIAN, P., AND LIANG, B. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proc. of CCS* (2017).

[60] ZHANG, S., OU, X., AND CARAGEA, D. Predicting cyber risks through national vulnerability database. *Information Security Journal: A Global Perspective* (2015).

[61] ZHONG, H., ZHANG, L., XIE, T., AND MEI, H. Inferring resource specifications from natural language api documentation. In *Proc. of ASE* (2009).

[62] ZHOU, P., SHI, W., TIAN, J., QI, Z., LI, B., HAO, H., AND XU, B. Attention-based bidirectional long short-term memory networks for relation classification. In *Proc. of ACL* (2016).

[63] ZHU, Z., AND DUMITRAS, T. Featuresmith: Automatically engineering features for malware detection by mining the security literature. In *Proc. of CCS* (2016).

Appendix

A Model Parameters

The have tested the model performance with respect to different parameter settings for the NER model and the RE model. Figure 15 shows the RE model performance under different position embedding dimensions. Figure 16 to Figure 18 show the NER model performance under different batch sizes, the number of network layers, and the number of epochs. The result shows that our model is not very sensitive to parameter settings.

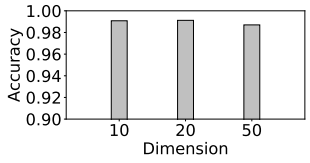


Figure 15: Position embed dim. vs. RE accuracy.

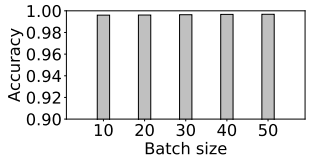


Figure 16: Batch size vs. NER accuracy.

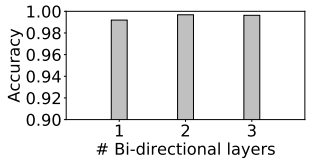


Figure 17: # Bi-directional layers vs. NER accuracy.

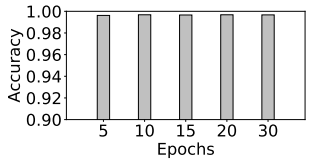


Figure 18: # Epochs vs. NER accuracy.

Understanding and Securing Device Vulnerabilities through Automated Bug Report Analysis

Xuan Feng[1,2,5*], Xiaojing Liao[2], XiaoFeng Wang[2], Haining Wang[3]
Qiang Li[4], Kai Yang[1,5], Hongsong Zhu[1,5], Limin Sun[1,5†]

[1] *Beijing Key Laboratory of IoT Information Security Technology, IIE[‡], CAS, China*
[2] *Department of Computer Science, Indiana University Bloomington, USA*
[3] *Department of Electrical and Computer Engineering, University of Delaware, USA*
[4] *School of Computer and Information Technology, Beijing Jiaotong University, China*
[5] *School of Cyber Security, University of Chinese Academy of Sciences , China*

Abstract

Recent years have witnessed the rise of Internet-of-Things (IoT) based cyber attacks. These attacks, as expected, are launched from compromised IoT devices by exploiting security flaws already known. Less clear, however, are the fundamental causes of the pervasiveness of IoT device vulnerabilities and their security implications, particularly in how they affect ongoing cybercrimes. To better understand the problems and seek effective means to suppress the wave of IoT-based attacks, we conduct a comprehensive study based on a large number of real-world attack traces collected from our honeypots, attack tools purchased from the underground, and information collected from high-profile IoT attacks. This study sheds new light on the device vulnerabilities of today's IoT systems and their security implications: ongoing cyber attacks heavily rely on these known vulnerabilities and the attack code released through their reports; on the other hand, such a reliance on known vulnerabilities can actually be used against adversaries. The same bug reports that enable the development of an attack at an exceedingly low cost can also be leveraged to extract vulnerability-specific features that help stop the attack. In particular, we leverage Natural Language Processing (NLP) to automatically collect and analyze more than 7,500 security reports (with 12,286 security critical IoT flaws in total) scattered across bug-reporting blogs, forums, and mailing lists on the Internet. We show that signatures can be automatically generated through an NLP-based report analysis, and be used by intrusion detection or firewall systems to effectively mitigate the threats from today's IoT-based attacks.

1 Introduction

The pervasiveness of Internet-of-Things (IoT) systems, ranging from cameras, routers, and printers to various home automation, industrial control and medical systems, also brings in new security challenges: they are more vulnerable than conventional computing systems. IoT systems may fall into an adversary's grip once their vulnerabilities are exposed. Indeed, recent years have seen a wave of high-profile IoT related cyber attacks, with prominent examples including IoT Reaper [25], Hajime [24], and Mirai [41]. Particularly, compromised IoT systems are becoming the mainstay for today's botnets, playing a central role in recent distributed denial-of-service attacks and other cybercrimes [39]. An example is Mirai, which involved hundreds of thousands of IoT devices and generated an attack volume at 600 Gbps, overwhelming Krebs on Security, OVH, and Dyn. It does not come as a surprise that these devices are reported to be hacked all through known vulnerabilities, including those caused by misconfiguration or mismanagement, just like unpatched servers and personal computers that are routinely exploited [42]. Still less clear, however, are the fundamental causes for this trend of malicious activities that are disproportionately related to IoT systems and these vulnerable devices' impact upon the cybercrime ecosystem.

Understanding the perilous IoT world. More specifically, we raise the following questions: Why are IoT devices more favorable to cybercriminals than other Internet hosts? How important are known vulnerabilities to the ongoing attacks on IoT systems? Is there an effective defense to mitigate ongoing attacks? The answers to these questions are critical for seeking an effective means to thwart the wave of malicious attacks that increasingly rely on a large number of vulnerable IoT devices. Finding these answers, however, is by no means trivial due to the challenges in (1) recovering disclosed IoT vulnerabilities from a large number of vulnerability reports scattered around forums, mailing lists, and blogs, and (2) collecting artifacts from the cybercrime underground to study how known flaws are utilized and how significant they are to ongoing criminal activities.

To understand how cybercriminals use these known vulnerabilities, we set up honeypots to collect the data of real-world IoT exploits and also analyzed four popular attack toolkits. From the adversary's end, our study shows that today's IoT

*Work was done while visiting Indiana University Bloomington.
†Corresponding author
‡Institute of Information Engineering

attacks almost *exclusively* use known vulnerabilities for exploitation. Specifically, among 81 different exploit scripts recovered from our honeypots, we found that 78 of them are on our vulnerability list. Also, each of the four attack toolkits we studied incorporates the exploits on at least 34 vulnerabilities, with all of them on our list. More importantly, we evidence that an adversary extensively leverages the exploit code released together with the vulnerability reports, and in most cases, directly copies the code or slightly adjusts it. More than 80% of the IoT-related reports come with working attack methods. Given our observations that *most of* IoT vulnerabilities can be exploited to attack target devices (compared with only 5% exploitable ones in Linux kernel vulnerabilities [43]), it is apparent that the cost for attacking IoT systems today is *exceedingly low*.

Automated protection generation. On the other hand, we show that adversaries' indulgence in such low-hanging fruits can actually be used against themselves. Specifically, the reliance on known vulnerabilities means that a large attack surface would be closed once these problems have been fixed. Interestingly, this turns out to be completely feasible, due to the simplicity of the problems and the availability of the vulnerability disclosure and attack code that carries all the information we need to fix the reported flaws. Based on this observation, we developed a framework, called *IoTShield*, which utilizes Natural-Language Processing (NLP) to automatically evaluate the content of vulnerability reports. In particular, IoTShield is based upon a set of automatic content analysis techniques that accurately discover IoT-related vulnerability disclosures from a large number of vulnerability reports published at different sources.

Using the approaches above, we automatically analyzed 430,000 vulnerability reports and discovered more than 7,500 IoT reports in the past 20 years. Then, IoTShield extracts key knowledge from these reports to automatically generate *vulnerability-specific* signatures, which can be used by existing intrusion detection systems (IDSes) or web application firewalls (WAFs) to screen the traffic received by IoT devices under protection. We validated the efficacy of IoTShield over 178,778 traces harvested by our honeypots, as well as 11,602 traces of eight real devices, including both attack and legitimate traffic. Furthermore, we evaluated the effectiveness of IoTShield over a long-time (more than one year) traffic captured in an industrial control system's human machine interface (HMI) Honeypot. The evaluation results show that IoTShield achieves a high precision (above 97%) and a very low false positive rate with minor performance impact.

Contributions. The major contributions of this work are outlined as follows:

• *New discovery*. Leveraging the collected traces of real-world IoT exploits and analysis results from popular attack toolkits, we bring to light new observations, including the adversary's exclusive use of known flaws and published code. These observations demonstrate that IoT devices are indeed more vulnerable, less patchable, and easier to attack than traditional Internet hosts, elucidating the ongoing cybercrime trend that heavily relies on these devices.

• *New defense*. More importantly, these findings also present us with an opportunity that could lead to the immediate defeat of most attack vectors in today's IoT-related attacks. We demonstrate that from the same sources adversaries use to build their exploits, vulnerability signatures can be *automatically* generated using NLP, and be quickly deployed to shield IoT devices from malicious attacks. Given the adversary's dependence on these known vulnerabilities, we believe that this simple, low cost yet effective defense will significantly raise the bar for future IoT-related attacks.

Roadmap. The rest of the paper is organized as follows. Section 2 presents the background and our threat model. Section 3 describes our new understandings on IoT vulnerabilities and real-world threats to IoT devices. Section 4 introduces the automated protection generation to defend against IoT attacks. Sections 5 and 6 present the implementation and evaluation of our automated protection generation, respectively. Section 7 discusses the limitations of this study and mitigation. Section 8 surveys related work, and finally, Section 9 concludes the paper.

2 Background

IoT vulnerability life cycle. Usually, an IoT device's vulnerability is a loophole in its firmware that enables an attacker to circumvent the deployed security measures [57]. Such a vulnerability has a life cycle with distinct phases characterized by its discovery, disclosure, exploitation, and patching. The first phase starts when the vulnerability is discovered by a vendor, a hacker, or a third-party security researcher. The security risk becomes particularly high if it is first found by hackers. The next phase is the public disclosure of the vulnerability by those who discover it, and is supposed to be done through a coordinated process [22], during which the vulnerability information is kept confidential allowing vendors to create a patch. However, this procedure is not always followed. Actual real-world disclosures could happen in different ways through sources across the Internet, including personal blogs, public forums, and security mailing lists. Once publicly disclosed, the information about a vulnerability is freely available to anyone. Thus, the level of security risk further increases as the hacker community is active in developing and releasing zero-day exploits [40]. From our observations, about 80% of IoT vulnerability reports are released together with exploitable methods, which can be readily utilized by hackers.

Even worse, a vendor may not provide any security updates or patches in response to a disclosure, even though it is supposed to do so. Also, even with the patches available, it is not uncommon that many users of the affected IoT devices do not

install them, given the complicated procedure (for ordinary users) to patch the firmware.

The life cycle of an IoT vulnerability ends when all IoT users install the patch to fix the vulnerability. However, even if an IoT device has a serious security vulnerability and vendors have released the patch, some users have no capability of updating patches in a timely manner due to their limited knowledge. From our observations, the life cycle of some IoT vulnerabilities lasts more than five years, during which these problems can be exploited at any time.

Signature-based IDS. An intrusion detection system (IDS) monitors a network or a system for malicious activities or policy violations. These systems can be signature-based or anomaly-based. Signature-based detection leverages known patterns (signatures) of malicious code or operations to identify malicious activities, while anomaly-based detection captures deviations from a system's normal profile. The focus of our protection is to provide automatic signature generation for the signature-based detection systems, such as Snort [37].

Signatures can be used to describe a specific attack on a vulnerability or model the vulnerability itself. The latter, which provides comprehensive protection against *all* related attacks on the weakness, is called a *vulnerability-specific* signature. Such a signature is typically created through manual analysis of a vulnerability. In our study, however, we found that the rich information about IoT vulnerabilities from various sources (blogs, mailing lists, and forums) can actually be leveraged to *automatically* generate such a signature, using NLP techniques.

Natural language processing. Our research utilizes various NLP techniques to analyze the text content of various vulnerability reports. A spell-checking [1] is used to filter out documents irrelevant to IoT. Regular expression based pattern matching is utilized to identify the vulnerability reports related to IoT. Further, semantic consistency analysis is used to examine the extracted IoT entities. To generate a vulnerability-specific signature, in-depth understanding of the vulnerability semantics is needed and can be achieved by using a grammatical dependency parser [51], which provides a representation of grammatical relations between words in a sentence.

Threat model. In this work, we consider an adversary who attempts to exploit the security flaws disclosed in a vulnerability report to compromise remote, Internet-connected IoT devices. For this purpose, the adversary can compromise the remote IoT devices and use them to launch malicious attacks. In particular, we focus on Internet-connected IoT devices (e.g., IP cameras, routers, and printers) that expose their attack surfaces on the Internet. Accordingly, adversaries can exploit those devices' security flaws remotely, and gain unauthorized control of those vulnerable devices (e.g., a compromised IoT device may become a botnet node).

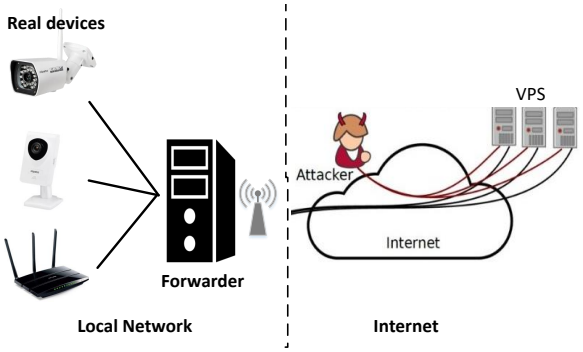


Figure 1: The infrastructure of our real device honeypots.

3 Understanding Real-World Threats

To understand how IoT vulnerabilities are exploited by adversaries, we deployed honeypots, analyzed underground attack toolkits, and studied prior attack reports. Here we elaborate our findings.

3.1 Honeypot

Our honeypots include real IoT devices and simulated devices for collecting real-world attack activities.

Real device honeypot. We deployed eight vulnerable IoT devices (three routers and five cameras) as honeypots from May 2018 to June 2018. These devices and their corresponding vulnerabilities are listed in Table 1. We chose these devices because they are typical IoT systems being exploited in various real-world attacks [41].

Figure 1 illustrates our honeypot system's infrastructure. To increase these devices' IP diversity, we rented Virtual Private Servers (VPSes) across different countries as relay hosts for each IoT device. Whenever an attacker connects to a VPS, we redirect this request to the corresponding IoT device and forward its response back to the attacker.

More specifically, we use the reverse SSH tunneling [33] to bridge the gap between the IoT devices behind NAT and VPSes. For this purpose, we set up a persistent SSH tunnel from our device to its corresponding VPS, which is configured with the SSH port forwarding command [38] to send the traffic received from the VPS' public IP on the HTTP port to FORWARD_PORT (pointing to the IoT device). For devices not supporting the SSH protocol, we set up a PC as FORWARD_PORT, passing the received traffic to the device.

Simulated honeypot. To study the attacks on more vulnerable devices, we also deployed four simulated IoT honeypots from May 2018 to July 2018[1] across two countries (Canada and the United States) and four cities (Buffalo, Los Angeles,

[1]The timeline for simulated honeypot data collection is not consistent with that of the real device honeypot, due to the data loss caused by a misconfiguration event in our real device honeypot.

Table 1: IoT device honeypot.

Products	Vulnerabilities
Linksys WVC54GCA	Multiple XSS [10]
	Absolute path traversal [12]
	Stored passwords/keys [13]
	Directory traversal (adm/file.cgi) [11]
TP-Link TL-SC3171G	Authentication bypass [19]
	OS Command injection [14]
	Hard-coded credentials [15]
	Unauthenticated file uploads [16]
	Unauthenticated firmware upgrades [17]
Dahua IPC-HF2100	Hard-coded Credentials [18]
D-Link DIR-645	Authentication Bypass [21]
TVT TD-9436T	Command execution [32]
Easyn Model:10D	Unreported 0-day
TP-link TL-WAR458L	Remote command execution [20]
TP-Link TL-WR941N	Backdoor [8]

Table 2: Four simulated honeypots.

Name	Vulnerability	Affected products
D-link SOAP [23]	command injection	at least 12 products
GoAhead webs [27]	7 CVEs	1,250 affected models
JAWS [30]	command injection	all DVR running JAWS
Avtech [4]	14 vulnerabilities	all Avtech devices

Table 3: Traffic analysis of deployed honeypots.

	Real devices	Simulated honeypots
Malicious (Targeted)	20	∼300
Malicious (Blind-scanned)	121	∼1,560
Benign	11,451	176,764
Unknown	10	∼154
Total	11,602	178,778

Canyon Country, and Beauharnois), which cover more than 2,000 devices and 23 vulnerabilities.

The settings of these simulators are listed in Table 2. The Avtech honeypot covers 14 vulnerabilities, such as authentication bypass and command injection, which affect all Avtech devices (i.e., IP camera, NVR, and DVR) and firmware versions. The "GoAhead webs" honeypot simulates the GoAhead IP camera using a GoAhead-webs HTTP server. The honeypot covers seven critical vulnerabilities, such as the backdoor account and the pre-auth information leakage. Particularly, the latter is found in more than 1,250 different camera models. This enables us to significantly increase the number of vulnerable devices in our study.

Specifically, each honeypot is an HTTP server (on Apache), whose default configurations (such as default page and HTTP response header/body) have been modified to simulate real devices. If a honeypot finds an IP address that attempts to connect to our honeypot, it records the request packets from the IP and their timestamps before sending back "200 OK" and redirecting all HTTP requests to our main page. Note that some attacks may first send a harmless request to identify their target devices. In this case, returning a "200 OK" often triggers follow-up attack behaviors. Also, all of our simulators are indexed as real IoT devices in Shodan [59], and so they can be discovered from the device search engine.

Analysis. From May to July in 2018, our honeypots gathered 190,380 HTTP requests from 47,089 IPs across 175 countries. We analyzed these traffic traces as follows: first we removed those confirmed to be legitimate, including legitimate login attempts, the requests like "`GET /`", and other ordinary HTTP GET requests; then we scanned the traces for attack attempts by searching the exploit code in our vulnerability dataset and further looking for the common attack traffic patterns, such as the presence of SQL commands and various execution commands (e.g., "`cmd=/usr/bin/telnetd`"). In this way, we identified nearly 2,000 IoT exploit attempts from 60 different countries, with an average of 38 attacks per day.

Table 3 presents the results in detail: for the real device honeypot, 141 unique attacks with 26 different scripts were captured, and 1,860 unique attacks through 81 attack scripts were found from our simulated honeypots. About 164 *unknown* requests still cannot be confirmed to be legitimate or malicious.

Analyzing these attacks (2,001 in total), we found that about 320 of them aimed at the honeypot (real or simulated) devices, while about 1,681 targeted the devices whose types are not covered in any honeypots. On one hand, this indicates that an adversary may blindly conduct an attack without first identifying the device type. On the other hand, this implies that our honeypots have a wide-spectrum attack coverage, not limited by the types of devices deployed at honeypots. More importantly, there are only 164 unknown requests observed by our honeypots. Even if we *conservatively* assume that all unknown requests originate from individual malicious attacks that exploit unknown security flaws, less than 10% (164 vs 2,001) of the total attacks exposed to our honeypots are such unknown attacks. In other words, more than 90% of malicious attacks exploit the *known* vulnerabilities. This observation is consistent with our analysis of underground attack toolkits (see Section 3.2).

Most commonly exploited vulnerabilities found in our study include unauthenticated command injection and information disclosure. These flaws can be easily attacked by sending a simple HTTP request to the vulnerable device to gain full control of the device. An intriguing observation is that 96% of the IoT attacks use the same or similar scripts included in the vulnerability reports we collected. For example, an attacker compromised our device TVT TD-9436T through a command execution vulnerability by using the exact same code documented in the report [32]. In another exploit (i.e., `/board.cgi?cmd=/usr/sbin/telnetd`) [28] on the same vulnerability location "board.cgi", the only change found in the attack code was an adjustment of a parameter from "`/usr/sbin/telnetd`" to the Linux command "`cat+/etc/passwd`".

Table 4: Underground IoT attack tools.

Name	Vulnerabilities
IPCAM exploits	Pre-Auth Info Leak
Huawei Exploits	Command Execution
iotNigger	Netis Backdoor
Brickerbot	More than 30 vulnerabilities

Table 5: Known IoT attack activities.

Name	Vulnerabilities	Year
IOT Reaper [25]	10 vulnerabilities	2017
Hajime [24]	at least 3 vulnerabilities	2016
Satori [34]	2 vulnerabilities	2018
Brickerbot [6]	21 vulnerabilities	2017
Masuta [26]	bypass & command execution	2018
Amnesia [3]	remote code execution	2017

3.2 Artifacts from Other Sources

To validate the findings made from the honeypots, we further analyzed four underground attack toolkits and six well-documented IoT botnets, which are elaborated below.

Underground attack tools. In this work, we searched popular underground marketplaces (such as openbazaar and dreammarket) by using a set of keywords related to attack toolkits (such as IoT malware names listed in Table 12 of Appendix), in an attempt to find the posts selling such tools. Once the posts were discovered, we contacted the sellers and purchased the tools. Altogether, we obtained four such tools with their source code (see Table 4), including Brickerbot, a variation for the one used in the famous Brickerbot attack [6].

By analyzing their code, we again found that *all* vulnerabilities they exploit are known ones, and their attack scripts are *all* copied from the vulnerability reports with minor changes (e.g., C&C server IP) to customize for specific attack campaigns. More specifically, from these tools, we identified 99 different attack scripts related to 34 vulnerabilities. Those vulnerabilities are all recorded in our dataset. Among the 99 attack scripts, three of them are exactly the same as those documented, while the remaining 96 all have small changes. As an example in Table 4, the Huawei exploit on an arbitrary command execution vulnerability apparently comes from exploits-db (https://www.exploit-db.com/exploits/43414/), with only its Linux command (e.g., "ls") changed to a new one (e.g., communicate with a specific remote server). Again, this confirms what we observed by analyzing our honeypot data: most IoT attacks utilize known vulnerabilities and even the attack code provided in the vulnerability reports, which on the other hand could be leveraged to quickly suppress this emerging attack wave.

Known attacks. Finally, we analyzed some well-known, well-documented IoT botnets that were recently reported (Table 5) to understand whether mostly known vulnerabilities and documented attack scripts were indeed used. These botnets were all aimed at the security flaws that affect many different products (like IoT Reaper) or those widely deployed on the Internet (like Masuta). Some of them (IoT Reaper and Amnesia) also focus on the vulnerabilities without patches.

Once again, we found that all the vulnerabilities exploited in these attacks are also included in the reports gathered in our research, and all the scripts attacking these flaws are the copies or variations of the code in the reports. For example, IoT Reaper attacks 37 vulnerabilities documented by 10 different reports: 10 for remote command execution, at least 24 not on any CVE and 7 without patches. Note that *all* these attacks took place *after* the disclosure of related vulnerabilities. For instance, IoT Reaper was brought to light in 2017, while the flaw it uses, Linksys E1500/E2500 vulnerabilities, was made public in 2013.

4 Automated Vulnerability-specific Signature Generation

As mentioned earlier, the IoT vulnerability ecosystem has a serious problem: the attack scripts are often publicly available in the vulnerability reports, making those known vulnerabilities easy to exploit. Such a problem has led to the spree of large-scale IoT based attacks in recent years, which almost exclusively exploit known flaws.

In the meantime, the adversary's reliance on the well-documented vulnerabilities also presents a new opportunity for mitigating such threats. From the same vulnerability reports, vulnerability-specific information can be extracted to form a protection strategy that stops all attacks on the vulnerability. In our research, we developed IoTShield, an automatic tool that collects IoT vulnerability reports from the Internet, analyzes the content of the IoT vulnerability reports, and recovers key knowledge to generate *vulnerability-specific* signatures, with their qualities determined by the comprehensiveness of the reports. These signatures can be easily deployed to existing intrusion detection systems (IDSes) or web application firewalls (WAFs) to detect exploit attempts on the target device from the traffic it receives. In the following, we elaborate on this approach.

4.1 Overview and Data

Collecting IoT vulnerability reports from Internet and extracting vulnerability-specific knowledge from the collected IoT vulnerability reports for signature generation is a non-trivial task, with unique technical challenges. First, a large number of vulnerability reports are scattered around forums, mailing lists, and blogs with different format written by different people. It is difficult to identify IoT vulnerability reports from other documents. Second, such identified IoT vulnerability reports describe security flaws in natural language, which makes a large-scale discovery of vulnerability information difficult.

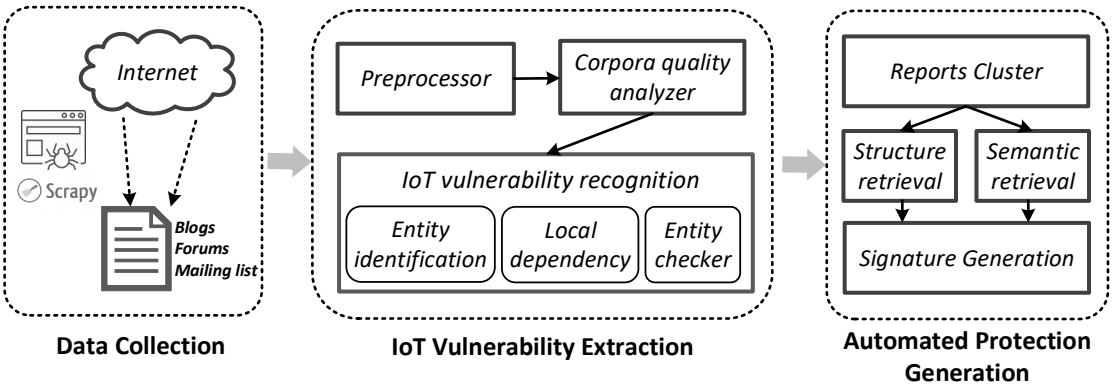


Figure 2: The architecture of IoTShield.

Third, identifying critical elements for a signature requires domain-specific knowledge to carefully distinguish between exploit-specific information and vulnerability-specific information. To address these challenges, in IoTShield, we use an IoT vulnerability extractor to remove irrelevant content and identify key information of IoT security flaws. After the collection of IoT vulnerability reports, we extract the semantics of vulnerability descriptions and other structured information (e.g., attack scripts) from the vulnerability reports. The descriptions here provide information about all circumstances under which a vulnerability can be exploited (e.g., all related parameters and locations), which enables us to leverage the attack surface observed from an attack script or other structured information like traffic logs to stop other attacks. A problem is that some of the vulnerability reports do not have vulnerability descriptions or structured information. Later we discuss how we handle this issue (see Section 4.3).

Architecture. Figure 2 illustrates the architecture of IoTShield, which has three major components: (1) data collection, (2) IoT vulnerability extraction, and (3) automated protection generation. Data collection is used to gather vulnerability reports from the Internet. The IoT vulnerability extraction is used to extract IoT vulnerability information from a large number of documents, including forums, blogs, and mailing lists. More specifically, we crawled popular online sources for bug disclosure and further ran a corpora quality analyzer to filter out the documents that are irrelevant to vulnerability reports. For the remaining documents, we used a recognizer to identify IoT vulnerability reports and extracted key information, such as their types, affected products, CVE numbers, authors, and published dates. This information will serve as the description for the later signature generation stage. Once given a set of IoT vulnerability reports, IoTShield first clusters the reports describing the same vulnerability together, and then utilizes NLP to discover the vulnerability semantics (e.g., vulnerability type, location, and parameters) from the descriptions. Then, it extracts the structured information (e.g., traffic logs, scripts, and Linux commands) to create an exploit (or PoC) template and find illegal parameters (e.g., those used to inject commands). After that, the signature generation component utilizes the vulnerability location and parameters to identify all related attack surfaces, which helps to determine all parameters for the template that can also lead to exploits. The parameters and template here form the signature. Finally, IoTShield automatically transforms the signature into the format used in existing IDSes or WAFs (the prototype built in our research outputs a Snort signature [37]) for a fast deployment.

Table 6: List of vulnerability reporting websites.

Categories	Website	Reports	IoT reports
Personal Blogs	s3cur1ty.de/advisories	28	16
	pierrekim.github.io	18	13
	gulftech.org	129	5
Forums	seclists.org/fulldisclosure	108,647	1,219
Team Blogs	coresecurity.com	390	31
	vulnerabilitylab.com	2,122	39
	blogs.securiteam.com	1,925	42
Mailing lists	seclists.org/bugtraq	85,593	1,591
Data Archive	exploit-db.com	39,380	895
	packetstormsecurity.com	97,093	1,951
	0day.today	30,177	834
	seebug.com	56,413	690
	myhack58	7,311	150
Total	-	42,9795	7,514

Dataset. As mentioned earlier, we ran our crawler across the vulnerability reporting websites listed in Table 6, including forums (`seclist.org/fulldisclosure`), mailing lists (`seclist.org/bugtraq`), personal blogs/advisories (`pierrekim.github.io`), research team advisories (`coresecurity.com`), and vulnerability archive websites (`packetstromsecurity.com`). These sources were collected from the external references included in CVEs, among which we selected the most frequently used ones. From these sources, we further manually picked out those related to IoT vulnerabilities, like `seclist.org`, and added them to the list, which also includes some research groups' websites known to report security vulnerabilities.

4.2 IoT Vulnerability Extraction

Preprocessing. Over the documents collected by the crawler, our IoT vulnerability extraction component removes the textual information irrelevant to vulnerabilities, such as advertisements, pictures, dynamical scripts, and navigation bar, while keeping the main content of each webpage with document URLs, document titles, authors, and publication dates. Since different websites have different templates and HTML structures, we manually analyzed each of them (13 vulnerability reporting sites in total, see Table 6) to identify useful content.

Corpora quality analyzer. After the preprocessing, we still need to filter out the documents irrelevant to IoT vulnerability reports, which is done as follows:

• *The percentage of dictionary words*. We removed the documents whose content contains mostly (above 82% in our research) dictionary words, which are recognized by enchant library [1], since a real vulnerability report always includes a significant amount of non-text information, like vulnerable paths and functions, PoC or scripts, etc. Otherwise, the text looks more like a survey article, white paper or notification.

• *The number of hyperlinks*. In general, a vulnerability report, particularly for IoT, is not supposed to include too many hyperlinks. Otherwise, it could be a summary for all the vulnerabilities disclosed, instead of a specific report with detailed vulnerability information. Thus, we discarded the documents with more than 25 hyperlinks.

•*Threshold justification*. The two threshold values above (i.e., 82% and 25) are based on our empirical experience, for the purpose of filtering out most non-IoT vulnerability reports with little collateral damage. To justify our threshold configuration, we attempt to estimate the possibility of a real IoT vulnerability report being wrongly discarded. To this end, we randomly sampled 100 documents being discarded and then manually examined whether there is any wrongly discarded case (i.e., a real IoT vulnerability report but discarded as non-IoT). What we found is that all of the sampled documents are irrelevant to IoT vulnerability reports (neither related to IoT vulnerability nor containing any vulnerability details). In the other words, no real IoT case exists among these 100 randomly sampled documents that are discarded as non-IoT. This implies that our empirical threshold configuration is very effective to filter out non-IoT vulnerability reports.

IoT vulnerability recognition. For the remaining documents, we further ran a recognizer to discover IoT vulnerability reports and extract key information (i.e., device type, vendor name, and vulnerability type). The retrieval of the vulnerability information is modeled as a named entity recognition problem [52] in NLP. More specifically, we first attempted to identify four IoT vulnerability-related entities, including device types, vendors, product names, and vulnerability types, and then utilize the dependency relationship across them to confirm the presence of vulnerability-related descriptions. Given the uniqueness of the descriptions, we adopted a set of IoT-specific recognition techniques to retrieve them, as elaborated on below.

Table 7: Context textual terms.

Entity	Context terms
Device Type	camera, ipcam, netcam, cam, dvr, router nvr, nvs, video server, video encoder, video recorder diskstation, rackstation, printer, copier, scanner switches, modem, switch, gateway, access point
Vendor	1,552 vendor names
Product	[A-Za-z]+[-]?[A-Za-z!]*[0-9]+[-]?[-]?[A-Za-z0-9] *∧[0-9]2,4[A-Z]+
Vuln type	733 CWE, 88 abbreviations
Version	(?:version[:.]*([\w-][\w.-]+) ve?r?s?i?o?n?s?[:.]*([\d-][\w.-]+)
CVE	CVE-[0-9]{4}-[0-9]{4,}

To identify these individual entities, we utilized keyword and regular expression based matching. For device types, vendor names, and vulnerability types, we used a set of keywords, as illustrated in Table 7: whenever a single word in the category (device type, vendor name, or vulnerability type) is found, we believe that its corresponding entity exists. These keywords are from the features of real-world devices. Specifically, we collected all common device types, including routers, cameras, modems, and printers, and found vendor names from Wikipedia. We also gathered vulnerability types from the Common Weakness Enumeration (CWE), which is a community-developed list of common software security weaknesses [9]. Further, we added to the list common acronyms of these vulnerabilities, such as CSRF and RCE. For product names, we built regular expressions to identify each entity, due to the large volume and difficulty of enumeration. These expressions are listed in Table 7.

In this way, our approach can identify all IoT vulnerability-related documents. However, given the pervasiveness of such entities (e.g., the term "switch" also appears in the documents unrelated to IoT flaws), using them alone could introduce a large number of false positives. To address this issue, we leveraged the dependency among these entities to ensure the correct recognition of these vulnerabilities.

Intuitively, when these entities are indeed used to describe an IoT vulnerability, they do not independently occur. Instead, they come together to present a concept. This implies the existence of dependency among them. In particular, the term for the vendor entity precedes the product entity or the device-type entity: e.g., D-Link DIR-600 or Foscam IPcamera. Also, the document needs to contain the vulnerability type, e.g., command injection. Using these rules, we can piece together these entities, linking IoT products to vulnerabilities. However, there are still several cases satisfying the rules above but as non-IoT device entities, such as "NAI NAI-0020" and "EE

Thomson TWG850 Wireless Router Multiple Vulnerabilities
Foscam IP Cameras Multiple Cross Site Request Forgery Vulnerabilities
Belkin Router N150 - Path Traversal Vulnerability
Dlink DIR-601 Command injection in ping functionality
Squirrelmail 1.4.22 Remote Code Execution
New Linux kernel 2.6.8 packages fix several vulnerabilities
Cisco Ironport Appliances - Privilege Escalation Vulnerability

Figure 3: Examples of the IoT vulnerability entities (first four) and non-IoT vulnerability entities (last three).

13EFF4". To suppress false positives, we further investigate the results by using an entity checker. More specifically, in the entity checker, we search for the extracted entities (e.g., D-Link DIR-600) in Google, and then calculate the cosine similarity between the extracted entities and the title of the search results. If the similarity is extremely low (e.g., 0.08), we regard the extracted entity as a non-IoT device.

Figure 3 shows examples of IoT vulnerabilities (the first four) and other vulnerabilities (the last three) in vulnerability reports. For each vulnerability extracting from the preprocessing stage, we used the keyword matching to identify the device type (e.g., router), vendor (e.g., Belkin), and vulnerability type (e.g., path traversal). Then, we used the regular expression based matching to extract product information (e.g., N150). After that, we checked if the extracted entities are combined to present a IoT vulnerability concept via local dependency and entity checker. For example, three entities "Belkin", "router", and "N150" together describe an IoT device. As we can see from Figure 3, the first four always include the affected IoT products (e.g., D-Link DIR-600) and the vulnerability types (e.g., command injection), while the latter three do not. Once such a report is found, our approach further extracts the firmware version and CVE number from its content when such information exists, for the follow-up analysis. The regular expressions for identifying these entities are also listed in Table 7.

4.3 Automatic Defense Rule Generation

After the IoT vulnerability recognition, we identified the entities (i.e., device types, vendors, product names, and vulnerability types) from the IoT vulnerability reports. These entities are then used to cluster IoT vulnerability reports describing the same IoT vulnerability. Then, given each cluster, we extracted the vulnerability semantics (e.g., vulnerability location[2] and exploit parameters) and other structured information (e.g., attack scripts) from the vulnerability reports. This generation process, as shown in Figure 4, enables us to leverage the attack surface to generate a vulnerability-specific signature.

[2] A vulnerability location is where flaws exist, such as "command injection in PwdGrp.cgi", "PwdGrp.cgi" is the vulnerability location.

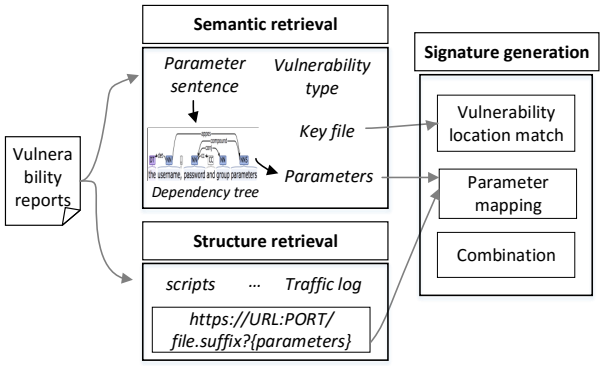


Figure 4: The architecture of signature generation.

Figure 5 presents an example, showing how we generate a signature from an extracted IoT vulnerability report. On the top-left of the figure is the vulnerability description. The bottom-left is the content of the structured information, which is a traffic log for an exploit on this vulnerability. IoTShield first locates the vulnerability semantics in the description. For instance, the sentence "command injection in PwdGrp.cgi" indicates that the vulnerability type is "command injection" and the affected location is "PwdGrp.cgi", together with the vulnerable parameters from another sentence "the username, password, and group parameters". Then, IoTShield parses the structured information (e.g., the traffic logs), and discovers the path of the vulnerable CGI "/cgi-bin/supervisor/PwdGrp.cgi" and the parameter used for command injection "pwd = ;reboot;", from the command indicator ";" and the list of legitimate commands.

Further, from the sentence about other vulnerable parameters "the username, password, and group parameters" (from the description), we can now infer that the same injection can also happen on "usr" and "grp", but not on the parameters "action" and "lifetime". In this way, we can build a vulnerability-specific signature for the injection flaw, and then transform it into the Snort format (based on vulnerability type) as presented in Figure 5. Here we elaborate on the individual components of IoTShield.

Report clustering. There are some different blogs describing the same vulnerability, and these reports may describe the vulnerability from different aspects. So, we need to cluster these reports together to form a complete vulnerability report before the rule generation stage. The challenge is that although two reports describe the same vulnerability, they may have different formats with different hash values. Our solution is to use the entities (i.e., device types, vendors, product names, and vulnerability types), which are recognized from IoT vulnerability reports, to cluster IoT vulnerability reports that describe the same IoT vulnerability.

Note that report clustering aims to supplement the missing information of a vulnerability report (e.g., missing script code or missing vulnerability description). There rarely exist

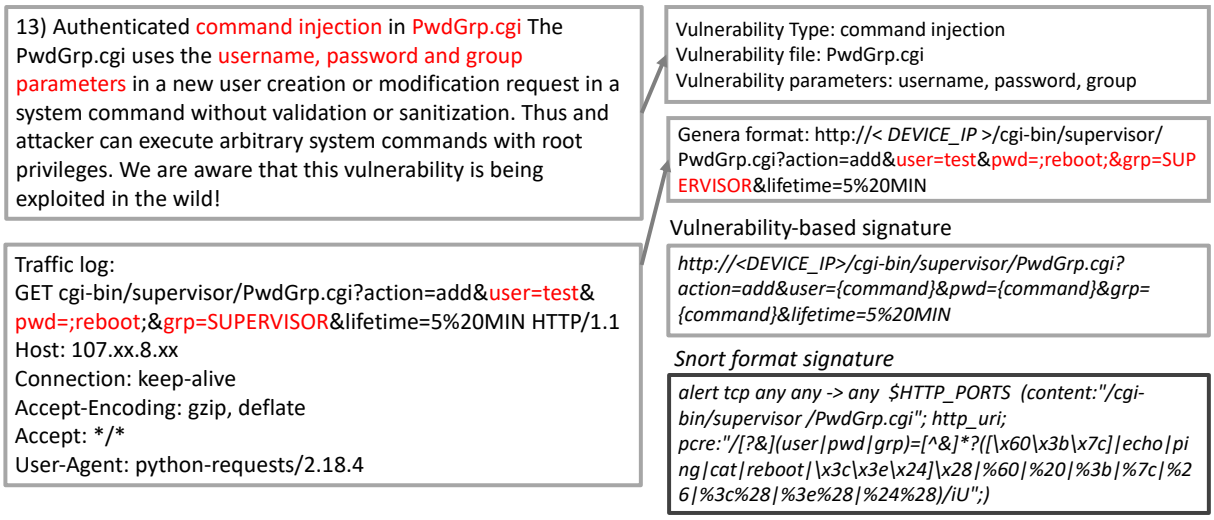


Figure 5: Example of vulnerability-specific signature generation from a vulnerability report.

different vulnerabilities with same device type, vendor, product name, and vulnerability type. However, even different vulnerabilities are clustered together, we can extract signatures for each vulnerability when matching the template (see Section 4.3)

Semantic and structured information retrieval. We first utilized NLP techniques to analyze the vulnerability descriptions to find vulnerability semantics, including vulnerability type, location, and short sentences with exploit parameters. Our approach is based on the observation that the semantic information of interest is presented through a relatively stable grammatical structure. Specifically, we utilized the vulnerability type list in Section 4.2 to determine the type of the problem documented and the regular expressions to find the vulnerability location, which should be a web content file, such as ".htm", ".cgi", ".php", ".asp", and ".html". When it comes to vulnerability parameters, our approach locates the sentences containing the keyword "parameter", "variable", "action", or "function", and leverages the grammatical relationship among these words and the values of the parameters to find them. For this purpose, we constructed a dependency tree using the Stanford Dependency Parser [51] to parse the whole sentence and then extract the nouns as the targeted terms to inspect their relationship with the keywords. These terms are considered to be parameters if they have a non-"nmod" relation [3] with the keyword, or have a "conj" relation [4] with an identified parameter. An example is shown in Figure 6.

Further, we used the regular expressions (listed in Table 13 of Appendix) to locate different kinds of the structured information, including the PoC using Linux commands (curl and wget), PoC URL, PoC HTML scripts, and PoC traffic log, etc. For each type of the structured information, we built a parser

[3] One element serves as a nominal modifier for the other.
[4] Two elements are connected by a coordinating conjunction.

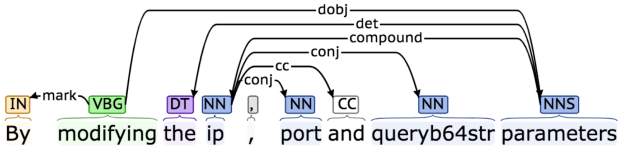


Figure 6: The dependency tree of a sentence with vulnerability parameters.

to transform it into a general template:

$$http://HOST:PORT/file.suffix?\{parameters\},$$

where HOST is the device's IP address, port is the application layer server port (default is 80), file.suffix is the vulnerability location, and {parameters} is the key-value format that includes the parameters used for the vulnerability file in the exploits. For each item in {parameters}, IoTShield checks whether it carries any illegal values like injected Linux command or Java scripts. Note that not all vulnerabilities need parameters, such as some information disclosure weaknesses, which could be exploited by simply requesting the vulnerable location. In this way, our approach acquires the semantic and structured information from the vulnerability reports.

Signature generation. After discovering vulnerability information from the reports, IoTShield utilizes it to build a signature. Specifically, we first compared the vulnerability location (recovered from the description) and "`file.suffix`" in the template (from the structured information); if matched, we believed that the semantic information (including vulnerability types and parameters) would be about the flaw modeled by the template (e.g., that attacked by the script). Then, based on the vulnerability type, we decided whether to ignore the parameter part of the template, since some vulnerabilities do not need parameters to exploit (e.g., information disclo-

sure and directory traversal). For the vulnerability types that do not need parameters to exploit (e.g., directory traversal ("/../../etc/passwd") in D-Link routers [CVE-2018-10822]), we ignore the parameter part. As of all the vulnerability types in our data, information disclosure and directory traversal are the only two typical vulnerability types not requiring a parameter to exploit. For those vulnerabilities that need parameters, IoTShield generalizes the parameter field in the template with those collected from the description. In this way, we built a vulnerability-specific signature. An example is shown in Figure 5.

A problem with this simple signature generation process is that the semantic information and the structured information may not always match in an exact fashion. For example, in Figure 5, the vulnerability parameters in the vulnerability description are presented in the natural language (i.e., username, password, and group) while they are abbreviated in the structured information (i.e., user, pwd, and grp). To address this problem, we manually collected a list mapping individual keywords to their corresponding abbreviations used in Linux, such as grp for group, for translating natural language terms into parameters in a signature.

Another issue is that, as mentioned earlier, some vulnerability reports do not contain both vulnerability descriptions and structured information; or vulnerability descriptions do not contain all vulnerability semantics. Next, we discuss how to handle them.

• *Vulnerability description only*. Without structured information, all we can get are just the vulnerability type, locations, and parameters. The parameters, however, can be less reliable due to the abbreviations they could have, which we cannot see. Therefore, we can only generate signatures for some vulnerabilities: those in which the knowledge of the vulnerability location alone (e.g., "PwdGrp.cgi") is enough for protection. Examples of such flaws include information disclosure and directory traversal, e.g., for an information disclosure problem, a request for `/QIS_wizard.htm` is sufficient for signature generation.

• *Structured information only*. Without vulnerability description, we cannot produce a generalized, vulnerability-specific signature. However, we can still utilize the structured information to build an exploit-specific signature to defeat the attack script described in a vulnerability report. As found in our measurement study (Section 3), today's IoT attacks often utilize these published scripts. These signatures can still contribute to the detection of many ongoing attacks, though they can be evaded once an adversary is willing to make more effort to better understanding the flaws he attacks. It is important to note that some level of generalization is still possible here; for example, once a parameter recovered from an attack script is found to contain a Linux command, we can add other commands commonly used in attacks to the parameter list for signature generation.

5 Implementation and Deployment

5.1 Implementation

We implemented a prototype system of IoTShield, which includes a set of building blocks. Here we briefly describe the system's nuts and bolts, and then show how they are assembled into the prototype system.

Nuts and bolts. Our prototype system was built upon three key functional components: *report crawler*, *vulnerability extractor*, and *rule generator*. Those components are extensively used across the whole system, and they were implemented as follows.

• The report crawler fetches the vulnerability reports from the Internet using wget and the scrapy crawling framework [36]. Specifically, for some websites that are well archived (e.g., `seclists.org/fulldisclosure`), we used "wget –mirror" to download their pages recursively (i.e., crawl the websites as deep as possible). For other websites, we used the scrapy crawling framework to crawl the whole websites. Since websites sometimes have crawling restrictions (e.g., `packetstorm.com` has a rate limit and `s3cur1ty.de` requires a cookie in the header of each request), our crawler simulates browser behaviors to mitigate those restrictions. It utilizes different user-agents for each request and sleeps for a random period of time after sending out multiple requests; additionally, once an access fails, a new attempt will be made later, after all pending requests in the current waiting queue have been delivered.

• The vulnerability extractor was implemented by 2,300 lines of python code. The Beautiful Soup Python library [5] was used to parse the vulnerability reports to extract main contents. The NLTK [29] package was used to split sentences, stem words, remove stop words, etc. The Aho–Corasick algorithm [2] is a string-searching algorithm to speed up the entity identification stage. A scikit-learn [35] library was used to calculate the TF-IDF (Term Frequency-Inverse Document Frequency) cosine similarity in the entity checker.

• The rule generator was implemented by 1,500 lines of Python code. We used a Simhash Algorithm [31] to detect near-duplicates, and the Stanford dependency parser [51] to establish the dependency tree.

5.2 Deployment

IoTShield can be deployed in two modes: coarse-grained and fine-grained. In the coarse-grained mode, all the generated rules are used in the IDS system, regardless of device types. All the rules are used to inspect the network traffic. This mode is easy to deploy but may have some false positives, since some rules can be device-specific. Also, in the coarse-grained mode, we suggest not to use the rules generated from descriptions only. This is because when ignoring device type, the rules generated from descriptions only may lead some false

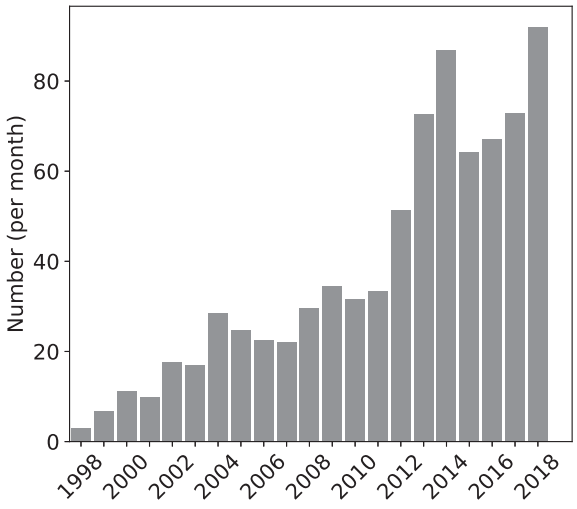


Figure 7: Vulnerability disclosure trend.

positives. For example, for a report describing an information disclosure vulnerability at the file of "/new/index.htm" in Merit Lilin IP Cameras, with description only, we generate a signature to block the traffic that attempts to extract information from the file of "/new/index.htm". However, such a scenario (retrieving the file of "/new/index.htm") may also occur in normal web servers and cause false positives. In the fine-grained mode, we take the network environments into account and deploy rules for given device types. For example, if there are just D-link devices in the local network, we only deploy the rules to protect D-link vulnerabilities. More specifically, in the fine-grained mode, IoTShield first analyzes the network traffic or actively probes the network to identify device types (e.g., models and brands) and their IP addresses. This step can be achieved by using a device list in the monitored network or using the device fingerprinting method proposed in [47]. After that, a signature selection process will be conducted to select the corresponding signatures, given the current device types in the network.

6 Evaluation

6.1 Effectiveness

To validate the efficacy of IoTShield, we first manually checked the extracted IoT vulnerabilities and obtained some basic statistics of them. Then, we used two different traffic traces to evaluate the effectiveness of generated signatures.

Vulnerability extractor. We randomly sampled 200 reports from those identified for manual validation and achieved a precision of 94%. In total, we collected 7,514 IoT vulnerability reports from 0.43 million articles (Table 6). These reports disclose 12,286 IoT vulnerabilities, with roughly 1.6 each on average. Figure 7 shows the average number of IoT vulnerabilities disclosed per month from 1998 to 2018. We can see that the number of disclosures has increased since 1998, and this increasing trend has further sped up since 2012 and slowed down since 2014, but it peaked in 2018 when about 90 IoT vulnerabilities per month were disclosed.

Table 8: List of top 10 vendors and device types of affected devices.

Device Vendor	Num	Device Type	Num
Cisco	1,264	router	3,700
D-Link	988	switch	1,422
Linksys	539	camera	1,248
Netgear	522	firewall	1,101
HP	485	gateway	1,032
Symantec	299	modem	843
TP-Link	255	access point	478
Zyxel	229	printer	408
Huawei	195	nas	338
Asus	180	scanner	176

Table 9: List of top 10 vulnerability types.

	Vulnerability type	Num
1	Denial of service	975
2	CSRF	902
3	Buffer overflow	869
4	Command injection	806
5	XSS	775
6	Authentication bypass	763
7	Command execution	458
8	Information disclosure	407
9	Directory traversal	307
10	Privilege escalation	276

These IoT vulnerabilities are related to device types and vendors. We found that the distribution of vulnerabilities among IoT vendors follows a long-tail: nearly 60% of vulnerable devices are from the top 10 vendors with the most security flaws. Table 8 lists the vulnerability distribution over these 10 device types and vendors. As we can see here, routers, switches, and cameras, which are perceived to be the most common IoT devices, also have the most vulnerabilities. In addition, the vendors responsible for the most vulnerabilities (i.e., Cisco, D-Link, and linksys) are all reputable and have the largest market shares.

Table 9 further lists the top 10 vulnerability types in our dataset. The majority of them are remotely exploitable (e.g., buffer overflow, denial-of-service, CSRF command injection, and authentication bypass), which could be easily used to compromise IoT devices. Moreover, cross-site scripting (XSS), command injection and command execution are commonly used by IoT malware to execute commands on a compromised device as a botnet node.

Table 10: Effectiveness of IoTShield.

Dataset	Precision	Recall	False Positive Rate
Real devices	97%	83%	0.01%
Honeypot	98%	93%	0.06%

Rule generation effectiveness. We first evaluated our IoTShield prototype on 190K HTTP requests collected from IoT devices and honeypots, using a Macbook Pro with 2.6GHz Intel Core i7 and 16GB of memory. Again, all signatures generated were in the Snort format.

Our HTTP requests include those gathered from real-device honeypots and those from the simulators. In our experiments, we labeled IoT device traffic as described in Section 3.1. Those traces include 178,778 HTTP requests received by the simulators, which are related to 141 attack activities generated by 26 unique attack scripts, and the rest is benign traffic. The remaining data come from the real-device honeypots, as described in Section 3.1, including 11,602 HTTP requests in 1,860 attacks generated by 81 unique attack scripts.

We evaluated the effectiveness of IoTShield using precision, recall, and the false positive rate (FPR). Precision is defined as $|TP|/|FP+TP|$, recall is $|TP|/|TP+FN|$, and FPR is $|FP|/|FP+TN|$, where TP is the number of true positives, FN is the number of false negatives, FP is the number of false positives, and TN is the number of true negatives. Table 10 presents the experimental results. Over the traces received by the simulators, the precision of our automatically generated signatures is 98%, the recall is 93%, and the FPR is 0.06%. Over the requests gathered from the real devices, our signatures can achieve a 97% precision, 83% recall, and 0.01% FPR.

We further used a long-time traffic captured in an industrial control system's HMI honeypot for the evaluation of IoTSheild. The simulated industrial control system's HMI honeypot is used to monitor the attack traffic with a blind scanning and attack. The duration was from October 2017 to November 2018 across from seven different cities. By replaying the traffic, IoTShield reported 7,396 alerts of exploiting the HMI system. By manually checking the 7,396 alerts, we confirmed that about 6,705 alerts were indeed IoT attacks. The rest of the alerts were confirmed to have attacked other vulnerabilities on common web servers. For instance, "`/level/77/exec/show/config/cr`", is found in exploit-db as a script to evade detection of HTTP attacks via non-standard "%u" Unicode encoding of ASCII characters in the requested URL. [7].

6.2 Performance

Signature generation. To understand the performance of IoTShield, we conducted experiments to measure the time cost of processing vulnerability reports at each individual stage: data collection, IoT vulnerability extraction, and automated rule generation. The IoTShield prototype runs on a commercial desktop computer (Ubuntu 18.04, 8GB of memory, 64-bit OS, with 4-core Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz), indicating that the CPU and memory requirements of IoTShield can be easily met. The IoTShield process runs in a single thread. Table 11 lists the average time cost of each stage of IoTShield for one rule generation. The acquisition of vulnerability reports from the Internet takes 0.386 seconds (51%). Note that this stage requires message transmissions, and the time cost is dependent upon the network conditions. The IoT vulnerability extraction takes 0.154 seconds (21%), while the rule generation costs more time, 0.21 seconds (28%), due to the fact that it needs to establish the dependency tree of the sentence with vulnerability parameters. Overall, the time cost of IoTShield for automatic rule generation is low in practice, and we could further reduce the time cost by running it in multiple threads. The results indicate that IoTShield is efficient and can be easily scaled to a desirable level to handle the massive amounts of vulnerabilities online with a timely update of the defense rules.

Table 11: Running time at different stages.

Stage	Running time (s)	Percentage
Data collection	0.386	51%
IoT vulnerability extraction	0.154	21%
Rule generation	0.210	28%
Overall	0.750	100%

Rule inspection. To further evaluate the performance of IoTShield in practice, we ran it as one component of an IDS for processing the real-world traffic captured on the edge router of a research institution, which consists of more than 100,000 Internet devices. The amount of traffic is about 53G, and the duration of this traffic collection is about two hours. We replayed the traffic to Snort with and without IoTShield. For the Snort without IoTShield, it costs 426.28 seconds to inspect all the collected packets; for the Snort with IoTShield, it only adds 0.13 seconds for rule inspection over the entire 53G of data, showing that IoTShield induces little overhead to IDSes for online data processing.

7 Discussion

Limitation. In the data collection, we crawled 13 different websites retrieving 0.4 million reports, and we plan to keep a periodic update in the future. However, we acknowledge that our methods cannot exhaustively collect all IoT vulnerabilities in the wild. Although we believe that we have collected the majority of IoT vulnerabilities in public, other methods

for data collection could also be considered to collect those vulnerabilities recorded in less popular websites, such as personal blogs or social networking logs. For example, since IoT vulnerabilities are usually targeted for some specific devices, we will search product names combined with vulnerability types as query keywords in search engines to crawl more reports.

In addition, although we did observe that the majority (80%) of the malicious HTTP requests are from blind scannings, not targeted at specific IoT devices, we acknowledge that the traffic logs collected by the honeypots could be biased, due to the IoT device types in the honeypots and deployment time periods of the honeypots. Ideally, IoTShield could be deployed in ISPs for a large-scale and real-scenario evaluation. However, we were unable to access ISP data. Alternatively, we performed a real-scenario evaluation based on the traffic we captured from edge routers of a research institution, whose data size is about 114G and the duration is about 12 hours. In this experiment, IoTShield only produced two false positive alerts; considering the substantial data size, the FPR is close to zero.

Since IoTShield deals with generally uncoordinated sources of vulnerability information, it may sometimes face incomplete source information (missing both vulnerability descriptions and structured information, or vulnerability descriptions not containing all vulnerability semantics), even we have used report clustering to supplement the missing information. As mentioned in Section 4.3, we indeed take the problem of incomplete information into serious considerations. For structured information only (about 9% observed in our dataset), we generate an exploit-specific signature that can contribute to the detection of many ongoing attacks. However, we acknowledge that we may miss some exploit variants, which leads to the decrease of recall. For vulnerability description only (about 20% observed in our dataset), we use vulnerability location alone (e.g., "PwdGrp.cgi") as signature for some specific vulnerabilities, which may lead to the increase of false positive rate. In this way, IoTShield can only generate signatures for limited vulnerability types (e.g., information disclosure and directory traversal). A natural follow-up step is to investigate the missing information and explore a systematic information supplement method. We will leave this as our future work.

Also, IoTShield cannot handle the exploits in some specific program languages, and its processing capability is limited to traffic logs, scripts, Linux commands, etc. This is because IoTShield cannot easily generate the general exploit template to identify vulnerability locations and parameters in a different program language. In our future work, we plan to develop a simulation system to execute these programs and generate the attack traffic. Thus, we will be able to produce the attack script in the traffic format (e.g., HTTP request) for whichever program language is used. Moreover, although we did observe a few IoT vulnerabilities in other application layer protocols, our defense only targets HTTP vulnerabilities, which cover most (90%) of the IoT vulnerabilities we observed in the honeypots and vulnerability reports. In our future work, we plan to cover more vulnerabilities, which exploit other application layer protocols, by extracting vulnerability semantic and structure information based on individual application layer protocol's domain knowledge.

Mitigation. Based on the results of our measurement study, we have identified several potentially effective mitigation strategies to restrain the fast-growing IoT-based attacks. In our study, we observed a large number of vulnerability reports in the wild, which are missed by vendors but exploited by attackers. We also observed the heterogeneity of IoT devices. IoT devices usually do not have an automatic update mechanism and are maintained by device users who may lack security awareness. Thus, the mitigation of an IoT-based attack requires a collaboration among users, vendors, and security researchers.

First, vendors should provide an official vulnerability report platform, and reply to vulnerability reports in a timely manner. In this work, we observed a relatively short duration of public disclosure: report authors usually disclosed the bugs after contacting vendors three times if no reply is received. In addition, vendors are expected to provide technical support for their discontinued products, especially those devices that are still widely used. Since most users lack security awareness, a vendor should increase efforts to avoid misconfiguration and notify these users about updating their devices in a more effective fashion (e.g., using an automatic update mechanism). Second, the authors of vulnerability reports should follow the guidelines for vulnerability reporting, such as a coordinated disclosure. We observed that at least 2,000 vulnerabilities were released before the vendors provided patches. Even worse, some report authors released a disclosure without attempting to contact the vendors or CVE. Finally, device users should pay more attention to the device configuration (e.g., default password) and quickly update vulnerable firmware when a new version becomes available.

Ethical issues. One ethical concern is the way by which the security reports were gathered, i.e., scraping various websites. We deployed certain mechanisms to bypass rate limiting and authentication. However, our process follows the robot exclusion protocol (robots.txt) of websites and causes no harm to them or their users. Another ethical concern is that we purchased attack tools on the black market. In our research, we consulted with our Institutional Review Board (IRB) (though no IRB review is required) and legal consul to ensure that the purchase has been done within the legal and ethical boundaries. The purchases did not violate any law and regulation. Also during the process, we refrained from gathering any information not supposed to collect, such as identity-related data.

8 Related Work

Vulnerability-specific signature generation. Cui et al. [46] presented a system for automatically generating a vulnerability-specific signature (or data patch) for an unknown vulnerability, given a zero-day attack instance. Their system injects the softwares/real devices to generate the variants of attack instances. However, it cannot be easily applied for the IoT vulnerability-specific signature generation, due to the large amount of different IoT device vulnerabilities being covered. Wang et al. [60] proposed a vulnerability-specific network filter, Shield, at an end-system to prevent known vulnerability exploits. Shield requires a manually-generated policy to describe the vulnerability. Specially, it requires a fairly deep understanding of the protocol over which the vulnerability is exploited. It is not acceptable for the IoT vulnerability signature generation, due to the significant amount of manual efforts to generate policies for each IoT device. Brumley et al. [44] proposed data-flow analysis techniques for automatically generating vulnerability-specific signatures. However, it cannot be deployed in IoT vulnerability signature generation, due to the lack of source code. By contrast, IoTShield analyzes the content of more than 7,500 IoT vulnerability reports and recovers key knowledge to generate vulnerability-specific signatures.

NLP for vulnerability assessment. Pandita et al. [54] used NLP techniques to analyze Android APP descriptions and API documents for determining unnecessary permissions. Sabottke et al. [55] explored the vulnerability-related information disseminated on Twitter and provided an early warning for the existence of real-world exploits by tweets. Liao et al. [50] presented a novel technique for automatic Indicator-of-Compromise extraction from unstructured text. You et al. [61] proposed leveraging vulnerability-related text (CVE reports and Linux git logs) to guide Linux kernel vulnerability fuzzing. Zhu et al. [62] mined Android documents and security literature to generate features for detecting Android malware. Caselli et al. [45] proposed to automatically mine parameter configuration rules of network control systems (e.g., BACnet-based building automation systems) from system specifications. In contrast to these previous works, we utilize a set of IoT vulnerability reports' syntax features to discover vulnerability-specific knowledge for IDS signature generation to protect IoT from being attacked.

Vulnerability-related measurement. Shahzad et al. [58] conducted a large-scale study on the software vulnerability life-cycle based on public vulnerability databases. They utilized association rule mining to extract the relationship between the representative exploitation behavior of hackers and the patching behavior of vendors. Nappa et al. [53] presented a systematic study of patch deployment in client-side vulnerabilities, in order to analyze how users deploy patches. They found that the patch mechanism has an important impact upon the patch deployment rate. Li et al. [48] performed an extensive study on the effectiveness of vulnerability notifications, with the aim of illuminating which fundamental aspects of notifications have the greatest impact. Li et al. [49] conducted a large-scale empirical study of security patches based on the open-source software projects. They sought to identify the differences between security and non-security bug fixes. Sarabi et al. [56] studied the vulnerability patching by analyzing vulnerabilities across four software products. Their focus is mainly on how individual behaviors influence the security state of an end-host. In contrast to previous works focusing on vulnerabilities in CVE or NVD, our work extensively studies the IoT-related vulnerabilities that are from a large number of vulnerability reports scattered around forums, mailing lists, and blogs, and we further explore the effectiveness of using such reports for IoT vulnerability defense.

9 Conclusion

To understand how cybercriminals launch IoT-related attacks, we leveraged honeypots to collect the traces of real-world IoT exploits and analyzed four popular attack toolkits. Our research sheds light on a largely overlooked cause of the pervasiveness of IoT attacks in recent years: IoT vulnerabilities are publicly available and easy to exploit, and today's IoT attacks almost exclusively use known vulnerabilities for mounting malicious attacks. More importantly, our findings lead to the design of IoTShield, a simple yet effective IoT vulnerability-specific signature generation system for intrusion detection. IoTShield first collects 430,000 vulnerability reports from the past 20 years and identifies content of 7,500 IoT vulnerability reports. IoTShield then retrieves key knowledge to generate vulnerability-specific signatures. These signatures can be easily deployed at existing intrusion detection systems or web application firewalls to detect exploit attempts on a target IoT device. Therefore, IoTShield significantly raises the bar for future IoT attacks to succeed.

Acknowledgments

We are grateful to our shepherd Adwait Nadkarni and anonymous reviewers for their insightful feedback. We also want to thank Haoran Lu and Jianzhou You for help collecting underground attack tools and honeypot data. The IIE authors are supported in part by National Key R&D Program of China (No. 2018YFB0803402), Key Program of National Natural Science Foundation of China (No. U1766215), and International Cooperation Program of Institute of Information Engineering, CAS (No. Y7Z0451104). The IU authors are supported in part by NSF CNS-1850725, 1527141, 1618493, 1838083 and 1801432 and ARO W911NF-16-1-0127. The support provided by China Scholarship Council (CSC) during a visit of Xuan Feng to IU is acknowledged.

References

[1] Abiword - Enchant. http://www.abisource.com/projects/enchant/.

[2] Aho-corasick algorithm. https://github.com/WojciechMula/pyahocorasick/.

[3] Amnesia. https://researchcenter.paloaltonetworks.com/2017/04/unit42-new-iotlinux-malware-targets-dvrs-forms-botnet/.

[4] AVTECH IP Camera, NVR, DVR multiple vulnerabilities. http://seclists.org/fulldisclosure/2016/Oct/36.

[5] Beautifulsoup. https://www.crummy.com/software/BeautifulSoup/.

[6] BrickerBot. https://www.trustwave.com/Resources/SpiderLabs-Blog/BrickerBot-mod_plaintext-Analysis/.

[7] Cisco Secure IDS 2.0/3.0 / Snort 1.x / ISS RealSecure 5/6 / NFR 5.0 - Encoded IIS Detection Evasion. https://www.exploit-db.com/exploits/21100.

[8] CNVD-2013-20783. http://www.cnvd.org.cn/webinfo/show/3205.

[9] Common Weakness Enumeration. https://cwe.mitre.org/.

[10] CVE-2009-1557. https://nvd.nist.gov/vuln/detail/CVE-2009-1557.

[11] CVE-2009-1558. https://nvd.nist.gov/vuln/detail/CVE-2009-1558.

[12] CVE-2009-1559. https://nvd.nist.gov/vuln/detail/CVE-2009-1559.

[13] CVE-2009-1560. https://nvd.nist.gov/vuln/detail/CVE-2009-1560.

[14] CVE-2013-2578. https://nvd.nist.gov/vuln/detail/CVE-2013-2578.

[15] CVE-2013-2579. https://nvd.nist.gov/vuln/detail/CVE-2013-2579.

[16] CVE-2013-2580. https://nvd.nist.gov/vuln/detail/CVE-2013-2580.

[17] CVE-2013-2581. https://nvd.nist.gov/vuln/detail/CVE-2013-2581.

[18] CVE-2013-3612. https://nvd.nist.gov/vuln/detail/CVE-2013-3612.

[19] CVE-2013-3688. https://nvd.nist.gov/vuln/detail/CVE-2013-3688.

[20] CVE-2017-16957. https://nvd.nist.gov/vuln/detail/CVE-2017-16957.

[21] D-Link DIR-645 Routers Remote Authentication Bypass Vulnerability. https://www.securityfocus.com/bid/58231.

[22] Guidelines for Security Vulnerability Reporting and Response . https://www.symantec.com/security/OIS_Guidelines%20for%20responsible%20disclosure.pdf.

[23] Hacking the D-Link DIR-890L. http://www.devttys0.com/2015/04/hacking-the-d-link-dir-890l/.

[24] Hajime. http://blog.netlab.360.com/hajime-status-report-en/.

[25] IoT reaper. https://krebsonsecurity.com/2017/10/reaper-calm-before-the-iot-security-storm/.

[26] Masuta. https://blog.newskysecurity.com/masuta-satori-creators-second-botnet-weaponizes-a-new-router-exploit-2ddc51cc52a7.

[27] Multiple vulnerabilities found in Wireless IP Camera (P2P) WIFICAM cameras and vulnerabilities in custom http serve. https://pierrekim.github.io/blog/2017-03-08-camera-goahead-0day.html.

[28] Multiple Vulnerabilities in TP-Link TL-SC3171 IP Cameras. https://www.coresecurity.com/advisories/multiple-vulnerabilities-tp-link-tl-sc3171-ip-cameras.

[29] Natural language toolkit. http://www.nltk.org/.

[30] Pwning CCTV cameras . https://www.pentestpartners.com/security-blog/pwning-cctv-cameras/.

[31] A python implementation of simhash algorithm. https://leons.im/posts/a-python-implementation-of-simhash-algorithm/.

[32] Remote Code Execution in CCTV-DVR affecting over 70 different vendors. http://www.kerneronsec.com/2016/02/remote-code-execution-in-cctv-dvrs-of.html.

[33] Reverse SSH tunneling. https://askubuntu.com/questions/598626/direct-ssh-tunnel-through-a-reverse-ssh-tunnel.

[34] Satori. https://www.trendmicro.com/vinfo/us/security/news/internet-of-things/source-code-of-iot-botnet-satori-publicly-released-on-pastebin.

[35] Scikit-learn machine learning in python. http://scikit-learn.org/stable/index.html.

[36] Scrapy: A fast and powerful scraping and web crawling framework. https://scrapy.org.

[37] Snort - Network Intrusion Detection & Prevention System. https://www.snort.org/.

[38] SSH Port forwarding. https://help.ubuntu.com/community/SSH/OpenSSH/PortForwarding.

[39] The Internet of Things Will Be Even More Vulnerable to Cyber Attacks. https://www.chathamhouse.org/expert/comment/internet-things-will-be-even-more-vulnerable-cyber-attacks.

[40] Ross Anderson. Security in open versus closed systems—the dance of boltzmann, coase and moore. Technical report, Cambridge University, England, 2002.

[41] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the mirai botnet. In *Proceedings of the USENIX Security Symposium*, pages 1093–1110, 2017.

[42] William A Arbaugh, William L Fithen, and John McHugh. Windows of vulnerability: A case study analysis. *Computer*, 33(12):52–59, 2000.

[43] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, 2014.

[44] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 2–16, 2006.

[45] Marco Caselli, Emmanuele Zambon, Johanna Amann, Robin Sommer, and Frank Kargl. Specification mining for intrusion detection in networked control systems. In *Proceedings of the USENIX Security Symposium*, pages 791–806, 2016.

[46] Weidong Cui, Marcus Peinado, Helen J Wang, and Michael E Locasto. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 252–266, 2007.

[47] Xuan Feng, Qiang Li, Haining Wang, and Limin Sun. Acquisitional rule-based engine for discovering internet-of-thing devices. In *Proceedings of the USENIX Security Symposium*, pages 327–341, 2018.

[48] Frank Li, Zakir Durumeric, Jakub Czyz, Mohammad Karami, Michael Bailey, Damon McCoy, Stefan Savage, and Vern Paxson. You've got vulnerability: Exploring effective vulnerability notifications. In *Proceedings of the USENIX Security Symposium*, pages 1033–1050, 2016.

[49] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 2201–2215, 2017.

[50] Xiaojing Liao, Kan Yuan, XiaoFeng Wang, Zhou Li, Luyi Xing, and Raheem Beyah. Acing the ioc game: Toward automatic discovery and analysis of open-source cyber threat intelligence. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 755–766, 2016.

[51] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014.

[52] David Nadeau and Satoshi Sekine. A survey of named entity recognition and classification. *Lingvisticae Investigationes*, 30(1):3–26, 2007.

[53] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras. The attack of the clones: A study of the impact of shared code on vulnerability patching. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 692–708, 2015.

[54] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. Whyper: Towards automating risk assessment of mobile applications. In *Proceedings of the USENIX Security Symposium*, pages 527–542, 2013.

[55] Carl Sabottke, Octavian Suciu, and Tudor Dumitras. Vulnerability disclosure in the age of social media: Exploiting twitter for predicting real-world exploits. In *Proceedings of the USENIX Security Symposium*, pages 1041–1056, 2015.

[56] Armin Sarabi, Ziyun Zhu, Chaowei Xiao, Mingyan Liu, and Tudor Dumitraş. Patch me if you can: A study on the effects of individual user behavior on the end-host

vulnerability state. In *Proceedings of the International Conference on Passive and Active Network Measurement*, pages 113–125. Springer, 2017.

[57] E Eugene Schultz Jr, David S Brown, and Thomas A Longstaff. Responding to computer security incidents: Guidelines for incident handling. Technical report, Lawrence Livermore National Lab., CA (USA), 1990.

[58] Muhammad Shahzad, Muhammad Zubair Shafiq, and Alex X Liu. A large scale exploratory analysis of software vulnerability life cycles. In *Proceedings of the IEEE International Conference on Software Engineering*, pages 771–781, 2012.

[59] Shodan. The search engine for Internet-connected devices. https://www.shodan.io/.

[60] Helen J Wang, Chuanxiong Guo, Daniel R Simon, and Alf Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of the ACM SIGCOMM Computer Communication Review*, pages 193–204, 2004.

[61] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 2139–2154, 2017.

[62] Ziyun Zhu and Tudor Dumitras. Featuresmith: Automatically engineering features for malware detection by mining the security literature. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 767–778, 2016.

A Keywords and Regular Expressions

Table 12: Keywords used in finding attack tools.

Keywords
Amnesia, next-gen Mirai botnets , Mirai Okiru, IoTroop
Satori IoT botnets, Brickerbot IoT botnets, Masuta,
Tsunami, Reaper, Hajime, IoT Mine Bitcoin, IoT hacking tool,
Persirai Botnet, Bashlite, Gafgyt, Qbot, Remaiten, Torlus,
DoubleDoor IoT botnet, JenX, IP Cameras GoAhead

Table 13: Regular expressions of structured information.

	Regular expressions
Linux command	curl [\w\W]+?\n\n
	wget [\w\W]+?\n\n
	wget [\w\W]+?\n
	curl [\w\W]+?\n)
Traffic log	POST [\w\W]+?\n\n
	GET[\n][\w\W]+?\n\n
	PUT [\w\W]+?\n\n
	POST [\w\W]+?\n\n [\w\W]+?\n
Exploit URL	((https?)://?[-A-Za-z0-9+&@#/%?\
	*=≃&$\\‘’\" \[\]() <>\]!:,.;]+[-A-Za-z0-9 +
	\[\]&’\" ()\]*<># \\/%= _])
Timeline	timeline[\w\W]+? \n\n[\-_]?
	time line[\w\W]+? \n\n[\-_]?

ATTACK2VEC: Leveraging Temporal Word Embeddings to Understand the Evolution of Cyberattacks

Yun Shen
Symantec Research Labs
yun_shen@symantec.com

Gianluca Stringhini
Boston University
gian@bu.edu

Abstract

Despite the fact that cyberattacks are constantly growing in complexity, the research community still lacks effective tools to easily monitor and understand them. In particular, there is a need for techniques that are able to not only track how prominently certain malicious actions, such as the exploitation of specific vulnerabilities, are exploited in the wild, but also (and more importantly) how these malicious actions factor in as attack steps in more complex cyberattacks. In this paper we present ATTACK2VEC, a system that uses temporal word embeddings to model how attack steps are exploited in the wild, and track how they evolve. We test ATTACK2VEC on a dataset of billions of security events collected from the customers of a commercial Intrusion Prevention System over a period of two years, and show that our approach is effective in monitoring the emergence of new attack strategies in the wild and in flagging which attack steps are often used together by attackers (e.g., vulnerabilities that are frequently exploited together). ATTACK2VEC provides a useful tool for researchers and practitioners to better understand cyberattacks and their evolution, and use this knowledge to improve situational awareness and develop proactive defenses.

1 Introduction

Modern cyberattacks have reached high levels of complexity. An attacker who is trying to compromise a computer system has to perform a number of *attack steps* to achieve her goal [16], including reconnaissance (i.e., identifying weaknesses on the victim machine), the actual exploitation, and installing mechanisms to ensure persistence (e.g., installing a remote access trojan (RAT) on the machine [10]). Moreover, getting access to the victim machine might not be enough for attackers to achieve what they want, therefore they might have to perform additional attack steps (e.g., exploiting another vulnerability to escalate privileges [36]). Additionally, for each of the attack steps that compose the attack, attackers have a choice of executing a variety of malicious actions (e.g., exploiting different known vulnerabilities on the victim system), depending on the exploits that they have available, on the software configuration of the victim machine, and on its security hygiene (i.e., which known vulnerabilities on it have not been patched).

Previous research studied how attack steps (e.g., specific Common Vulnerabilities and Exposures (CVEs) being exploited) evolve and are used in isolation [5, 33, 39]. While doing so is useful to understand how prominently certain attack steps are exploited in the wild, it does not tell us anything on *how* these attack steps are used as part of complex cyberattacks. Instead, looking at attack steps in relation to each other can provide researchers and practitioners with invaluable insights into the *modus operandi* of attackers, highlighting important trends in the way attacks are conducted. In this paper, we define the sequence of attack steps that are commonly performed together with an attack step of interest as its *context*.

Understanding the context in which a vulnerability is exploited in the wild as well as detecting when this context suddenly changes can be very useful for researchers, to better understand the modus operandi of attackers, to improve situational awareness in organizations, and to develop more proactive defenses. For example, when a new CVE is published, attackers will start attempting to exploit it, and in this process they will first try a number of strategies. Eventually, once an attacker will succeed in developing an attack that reliably compromises machines, we will observe this strategy being consistently exploited in the wild, potentially because this consolidated attack was commoditized and added to an exploit kit for multiple attackers to use [12]. This information is useful for defenders, since it allows to design better mitigation strategies that take into account the entire attack, and it can possibly also be used for attack attribution, since the same attacker often uses similar strategies to carry out their attacks [38].

However, attack strategies are not stable over time, because new defenses might be deployed that make them ineffective (e.g., vulnerabilities getting patched), or simply because the

attackers might develop more efficient strategies. Looking at the context of an attack step (e.g., a particular CVE) can help identifying these sudden changes in the way attacks are performed, and prompt proactive defenses. For example, a number of systems have been proposed that use supervised learning to detect attacks [4, 13, 14, 41]. These systems typically need periodic retraining due to the fact that the evolution of attacks over time makes the model that the system was trained on obsolete [18]. Having a system able to track significant changes in the context associated to a security event could be used to perform a timely retraining of such systems.

To model the context of an attack step, in this paper we adapt techniques that have been proposed in the area of natural language processing. Word embeddings [30, 35] are a powerful tool for modeling relationships between words. This technique represents words with low-dimensional vectors based on the surrounding words that appear in the same sentence (i.e., the context). These vectors are able to capture the context of a word and its relationship with the other words, allowing researchers to understand the way in which words are used in various types of language (e.g., on social media [8]). In a similar way, we can calculate the embedding of an attack step by considering the entire attack sequence as a sentence, and each step as a word. Upon encoding the relationship between attack steps within the vector space, we can quantitatively study the attack steps appearing in similar contexts in the latent space and understand them in a more meaningful and measurable way.

As a proxy for the attack steps performed by attackers in the wild, we use the security alerts generated by a commercial Intrusion Prevention System (IPS), collected over a period of two years. Throughout this observation period, we collect 102 snapshots on a weekly basis. Each snapshot contains over 190 million alerts collected from tens of millions unique machines. Each alert is indicative of the attack step that is performed by an attacker, and our dataset contains over 8k possible alert types, spanning from port scans to exploits for specific CVEs. Similar data was used in our previous work, which showed that, although a proxy (e.g., they can only monitor attacks for which a detection signature exists), these alerts are useful to study the behavior of attackers in the wild [40]. In the remainder of the paper, we define each alert generated by the IPS a *security event*.

We implement our approach in a system, ATTACK2VEC. Our system takes a stream of security events and computes their context by using temporal word embeddings. By running ATTACK2VEC on our data, we show that our approach is able to effectively monitor how security events are exploited in the wild. For example, we can identify when a certain CVE starts getting exploited, when its exploitation becomes stable, and when attackers change strategy in exploiting it. By leveraging the similarity between the context of different security events, we can infer which events are often used as part of the same malicious campaign, and this allows us to identify emerging attacks in a more timely manner than the state of the art. For example, we were able to identify a variant of the Mirai botnet that was scanning the Internet attempting to exploit a CVE relative to Apache Struts, together with IoT-related exploits over 72 weeks before this variant was officially identified. These findings show that ATTACK2VEC can be an effective tool for researchers and practitioners who need to understand how security events are exploited in the wild and react to sudden changes.

In summary, this paper makes the following contributions:

- We show that temporal word embeddings are an effective way to study how attack steps are exploited in the wild and how they evolve.
- We show how ATTACK2VEC can be used to understand the emergence, the evolution, and the characteristics of attack steps in relation to the wider context in which they are exploited.
- We discuss how ATTACK2VEC can be effectively used to identify emerging attack campaigns several weeks before they are publicly disclosed.

2 Motivation

This paper presents the first approach to characterize not only single security events, but the context in which they are used in the wild. The problem of characterizing the evolution of security events, however, is a complex one and presents multiple challenges. To illustrate its complexity, consider the real-world example in Figure 1, showing several machines undergoing two coordinated attacks across time, C_1 and C_2. Both attacks leverage the attack step e_{11}, "CVE-2018-7602 Drupal core RCE." C_1: $\{e_4, e_{10}, e_{11}, e_{12}\}$ mainly functions as a reconnaissance attack including "Joomla JCE security bypass and XSS vulnerabilities" (e_4), "Wordpress RevSlider/ShowBiz security byPass" (e_{10}) and "Symposium plugin shell upload" (e_{12}), together with e_{11}. C_2: $\{e_7, e_5, e_{11}, e_6\}$, is an attack targeted at the Drupal ecosystem, consisting of "phpMyAdmin RFI CVE-2018-12613" (e_7), "Drupal SQL Injection CVE-2014-3704" (e_5), and "Apache Flex BlazeDS RCE CVE-2017-3066" (e_6), and the aforementioned e_{11}. Our goal is to develop a system that allows to automatically analyze the context in which e_{11} is exploited, and identify changing trends.

The first challenge that we can immediately notice from Figure 1 is that even though the machines at a certain timestamp are going through the same type of attack (e.g., C_1 at t_i), there are no obvious event relationships reflected in the telemetry recorded by the IPS due to noise (e.g., other security events not related to the coordinated attack observed, or certain events relating to the coordinated attack being not observed). If we take the IPS data recorded at timestamp t_i, it is not trivial to understand how e_{11} is leveraged by the attackers by directly inspecting the security events, what attack vectors

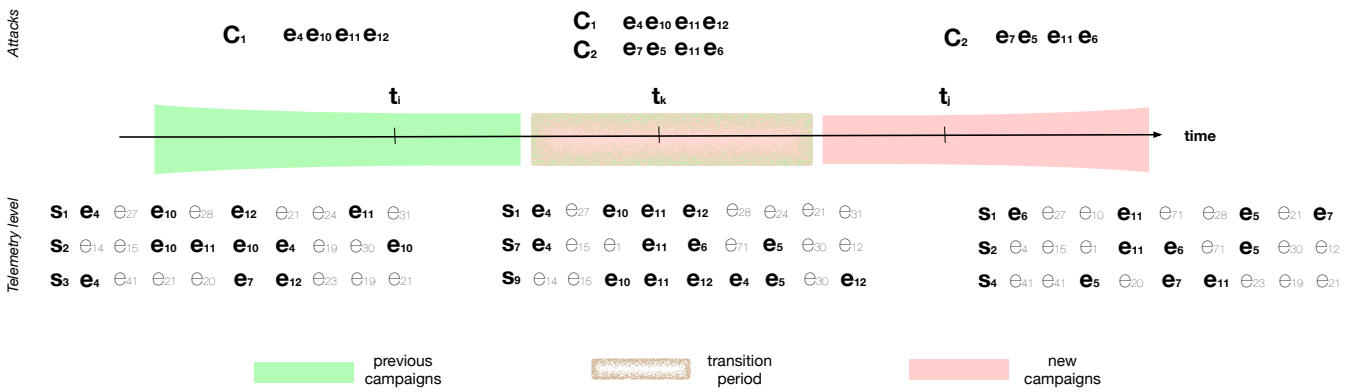


Figure 1: A real-world example of security event evolution. Over time, e_{11} evolves from being an "add-on" reconnaissance vector to part of a targeted attack on the Drupal ecosystem.

are used together with e_{11}, etc. Additionally, it is worth noting that not all security events may be observed in a given observation period. For example, e_7 is not observed until timestamp t_j.

The second challenge is that attacks change over time, consequently, the context of a security event and its relationship with other attack steps may drift. It is possible that C_1 and C_2 can be operated by the same attackers, and that at some point they changed their attack scripts to leverage newly disclosed vulnerabilities (i.e., , phpMyAdmin RFI CVE-2018-12613 (e_7)). As we can see in Figure 1, from timestamp t_i to t_j attack C_1 gradually migrated to or was replaced by attack C_2. However, it is difficult to determine if these new relationships (e.g., e_{11} starting to appear in close proximity of e_5) at timestamp t_k with respect to those of timestamp t_i are due to noise or are actually indicators of a change in the way e_{11} is being used in the wild. Considering all these temporal factors, it is desirable to have a model that is able to understand the context of a security event and its changes over time, and whose output can be quantitatively measured and studied. This is what ATTACK2VEC aims to do.

Problem formulation. We formalize our temporal security event evolution approach as follows. A security event $e_i \in \mathcal{E}$ is a timestamped observation recorded at timestamp i, where $\mathcal{E}$ denotes the set of all unique events and $|\mathcal{E}|$ denotes the size of $\mathcal{E}$. A security event sequence observed in an endpoint s_j is a sequence of events ordered by their observation time, $s_j = \{e_1^{(j)}, e_2^{(j)}, ..., e_l^{(j)}\}$. Let $\mathcal{S}_t = \{s_1^t, ..., s_i^t, ..., s_z^t\}$ denote the set of the security events from z endpoints during the t-th observation period. Finally we denote $\mathcal{S} = \{\mathcal{S}_1, ..., \mathcal{S}_t, ..., \mathcal{S}_T\}$, $t = 1, ..., T$, as the total security events over time T. It is worth noting that not all security events may be observed in a given $\mathcal{S}_t$. For example, security events associated with CVEs reported in 2018 are not present in the set of security events collected in 2017. Our goal is to find a mapping function $\mathcal{M}(e_i, \mathcal{S}, T) \rightarrow \{\eta_{e_i}^t\}$, where $t = 1, ..., T$ and $\eta_{e_i}^t \in \mathbb{R}^d$, $d \ll |\mathcal{E}|$ denotes a d-dimensional vector representation of the security event e_i at timestamp t. In the next section, we describe the data used in this paper in more detail. Then, in Section 4 we describe the methodology used by ATTACK2VEC.

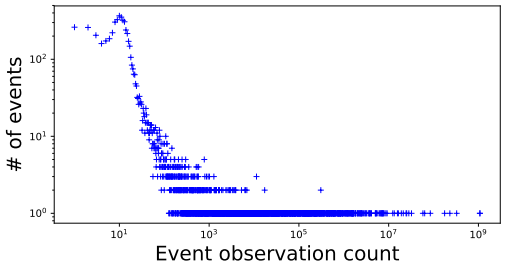


Figure 2: The distribution of events in our IPS security event dataset follows a power-law, much like the distribution of words in natural language, confirming the appropriateness of word embeddings to study the evolution of the use of security events over time.

3 Dataset

Data origin. As a proxy for the attack steps performed by miscreants in the wild, we use security event data collected from Symantec's intrusion prevention system (IPS). The company offers end users to explicitly opt in to its data sharing program to help improving its detection capabilities. To preserve the anonymity of users, endpoint identifiers are anonymized and it is not possible to link the collected data back to the users that originated it. Meta-information associated with a security event is recorded when the product detects network-level or system-level activity that matches a predefined signature (i.e., a security event).

Data collection. To thoroughly investigate security event evolution, we collected 102 days (one observation day per week for 102 consecutive weeks) of data between December 1, 2016 and November 08, 2018. From this data we extract the

following information: anonymized machine ID, timestamp, security event ID, event description, system actions, etc. On average, we collect 190 million security events collected from tens of millions unique machines per day. These security events were then reconstructed on a per machine basis and sorted chronologically. Note that for privacy reasons we use the anonymized endpoint ID to reconstruct a series of security events detected in a given mahcine and discard it after the reconstruction process is done. In total, the monitored machines generated 8,087 unique security events over the 102 observation days.

Data Limitations. It is important to note that the security event data is collected passively. That is, these security events are recorded only when corresponding attack signatures are triggered. Any events preemptively blocked by other security products cannot be observed. Additionally, any events that did not match the predefined signatures are also not observed. Hence the findings in this paper reflect security event evolution observed by Symantec's IPS, and the data can only be considered as a proxy for the actual attacker behavior in the wild. For example, we are unable to trace how zero day attacks are exploited in the wild [5]. However, as we show in Section 5 this data still allows ATTACK2VEC to identify meaningful trends in how security events are used and evolve. Additionally, ATTACK2VEC could be applied to any dataset with similar characteristics (i.e., a sequence of security events). Another limitation is that our dataset is composed of weekly snapshots, and we are therefore unable to characterize the evolution of security events that are faster than that. While this could prevent us from detecting quick anomalies in the way security events are used (i.e., those that go back to "normal" in a matter of a few days), this data is still representative enough to identify long term trends. We provide a more detailed discussion on the limitations of our data in Section 6.

Appropriateness for word embeddings. As we mentioned, the word embedding techniques used by ATTACK2VEC come from the natural language processing field. Word frequency in natural language follows a power-law distribution, and techniques from language modeling account for this distributional behavior. For these techniques to be appropriate to our data, therefore, it is ideal that our security events follow a similar distribution. Figure 2 shows that the events in our dataset indeed follows a power-law distribution. This similarity forms a solid theoretical foundation for us to use word embedding techniques to encode latent forms of security events, by considering sequences of security events in the IPS logs as short sentences and phrases in a special language. In the next section, we describe how ATTACK2VEC builds temporal embeddings from a sequence of security events in detail.

4 Methodology

In this section we first define the context window used in this work. We then formalize the techniques used to generate vector embeddings of security events. Finally, we describe ATTACK2VEC's architecture.

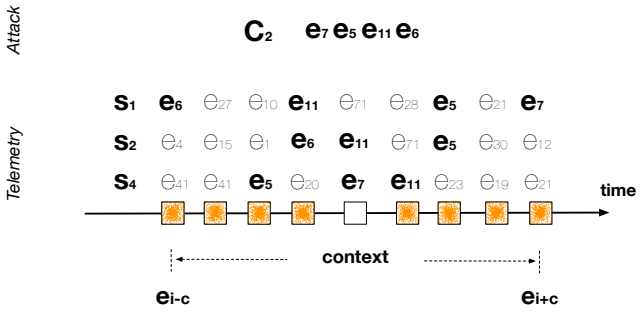


Figure 3: Illustration of context window ($c = 4$).

4.1 Context Window

Previous research made several interesting observations that different attack vectors are often packed together by attackers for a given period of time. For example, Kwon *et al.* [21] observed that silent delivery campaigns exhibit synchronized activity among a group of downloaders or domains and access the same set of domains to retrieve payloads within a short bounded time period. Shen *et al.* [40] pointed out that some machines may potentially observe different attacks from various adversary groups happening at the same time, and one coordinated attack may be observed by different machines. On the defense side (i.e., IPS telemetry), we consequently observe that related security events co-occur within a context (i.e., the sequence of attack steps that are commonly performed together with an attack step of interest). Note that this context can be defined as a time window [21] or a rollback window [40].

In this paper, we define the context as a sliding window, denoted as c, centering around a given security event e_i (see Figure 3). The purpose of using this symmetric context window is to deal with the noise incurred by concurrency at the telemetry level (see Section 2). For example, given a real-world coordinated attack e_7, e_5, e_{11}, e_6 (highlighted in bold in Figure 3), each endpoint may observe the attack vectors in different order (e.g., e_7 and e_5 may switch orders), attack vectors might be diluted by other unrelated security events (e.g., e_{71} observed between e_6 and e_5 in s_2), or certain security events are not observed, for example because they have been blocked by other security products before the IPS was able to log them (e.g., e_6 not observed in s_4). The proposed context window mechanism is able to capture the events surrounding a given security event (i.e., before and after), minimizing the impact of noise incurred by concurrency.

4.2 Temporal Security Event Embedding

The proposed temporal security event embedding is adapted from dynamic word embeddings by Yao *et al*. [48]. We use pointwise mutual information (PMI), a popular measure for word associations, to calculate weights between two security events given a contextual window c and an observation period t. PMI measures the extent to which the events co-occur more than by chance or are independent. The assumption is that if two events co-occur more than expected under independence there must be some kind of relationship between them. For each t-th observation period, we build a $|\mathcal{E}| \times |\mathcal{E}|$ PMI matrix, where a PMI value between e_i and e_j is defined as follows.

$$
\begin{aligned}
PMI_t(e_i, e_j, c, \mathcal{S}) &= max(log\left(\frac{p_t(e_i,e_j)}{p_t(e_i)p(e_j)}\right), 0), \\
p_t(e_i,e_j) &= \frac{W(e_i,e_j)}{|\mathcal{S}_t|}, \\
p_t(e_i) &= \frac{W(e_i)}{|\mathcal{S}_t|},
\end{aligned} \tag{1}
$$

where $W(e_i)$ and $W(e_j)$ respectively count the occurrences of security events e_i and e_j in $\mathcal{S}_t$, and $W(e_i,e_j)$ counts the number of times e_i and e_j co-occur within a context window (see Figure 3, Section 4.1) in $\mathcal{S}_t$. Note that when $W(e_i,e_j)$, the number of times e_i and e_j co-occurring in a given contextual window is small, $log\left(\frac{p_t(e_i,e_j)}{p_t(e_i)p(e_j)}\right)$ can be negative and affects the numerical stability. Therefore, we only keep the positive values in Eq 1 (see [22]).

Following the definition of PMI_t, the security event embedding $H(t)$, e.g., $\eta^t_{e_i} \in H(t)$, at t-th observation time is defined as a factorization of $PMI_t(c, \mathcal{S})$,

$$
H(t)H(t)^T \approx PMI_t(c, \mathcal{S}). \tag{2}
$$

The denser representation $H(t)$ reduces the noise [37] and is able to capture events with high-order co-occurrence (i.e., that appear in similar contexts) [30, 35]. These characteristics enable us to use word embedding techniques to encode latent forms of security events, and interpret the security event evolution in a meaningful and measurable way. Note that Li *et al*. [25] and Levy *et al*. [22] have theoretically proven that the skip-gram negative sampling (SGNS) used by the word2vec model can be viewed as explicitly (implicitly) factorizing a word co-occurrence matrix. We refer interested readers to [22, 25] for theoretical proofs.

Across time T, we also require that $\eta^t_{e_i} \approx \eta^{t+1}_{e_i}$. This means that the same security event should be placed in the same latent space so that their changes across time can be reliably studied. This requirement roots upon a practical implication. For example, a security event was observed after its associated CVE was disclosed. Its embeddings must therefore approximately stay the same before the disclosure date. Otherwise, we would observe unwanted embedding changes and invalidate the findings. To this end, Yao *et al*. [48] identified the solution of the following joint optimization problem as the temporal embedding results. Note that throughout this section, $\|.\|$ denotes squared Frobenius norm of a vector.

$$
\begin{aligned}
\min_{H(1),\ldots,H(T)} \frac{1}{2}\sum_{t=1}^{T} \|PMI_t(c,\mathcal{S}) - H(t)H(t)^T\|_2 \\
+ \frac{\alpha}{2}\sum_{t=1}^{T}\|H(t)\|^2 + \frac{\beta}{2}\sum_{t=1}^{T}\|H(t-1) - H(t)\|^2,
\end{aligned} \tag{3}
$$

where α and β are parameters respectively regularizing $H(t)$, and making sure that $H(t-1)$ and $H(t)$ are aligned (i.e., embeddings should be close if their associated contexts don't change between subsequent times.). In this way, all embeddings across time T are taken into consideration. At the same time, this method can accommodate extreme cases such as the one in which security event e_i is not observed in $(S)_t$ since the optimization is applied across all time slices in Eq 3. We refer interested readers to [48] for theoretical proofs and empirical comparison studies with other state-of-the-art embedding approaches. Following [48], we use grid search to identify the best parameters and experimentally set $\alpha = 10$, $\beta = 40$, $c = 8$, $d = 50$ and run 5 epochs for all the evaluations throughout our paper.

4.3 ATTACK2VEC Architecture

The architecture and workflow of ATTACK2VEC is depicted in Figure 4. Its operation consists of three phases: ❶ data collection and preprocessing, ❷ temporal event embedding, and ❸ event tracking and monitoring.

Data collection and preprocessing (❶). ATTACK2VEC takes the security event stream generated by endpoints (e.g., computers that installed an IPS). The goal of the data collection and preprocessing module is to prepare the data for the temporal event embedding method detailed in Section 4.2. ATTACK2VEC then consumes this timestamped security event data generated from millions of machines that send back their activity reports. The collection and preprocessing module reconstructs the security events observed on a given machine s_j as a sequence of events ordered by timestamps, in the format of $s_j = \{e^{(j)}_1, e^{(j)}_2, ..., e^{(j)}_l\}$. The output of the data collection and preprocessing module is $\mathcal{S}_t = \{s^t_1, ..., s^t_i, ..., s^t_z\}$ where z denotes the number of machines.

Temporal event embedding (❷). The core operation of ATTACK2VEC is embedding these security events into a low dimensional space over time. This phase takes $\mathcal{S}$ as input and encodes latent forms of security events, by considering sequences of security events in the IPS logs as short sentences and phrases in a special language. In this way, each security

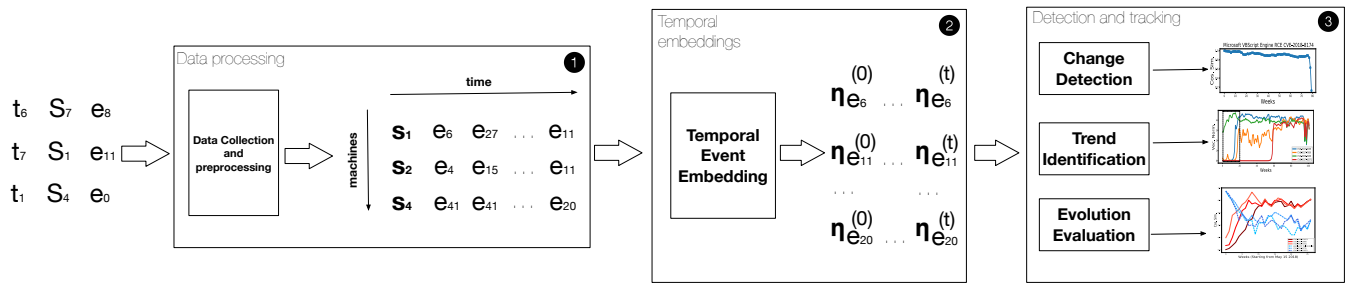


Figure 4: ATTACK2VEC's Architecture.

event, at a timestamp t, is represented by a d-dimensional vector representation $\eta^t_{e_i}$, and later aligned across time.

Detection and monitoring (❸). Once the security events are encoded in low-dimensional space, ATTACK2VEC is able to use various metrics (Section 5.1) to detect changes (Section 5.2), identify event trends (Section 5.3), and monitor how security events are exploited in the wild (Section 5.4) in a measurable and quantifiable way.

5 Evaluation

In this section, we provide a thorough evaluation of temporal event embeddings and ATTACK2VEC. We designed a number of experiments that allow us to answer the following questions:

- Can we use the temporal embeddings calculated by ATTACK2VEC to identify changes in how a security event is used in the wild (see Section 5.2)? To this end, we need our temporal embeddings to present high fidelity over time. The rationale behind this question is that the same security event should be placed in the same latent space by the proposed temporal event embedding method (see Section 4). Their changes across time can be reliably studied (see Section 5.6).
- Can we leverage temporal embeddings to identify trends in the use of security events (see Section 5.3)? The rationale behind this evaluation is that embedding vector norms across time should be more robust to the changes than word frequency which is static (i.e., calculated at a specific point of time) and sporadic.
- Can we leverage temporal embeddings to meaningfully understand the evolution of security events, and monitor how security events are exploited in the wild (see Section 5.4)?

In the following, we first define the metrics used by our evaluation. We then proceed to show that ATTACK2VEC is effective in answering these three research questions, and discuss the performance of our approach, showing that ATTACK2VEC is able to process a day of data within minutes. Finally, we present further evaluation of ATTACK2VEC, showing an end-to-end case on how our system can be used to assess the evolution in the use of a specific vulnerability in the wild.

5.1 Evaluation Metric

We use *cosine similarity* as the distance metric to quantify the temporal embedding changes at time t in the latent space. That is, for any two embeddings (i.e., $\eta^{(t)}_{e_i}$ and $\eta^{(t)}_{e_j}$), the similarity is measured as

$$similarity(\eta^{(t)}_{e_i}, \eta^{(t)}_{e_j}) = \frac{{\eta^{(t)}_{e_i}}^T \eta^{(t)}_{e_j}}{\|\eta^{(t)}_{e_i}\|_2 \|\eta^{(t)}_{e_j}\|_2}. \quad (4)$$

Note that in this paper the cosine similarity is used in positive space, where the outcome is bounded in [0, 1]. That is, two vectors with the same orientation have a cosine similarity of 1 (most similar), two vectors oriented at 90° relative to each other have a similarity of 0 (not similar).

Following Eq 4, we denote the *neighborhood* of a security event embedding $e^{(t)}_i$ as $\mathcal{N}(e^{(t)}_i)$, and accordingly defined as

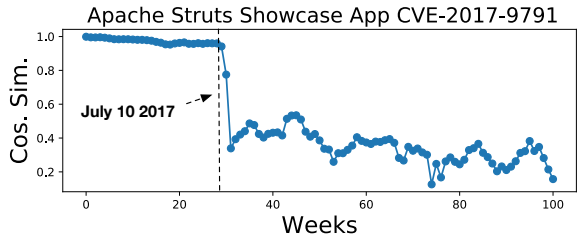
$$\mathcal{N}(e^{(t)}_i) = argsort_{e^{(t)}_j}(similarity(e^{(t)}_i, e^{(t)}_j)). \quad (5)$$

$\mathcal{N}(e^{(t)}_i)$ enables us to use temporal embeddings to discover and analyze how different security events are used together with e_i. We use $\mathcal{N}_k(e^{(t)}_i)$ to denote the top k closest neighbors of e_i. As we show in Section 5.4, this can be used to identify security events that are frequently used together as part of a multi-step attack.

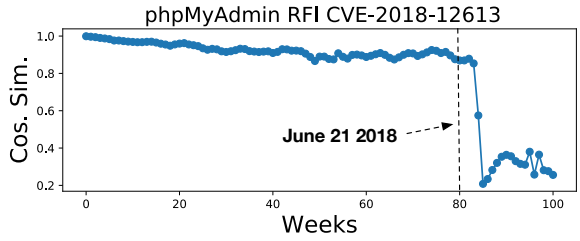
We also use a *weighted drift* metric to measure a security event *relative* changes. This metric is defined in Eq 6 as

$$weighted_drift(e_i) = argsort_t \left(\frac{\|\eta^{(t-1)}_{e_i}, \eta^{(t)}_{e_i}\|}{\sum_{e \in \mathcal{E}} \|\eta^{(t-1)}_{e}, \eta^{(t)}_{e}\|} \right). \quad (6)$$

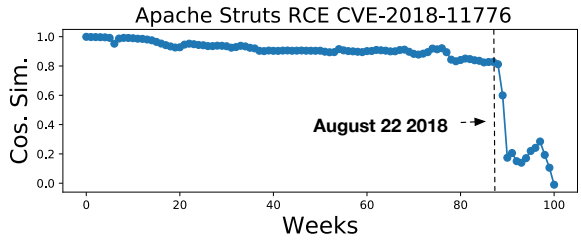
Eq 6 normalizes a security event's embedding change by the sum of all security event changes within that observation period. This metric enables us to measure how a security event changes comparing to the other security events within a given observation point.



(a) CVE-2017-9791 disclosure date: July 10 2017



(b) CVE-2018-12613 disclosure date: June 21 2018



(c) CVE-2018-11776 disclosure date: August 22 2018

Figure 5: Temporal embedding results of "Apache Struts Showcase App CVE-2017-9791" (5a), "phpMyAdmin RFI CVE-2018-12613" (5b), and "Apache Struts RCE CVE-2018-11776" (5c). The cosine similarities of the CVE embeddings are stable before they are publicly disclosed, and decline swiftly after the disclosure.

5.2 Change Detection

One of the key practical questions when evaluating the temporal security event embeddings built by ATTACK2VEC is determining the fidelity of the embedding results over time. In this paper, fidelity refers to the condition that the same security event should be placed in the same latent space. That is, if the frequency and the contexts of a security event between subsequent time slices don't change, its latent embedding should stay the same. This consistency allows the change to be reliably detected. This requirement lays the foundation to quantitatively study their changes. The concept of fidelity is different from the stability term used in previous research approaches in which stability was used to evaluate how classifiers perform after certain period of time. Bearing this difference in mind, we use the following two criteria to evaluate the fidelity of temporal embeddings and show how ATTACK2VEC can faithfully capture both single event usage change and global changes:

- **criterion a.** The cosine similarity of the event embeddings must be stable when an event usage does not change between subsequent time slices.
- **criterion b.** The cosine similarity of these embeddings should change swiftly if these events are used in different attacks or emerge as a new attack vector.

It is important to note that while these criteria are helpful in demonstrating the power of word embeddings extracted by ATTACK2VEC, they are self-referential and not in themselves sufficient to validate the effectiveness of our approach.

Single event change detection. To evaluate whether our two criteria hold for our dataset, we use three CVEs, "Apache Struts Showcase App CVE-2017-9791" (Figure 5a), "phpMyAdmin RFI CVE-2018-12613" (Figure 5b), and "Apache Struts RCE CVE-2018-11776" (Figure 5c). These CVEs were disclosed between 2017 and 2018. Regarding the aforementioned two evaluation criteria, these vulnerabilities were not disclosed in 2016, and therefore they did not have a matching signature in the IPS from which we collected our data. Thus, they form a good baseline for temporal fidelity evaluation. We therefore expect the following properties to hold:

response to a. Before a vulnerability was disclosed, its corresponding signature does not exist hence its non-existent context should stay the same until timestamp t. That is, if the vulnerability's disclosure date is t, $similarity(\eta_{e_i}^{(0)}, \eta_{e_i}^{(z)})$, where $z \in (0,t]$, should be stable.

response to b. After the disclosure date, the cosine similarity values of its embeddings should change swiftly. The justification is obvious. If attackers start exploiting a vulnerability, its corresponding security event moves away from its non-existent context and such drift leads to embedding changes.

For each CVE, we calculate the cosine similarity between each event's current representation (i.e., at timestamp t) and its *original* representation (i.e., at timestamp 0, on December 1 2016) over our observation period (i.e., $similarity(\eta_{e_i}^{(0)}, \eta_{e_i}^{(t)})$, where $t = 1...T$. See Eq 4). The results are shown in Figure 5. As we can observe, the temporal embeddings of CVE-2017-9791, CVE-2018-12613 and CVE-2018-11776 are stable across time and their cosine similarity values are above 0.9 before their respective disclosure dates (see **criterion a**). The way to interpret **criterion a** is that before a vulnerability is disclosed, its corresponding signature does not exist, and therefore its context is *non-existing*. As such, this context should remain constant until the vulnerability starts being exploited in the wild. Figure 5a shows that the cosine similarity between the embeddings of CVE-2017-9791 calculated daily and the original one recorded on

day 0 is stable and above 0.95 before July 10 2017, which is when the vulnerability was disclosed. Note that the similarity is not strictly 1.0 because of marginal deviation incurred by joint optimization across time slices (see Eq 3). Nevertheless, the high similarity before the disclosure date shows that ATTACK2VEC obtains correct temporal embeddings. After their public disclosure of each CVE, on the other hand, we expect the context in which each vulnerability is exploited to quickly change. This can be measured by ATTACK2VEC with the fact that the cosine similarity values of CVE-2017-9791, CVE-2018-12613, and CVE-2018-11776 decline quickly and move away from the original non-existing context built for those CVEs (see **criterion b**). This phenomenon exemplifies that the temporal embeddings capture the changes in the context in which a security event is used.

It is also worth noting that the temporal embeddings of CVE-2018-11776 show an immediate change after disclosure, while those of CVE-2018-12613 are slightly delayed for a couple of weeks (i.e., CVE-2018-12613 was officially published on June 21 2018 and the embedding starts to drift on July 12 2018). This phenomenon, i.e., the gap between public disclosure dates and real world exploits was well discussed in Sabottke *et al*. [39], and ATTACK2VEC allows to easily observe it.

The temporal embeddings generated by ATTACK2VEC not only allow us to identify when a vulnerability starts being exploited in the wild, but also how these event embeddings change after the disclosure date. To monitor and evaluate these changes, instead of comparing the context of a security event with the one extracted from the first day of observation, we compare the cosine similarity of the contexts extracted on subsequent time slices – between each event's current representation (i.e., at timestamp t) and its previous representation (i.e., at timestamp $t-1$). In short, we calculate $similarity(\eta_{e_i}^{(t)}, \eta_{e_i}^{(t-1)})$ (see Eq 4), which enables us to capture how the context of each event evolves between two subsequent observations. If the use of an event remains stable, the cosine similarity between $\eta_{e_i}^{(t)}$ and $\eta_{e_i}^{(t-1)}$ will remain high. If, on the other hand, the event experiences a sudden change in the way it is used in the wild, then its context will also significantly change and the cosine similarity with the previous observation will suddenly decrease, allowing an analyst to identify the point in time in which this change happened. To demonstrate this, we reuse the security event "Apache Struts Showcase App CVE-2017-9791" from earlier in this section. We calculate the cosine similarity between subsequent snapshots, where t_0 starts from July 10 2017 (the public disclosure date). This evolution is depicted in Figure 6.

We can observe the following: ❶ The cosine similarity values decline for the first three weeks. This phenomenon implies that CVE-2017-9791 started being exploited in different attacks after its disclosure date, with attackers trying different strategies to reliably exploit this vulnerability. ❷ the cosine similarity increases between the $3rd$ week and the $10th$ week. This phenomenon implies that CVE-2017-9791 became being exploited in less diversified attacks, indicating that attackers were converging towards a stable way to exploit the CVE. ❸ The cosine similarity stabilizes after the $10th$ week, which means that CVE-2017-9791 started being exploited in a *stable* context. This could indicate that attackers weaponized the CVE into a reliable attack, and possibly developed methods to exploit it at scale (e.g., by including it in an exploit kit). Later in the timeline, we can see other changes in the way in which attacks are exploited, but after each sudden change we observe a stabilization in how the CVE is exploited, indicating that attackers keep the same modus operandi over long periods of time.

A possible concern is that the changes in context identified by ATTACK2VEC might be due to noise and not representative of actual changes in the modus operandi of attackers. To demonstrate that this is not the case, we use the event co-occurrence matrix $PMI_t(c, 8)$ as defined in Section 4. This matrix captures the co-occurrence of any two events within the context window. For each observation time t, we select the top events that co-occurred with CVE-2017-9791 to better understand the phenomenon. If the changes in the use of a CVE identified by ATTACK2VEC are meaningful, we expect the co-occurrence matrix on that day to suddenly change, but to later stabilize and remain similar over time. For the first three weeks after disclosure in Figure 6 (❶), CVE-2017-9791 was used in conjunction with known attack vectors such as Apache Struts RCE CVE-2013-2251, HTTP Apache Tomcat UTF-8 dir traversal CVE-2008-2938, and malicious OGNL expression upload. By the third week, while some attack vectors were still associated with CVE-2017-9791, the vulnerability gradually started being used together with more recent server attack vectors (e.g., Apache Struts RCE CVE-2016-3087) and application vulnerabilities (e.g., WebNMS RCE CVE-2016-6603 and Web CMS Think PHP RCE). After ❷, CVE-2017-9791 started being used consistently with the aforementioned attack vectors and with several additional attack vectors (e.g., Apache Struts dynamic method invocation RCE CVE-2016-3081, Drupal PHP RCE, and generic PHP REC) Once CVE-2017-9791 reached ❸, its usage patterns became reasonably stable. Note that small fluctuations still happen when new Apache Struts related vectors were disclosed and exploited (e.g., . Apache Struts CVE-2017-9805 (week 11), CVE-2017-12611 (week 15) and CVE-2017-12617 (week 21), and temporary withdrawn of CVE-2017-12617 (around week 45) in the attacks. These changes are reliably detected by ATTACK2VEC.

In summary, our method is able to capture changes in the security event embeddings with high fidelity.

Global change detection. Recall that temporal event embeddings are the solution of a joint optimization problem across all time slices (see Section 4). Therefore, such embeddings not only encode their respective usage and context in a given time slice, but also its history across all time slices. In this

Rank (by changes)	Mar. 9 2017	Jan. 4 2018	Oct. 15 2018
1	ZyNOS Information Disclosure	Rig Exploit Kit Website	Unwanted Extension or Scam Sites Redirection
2	WordPress Mobile-Detector Arbitrary File Upload	Malicious Javascript Website	Fake Tech Support Website
3	Netgear Router Remote Command Execute	Fake Tech Support Website	Drupal Core RCE
4	Wordpress Arbitrary File Download	Malicious Redirection	Fake Browser History Injection
5	Fake Flash Player Download	JSCoinminer Download	Mass Injection Website

Table 1: Top 5 events with most usage changes in selective dates.

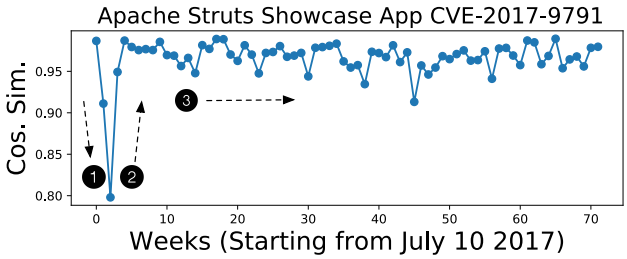


Figure 6: Temporal embedding result of Apache Struts Showcase App CVE-2017-9791. Cosine similarity values for each plot is calculated as $similarity(\eta_{e_i}^{(t-1)}, \eta_{e_i}^{(t)})$ (i.e., cosine similarity between subsequent time slices), where t starts from July 10 2017 (the public disclosure date).

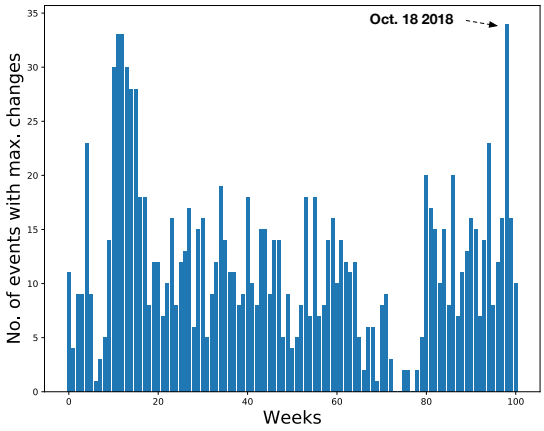


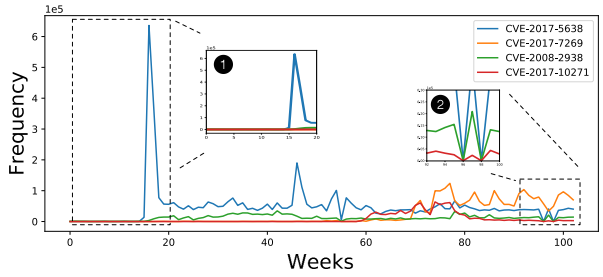
Figure 7: Summary of the security event changes between December 1 2016 and November 15 2018.

section, we show how to leverage the temporal embeddings generated by ATTACK2VEC to find times where we observe an anomalous high number of changes in the use of multiple security events.

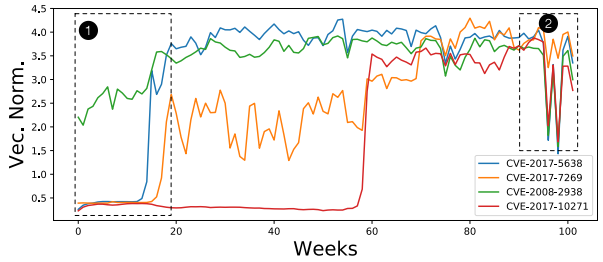
ATTACK2VEC computes a list of changes for each security event e_i in all time slices using the weighted drift metric (see Eq 6). We use this to identify the observation periods in which many security events exhibit most usage changes. These points are interesting candidates for scrutiny for security analysts, since multiple changes in word embeddings might indicate the emergence of new pervasive attacks. Note that we remove all the security events with less than 100 observations in 2 years time (see Section 6 for a rationale for this). Figure 7 shows a histogram of the time slices in which security event usage changes most. As we can see, we observe 34 security events with most usage changes between October 11 and October 18 2018. We selectively list the top 5 events with most usage changes in different dates (see Table 1). These changes demonstrate how security event evolve over the time. For example, at the beginning of 2017, we can observe many changes relating to exploits affecting routers. This can be an indicator of a large attack campaign targeting such devices unfolding. Across time, more attacks change over fake tech support websites, coinminer and content management systems. This context information is usually for analysts to improve situational awareness and be promptly warned about emerging attacks.

5.3 Trend Identification

One *de facto* method used in empirical studies [5, 24, 32, 46] to analyze temporal usage changes is leveraging frequencies to reveal patterns. That is, previous approaches often start by determining the occurrence frequency of events across the data, and using the event frequency time series as a criterion to reveal the significance of these events. Despite its straightforwardness in analyzing and visualizing temporal data, one drawback of this approach is that frequencies are prone to noise because they are only counted at a specific timestamp. Another drawback of using temporal frequency, especially in practical analysis, is that it is more difficult to compare two time series when they are both trending but at a different magnitude, and a sudden spike in the occurrence of one security event might make it difficult to identify other important events that are not happening as frequently. To demonstrate this problem, we use four popular remote Web server attack vectors (see Figure 8a). We can observe in Figure 8a (inset ❶) , that Apache Struts CVE-2017-5638 (blue line) was overwhelmingly exploited by attackers after it was disclosed. Its preponderant usage overshadows (see the zoom region inset ❶ in Figure 8a) the other popular remote Web server attack vectors (e.g., HTTP Apache Tomcat UTF-8 Dir

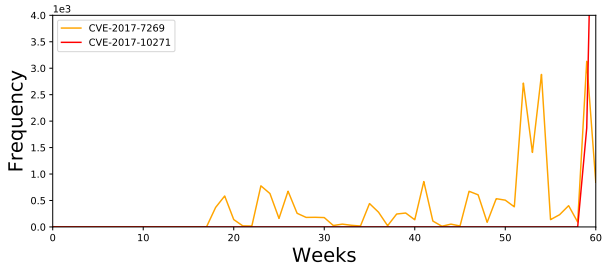


(a)

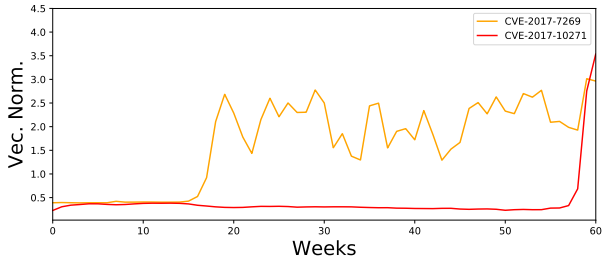


(b)

Figure 8: Event frequencies (8a) and event embedding vector norms (8b). We select four CVEs relating to different Web services to demonstrate the robustness of temporal event embedding in trend changes (CVE-2008-2938 in green line, CVE-2017-5638 in blue line, CVE-2017-10271 in red line, and CVE-2017-7269 in orange line). The dashed box highlights the observation period discussed in Section 5.3.



(a)



(b)

Figure 9: Comparison study of CVE-2017-7269 (orange line) and CVE-2017-10271 (red line). ATTACK2VEC can trace the emerging popularity of CVE-2017-7269 even though it is barely observable in Figure 8.

Traversal CVE-2008-2938) when using event frequency time series to comparatively study attack vector popularity.

In the rest of this section, we demonstrate that the word embeddings calculated by ATTACK2VEC are more robust than temporal frequencies to reveal trend changes. Recall the definition of $PMI_t(c, \mathcal{S})$ (defined in Section 4). It contains information from $W(e_i, e_j)$, the number of times e_i and e_j co-occurring in a given contextual window, and $W(e_i)$ and $W(e_j)$ respectively count the occurrences of security events e_i and e_j. The norms of word embeddings η_{e_i}, as a consequence of matrix factorization of $PMI_t(c, \mathcal{S})$, grow with word frequency and are averaged by their contexts. Therefore, we can leverage these norms to identify event usage trends. We compare temporal frequencies (Figure 8a) to the embedding vector norm (Figure 8b) in the context of trend identification. It is straightforward to see that the security event "Apache Struts CVE-2017-5638" was predominantly leveraged by the attackers for a short period of time. As we said, such sudden spike makes other trendy CVEs used at that same period of time less detectable (Figure 8a). However, we can see that the temporal event embeddings (Figure 8b) reveal the real patterns behind the frequencies, and are able to capture trends more reliably. For example, we can clearly see in Figure 8b that CVE-2008-2938 ("HTTP Apache Tomcat UTF-8 Dir Traversal," green line in Figure 8a) is a persistent vector used by the attackers. This was not observable in Figure 8a because CVE-2017-5638 dominates during that period of time. Nevertheless, it is worth noting that we study the independent popularity of these attack vectors and reveal their underlying usage trend beneath the event frequency time series. Hence in this paper we do not intend to study how such attack vectors influence each other. It is coincidental that the red (CVE-2017-10271) and green (CVE-2008-2938) lines show a similar vector norm shift. In fact, these vector norm shifts were incurred by their similar usage changes (e.g., dropping close to zero) during the same period (see inset ❷, Figure 8a). ATTACK2VEC still preserves the trends of red (CVE-2017-10271) and green (CVE-2008-2938) lines (i.e., larger than zero) in this extreme case and their respective vector norms become stable immediately after.

Our temporal embedding can also capture how two CVEs, CVE-2017-10271 ("Oracle WebLogic RCE," red line in Figure 8a) and CVE-2017-7269 ("Buffer overflow in the ScStoragePathFromUrl function in the WebDAV service in IIS 6.0," orange line in Figure 8a) gradually emerge as a trendy attack vector over time. That is, both CVE-2017-10271 (red line) and CVE-2017-7269 (orange line), before their disclosure dates (May 27, 2017 and Oct 17, 2017), are flat. This proves that ATTACK2VEC faithfully captures their non-existent trend. After their disclosure, ATTACK2VEC is able to correctly keep track of their trends. For example, in Figure 9 we show that

Drupal core RCE (CVE-2018-7602)	
May 15 2018	Nov. 08 2018
Joomla JCE Vulnerability	phpMyAdmin RFI (CVE-2018-12613)
Wordpress RevSlider/ShowBiz Bypass (CVE-2014-9735)	Drupal SQL Injection (CVE-2014-3704)
WordPress Symposium Plugin Shell Upload	Adobe Flex BlazeDS RCE (CVE-2017-3066)

Table 2: Top 3 security events associated with CVE-2018-7602 at the beginning (May 15 2018) and the end (November 18 2018) of our time span.

our approach can reliably trace the trends of CVE-2017-7269 (orange line) from approximately week 19 in the figure. Although its usage (e.g., frequency) is less observable than CVE-2008-2938 and CVE-2017-5638 during the same period of time (see Figure 8), its emerging trend is still recognized by the increase in vector norm as we can see in Figure 9b. Additionally, despite the five times of frequency difference between it initial disclosure (week 19 in Figure 9a) and later weeks (after week 51 in Figure 9a), ATTACK2VEC captures the trend changes with a small fluctuation of less than 0.6. ATTACK2VEC also reliably captures CVE-2017-10271 (red line) non-existent trend before its public disclosure.

5.4 Event Evolution

Another useful functionality for which ATTACK2VEC can be used is understanding how attacks evolve in the wild, and in particular monitoring which attack steps are often performed together by attackers. In a nutshell, security events that are often used together will have similar contexts. Identifying events with such similar contexts could help detecting emerging threats such as new botnets scanning for specific vulnerabilities (e.g., Mirai or WannaCry) or new exploit kits that are probing for specific weaknesses in victim systems [3, 33].

To evaluate ATTACK2VEC's capability of tracking the evolution of security event contexts over time in relation to each other we use the same CVE from Section 2, "Drupal core RCE (CVE-2018-7602)." This CVE is for a highly critical remote code execution (RCE) vulnerability that exists within multiple subsystems of Drupal 7.x and 8.x, enabling attackers to compromise a machine running a Drupal website. This vulnerability was first disclosed on April 23 2018, and we observe its activities in the our data starting from May 15 2018. Due to its high severity, it is interesting to see how such a critical vulnerability was exploited in different contexts across time.

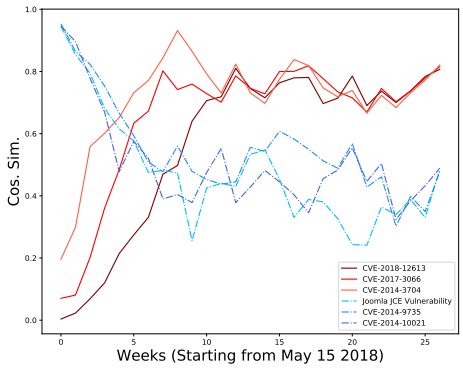
We use Eq 5 to identify the top 3 security events associated with CVE-2018-7602 at the beginning and the end of our time span. These events are the ones that have the closest context to CVE-2018-7602, and are therefore used in association with it. Table 2 shows a detailed description of these events. It is interesting to note that the top 3 attack vectors used together with CVE-2018-7602 at the beginning resemble a reconnaissance attack aiming at all three major content management systems - Joomla, Wordpress, and Drupal. Toward the end of the time span, however, we notice that CVE-2018-7602 migrates to be part of a more specific multi-step attack, aiming at the ecosystem surrounding the Drupal CMS - php (phpMyAdmin), SQL (Drupal SQL Injection), and Flex (BlazeDS RCE).

Leveraging Eq 4, we show in Figure 10a that the temporal security event embedding computed by ATTACK2VEC can meaningfully capture the aforementioned usage changes across time. "Joomla JCE Vulnerability," CVE-2014-9735, CVE-2014-10021 (blue dashed lines), and CVE-2018-12613, CVE-2017-3066, CVE-2014-3704 (red solid lines) respectively are the top three closest security events associated with "Drupal core RCE (CVE-2018-7602)" at the beginning (end) of the observation span (starting from May 15 2018). We can clearly see that the red lines are rising (i.e., these security events are used more closely with CVE-2018-7602), and the blue lines are moving away from CVE-2018-7602. In general, we can see that the attackers change their modus operandi, using CVE-2018-7602 as part of a more targeted attack on Drupal, 8 weeks after its initial observation in the telemetry data. This can be indicative of the old reconnaissance campaign fading and of the new one more targeted towards Drupal emerging, or of an attacker changing their behavior. Note that such usage changes can be automatically detected using various change point detection algorithms [2].

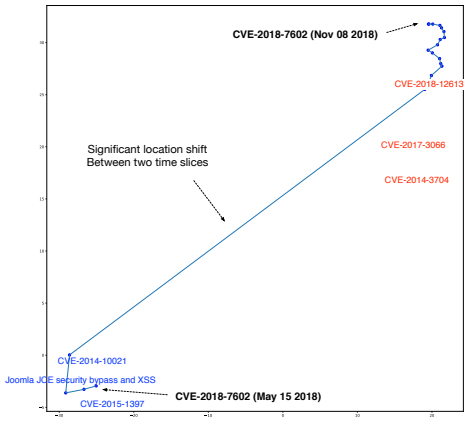
Trajectory visualization. The trajectory of a security event in the embedded latent space can assist security analysts to understand its context changes over time. To show this, we collect the top k security events associated with CVE-2018-7602 using Eq 5 in each time slice to form our trajectory data $D_{e_i} = \{\mathcal{N}_k(e_i^{(t)})\}$,where t starts from May 15, 2018. We accordingly plot the 2-D t-SNE projection of the temporal embeddings of CVE-2018-7602 (and the aforementioned security events) in Figure 10b to visualize its context change over time. In Figure 10b, each blue dot represents the 2-D location in the latent space at a given timestamp. We can observe a considerable drift in Figure 10b between the fourth and the fifth blue dot. This is correlated to the trend we observed in Figure 10a. The top 3 security events associated with CVE-2018-7602 at the beginning and the end of our time span are closer to the respectively locations in Figure 10b. Note that such changes in Figure 10b can be quantitatively detected as these locations are bounded in a Euclidean space.

5.5 System Performance

ATTACK2VEC is implemented in Python 3.7.3 and tested on a server with dual Xeon E5-2630 CPUs and 256GB memory running Ubuntu Linux 14.04. In this setup, ATTACK2VEC takes 859.86 seconds to construct the PPMI matrices for all 102 snapshots. Once the PPMI matrices are constructed, ATTACK2VEC takes 3014.18 seconds per epoch to optimize the temporal embeddings (see Section 4). We empirically run 5 epochs for ATTACK2VEC to reach the optimum embedding results. This leads to approximately 4.18 hours for



(a) Joomla JCE Vulnerability, CVE-2014-9735, CVE-2014-10021 (blue dashed lines) and CVE-2018-12613, CVE-2017-3066, CVE-2014-3704 (red solid lines) respectively are the top three closest security events associated with Drupal core RCE (CVE-2018-7602) at the beginning (end) of the observation span (starting from May 15 2018).



(b) Temporal t-SNE trajectory of Drupal core RCE (CVE-2018-7602). The significant shift in this figure can be indicative of the old reconnaissance campaign fading and of the new one more targeted towards Drupal emerging, or of an attacker changing their behavior.

Figure 10: Drupal core RCE (CVE-2018-7602) Evolution between May 15 2018 and November 8 2018.

Apache Struts Jakarta Multipart parser RCE (CVE-2017-5638)	
Mar. 23 2017	Nov. 08 2018
WifiCam Authentication Bypass	Malicious OGNL Expression Upload
CCTV-DVR Remote Code Execution	Apache Struts CVE-2017-12611
ZyNOS Information Disclosure	Malicious Serialized Object Upload

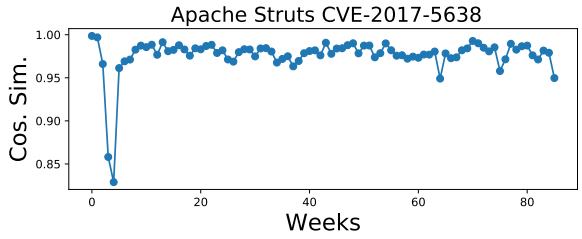
Table 3: Top 3 security events associated with CVE-2017-5638 at the beginning (March 9 2017) and the end (November 8 2018) of our time span.

ATTACK2VEC to generate final temporal embeddings for all 8,087 security events across 102 snapshots. This enables us to deploy ATTACK2VEC to understand the long term evolution of different security events at scale by security analysts.

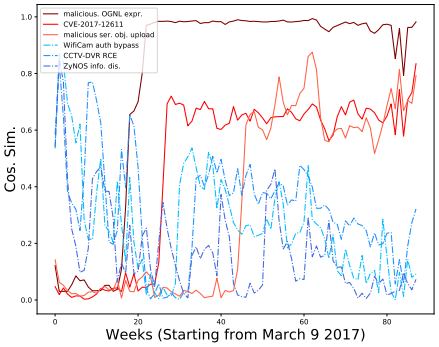
5.6 End-to-end Evaluation of ATTACK2VEC

In the previous sections we evaluated the ability of ATTACK2VEC to study various aspects of how a security event is exploited in the wild. We envision that ATTACK2VEC could be used by security analysts to understand the evolution of the use of a security event (e.g., a vulnerability) over time. In this section we provide an end-to-end example to show how an analyst could be using our tool to better understanding the context surrounding a security vulnerability and its evolution. To this end, we study the evolution of the security event "Apache Struts Jakarta Content-Type RCE (CVE-2017-5638)," a remote code execution vulnerability targeting Apache Struts. This vulnerability is classified by NVD as a critical bug with CVSS score 10.0, and was the culprit of the Equifax data breach [11].

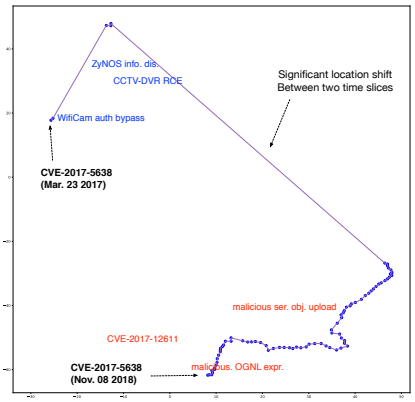
Figure 11a shows that CVE-2017-5638 experienced a change in the way it is being exploited between April 5, 2017 and April 13, 2017. ATTACK2VEC is able to identify this change in the way the vulnerability is exploited, and an analyst could easily identify this. We then want to understand what these two contexts looked like, and evaluate whether this information can help us understand the types of attacks that CVE-2017-5638 was used in. To this end we perform the analysis described in Section 5.4, whose results are shown in Figure 11b. In particular, we trace the top three security events with the closest context to CVE-2017-5638 at the beginning and at the end of our analysis period (see Table 3 for a detailed description of these events). The top 3 security events tightly associated with CVE-2017-5638 at the beginning of our observation span are IoT specific attack vectors. Figure 11b shows that these three IoT attack vectors maintain similar contexts to CVE-2017-5638 for approximately 10 weeks, indicating that the four vulnerabilities were frequently exploited together in the wild as part of a multi-step attack. Later in the analysis period (starting from May 8, 2017) we see that this attack is substituted by another attack that is targeted at the Apache Struts ecosystem, consisting of Malicious OGNL expression upload, CVE-2017-12611 and Malicious Serialized Object Upload. Figure 11c shows a similar pattern, with CVE-2017-5638 migrating from being close to the IoT security events to the Apache Struts related ones. This information could inform an analyst about the change in which CVE-2017-5638 was



(a) Temporal embedding result of Apache Struts Jakarta Multipart parser RCE (CVE-2017-5638). Cosine similarity values for each plot is calculated as $similarity(\eta_{e_i}^{(t-1)}, \eta_{e_i}^{(t)})$, where t starts from March 9 2017.



(b) WifiCam Authentication Bypass, CCTV-DVR Remote Code Execution, ZyNOS Information Disclosure (blue dashed lines) and Malicious OGNL expression upload, CVE-2017-12611 and Malicious Serialized Object Upload (red solid lines) are the top three closest security events associated with CVE-2017-5638 at the beginning (end) of the observation span (starting from March 9 2017), respectively.



(c) Temporal t-SNE trajectory of Apache Struts Jakarta Multipart parser RCE (CVE-2017-5638).

Figure 11: Apache Struts Jakarta Multi- part parser RCE (CVE-2017-5638) evolution between March 9 2017 and November 8 2018.

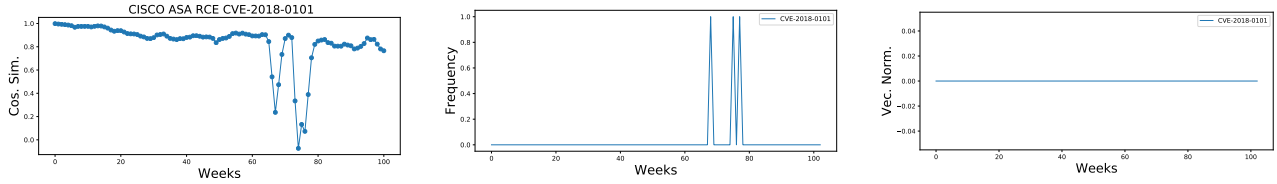
exploited in the wild, switching from an attack step as part of an IoT-centered attack to part of an attack centered around Apache Struts.

To further evaluate the accuracy of the embeddings calculated by ATTACK2VEC, and the meaningfulness of our analysis, we further investigate the contexts calculated at the beginning of our analysis period, when CVE-2017-5638 was exploited in conjunction with IoT-related vulnerabilities.To this end, we retrieve the remote IP addresses from which the security events originated on selected dates. We find that in many cases these four (including CVE-2017-5638) security events were generated from connections originating from the same IP addresses (for example, 701 unique IP addresses on March 23 2018), indicating that the relation depicted by the context is not an artifact of ATTACK2VEC, but it is indeed a large scale attack performed by the same malicious actors. These IP addresses were often located in residential ISPs, indicating a potential botnet infection.

We later found confirmation that a variant of the Mirai IoT malware was active during that period and was explicitly exploiting CVE-2017-5638, "WifiCam auth bypass," "CCTV-DVR RCE," and "ZyNOS information disclosure," the same vulnerabilities picked up by ATTACK2VEC as being related (see Table 3) [1]. This shows that ATTACK2VEC can help identifying complex attacks in an effective way. An additional advantage of our approach is that our system was able to flag this attack as emerging in the wild 72 weeks before it was actually discussed by security researchers, highlighting the potential of the use of temporal embeddings for early warning and situational awareness.

6 Limitations and Discussion

Rare events. The temporal security event embedding used in this paper builds on top of event frequency and event co-occurrence (see Section 4). By design the learned temporal event embeddings are biased towards word frequency per observation time. When analyzing the security events using the proposed method, we need to pay attention to the events that appear less frequently. Broadly speaking, these rare events can be grouped into two categories - i) the new security events associated with recently disclosed vulnerabilities and ii) those rarely observed in the IPS. For the new events, their corresponding disclosure dates are good indicators when interpreting the embedding results. For the events that rarely observed, frequency and popularity are two good reference points when interpreting the embedding results. Take "CVE-2018-0101 (Cisco Adaptive Security Appliance (ASA) RCE vulnerability)" for example (see Figure 12). ATTACK2VEC faithfully identifies its changes over time (see Figure 12a). However, when referring to event frequency (see Figure 12b), we can see that these changes do not represent that the attackers are changing their attack campaign strategy. Moreover, it is important to notice that our proposed system is robust



(a) (b) (c)

Figure 12: Rare event CVE-2018-0101 (Cisco Adaptive Security Appliance (ASA) RCE vulnerability).

and can correctly indicate that such event is not actively being exploited by attackers (see Figure 12c), and therefore the changes flagged at the previous steps are spurious.

Distraction from attackers. Our proposed temporal event embedding may be subject to distraction from malicious attackers, leading to inaccurate insights. For example, attackers could generate large amounts of fake events by targeting a considerable number of machines (e.g., hundreds of thousands) over a certain period of time (e.g., weeks). However, we argue that this would make the attackers more visible to the security companies who could track such malicious activities and block them accordingly. Additionally, such distraction operations would not bring financial incentive to the attackers. Note that once they switch back to real campaigns, our method would faithfully capture the new trend.

Limitations. ATTACK2VEC relies on a dataset of pre-labeled security events to generate insights and understand their evolution. An inherent limitation of this type of data is that an event can be detected only if it belongs to a known attack vector. If, for example, a new zero-day vulnerability started being exploited in the wild, this would not be reflected in the data until its signature is created. Our method is data-driven hence it can not deduce insights before an event was detected. However, such delay can be reduced since security companies typically use threat intelligence systems and employ human specialists to analyze intrusion data identifying new attack trends. We refer interested readers to Bilge *et al*. [5] for a detailed study on zero-day vulnerabilities.

7 Related Work

7.1 Embedding Applications in Security

Xu *et al*. [47] proposed to use network-based graph embedding to accomplish cross-platform binary code similarity detection task. The authors adopted the structure2vec approach to effectively compute embedding vectors for the control flow graph of binary functions. This allows for efficient similarity detection by comparing the embeddings for two functions. Song *et al*. [42] propose DeepMem, a graph-based deep learning approach to automatically generate abstract representations for kernel objects and recognize these objects from raw memory dumps. The key idea is building a memory graph and embed the graph nodes into a low-dimensional vector space using a node's actual content and the embeddings of its four kinds of neighboring nodes. These embeddings are then used as features for classification. Ding *et al*. [9] developed an assembly code representation learning model called Asm2Vec. The key idea is to encode assembly code syntax and control flow graph into a feature vector. At the query/estimation stage, the previously unknown assembly code is encoded into a lower-dimensional vector and compared using cosine similarity. Li *et al*. [23] introduced a data poisoning attack on matrix factorization. The authors demonstrated that, with the full knowledge of the learner, several attacks can be achieved.

7.2 Other Related Work

Concept drift. Concept drift refers to the phenomenon that the statistical properties of the target variable change over time. Such causes less accurate predictions across time. Within the context of security research, Maggi *et al*. [27] addressed concept drift in Web application security, while Kantchelian *et al*. [20] discussed adversarial drift. In recent years, Jordaney *et al*. [18] proposed Transcend, a statistical framework to identify aging classification models. The authors used a statistical comparison of samples seen during deployment with those used to train the model, thereby building metrics for prediction quality. They then combine both decision assessment (i.e., the robustness of the prediction results) and alpha assessment (i.e., the quality of the nonconformity measure) to detect concept drift.

Empirical studies on cyberattacks. Bilge *et al*. [5] conducted a systematic study of the characteristics of zero-day attacks through the data collected from 11 million endpoints. Nappa *et al*. [32] conducted a systematic analysis of the patching process of 1,593 vulnerabilities in 10 client-side applications over 5 year time, especially on measuring the patching delay and several patch deployment milestones for each vulnerability. Nayak *et al*. [33] carried an empirical study on vulnerability using field data and proposed several count-based metrics for attack surface evaluation. Vervier *et al*. [46] analyzed 18 months of data collected by an infrastructure specifically built to address BGP hijacks. The author characterized the BGP hijacks in this longitudinal study and provide a thorough investigation and validation of the candidate malicious BGP hijacks. Li *et al*. [24] conducted a large-scale empirical study of security patches that affected 862 open-

source projects.

Vulnerability prediction. Vulnerability prediction techniques learn the attack history from previous events (e.g., historical compromise data) and use the acquired knowledge to predict future ones. What learned in the history can offer insights to evolution and is therefore relevant to our work. Sabottke *et al*. [39] conducted a quantitative and qualitative exploration of the vulnerability-related information disseminated on Twitter. The authors built a twitter-based exploit detector, which was capable of providing early warnings for the existence of real-world exploits. Similarly, Bozorgi *et al*. [6] showed how to train linear support vector machines (SVMs) that predict whether and how soon a vulnerability is likely to be exploited (i.e., predict time to exploit). Recently, Shen *et al*. [40] demonstrated that recurrent neural networks (RNNs) can be leveraged to predict the specific steps (i.e., vulnerability that may be exploited) that would be taken by an adversary when performing an attack. Liu *et al*. [26] explored the effectiveness of forecasting security incidents. This study collected 258 externally measurable features about an organization's network covering two main categories: mismanagement symptoms (e.g., misconfigured DNS) and malicious activities (e.g., spam, scanning activities originated from this organization's network). Based on the data, the study trained and tested a Random Forest classifier on these features, and are able to achieve with 90% True Positive (TP) rate, 10% False Positive (FP) rate and an overall accuracy of 90% in forecasting security incidents. In summary, these approaches learn the attack history from previous events (e.g., historical compromise data) and use the acquired knowledge to predict future ones. They don't provide thorough investigations on how security events evolve over time.

Alert correlation. Alert correlation [7, 45] refers to a process that analyzes the alert logs produced by IDS and forms higher-level information on attempted intrusions. Once alerts are correlated among multiple monitors, the results can provide IDS a holistic view of the network monitored. A lot of work has been done in this areas such CRIM [7], DIDMA [19], ACARM [44], INDRA [17], etc. Vasilomanolakis *et al*. [45] summarized the current state of the art in the area of distributed and collaborative intrusion detection. In contrast to this previous work, this paper focuses on understanding the emergence, the evolution, and the characteristics of attack steps in relation to the wider context in which they are exploited.

Automated causality analysis. Causality is an orthogonal but interesting problem relating to ATTACK2VEC. Hercule [34] uses tainted path s from t to model the causality. SteamSpot [28] uses a new similarity function to compare graphs and builds information flow graph clusters to detect anomalies. NoDoze [15] builds provenance graph of a given event and use a novel diffusion algorithm to efficiently propagate and aggregate the anomalous scores. HOLMES [31] also leverages provenance graph and identify APT attacks via information flow graphs. Conversely, our goal is to provide a reliable new method for analyzing attack trends than frequency analysis.

Closest work. One of the closest work to this paper is DarkEmbed [43]. It used paragraph vector to learn low dimensional distributed representations, i.e., embeddings, of darkweb/deepweb discussions. These embeddings effectively captured the meaning of these discussions and their other characteristics, such as language, and indicator words. DarkEmbed then trained a classifier to recognize posts discussing vulnerabilities that would be exploited in the wild. DarkEmbed is essentially a NLP analysis. Different from DarkEmbed, our work focuses on using representation vectors to parameterize the conditional probabilities of security events in the context of other events, and study how these security events evolve from a temporal perspective. Another closest work is [29]. The authors carried out a longitudinal analysis of a large corpus of cyber threat descriptions. It quantifies the severity and types (e.g., worms, viruses and trojans) of 12,400 threats detected by Symantec's AV and 2,700 attacks detected by Symantec's IPS. Different from [29], our work focuses on how the security events evolve and monitor how security events are exploited in the wild from real-world intrusion prevention data.

8 Conclusion

In this paper, we showed that techniques that were developed in the area of natural language processing can be used to effectively model and monitor the evolution of cyberattacks. To demonstrate this, we developed ATTACK2VEC, a tool that leverages word embeddings to understand the context in which attack steps are exploited. We showed that ATTACK2VEC is effective in flagging changes in the way attacks unfold. In future work we plan to investigate how the use of ATTACK2VEC could make the work of security analysts easier in studying emerging attacks.

Acknowledgments

We wish to thank the anonymous reviewers for their feedback and our shepherd Brad Reaves for his help in improving this paper.

References

[1] Unit 42. Multi-exploit iot/linux botnets mirai and gafgyt target apache struts, sonicwall. https://unit42.paloaltonetworks.com/unit42-multi-exploit-iotlinux-botnets-mirai-gafgyt-target-apache-struts-sonicwall/, 2018.

[2] Samaneh Aminikhanghahi and Diane J Cook. A survey of methods for time series change point detection. *KIS*, 51(2), 2017.

[3] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the mirai botnet. In *USENIX Security Symposium*, 2017.

[4] Leyla Bilge, Davide Balzarotti, William Robertson, Engin Kirda, and Christopher Kruegel. Disclosure: detecting botnet command and control servers through large-scale netflow analysis. In *ACSAC*, 2012.

[5] Leyla Bilge and Tudor Dumitras. Before we knew it: an empirical study of zero-day attacks in the real world. In *ACM CCS*, 2012.

[6] Mehran Bozorgi, Lawrence K Saul, Stefan Savage, and Geoffrey M Voelker. Beyond heuristics: learning to classify vulnerabilities and predict exploits. In *KDD*, 2010.

[7] Frédéric Cuppens and Alexandre Miege. Alert correlation in a cooperative intrusion detection framework. In *IEEE S&P*, 2002.

[8] Bhuwan Dhingra, Zhong Zhou, Dylan Fitzpatrick, Michael Muehl, and William W Cohen. Tweet2vec: Character-based distributed representations for social media. *arXiv preprint arXiv:1605.03481*, 2016.

[9] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *IEEE S&P*, 2019.

[10] Brown Farinholt, Mohammad Rezaeirad, Paul Pearce, Hitesh Dharmdasani, Haikuo Yin, Stevens Le Blond, Damon McCoy, and Kirill Levchenko. To catch a ratter: Monitoring the behavior of amateur darkcomet rat operators in the wild. In *IEEE S&P*, 2017.

[11] Apache Software Foundation. The apache software foundation confirms equifax data breach due to failure to install patches provided for apache struts exploit. https://blogs.apache.org/foundation/entry/media-alert-the-apache-software, 2017.

[12] Chris Grier, Lucas Ballard, Juan Caballero, Neha Chachra, Christian J Dietrich, Kirill Levchenko, Panayiotis Mavrommatis, Damon McCoy, Antonio Nappa, Andreas Pitsillidis, et al. Manufacturing compromise: the emergence of exploit-as-a-service. In *ACM CCS*, 2012.

[13] Guofei Gu, Roberto Perdisci, Junjie Zhang, and Wenke Lee. Botminer: Clustering analysis of network traffic for protocol-and structure-independent botnet detection. In *USENIX Security Symposium*, 2008.

[14] Guofei Gu, Phillip A Porras, Vinod Yegneswaran, Martin W Fong, and Wenke Lee. Bothunter: Detecting malware infection through ids-driven dialog correlation. In *USENIX Security Symposium*, 2007.

[15] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. Nodoze: Combatting threat alert fatigue with automated provenance triage. In *NDSS*, 2019.

[16] Eric M Hutchins, Michael J Cloppert, and Rohan M Amin. Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains. *Leading Issues in Information Warfare & Security Research*, 2011.

[17] Ramaprabhu Janakiraman, Marcel Waldvogel, and Qi Zhang. Indra: A peer-to-peer approach to network intrusion detection and prevention. In *WET ICE*, 2003.

[18] Roberto Jordaney, Kumar Sharad, Santanu K Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. Transcend: Detecting concept drift in malware classification models. In *USENIX Security Symposium*, 2017.

[19] Pradeep Kannadiga and Mohammad Zulkernine. Didma: A distributed intrusion detection system using mobile agents. In *SNPD-SAWN*, 2005.

[20] Alex Kantchelian, Sadia Afroz, Ling Huang, Aylin Caliskan Islam, Brad Miller, Michael Carl Tschantz, Rachel Greenstadt, Anthony D Joseph, and JD Tygar. Approaches to adversarial drift. In *AIsec*, 2013.

[21] Bum Jun Kwon, Virinchi Srinivas, Amol Deshpande, and Tudor Dumitraş. Catching worms, trojan horses and pups: Unsupervised detection of silent delivery campaigns. In *NDSS*, 2017.

[22] Omer Levy and Yoav Goldberg. Neural word embedding as implicit matrix factorization. In *NIPS*, 2014.

[23] Bo Li, Yining Wang, Aarti Singh, and Yevgeniy Vorobeychik. Data poisoning attacks on factorization-based collaborative filtering. In *NIPS*, 2016.

[24] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *ACM CCS*, 2017.

[25] Yitan Li, Linli Xu, Fei Tian, Liang Jiang, Xiaowei Zhong, and Enhong Chen. Word embedding revisited: A new representation learning and explicit matrix factorization perspective. In *AAAI*, 2015.

[26] Yang Liu, Armin Sarabi, Jing Zhang, Parinaz Naghizadeh, Manish Karir, Michael Bailey, and Mingyan Liu. Cloudy with a chance of breach: Forecasting cyber security incidents. In *USENIX Security Symposium*, 2015.

[27] Federico Maggi, William Robertson, Christopher Kruegel, and Giovanni Vigna. Protecting a moving target: Addressing web application concept drift. In *RAID*, 2009.

[28] Emaad Manzoor, Sadegh M Milajerdi, and Leman Akoglu. Fast memory-efficient anomaly detection in streaming heterogeneous graphs. In *KDD*, pages 1035–1044. ACM, 2016.

[29] Ghita Mezzour, L Richard Carley, and Kathleen M Carley. Longitudinal analysis of a large corpus of cyber threat descriptions. *J Comput Virol Hack Tech*, 12(1), 2016.

[30] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, 2013.

[31] Sadegh M Milajerdi, Rigel Gjomemo, Birhanu Eshete, R Sekar, and VN Venkatakrishnan. Holmes: real-time apt detection through correlation of suspicious information flows. In *IEEE S&P*, 2019.

[32] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras. The attack of the clones: A study of the impact of shared code on vulnerability patching. In *IEEE S&P*, 2015.

[33] Kartik Nayak, Daniel Marino, Petros Efstathopoulos, and Tudor Dumitraş. Some vulnerabilities are different than others. In *RAID*, 2014.

[34] Kexin Pei, Zhongshu Gu, Brendan Saltaformaggio, Shiqing Ma, Fei Wang, Zhiwei Zhang, Luo Si, Xiangyu Zhang, and Dongyan Xu. Hercule: Attack story reconstruction via community discovery on correlated log graph. In *ACSAC*. ACM, 2016.

[35] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *EMNLP*, 2014.

[36] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *USENIX Security Symposium*, 2003.

[37] Reinhard Rapp. Word sense discovery based on sense descriptor dissimilarity. In *MT Summit*, 2003.

[38] Thomas Rid and Ben Buchanan. Attributing cyber attacks. *Journal of Strategic Studies*, 2015.

[39] Carl Sabottke, Octavian Suciu, and Tudor Dumitras. Vulnerability disclosure in the age of social media: Exploiting twitter for predicting real-world exploits. In *USENIX Security Symposium*, 2015.

[40] Yun Shen, Enrico Mariconti, Pierre Antoine Vervier, and Gianluca Stringhini. Tiresias: Predicting security events through deep learning. In *ACM CCS*, 2018.

[41] Robin Sommer and Vern Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *IEEE S&P*, 2010.

[42] Wei Song, Heng Yin, Chang Liu, and Dawn Song. Deepmem: Learning graph neural network models for fast and robust memory forensic analysis. In *ACM CCS*, 2018.

[43] Nazgol Tavabi, Palash Goyal, Mohammed Almukaynizi, Paulo Shakarian, and Kristina Lerman. Darkembed: Exploit prediction with neural language models. In *IAAI*, 2018.

[44] Fredrik Valeur, Giovanni Vigna, Christopher Kruegel, and Richard A Kemmerer. Comprehensive approach to intrusion detection alert correlation. *IEEE Transactions on dependable and secure computing*, 1(3), 2004.

[45] Emmanouil Vasilomanolakis, Shankar Karuppayah, Max Mühlhäuser, and Mathias Fischer. Taxonomy and survey of collaborative intrusion detection. *ACM CSUR*, 47(4):55, 2015.

[46] Pierre-Antoine Vervier, Olivier Thonnard, and Marc Dacier. Mind your blocks: On the stealthiness of malicious bgp hijacks. In *NDSS*, 2015.

[47] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *ACM CCS*, 2017.

[48] Zijun Yao, Yifan Sun, Weicong Ding, Nikhil Rao, and Hui Xiong. Dynamic word embeddings for evolving semantic discovery. In *WSDM*, 2018.

Leaky Images: Targeted Privacy Attacks in the Web

Cristian-Alexandru Staicu
Department of Computer Science
TU Darmstadt

Michael Pradel
Department of Computer Science
TU Darmstadt

Abstract

Sharing files with specific users is a popular service provided by various widely used websites, e.g., Facebook, Twitter, Google, and Dropbox. A common way to ensure that a shared file can only be accessed by a specific user is to authenticate the user upon a request for the file. This paper shows a novel way of abusing shared image files for targeted privacy attacks. In our attack, called *leaky images*, an image shared with a particular user reveals whether the user is visiting a specific website. The basic idea is simple yet effective: an attacker-controlled website requests a privately shared image, which will succeed only for the targeted user whose browser is logged into the website through which the image was shared. In addition to targeted privacy attacks aimed at single users, we discuss variants of the attack that allow an attacker to track a group of users and to link user identities across different sites. Leaky images require neither JavaScript nor CSS, exposing even privacy-aware users, who disable scripts in their browser, to the leak. Studying the most popular websites shows that the privacy leak affects at least eight of the 30 most popular websites that allow sharing of images between users, including the three most popular of all sites. We disclosed the problem to the affected sites, and most of them have been fixing the privacy leak in reaction to our reports. In particular, the two most popular affected sites, Facebook and Twitter, have already fixed the leaky images problem. To avoid leaky images, we discuss potential mitigation techniques that address the problem at the level of the browser and of the image sharing website.

1 Introduction

Many popular websites allow users to privately share images with each other. For example, email services allow attachments to emails, most social networks support photo sharing, and instant messaging systems allow files to be sent as part of a conversation. We call websites that allow users to share images with each other *image sharing services*.

This paper presents a targeted privacy attack that abuses a vulnerability we find to be common in popular image sharing services. The basic idea is simple yet effective: An attacker can determine whether a specific person is visiting an attacker-controlled website by checking whether the browser can access an image shared with this person. We call this attack *leaky images*, because a shared image leaks the private information about the victim's identity, which otherwise would not be available to the attacker. To launch a leaky images attack, the attacker privately shares an image with the victim through an image sharing service where both the attacker and the victim are registered as users. Then, the attacker includes a request for the image into the website for which the attacker wants to determine whether the victim is visiting it. Since only the victim, but no other user, is allowed to successfully request the image, the attacker knows with 100% certainty whether the victim has visited the site.

Beyond the basic idea of leaky images, we describe three further attacks. First, we describe a targeted attack against groups of users, which addresses the scalability issues of the single-victim attack. Second, we show a pseudonym linking attack that exploits leaky images shared via different image sharing services to determine which user accounts across these services belong to the same individual. Third, we present a scriptless version of the attack, which uses only HTML, and hence, works even for users who disable JavaScript in their browsers.

Leaky images can be (ab)used for targeted attacks in various privacy-sensitive scenarios. For example, law enforcement could use the attack to gather evidence that a suspect is visiting particular websites. Similarly but perhaps less noble, a governmental agency might use the attack to deanonymize a political dissident. As an example of an attack against a group, consider deanonymizing reviewers of a conference. In this scenario, the attacker would gather the email addresses of all committee members and then share leaky images with each reviewer through some of the various websites providing that service. Next, the attacker would embed a link to an external website into a paper under review, e.g.,

Table 1: Leaky images vs. related web attacks. All techniques assume that the victim visits an attacker-controlled website.

Threat	Who can attack?	What does the attacker achieve?	Usage scenario
Tracking pixels	Widely used ad providers and web tracking services	Learn that user visiting site A is the same as user visiting site B	Large-scale creation of low-entropy user profiles
Social media fingerprinting	Arbitrary website provider	Learn into which sites the victim is logged in	Large-scale creation of low-entropy user profiles
Cross-site request forgery	Arbitrary website provider	Perform side effects on a target site into which the victim is logged in	Abuse the victim's authorization by acting on her behalf
Leaky images	Arbitrary website provider	Precisely identify the victim	Targeted, fine-grained deanonymization

a link to a website with additional material. If and when a reviewer visits that page, while being logged into one of the image sharing services, the leaky image will reveal to the attacker who is reviewing the paper. The prerequisite for all these attacks is that the victim has an account at a vulnerable image sharing service and that the attacker is allowed to share an image with the victim. We found at least three highly popular services (Google, Microsoft Live, and Dropbox) that allow sharing images with any registered user, making it straightforward to implement the above scenarios.

The leak is possible because images are exempted from the same-origin policy, and because image sharing services authenticate users through cookies. When the browser makes a third-party image request, it attaches the user's cookie of the image sharing website to it. If the decision of whether to authorize the image request is cookie-dependent, then the attacker can infer the user's identity by observing the success of the image request. Related work discusses the dangers of exempting JavaScript from the same-origin policy [24], but to the best of our knowledge, there is no work discussing the privacy implications of observing the result of cross-origin requests to privately shared images.

Leaky images differ from previously known threats by enabling arbitrary website providers to precisely identify a victim (Table 1). One related technique are tracking pixels, which enable tracking services to determine whether two visitors of different sites are the same user. Most third-party tracking is done by a few major players [13], allowing for regulating the way these trackers handle sensitive data. In contrast, our attack enables arbitrary attackers and small websites to perform targeted privacy attacks. Another related technique is social media fingerprinting, where the attacker learns whether a user is currently logged into a specific website.[1] In contrast, leaky images reveal not only whether a user is logged in, but precisely which user is logged in. Leaky images resemble cross-site request forgery (CSRF) [33], where a malicious website performs a request to a target site on behalf of the user. CSRF attacks typically cause side effects on the server, whereas our attack simply retrieves an image. We discuss in Section 5 under what conditions defenses proposed against CSRF, as well as other mitigation techniques, can reduce the risk of privacy leaks due to leaky images.

To understand how widespread the leaky images problem is, we study 30 out of the 250 most popular websites. We create multiple user accounts on these websites and check whether one user can share a leaky image with another user. The attack is possible if the shared image can be accessed through a link known to all users sharing the image, and if access to the image is granted only to certain users. We find that at least eight of the 30 studied sites are affected by the leaky images privacy leak, including some of the most popular sites, such as Facebook, Google, Twitter, and Dropbox. We carefully documented the steps for creating leaky images and reported them as privacy violations to the security teams of the vulnerable websites. In total, we informed eight websites about the problem, and so far, six of the reports have been confirmed, and for three of them we have been awarded bug bounties. Most of the affected websites are in the process of fixing the leaky images problem, and some of them, e.g., Facebook and Twitter, have already deployed a fix.

In summary, this paper makes the following contributions:

- We present leaky images, a novel targeted privacy attack that abuses image sharing services to determine whether a victim visits an attacker-controlled website.
- We discuss variants of the attack that aim at individual users, groups of users, that allow an attacker to link user identities across image sharing services, and that do not require any JavaScript.
- We show that eight popular websites, including Facebook, Twitter, Google, and Microsoft Live are affected by leaky images, exposing their users to be identified on third-party websites.
- We propose several ways to mitigate the problem and discuss their benefits and weaknesses.

2 Image Sharing in the Web

Many popular websites, including Dropbox, Google Drive, Twitter, and Facebook, enable users to upload images and to

[1] See https://robinlinus.github.io/socialmedia-leak/ or https://browserleaks.com/social.

share these images with a well-defined set of other users of the same site. Let i be an image, U be the set of users of an image sharing service, and let $u^i_{owner} \in U$ be the owner of i. By default, i is not accessible to any other users than u^i_{owner}. However, an owner of an image can share the image with a selected subset of other users $U^i_{shared} \subseteq U$, which we define to include the owner itself. As a result, all users $u \in U^i_{shared}$, but no other users of the service and no other web users, have read access to i, i.e., can download the image via a browser.

Secret URLs To control which users can access an image, there are several implementation strategies. One strategy is to create a *secret URL* for each shared image, and to provide this URL only to users allowed to download the image. In this scenario, there is a set of URLs L^i (L stands for "links") that point to a shared image i. Any user who knows a URL $l^i \in L^i$ can download i through it. To share an image i with multiple users, i.e., $|U^i_{shared}| > 1$, there are two variants of implementing secret URLs. On the one hand, each user u may obtain a personal secret URL l^i_u for the shared image, which is known only to u and not supposed to be shared with anyone. On the other hand, all users may share the same secret URL, i.e., $L^i = \{l^i_{shared}\}$. A variant of secret URLs are URLs that expire after a given amount of time or after a given number of uses. We call these URLs session URLs.

Authentication Another strategy to control who accesses an image is to authenticate users. In this scenario, the image sharing service checks for each request to i whether the request comes from a user in U^i_{shared}. Authentication may be used in combination with secret URLs. In this case, a user u may access an image i only if she knows a secret URL l^i and if she is authenticated as $u \in U^i_{shared}$. The most common way to implement authentication in image sharing services are cookies. Once a user logs into the website of an image sharing service, the website stores a cookie in the user's browser. When the browser requests an image, the cookie is sent along with the request to the image sharing service, enabling the server-side of the website to identify the user.

Image Sharing in Practice Different real-world image sharing services implement different strategies for controlling who may access which image. For example, Facebook mostly uses secret URLs, which initially created confusion among users due to the apparent lack of access control[2]. Gmail relies on a combination of secret URLs and authentication to access images attached to emails. Deciding how to implement image sharing is a tradeoff between several design goals, including security, usability, and performance. The main advantage of using secret URLs only is that third-party content delivery networks may deliver images, without any cross-domain access control checks. A drawback of secret URLs is that they should not be used over non-secret channels, such as HTTP, since these channels are unable to protect the secrecy of requested URLs. The main advantage of authentication is to not require links to be secret, enabling them to be sent over insecure channels. On the downside, authentication-based access control makes using third-party content delivery networks harder, because cookie-based authentication does not work across domains.

[2] `https://news.ycombinator.com/item?id=13204283`

Same-Origin Policy The same-origin policy regulates to what extent client-side scripts of a website can access the document object model (DOM) of the website. As a default policy, any script loaded from one origin is not allowed to access parts of the DOM loaded from another origin. Origin here means the URI scheme (e.g., *http*), the host name (e.g., *facebook.com*), and the port number (e.g., 80). For example, the default policy implies that a website *evil.com* that embeds an `iframe` from *facebook.com* cannot access those parts of the DOM that have been loaded from *facebook.com*. There are some exceptions to the default policy described above. One of them, which is crucial for the leaky images attack, are images loaded from third parties. In contrast to other DOM elements, a script loaded from one origin can access images loaded from another origin, including whether the image has been loaded at all. For the above example, *evil.com* is allowed to check whether an image requested from *facebook.com* has been successfully downloaded.

3 Privacy Attacks via Leaky Images

This section presents a series of attacks that can be mounted using leaky images. At first, we describe the conditions under which the attack is possible (Section 3.1). Then, we present a basic attack that targets individual users (Section 3.2), a variant of the attack that targets groups of users (Section 3.3), and an attack that links identities of an individual registered at different websites (Section 3.4). Next, we show that the attack relies neither on JavaScript nor CSS, but can be performed by a purely HTML-based website (Section 3.5). Finally, we discuss how leaky images compare to previous privacy-related issues, such as web tracking (Section 3.6).

3.1 Attack Surface

Our attack model is that an attacker wants to determine whether a specific victim is visiting an attacker-controlled website. This information is important from a privacy point of view and usually not available to operators of a website. An operator of a website may be able to obtain some information about clients visiting the website, e.g., the IP and the browser version of the client. However, this information is limited, e.g., due to multiple clients sharing

Table 2: Conditions that enable leaky image attacks.

Authentication (e.g., cookies)	URL of image: Publicly known	Secret URL shared among users	Per-user secret URL
Yes	(1) Leaky image	(2) Leaky image	(3) Secure
No	(4) Irrelevant	(5) Secure	(6) Secure

the same IP or the same browser version, and often insufficient to identify a particular user with high confidence. Moreover, privacy-aware clients may further obfuscate their traces, e.g., by using the Tor browser, which hides the IP and other details about the client. Popular tracking services, such as Google Analytics, also obtain partial knowledge about which users are visiting which websites. However, the use of this information is legally regulated, available to only a few tracking services, and shared with website operators only in anonymized form. In contrast, the attack considered here enables an arbitrary operator of a website to determine whether a specific person is visiting the website.

Leaky image attacks are possible whenever all of the following four conditions hold. First, we assume that the attacker and the victim are both users of the same image sharing service. Since many image sharing services provide popular services beyond image sharing, such as email or a social network, their user bases often cover a significant portion of all web users. For example, Facebook announced that it has more than 2 billion registered users[3], while Google reported to have more than 1 billion active Gmail users each month[4]. Moreover, an attacker targeting a specific victim can simply register at an image sharing service where the victim is registered. Second, we assume that the attacker can share an image with the victim. For many image sharing services, this step involves nothing more than knowing the email address or user name of the victim, as we discuss in more detail in Section 4. Third, we assume that the victim visits the attacker-controlled website while the victim's browser is logged into the image sharing service. Given the popularity of some image sharing services and the convenience of being logged in at all times, we believe that many users fulfill this condition for at least one image sharing service. In particular, in Google Chrome and the Android operating system, users are encouraged immediately after installation to login with their Google account and to remain logged in at all times.

The fourth and final condition for leaky images concerns the way an image sharing service determines whether a request for an image is from a user supposed to view that image. Table 2 shows a two-dimensional matrix of possible implementation strategies, based on the description of secret URLs and authentication-based access control in Section 2. In one dimension, a website can either rely on authentication or not. In the other dimension, the site can make an image available through a publicly known URL, a secret URL shared among the users allowed to access the image, or a per-user secret URL. Out of the six cases created by these two dimensions, five are relevant in practice. The sixth case, sharing an image via a publicly known URL without any authentication, would make the image available to all web users, and therefore is out of the scope of this work. The leaky image attack works in two of the five possible cases in Table 2, cases 1 and 2. Specifically, leaky images are enabled by sites that protect shared images through authentication and that either do not use secret URLs at all or that use a single secret URL per shared image. Section 4 shows that these cases occur in practice, and that they affect some of today's most popular websites.

[3] https://techcrunch.com/2017/06/27/facebook-2-billion-users/

[4] https://www.businessinsider.de/gmail-has-1-billion-monthly-active-users-2016-2

3.2 Targeting a Single User

After introducing the prerequisites for leaky images, we now describe several privacy attacks based on them. We start with a basic version of the attack, which targets a single victim and determines whether the victim is visiting an attacker-controlled website. To this end, the attacker uploads an image i to the image sharing service and therefore becomes the owner of the image, i.e., $u_{attacker} = u^i_{owner}$. Next, the attacker configures the image sharing service to share i with the victim user u_{victim}. As a result, the set of users allowed to access the image is $U^i_{shared} = \{u_{attacker}, u_{victim}\}$. Then, the attacker embeds a request for i into the website s for which the attacker wants to determine whether the victim is visiting the site. Because images are exempted from the same-origin policy (Section 2), the attacker-controlled parts of s can determine whether the image gets loaded successfully and report this information back to the attacker. Once the victim visits s, the image request will succeed and the attacker knows that the victim has visited s. If any other client visits s, though, the image request fails because s cannot authenticate the client as a user in U^i_{shared}. We assume that the attacker does not visit s, as this might mislead the attacker to believe that the victim is visiting s.

Because the authentication mechanism of the image sharing service ensures that only the attacker and the victim can access the image, a leaky image attack can determine with 100% accuracy whether the targeted victim has visited the site. At the same time, the victim may not notice that she was tracked, because the image can be loaded in the background.

For example, Figure 1 shows a simple piece of HTML code with embedded JavaScript. The code requests a leaky image, checks whether the image is successfully loaded, and sends this information back to the attacker-controlled web

```
<script>
window.onload = function() {
 var img = document.getElementById("myPic");
 img.src = "https://imgsharing.com/leakyImg.png";
 img.onload = function() {
   httpReq("evil.com", "is the target");
 }
 img.onerror = function() {
   httpReq("evil.com", "not the target");
 }
}
</script>
<img id="myPic">
```

Figure 1: Tracking code included in the attacker's website.

server via another HTTP request. We assume `httpReq` is a method that performs such a request using standard browser features such as `XMLHttpRequest` or `innerHTML` to send the value of the second argument to the domain passed as first argument. Alternatively to using `onload` to detect whether the image has been loaded, there are several variations, which, e.g., checking the width or height of the loaded image. As we show below (Section 3.5), the attack is also possible within a purely HTML-based website, i.e., without JavaScript.

The described attack works because the same-origin policy does not apply to images. That is, the attacker can include a leaky image through a cross-origin request into a website and observe whether the image is accessible or not. In contrast, requesting an HTML document does not cause a similar privacy leak, since browsers implement a strict separation of HTML coming from different origins. A second culprit for the attack's success is that today's browsers automatically include the victim's cookie in third-party image requests. As a result, the request passes the authentication of the image sharing service, leaking the fact that the request comes from the victim's browser.

3.3 Targeting a Group of Users

The following describes a variant of the leaky images attack that targets a group of users instead of a single user. In this scenario, the attacker considers a group of n victims and wants to determine which of these victims is visiting a particular website.

As an example, consider a medium-scale spear phishing campaign against the employees of a company. After preparing the actual phishing payload, e.g., personalized emails or cloned websites, the attacker may include a set of leaky images to better understand which victims interact with the payload and in which way. In this scenario, leaky images provide a user experience analysis tool for the attacker.

A naive approach would be to share one image i_k ($1 \leq k \leq n$) with each of the n victims. However, this naive approach does not scale well to larger sets of users: To track a group of 10,000 users, the attacker needs 10,000 shared images and 10,000 image requests per visit of the website. In other words, this naive attack has $\mathcal{O}(n)$ complexity, both in the number of leaky images and in the number of requests. For the above example, this naive way of performing the attack might raise suspicion due to the degraded performance of the phishing site and the increase in the number of network requests.

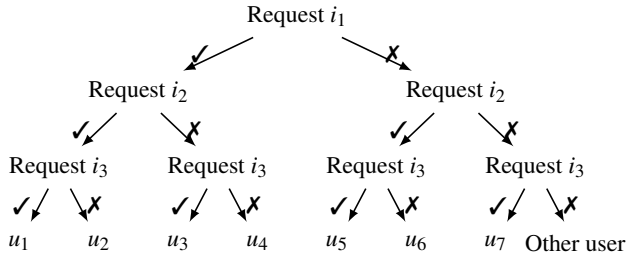


Figure 2: Binary search to identify individuals in a group of users u_1 to u_7 through requests to leaky images i_1 to i_3.

To efficiently attack a group of users, an attacker can use the fact that image sharing services allow sharing a single image with multiple users. The basic idea is to encode each victim with a bit vector and to associate each bit with one shared image. By requesting the images associated with each bit, the website can compute the bit vector of a user and determine if the user is among the victims, and if yes, which victim it is. This approach enables a binary search on the group of users, as illustrated in Figure 2 for a group of seven users. The website includes code that requests images i_1, i_2, and i_3, and then determines based on the availability of the images which user among the targeted victims has visited the website. If none of the images is available, then the user is not among the targeted victims. In contrast to the naive approach, the attack requires only $\mathcal{O}(log(n))$ shared images and only $\mathcal{O}(log(n))$ image requests, enabling the attack on larger groups of users.

In practice, launching a leaky image attack against a group of users requires sharing a set of images with different subsets of the targeted users. This process can be automated, either through APIs provided by image sharing services or through UI-level web automation scripts. However, this process will most likely be website-specific which makes it expensive for attacking multiple websites at once.

3.4 Linking User Identities

The third attack based on leaky images aims at linking multiple identities that a single individual has at different image sharing services. Let *siteA* and *siteB* be two image sharing services, and let u_{siteA} and u_{siteB} be two user accounts, registered at the two image sharing services, respectively. The

```
1  <!-- Three users (u1, u2, u3) have access to two
2  images (i1, i2) as follows: u1 to (i1);
3  u2 to (i2); u3 to (i1, i2) -->
4  <object data="leaky-domain.com/i1.png">
5   <object data="evil.com?info=not_i1?sid=2342"/>
6  </object>
7  <object data="leaky-domain.com/i2.png">
8   <object data="evil.com?info=not_i2?sid=2342"/>
9  </object>
10
11 <object data="leaky-domain.com/invalidImg.png">
12  <object data="leaky-domain.com/invalidImg2.png">
13   <object data="leaky-domain.com/invalidImg3.png">
14    <object data="evil.com?info=loaded?sid=2342"/>
15   </object>
16  </object>
17 </object>
```

Figure 3: HTML-only variant of the leaky image group attack. All the `object` tags should have the `type` property set to `image/png`.

attacker wants to determine whether u_{siteA} and u_{siteB} belong to the same individual. For example, this attack might be performed by law enforcement entities to check whether a user account that is involved in criminal activities matches another user account that is known to belong to a suspect.

To link two user identities, the attacker essentially performs two leaky image attacks in parallel, one for each image sharing service. Specifically, the attacker shares an image i_{siteA} with u_{siteA} through one image sharing service and an image i_{siteB} with u_{siteB} through the other image sharing service. The attacker-controlled website requests both i_{siteA} and i_{siteB}. Once the targeted individual visits this site, both requests will succeed and establish the fact that the users u_{siteA} and u_{siteB} correspond to the same individual. For any other visitors of the site, at least one request will fail because the two requests only succeed if the browser is logged into both user accounts u_{siteA} and u_{siteB}.

The basic idea of linking user accounts generalizes to more than two image sharing services and to user accounts of more than a single individual. For example, by performing two attacks on groups of users, as described in Section 3.3, in parallel, an attacker can establish pairwise relationships between the two groups of users.

3.5 HTML-only Attack

The leaky image attack is based on the ability of a client-side website to request an image and to report back to the attacker-controlled server-side whether the request was successful or not. One way to implement it is using client-side JavaScript code, as shown in Figure 1. However, privacy-aware users may disable JavaScript completely or use a security mechanism that prevents JavaScript code from reading details about images loaded from different domains.

We present a variant of the leaky image attack implemented using only HTML code, i.e., without any JavaScript or CSS. The idea is to use the `object` HTML tag, which allows a website to specify fallback content to be loaded if there is an error in loading some previously specified content.[5] When nesting such `object` elements, the browser first requests the resource specified in the outer element, and in case it fails, it performs a request to the inner element instead. Essentially, this behavior corresponds to a logical *if-not* instruction in pure HTML which an attacker may use to implement the leaky image attack.

Figure 3 shows an example of this attack variant. We assume that there are three users u_1, u_2, and u_3 in the target group and that the attacker can share leaky images from leaky-domain.com with each of them. The comment at the beginning of Figure 3 specifies the exact sharing configuration. We again need $log(n)$ images to track n users, as for the JavaScript-based attack against a group of users (Section 3.3). We assume that the server-side generates the attack code upon receiving the request, and that the generated code contains a session ID as part of the reporting links pointing to evil.com. In the example, the session ID is 2342. Its purpose is to enable the server-side code to link multiple requests coming from the same client.

The main insight of this attack variant is to place a request to the attacker's domain as fallbacks for leaky image requests. For example, if the request to the leaky image i_1 at line 4 fails, a request is made to evil.com for an alternative resource in line 5. This request leaks the information that the current user cannot access i_1, i.e., `info=not_i1`. By performing similar requests for all the leaky images, the attacker leaks enough information for precisely identifying individual users. For example, if in a given session, evil.com receives `not_i1`, but not `not_i1`, the attacker can conclude that the user is u_2. Because the server-side infers the user from the absence of requests, it is important to ensure that the current tracking session is successfully completed before drawing any conclusions. Specifically, we must ensure that the user or the browser did not stop the page load before all the nested `object` tags were evaluated. One way to ensure this property is to add a sufficiently high number of nested requests to non-existent images in lines 11 to 13 followed by a request that informs the attacker that the tracking is completed, in line 14. The server discards every session that does not contain this last message.

As a proof of concept, we tested the example attack and several variants of it in the newest Firefox and Chrome browsers and find the HTML-only attack to work as expected.

[5] `https://html.spec.whatwg.org/multipage/iframe-embed-object.html#the-object-element`

3.6 Discussion

Tracking pixels Leaky images are related to the widely used tracking pixels, also called web beacons [14, 8, 47], but both differ regarding who learns about a user's identity. A tracking pixel is a small image that a website s loads from a tracker website s_{track}. The image request contains the user's cookie for s_{track}, enabling the tracker to recognize users across different page visits. As a result, the tracking service can analyze which pages of s users visit and show this information in aggregated form to the provider of s. If the tracker also operates services where users register, it can learn which user visits which site. In contrast, leaky images enable the operator of a site s to learn that a target user is visiting s, without relying on a tracker to share this information, but by abusing an image sharing service. As for tracking pixels, an attacker can deploy leaky image attacks with images of 1x1 pixel size to reduce its impact on page loading time.

Fingerprinting Web fingerprinting techniques [12, 29, 10, 22, 1, 2, 30] use high-entropy properties of web browsers, such as the set of installed fonts or the size of the browser window, to heuristically recognize users. Like fingerprinting, leaky images aim at undermining the privacy of users. Unlike fingerprinting, the attacks presented here enable an attacker to determine specific user accounts, instead of recognizing that one visitor is likely to be the same as another visitor. Furthermore, leaky images can determine a visitor's identity with 100% certainty, whereas fingerprinting heuristically relies on the entropy of browser properties.

Targeted attacks versus large-scale tracking Leaky images are well suited for targeted attacks [37, 6, 26, 16], but not for large-scale tracking of millions of users. One reason is that leaky images require the attacker to share an image with each victim, which is unlikely to scale beyond several hundreds users. Another reason is that the number of image requests that a website needs to perform increases logarithmically with the number of targeted users, as discussed in Section 3.3. Hence, instead of aiming at large-scale tracking in the spirit of tracking pixels or fingerprinting, leaky images are better suited to target (sets of) individuals. However, this type of targeted attacks is reported to be increasingly popular, especially when dealing with high-value victims [37].

4 Leaky Images in Popular Websites

The attacks presented in the previous section make several assumptions. In particular, leaky images depend on how real-world image sharing services implement access control for shared images. To understand to what extent popular websites are affected by the privacy problem discussed in this paper, we systematically study the prevalence of leaky images. The following presents our methodology (Section 4.1), our main findings (Section 4.2), and discusses our ongoing efforts toward disclosing the detected problems in a responsible way (Section 4.3).

4.1 Methodology

Selection of websites To select popular image sharing services to study, we examined the top 500 most popular websites, according to the "Top Moz 500" list[6]. We focus on websites that enable users to share data with each other. We exclude sites that do not offer an English language interface and websites that do not offer the possibility to create user accounts. This selection yields a list of 30 websites, which we study in more detail. Table 3 shows the studied websites, along with their popularity rank. The list contains all of the six most popular websites, and nine of the ten most popular websites. Many of the analyzed sites are social media platforms, services for sharing some kind of data, and communication platforms.

Image sharing One condition for our attacks is that an attacker can share an image with a victim. We carefully analyze the 30 sites in Table 3 to check whether a site provides an image sharing service. To this end, we create multiple accounts on each site and attempt to share images between these accounts using different channels, e.g., chat windows or social media shares. Once an image is shared between two accounts, we check if the two accounts indeed have access to the image. If this requirement is met, we check that a third account cannot access the image.

Access control mechanism For websites that act as image sharing services, we check whether the access control of a shared image is implemented in a way that causes leaky images, as presented in Table 2. Specifically, we check whether the access to a shared image is protected by authentication and whether both users access the image through a common link, i.e., a link known to the attacker. A site that fulfills also this condition exposes its users to leaky image attacks.

4.2 Prevalence of Leaky Images in the Wild

Among the 30 studied websites, we identify a total of eight websites that suffer from leaky images. As shown in Table 3 (column "Leaky images"), the affected sites include the three most popular sites, Facebook, Twitter, and Google, and represent over 25% of all sites that we study. The following discusses each of the vulnerable sites in detail and explains how an attacker can establish a leaky image with a target user. Table 4 summarizes the discussion in a concise way.

[6] `https://moz.com/top500`

Table 3: List of analyzed websites, whether they suffer from leaky images, and how the respective security teams have reacted to our notifications about the privacy leak.

Rank	Domain	Leaky images	Confirmed	Fix	Bug bounty
1	**facebook.com**	**yes**	**yes**	**yes**	**yes**
2	**twitter.com**	**yes**	**yes**	**yes**	**yes**
3	**google.com**	**yes**	**yes**	**planned**	**no**
4	youtube.com	no			
5	instagram.com	no			
6	linkedin.com	no			
8	pinterest.com	no			
9	wikipedia.org	no			
10	**wordpress.com**	**yes**	**no**	**no**	**no**
15	tumblr.com	no			
18	vimeo.com	no			
19	flickr.com	no			
25	vk.com	no			
26	reddit.com	no			
33	blogger.com	no			
35	**github.com**	**yes**	**no**	**no**	**no**
39	myspace.com	no			
54	stumbleupon.com	no			
65	**dropbox.com**	**yes**	**yes**	**planned**	**yes**
71	msn.com	no			
72	slideshare.net	no			
91	typepad.com	no			
126	**live.com**	**yes**	**yes**	**planned**	**no**
152	spotify.com	no			
160	goodreads.com	no			
161	scribd.com	no			
163	imgur.com	no			
166	photobucket.com	no			
170	deviantart.com	no			
217	**skype.com**	**yes**	**yes**	**planned**	**no**

Facebook Images hosted on Facebook are in general delivered by content delivery networks not hosted at the facebook.com domain, but, e.g., at fbcdn.net. Hence, the fact that facebook.com cookie is not sent along with requests to shared images disables the leaky image attacks. However, we identified an exception to this rule, where a leaky image can be placed at `https://m.facebook.com/photo/view\_full\_size/?fbid=xxx`. The `fbid` is a unique identifier that is associated with each picture on Facebook, and it is easy to retrieve this identifier from the address bar of an image page. The attacker must gather this identifier and concatenate it with the leaky image URL given above. By tweaking the picture's privacy settings, the attacker can control the subset of friends that are authorized to access the image, opening the door for individual and group attacks. A prerequisite of creating a leaky image on Facebook is that the victim is a "friend" of the attacker.

Twitter Every image sent in a private chat on Twitter is a leaky image. The victim and the attacker can exchange messages on private chats, and hence send images, if one of them checked "Receive direct messages from anyone" in their settings or if one is a follower of the other. An image sent on a private chat can only be accessed by the two participants, based on their login state, i.e., these images are leaky images. The attacker can easily retrieve the leaky image URL from the conversation and include it in another page. A limitation of the attack via Twitter is that we are currently not aware of a way of sharing an image with multiple users at once.

Google We identified two leaky image channels on Google's domains: one in the thumbnails of Google Drive documents and one in Google Hangouts conversations. To share documents with the victim, an attacker only needs the email address of the victim, while in order to send Hangouts messages, the victim needs to accept the chat invitation from the attacker. The thumbnail-based attack is more powerful since it allows to easily add and remove users to the group of users that have access to an image. Moreover, by unselecting the "Notify people" option when sharing, the victim users are not even aware of this operation. An advantage of the Hangouts channel, though, is that the victim has no way to revoke its rights to the leaky image, once the image has been received in a chat, as opposed to Drive, where the victim can remove a shared document from her cloud.

Wordpress To create a leaky image via Wordpress, the attacker needs to convince the victim to become a reader of his blog, or the other way around. Once this connection is established, every image posted on the shared private blog is a leaky image between the two users. Fulfilling this strong prerequisite may require non-trivial social engineering.

GitHub Private repositories on GitHub enable leaky images. Once the victim and the attacker share such a repository, every committed image can be accessed through a link in the web interface, e.g., `https://github.com/johndoe/my-awesome-project/raw/master/car.jpg`. Only users logged into GitHub who were granted access to the repository *my-awesome-project* can access the image. To control the number of users that have access to the image, the attacker can remove or add contributors to the project.

Dropbox Every image uploaded on Dropbox can be accessed through a leaky image endpoint by appending the HTTP parameter `dl=1` to a shared image URL. Dropbox allows the attacker to share such images with arbitrary email addresses and to fine-tune the permissions to access the image by including and excluding users at any time. Once the image is shared, our attack can be successfully deployed,

Table 4: Leaky images in popular websites, the attack's preconditions, the image sharing channel and the implemented authentication mechanism as introduced in Table 2

Domain	Prerequisites	Image sharing channel	Authentication mechanism
facebook.com	Victim and attacker are "friends"	Image sharing	(5), (2)
twitter.com	Victim and attacker can exchange messages	Private message	(2)
google.com	*None*	Google Drive document Private message	(3), (2)
wordpress.com	Victim is a viewer of the attacker's private blog	Posts on private blogs	(2)
github.com	Victim and attacker share a private repository	Private repository	(3), (2)
dropbox.com	*None*	Image sharing	(3), (6), (2)
live.com	*None*	Shared folder on OneDrive	(3), (2)
skype.com	Victim and attacker can exchange messages	Private message	(2)

without requiring the victim to accept the shared image. However, the victim can revoke its rights to access an image by removing it from the "Sharing" section of her account.

Live.com Setting up a leaky image on One Drive, a cloud storage platform available on a live.com subdomain, is very similar to the other two file sharing services that we study, Google Drive and Dropbox. The attacker can share images with arbitrary email addresses and the victim does not need to acknowledge the sharing. Moreover, the attacker can easily deploy a group attack due to the ease in changing the group of users that have access to a particular image.

Skype In the Skype web interface, every image sent in a chat is a leaky image. Note that most of the users probably access the service through a desktop or mobile standalone client, hence the impact of this attack is limited to the web users. Moreover, Skype automatically logs out the user from time to time, limiting the time window for the attack.

Our study of leaky images in real-world sites enables several observations.

Leaky images are prevalent The first and perhaps most important observation is that many of the most popular websites allow an attacker to create leaky images. From an attacker's point of view, a single leaky image is sufficient to track a user. If a victim is registered as a user with at least one of the affected image sharing services, then the attacker can create a user account at that service and share a leaky image with the victim.

Victims may not notice sharing a leaky image Several of the affected image sharing services enable an attacker to share an image with a specific user without any notice given to the user. For example, if the attacker posts an image on her Facebook profile and tweaks the privacy settings so that only the victim can access it, then the victim is not informed in any way. Another example is Google Drive, which allows sharing files with arbitrary email addresses while instructing the website to not send an email that informs the other user.

Victims cannot "unshare" a leaky image For some services, the victim gets informed in some way that a connection to the attacker has been established. For example, to set up a leaky image on Twitter, the attacker needs to send a private message to the victim, which may make the victim suspicious. However, even if the victim knows about the shared image, for most websites, there is no way for a user to revoke its right to access the image. Specifically, let's assume the victim receives a cute cat picture from a Google Hangouts contact. Let us now assume that the victim is aware of the leaky image attack and that she suspects the sender of the image tracking her. We are not aware of any way in which the victim can revoke the right to access the received image.

Image sharing services use a diverse mix of implementation strategies Secret URLs and per-user authenticated URLs are widely implemented techniques that protects against our attack. However, many websites use multiple such strategies and hence, it is enough if one of the API endpoints uses leaky images. Identifying this endpoint is often a hard task: for example, in the case of Facebook, most of the website rigorously implements secret URLs, but one API endpoint belonging to a mobile subdomain exposes leaky images. After identifying this endpoint we realized that it can be accessed without any problem from a desktop browser as well, enabling all the attacks we describe in Section 3.

The attack surface varies from site to site Some but not all image sharing services require a trust relation between the attacker and the victim before a leaky image can be shared. For example, an attacker must first befriend a victim on Facebook before sharing an image with the victim, whereas no such requirement exists on Dropbox or Google Drive. However, considering that most users have hundreds of friends on social networks, there is a good chance that a trust channel is established before the attack starts. In the case of Wordpress the prerequisite that the "victim is a viewer of the attacker's private blog" appears harder to meet and may require advanced social engineering. Nonetheless, we believe that such leaky images may still be relevant in certain targeted attacks.

Moreover, three of the eight vulnerable sites allow attackers to share images with arbitrary users, without any prerequisites (Table 4).

Since our study of the prevalence of leaky images is mostly manual, we cannot guarantee that the 22 sites for which we could not create a leaky image are not affected by the problem. For some sites, though, we are confident that they are not affected, as these sites do not allow users to upload images. A more detailed analysis would require in-depth knowledge of the implementation of the studied sites, and ideally also access to the server-side source code. We hope that our results will spur future work on more automated analyses that identify leaky images.

4.3 Responsible Disclosure and Feedback from Image Sharing Services

After identifying image sharing services that suffer from leaky images, we contacted their security teams to disclose the problem in a responsible way. Between March 26 and March 29, 2018, we sent a detailed description of the general problem, how the specific website can be abused to create leaky images, and how it may affect the privacy of users of the site. Most security teams we contacted were very responsive and eager to collaborate upon fixing the issue.

Confirmed reports The last three columns of Table 3 summarize how the security teams of the contacted companies reacted to our reports. For most of the websites, the security teams confirmed that the reported vulnerability is worth fixing, and at least six of the sites have already fixed the problem or have decided to fix it. In particular, the top three websites all confirmed the reported issue and all have been working on fixing it. Given the huge user bases of these sites and the privacy implications of leaky images for their users, this reaction is perhaps unsurprising. As another sign of appreciation of our reports, the authors have received bug bounties from (so far) three of the eight affected sites.

Dismissed reports Two of our reports were flagged as false positives. The security teams of the corresponding websites replied by saying that leaky images are a "desired behavior" or that the impact on privacy of their user is limited. Comparing Table 3 with Table 4 shows that the sites that dismiss our report are those where the prerequisites for creating a leaky image are harder to fulfill than for the other sites: Creating a leaky image on GitHub requires the attacker and the victim to share a private repository, and Wordpress requires that the victim is a viewer of the attacker's private blog. While we agree that the attack surface is relatively small for these two sites, leaky images may nevertheless cause surprising privacy leaks. For example, an employee might track her colleagues or even her boss if their company uses private GitHub repositories.

Case study: Fix by Facebook To illustrate how image sharing services may fix a leaky images problem, we describe how Facebook addressed the problem in reaction to our report. As mentioned earlier, Facebook employs mostly secret URLs and uses content delivery networks to serve images. However, we were able to identify a mobile API endpoint that uses leaky images and redirects the user to the corresponding content delivery network link. This endpoint is used in the mobile user interface for enabling users to download the full resolution version of an image. The redirection was performed at HTTP level, hence it resulted in a successful image request when inserted in a third-party website using the `<a>` HTML tag. The fix deployed by Facebook was to perform a redirection at JavaScript level, i.e. load an intermediate HTML that contains a JavaScript snippet that rewrites `document.location.href`. This fix enables a benign user to still successfully download the full resolution image through a browser request, but disables third-party image inclusions. However, we believe that such a fix does not generalize and cannot be deployed to the other identified vulnerabilities. Hence, we describe alternative ways to protect against a leaky image attacks in Section 5.

Case study: Fix by Twitter A second case study of how websites can move away from leaky images comes from Twitter that changed its API[7] in response to our report[8]. First, they disabled cookie-based authentication for images. Second, they changed the API in a way that image URLs are only delivered on secure channels, i.e., only authenticated HTTPS requests. Last, Twitter also changed the user interface to only render images from strangers when explicit consent is given. Essentially, Twitter moved from implementation strategy (2) to (5) in Table 2 in response to our report.

Overall, we conclude from our experience of disclosing leaky images that popular websites consider it to be a serious privacy problem, and that they are interested in detecting and avoiding leaky images.

5 Mitigation Techniques

In this section, we describe several techniques to defend against leaky image attacks. The mitigations range from server-side fixes that websites can deploy, over improved privacy settings that empower users to control what is shared with them, to browser-based mitigations.

[7] `https://twitter.com/TwitterAPI/status/1039631353141112832`

[8] `https://hackerone.com/reports/329957`

5.1 Server-Side Mitigations

The perhaps easiest way to defend against the attack presented in this paper is to modify the server-side implementation of an image sharing service, so that it is not possible anymore to create leaky images. There are multiple courses of actions to approach this issue.

First, a controversial fix to the problem is to disable authenticated image requests altogether. Instead of relying on, e.g., cookies to control who can access an image, an image sharing service could deliver secret links only to those users that should access an image. Once a user knows the link she can freely get the image through the link, independent of whether she is logged into the image sharing service or not. This strategy corresponds to case 5 in Table 2. Multiple websites we report about in Table 3 implement such an image sharing strategy. The most notable examples are Facebook, which employs this technique in most parts of their website, and Dropbox, which implements this technique as part of their link sharing functionality. The drawback of this fix is that the link's secrecy might be compromised in several ways outside of the control of the image sharing service: by using insecure channels, such as HTTP, through side-channel attacks in the browser, such as cache attacks [20], or simply by having the users handle the links in an insecure way because they are not aware of the secrecy requirements.

Second, an alternative fix is to enforce an even stricter cookie-based access control on the server-side. In this case, the image sharing service enforces that each user accesses a shared image through a secret, user-specific link that is not shared between users. As a result, the attacker does not know which link the victim could use to access a shared image, and therefore the attacker cannot embed such a link in any website. This implementation strategy corresponds to case 3 in Table 2. On the downside, implementing this defense may prove challenging due to the additional requirement of guaranteeing the mapping between users and URLs, especially when content delivery networks are involved. Additionally, it may cause a slowdown for each image request due to the added access control mechanism.

Third, one may deploy mitigations against CSRF.[9] One of them is to use the `origin` HTTP header to ensure that the given image can only be embedded on specific websites. The `origin` HTTP header is sent automatically by the browser with every request, and it precisely identifies the page that requests a resource. The server-side can check the request's `origin` and refuse to respond with an authenticated image to unknown third-party request. For example, facebook.com could refuse to respond with a valid image to an HTTP request with the `origin` set to evil.com. However, this mitigation cannot defend against tracking code injected into a trusted domain. For example, until recently Facebook allowed users to post custom HTML code on their profile page. If a user decides to insert leaky image-based tracking code on the profile page, to be notified when a target user visits the profile page, then the CSRF-style mitigation does not prevent the attack. The reason for this is that the request's origin would be set to facebook.com, and hence the server-side code will trust the page and serve the image.

Similarly, the server-side can set the `Cross-Origin-Resource-Policy` response header on authenticated image requests and thus limit which websites can include a specific image. Browsers will only render images for which the origin of the request matches the origin of the embedding website or if they correspond to the same site. This solution is more coarse-grained than the previously discussed `origin` checking since it does not allow for cross-origin integration of authenticated images, but it is easier to deploy since it only requires a header set instead of a header check. The `From-Origin` header was proposed for allowing a more fine-grained integration policy, but to this date there is no interest from browser vendors side to implement such a feature.

Another applicable CSRF mitigation is the `SameSite` cookie attribute. When set to "strict" for a cookie, the attribute prevents the browser from sending the cookie along with cross-site requests, which effectively prevents leaky images. However, the "strict" setting may be too strict for most image sharing services, because it affects all links to the service's website. For example, a link in a corporate email to a private GitHub project or to a private Google Doc would not work anymore, because when clicking the link, the session cookie is not sent along with the request. The less restrictive "lax" setting of the `SameSite` attribute does not suffer from these problems, but it also does not prevent leaky images attacks, as it does not affect GET requests.

A challenge with all the described server-side defenses is that they require the developers to be aware of the vulnerability in the first place. From our experience, a complex website may allow sharing images in several ways, possibly spanning different UI-level user interactions and different API endpoints supported by the image sharing service. Since rigorously inspecting all possible ways to share an image is non-trivial, we see a need for future work on automatically identifying leaky images. At least parts of the methodology we propose could be automated with limited effort. To check whether an image request requires authentication, one can perform the request in one browser where the user is logged in, and then try the same request in another instance of the browser in "private" or "incognito" mode, i.e., without being logged in. Comparing the success of the two requests reveals whether the image request relies in any form of authentication, such as cookies. Automating the rest of our methodology requires some support by image sharing services. In particular, automatically checking that a leaky image is accessible only by a subset of a website's users, requires APIs

[9] `https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet`

to handle user accounts and to share images between users.

Despite the challenges in identifying leaky images, we believe that server-side mitigations are the most straightforward solution, at least in the short term. In the long term, a more complete solution would be desirable, such as those described in the following.

5.2 Browser Mitigations

The current HTTP standard does not specify a policy for third-party cookies[10], but it encourages browser vendors to experiment with different such policies. More precisely, the current standard lets the browser decide whether to automatically attach the user's cookie to third-party requests. Most browsers decide to attach third-party cookies, but there are certain counter-examples, such as the Tor browser. In Tor, cookies are sent only to the domain typed by the user in the address bar.

Considering the possible privacy implications of leaky images and other previously reported tracking techniques [8], one possible mitigation would be that browsers specify as default behavior to not send cookies with third-party (image) requests. If this behavior is overwritten, possibly using a special HTTP header or tag, the user should be informed through a transparent mechanism. Moreover, the user should be offered the possibility to prevent the website from overwriting the default behavior. We believe this measure would be in the spirit of the newly adopted European Union's General Data Protection Regulation which requires *data protection by design and by default*. However, such an extreme move may impact certain players in the web ecosystem, such as the advertisement ecosystem. To address this issue, advertisers may decide to move towards safer distribution mechanisms, such as the one popularized by the Brave browser.

An alternative to the previously discussed policy is to allow authenticated image requests, but only render them if the browser is confident that there are no observable differences between an authenticated request and a non-authenticated one. To this end, the browser could perform two image requests instead of one: one request with third-party cookies and one request without. If the browser receives two equivalent responses, it can safely render the content, since no sensitive information is leaked about the authenticated user. This solution would still allow most of the usages of third-party cookies, e.g. tracking pixels, but prevent the leaky image attack described here. A possible downside might be the false positives due to strategy (3) in Table 2, but we hypothesize that requests to such images rarely appear in benign third-party image requests. A second possible drawback of this solution may be the increase in web traffic and the potential performance penalties. Future work should test the benefits of this defense and the cost imposed by the additional image request.

[10] `https://tools.ietf.org/html/rfc6265#page-28`

To reduce the cost imposed by an additional image request, a hybrid mechanism could disable authenticated image requests by default, and allow them only for the resources specified by a CSP directive. For the allowed authenticated images, the browser deploys the double image requests mechanism described earlier. We advocate this as our preferred browser-level defense since it can also defend against other privacy attacks, e.g. reading third-party image pixels through a side channel [21], while still permitting benign uses.

Similarly to ShareMeNot [32], one can also implement a browser mechanism in which all third-party image requests are blocked unless the user authorizes them by providing explicit consent. To release the burden from the user, a hybrid mechanism can be deployed in which the website requires authenticated requests only for a subset of images for which the user needs to provide consent.

Another solution for when third-party cookies are allowed is for browsers to implement some form of information flow control to ensure that the fact whether a third-party request was successfully loaded or not, cannot be sent outside of the browser. A similar approach is deployed in *tainted canvas*[11], which disallows pixel reads after a third-party image is painted on the canvas. Implementing such an information flow control for third-party images may, however, be challenging in practice, since the fact whether an image has successfully loaded or not can be retrieved through multiple side channels, such as the `object` tag or by reading the size of the contained div.

The mechanisms described in this section vary both in terms of implementation effort required for deploying them and in terms of their possible impact on the existing state of the web, i.e., incompatibility with existing websites. Therefore, to aid the browser vendors to take an informed decision, future work should perform an in-depth analysis of all these defenses in terms of usability, compatibility and deployment cost, in the style of Calzavara et al. [9], and possibly propose additional solutions.

5.3 Better Privacy Control for Users

A worrisome finding of our prevalence study is that a user has little control over the image sharing process. For example, for some image sharing services, the user does not have any option to restrict which other users can privately share an image with her. In others, there is no way for a user to revoke her right to access a specific image. Moreover, in most of the websites we analyzed, it is difficult to even obtain a complete list of images privately shared with the current account. For example, a motivated user who wants to obtain this list must check all the conversations in a messaging platform, or all the images of all friends on a social network.

[11] `https://html.spec.whatwg.org/multipage/canvas.html#security-with-canvas-elements`

We believe that image sharing services should provide users more control over image sharing policies, to enable privacy-aware users to protect their privacy. Specifically, a user should be allowed to decide who has the right to share an image with her and she should be granted the right to revoke her access to a given image. Ideally, websites would also offer the user a list of all the images shared with her and a transparent notification mechanism that announces the user when certain changes are made to this list. Empowering the users with these tools may help mitigate some of the leaky image attacks by attracting user's attention to suspicious image sharing, allowing users to revoke access to leaky images.

The privacy controls for web users presented in this section will be useful mostly for advanced users, while the majority of the users are unlikely to take advantage of such fine-grained controls. Therefore, we believe that the most effective mitigations against leaky images are at the server side or browser level.

6 Related Work

Previous work shows risks associated with images on the web, such as malicious JavaScript code embedded in SVGs [17], image-based fingerprinting of browser extensions [35], and leaking sensitive information, such as the gender or the location of a user uploading an image [11]. This work introduces a new risk: privacy leaks due to shared images. Lekies et al. [24] describe privacy leaks resulting from dynamically generated JavaScript. The source of this problem is the same as for leaky images: both JavaScript code and images are excepted from the same-origin policy. While privacy leaks in dynamic JavaScript reveal confidential information about the user, such as credentials, leaky images allow for tracking specific users on third-party websites. Heiderich et al. [18] introduce a scriptless, CSS-based web attacks. The HTML-only variant of leaky images does not rely on CSS and also differ in the kinds of leaked information: While the attack by Heiderich et al. leaks content of the current website, our attacks leak the identity of the user.

Wondracek et al. [46] present a privacy leak in social networks related to our group attack. In their work, the attacker neither has control over the group structure nor can she easily track individuals. A more recent attack [41] deanonymizes social media users by correlating links on their profiles with browsing histories. In contrast, our attack does not require such histories. Another recent attack [44] retrieves sensitive information of social media accounts using the advertisement API provided by a social network. However, their attack cannot be used to track users on third-party websites.

Cross-Site Request Forgery (CSRF) is similar in spirit to leaky image attacks: both rely on the fact that browsers send cookies with third-party requests. For CSRF, this behavior results in an unauthorized action on a third-party website, whereas for leaky images, it results in deanonymizing the user. Existing techniques for defending [5] and detecting [31] CSRF partially address but do not fully solve the problem of leaky images (Section 5).

Browser fingerprinting is a widely deployed [1, 2, 30] persistent tracking mechanism. Various APIs have been proposed for fingerprinting: user agent and fonts [12], canvas [29, 10], ad blocker usage, and WebGL Renderer [22]. Empirical studies [12, 22] suggest that these technique have enough entropy to identify most of the users, or at least, to place a user in a small set of possible users, sometimes even across browsers [10]. The leaky image attack is complementary to fingerprinting, as discussed in detail in Section 3.6.

Another web tracking mechanism is through third-party requests, such as tracking pixels. Mayer and Mitchell [27] describe the tracking ecosystem and the privacy costs associated with these practices. Lerner et al. [25] show how tracking in popular websites evolves over time. Several other studies [42, 47, 13, 32, 8, 14] present a snapshot of the third-party tracking on the web at various moments in time. One of the recurring conclusion of these studies was that few big players can track most of the traffic on the Internet. We present the first image-based attack that allows a less powerful attacker to deanonymize visitors of a website.

Targeted attacks or advanced persistent threats are an increasingly popular class of cybersecurity incidents [37, 26]. Known attacks include spear phishing attacks [6] and targeted malware campaigns [26, 16]. Leaky images adds a privacy-related attack to the set of existing targeted attacks.

Several empirical studies analyze different security and privacy aspects of websites in production: postMessages [36], cookie stealing [4, 34], credentials theft [3], cross-site scripting [28, 23], browser fingerprinting [30, 1], deployment of CSP policies [45], and ReDoS vulnerabilities [38]. User privacy can also be impacted by security issues in browsers, such as JavaScript bindings bugs [7], micro-architectural bugs [20], and insufficient isolation of web content [19]. Neither of these studies explores privacy leaks caused by authenticated cross-origin image requests.

Van Goethem et al. [43] propose the use of timing channels for estimating the file size of a cross-origin resource. One could combine leaky images with such a channel to check if a privately shared image is accessible for a particular user, enabling the use of leaky images even if the browser would block cross-origin image requests. One difference between our attack and theirs is that leaky images provide 100% certainty that a victim has visited a website, which a a probabilistic timing channel cannot provide.

Several researchers document the difficulty of notifying the maintainers of websites or open-source projects about security bugs in software [24, 40, 39]. We experienced quick and helpful responses by all websites we contacted, with an initial response within less than a week. One reason for this difference may be that we used the bug bounty channels provided by the websites to report the problems [15, 48].

7 Conclusions

This paper presents leaky images, a targeted deanonymization attack that leverages specific access control practices employed in popular websites. The main insight of the attack is a simple yet effective observation: Privately shared resources that are exempted from the same origin policy can be exploited to reveal whether a specific user is visiting an attacker-controlled website. We describe several flavors of this attack: targeted tracking of single users, group tracking, pseudonym linking, and an HTML-only attack.

We show that some of the most popular websites suffer from leaky images, and that the problem often affects any registered users of these websites. We reported all the identified vulnerabilities to the security teams of the affected websites. Most of them acknowledge the problem and some already proceeded to fixing it. This feedback shows that the problem we identified is important to practitioners. Our paper helps raising awareness among developers and researchers to avoid this privacy issue in the future.

Acknowledgments
Thanks to Stefano Calzavara and the anonymous reviewers for their feedback on this paper. This work was supported by the German Federal Ministry of Education and Research and by the Hessian Ministry of Science and the Arts within CRISP, by the German Research Foundation within the ConcSys and Perf4JS projects, and by the Hessian LOEWE initiative within the Software-Factory 4.0 project.

References

[1] G. Acar, C. Eubank, S. Englehardt, M. Juárez, A. Narayanan, and C. Díaz, "The web never forgets: Persistent tracking mechanisms in the wild," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, 2014, pp. 674–689. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660347

[2] G. Acar, M. Juárez, N. Nikiforakis, C. Díaz, S. F. Gürses, F. Piessens, and B. Preneel, "Fpdetective: dusting the web for fingerprinters," in *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, 2013, pp. 1129–1140. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516674

[3] S. V. Acker, D. Hausknecht, and A. Sabelfeld, "Measuring login webpage security," in *Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017*, 2017, pp. 1753–1760. [Online]. Available: http://doi.acm.org/10.1145/3019612.3019798

[4] D. J. andZhaoGL15 Ranjit Jhala, S. Lerner, and H. Shacham, "An empirical study of privacy-violating information flows in javascript web applications," in *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, 2010, pp. 270–283. [Online]. Available: http://doi.acm.org/10.1145/1866307.1866339

[5] A. Barth, C. Jackson, and J. C. Mitchell, "Robust defenses for cross-site request forgery," in *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, 2008, pp. 75–88. [Online]. Available: http://doi.acm.org/10.1145/1455770.1455782

[6] S. L. Blond, A. Uritesc, C. Gilbert, Z. L. Chua, P. Saxena, and E. Kirda, "A look at targeted attacks through the lense of an NGO," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, 2014, pp. 543–558. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/le-blond

[7] F. Brown, S. Narayan, R. S. Wahby, D. R. Engler, R. Jhala, and D. Stefan, "Finding and preventing bugs in javascript bindings," in *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, 2017, pp. 559–578. [Online]. Available: https://doi.org/10.1109/SP.2017.68

[8] A. Cahn, S. Alfeld, P. Barford, and S. Muthukrishnan, "An empirical study of web cookies," in *Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, April 11 - 15, 2016*, 2016, pp. 891–901. [Online]. Available: http://doi.acm.org/10.1145/2872427.2882991

[9] S. Calzavara, R. Focardi, M. Squarcina, and M. Tempesta, "Surviving the web: A journey into web session security," *ACM Comput. Surv.*, vol. 50, no. 1, pp. 13:1–13:34, 2017. [Online]. Available: https://doi.org/10.1145/3038923

[10] Y. Cao, S. Li, and E. Wijmans, "(cross-)browser fingerprinting via OS and hardware level features," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017. [Online]. Available: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/cross-browser-fingerprinting-os-and-hardware-level-features/

[11] M. Cheung and J. She, "Evaluating the privacy risk of user-shared images," *ACM Transactions on Multi-*

media Computing, Communications, and Applications (TOMM), vol. 12, no. 4s, p. 58, 2016.

[12] P. Eckersley, "How unique is your web browser?" in *Privacy Enhancing Technologies, 10th International Symposium, PETS 2010, Berlin, Germany, July 21-23, 2010. Proceedings*, 2010, pp. 1–18. [Online]. Available: https://doi.org/10.1007/978-3-642-14527-8_1

[13] S. Englehardt and A. Narayanan, "Online tracking: A 1-million-site measurement and analysis," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, 2016, pp. 1388–1401. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978313

[14] S. Englehardt, D. Reisman, C. Eubank, P. Zimmerman, J. Mayer, A. Narayanan, and E. W. Felten, "Cookies that give you away: The surveillance implications of web tracking," in *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*, 2015, pp. 289–299. [Online]. Available: http://doi.acm.org/10.1145/2736277.2741679

[15] M. Finifter, D. Akhawe, and D. A. Wagner, "An empirical study of vulnerability rewards programs," in *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, 2013, pp. 273–288. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/finifter

[16] S. Hardy, M. Crete-Nishihata, K. Kleemola, A. Senft, B. Sonne, G. Wiseman, P. Gill, and R. J. Deibert, "Targeted threat index: Characterizing and quantifying politically-motivated targeted malware," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, 2014, pp. 527–541. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/hardy

[17] M. Heiderich, T. Frosch, M. Jensen, and T. Holz, "Crouching tiger - hidden payload: security risks of scalable vectors graphics," in *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, 2011, pp. 239–250. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046735

[18] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk, "Scriptless attacks: Stealing more pie without touching the sill," *Journal of Computer Security*, vol. 22, no. 4, pp. 567–599, 2014. [Online]. Available: https://doi.org/10.3233/JCS-130494

[19] Y. Jia, Z. L. Chua, H. Hu, S. Chen, P. Saxena, and Z. Liang, ""the web/local" boundary is fuzzy: A security study of chrome's process-based sandboxing," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, 2016, pp. 791–804. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978414

[20] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *arXiv preprint arXiv:1801.01203*, 2018.

[21] R. Kotcher, Y. Pei, P. Jumde, and C. Jackson, "Cross-origin pixel stealing: timing attacks using CSS filters," in *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, A. Sadeghi, V. D. Gligor, and M. Yung, Eds. ACM, 2013, pp. 1055–1062. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516712

[22] P. Laperdrix, W. Rudametkin, and B. Baudry, "Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints," in *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, 2016, pp. 878–894. [Online]. Available: https://doi.org/10.1109/SP.2016.57

[23] S. Lekies, B. Stock, and M. Johns, "25 million flows later: large-scale detection of dom-based XSS," in *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, 2013, pp. 1193–1204. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516703

[24] S. Lekies, B. Stock, M. Wentzel, and M. Johns, "The unexpected dangers of dynamic javascript," in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, 2015, pp. 723–735. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/lekies

[25] A. Lerner, A. K. Simpson, T. Kohno, and F. Roesner, "Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, 2016. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lerner

[26] W. R. Marczak, J. Scott-Railton, M. Marquis-Boire, and V. Paxson, "When governments hack opponents: A look at actors and technology,"

in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, 2014, pp. 511–525. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/marczak

[27] J. R. Mayer and J. C. Mitchell, "Third-party web tracking: Policy and technology," in *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, 2012, pp. 413–427. [Online]. Available: https://doi.org/10.1109/SP.2012.47

[28] W. Melicher, A. Das, M. Sharif, L. Bauer, and L. Jia, "Riding out domsday: Toward detecting and preventing dom cross-site scripting," 2018.

[29] K. Mowery and H. Shacham, "Pixel perfect: Fingerprinting canvas in html5," *Proceedings of W2SP*, pp. 1–12, 2012.

[30] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "Cookieless monster: Exploring the ecosystem of web-based device fingerprinting," in *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, 2013, pp. 541–555. [Online]. Available: https://doi.org/10.1109/SP.2013.43

[31] G. Pellegrino, M. Johns, S. Koch, M. Backes, and C. Rossow, "Deemon: Detecting CSRF with dynamic analysis and property graphs," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, 2017, pp. 1757–1771. [Online]. Available: http://doi.acm.org/10.1145/3133956.3133959

[32] F. Roesner, T. Kohno, and D. Wetherall, "Detecting and defending against third-party tracking on the web," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, 2012, pp. 155–168. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/roesner

[33] C. Shiflett, "Cross-site request forgeries," http://shiflett.org/articles/cross-site-request-forgeries.

[34] S. Sivakorn, I. Polakis, and A. D. Keromytis, "The cracked cookie jar: HTTP cookie hijacking and the exposure of private information," in *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, 2016, pp. 724–742. [Online]. Available: https://doi.org/10.1109/SP.2016.49

[35] A. Sjösten, S. V. Acker, and A. Sabelfeld, "Discovering browser extensions via web accessible resources," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, CODASPY 2017, Scottsdale, AZ, USA, March 22-24, 2017*, 2017, pp. 329–336. [Online]. Available: http://doi.acm.org/10.1145/3029806.3029820

[36] S. Son and V. Shmatikov, "The postman always rings twice: Attacking and defending postmessage in HTML5 websites," in *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*, 2013. [Online]. Available: https://www.cs.utexas.edu/~shmat/shmat_ndss13postman.pdf

[37] A. K. Sood and R. J. Enbody, "Targeted cyberattacks: A superset of advanced persistent threats," *IEEE Security & Privacy*, vol. 11, no. 1, pp. 54–61, 2013. [Online]. Available: https://doi.org/10.1109/MSP.2012.90

[38] C. Staicu and M. Pradel, "Freezing the web: A study of ReDoS vulnerabilities in JavaScript-based web servers," in *USENIX Security Symposium*, 2018, pp. 361–376.

[39] C.-A. Staicu, M. Pradel, and B. Livshits, "Understanding and automatically preventing injection attacks on Node.js," in *25th Annual Network and Distributed System Security Symposium, NDSS*, 2018.

[40] B. Stock, G. Pellegrino, C. Rossow, M. Johns, and M. Backes, "Hey, you have a problem: On the feasibility of large-scale web vulnerability notification," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, 2016, pp. 1015–1032. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/stock

[41] J. Su, A. Shukla, S. Goel, and A. Narayanan, "De-anonymizing web browsing data with social networks," in *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*, 2017, pp. 1261–1269. [Online]. Available: http://doi.acm.org/10.1145/3038912.3052714

[42] M. Tran, X. Dong, Z. Liang, and X. Jiang, "Tracking the trackers: Fast and scalable dynamic analysis of web content for privacy violations," in *Applied Cryptography and Network Security - 10th International Conference, ACNS 2012, Singapore, June 26-29, 2012. Proceedings*, 2012, pp. 418–435. [Online]. Available: https://doi.org/10.1007/978-3-642-31284-7_25

[43] T. van Goethem, W. Joosen, and N. Nikiforakis, "The clock is still ticking: Timing attacks in the modern web," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, 2015, pp. 1382–1393. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813632

[44] G. Venkatadri, A. Andreou, Y. Liu, A. Mislove, K. P. Gummadi, P. Loiseau, and O. Goga, "Privacy risks with Facebook's pii-based targeting: Auditing a data Broker's advertising interface," 2018.

[45] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc, "CSP is dead, long live csp! on the insecurity of whitelists and the future of content security policy," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, 2016, pp. 1376–1387. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978363

[46] G. Wondracek, T. Holz, E. Kirda, and C. Kruegel, "A practical attack to de-anonymize social network users," in *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, 2010, pp. 223–238. [Online]. Available: https://doi.org/10.1109/SP.2010.21

[47] Z. Yu, S. Macbeth, K. Modi, and J. M. Pujol, "Tracking the trackers," in *Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, April 11 - 15, 2016*, 2016, pp. 121–132. [Online]. Available: http://doi.acm.org/10.1145/2872427.2883028

[48] M. Zhao, J. Grossklags, and P. Liu, "An empirical study of web vulnerability discovery ecosystems," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, 2015, pp. 1105–1117. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813704

All Your Clicks Belong to Me: Investigating Click Interception on the Web

Mingxue Zhang
Chinese University of Hong Kong

Wei Meng
Chinese University of Hong Kong

Sangho Lee
Microsoft Research

Byoungyoung Lee
Seoul National University
Purdue University

Xinyu Xing
Pennsylvania State University

Abstract

Click is the prominent way that users interact with web applications. For example, we click hyperlinks to navigate among different pages on the Web, click form submission buttons to send data to websites, and click player controls to tune video playback. Clicks are also critical in online advertising, which fuels the revenue of billions of websites. Because of the critical role of clicks in the Web ecosystem, attackers aim to intercept genuine user clicks to either send malicious commands to another application on behalf of the user or fabricate realistic ad click traffic. However, existing studies mainly consider one type of click interceptions in the cross-origin settings via iframes, *i.e.*, clickjacking. This does not comprehensively represent various types of click interceptions that can be launched by malicious third-party JavaScript code.

In this paper, we therefore systematically investigate the click interception practices on the Web. We developed a browser-based analysis framework, OBSERVER, to collect and analyze click related behaviors. Using OBSERVER, we identified three different techniques to intercept user clicks on the Alexa top 250K websites, and detected 437 third-party scripts that intercepted user clicks on 613 websites, which in total receive around 43 million visits on a daily basis.

We revealed that some websites collude with third-party scripts to hijack user clicks for monetization. In particular, our analysis demonstrated that more than 36% of the 3,251 unique click interception URLs were related to online advertising, which is the primary monetization approach on the Web. Further, we discovered that users can be exposed to malicious contents such as scamware through click interceptions. Our research demonstrated that click interception has become an emerging threat to web users.

1 Introduction

Clicking an HTML element is the primary way that users interact with web applications. We click hyperlinks to navigate among different documents that are interconnected through the hyperlinks on the Web. We click form submission buttons (*e.g.*, the Facebook like button and the Twitter tweet button) to share data with websites and other people on the Internet. We click custom user interface components (*e.g.*, the video or audio player controls) to command various web applications.

Since clicks are important in modern web applications, attackers have launched UI redressing attacks, namely Clickjacking [26], to hijack user clicks. In particular, malicious websites trick a user into clicking components (*e.g.*, a Facebook like button) different from what the user perceives to click, in order to send commands on behalf of the user to the different application they secretly embed (typically in an iframe tag). To defend against Clickjacking, a rich collection of works has been proposed, which has shown great performance [1, 3, 10, 15, 29, 30].

Clicks are also critical in one pervasive application—online display advertising, which powers billions of websites on the Internet. The publisher websites earn a commission when a user clicks an advertisement they embed from an online advertising network (ad network in short). However, the ad click-through rate is usually very low, *e.g.*, around 2% in business-to-consumer banner ads [18]. To increase revenue that can be made through ad clicks, malicious websites have used bots to automatically and massively send fake click traffic to the ad networks, which is known as ad click fraud [5, 22, 27]. To combat against click frauds, ad networks have developed advanced techniques to determine the authenticity of click traffic [2, 6, 9, 38]. Consequently, traditional bot-based ad click fraud has then become less effective.

Instead of relying on click bots, attackers recently started to *intercept and redirect* clicks or page visits from real users to fabricate realistic ad clicks. First, they infect a victim user's computer with malware to either force or trick a user into submitting an ad click. For example, some "browser redirect viruses" modify a user's default search engine to a malicious one, redirecting the user to an advertiser's page when the user clicks a search result [19]. Second, malicious third-party iframes can automatically redirect users to an ad page. Similarly, a user's current tab may be automatically redirected

to unintended destinations when a script opens a new tab upon click. Google recently released a new version of the Chrome browser to automatically prevent these two types of automatic redirects [8]. Nevertheless, Chrome still cannot detect and prevent other possible ways to intercept user clicks, including but not limited to links modified by third-party scripts, third-party contents disguised as first-party contents, and transparent overlays.

A systematic study on click interceptions is necessary to deeply understand this emerging threat to web users. We aim to develop a system to automatically detect such practices on the Web, and investigate what kinds of techniques are exploited and who are involved in. We first design and develop a system to detect various techniques employed by JavaScript to intercept user clicks. Using this system, we then perform a large-scale measurement with the goal of finding out those practitioners that hijack links and deceive user clicks. Finally, we analyze our measurement results, and explore the intents and consequences hidden behind the click interception practices.

However, it is challenging to perform the aforementioned systematic study because of the dynamic and event-driven characteristics of web applications. First, JavaScript code can be dynamically loaded. Statically analyzing the HTML source code is insufficient to cover all scripts that can intercept user clicks. Second, hyperlinks can be dynamically created and modified by any scripts. To pinpoint the scripts truly accountable for the interception, we need to re-engineer a browser to differentiate the actions of different scripts in runtime. Third, JavaScript can dynamically bind a URL to user click on an arbitrary HTML element through event listeners (handlers). Monitoring hyperlink creation and modification is insufficient to catch all the click interception practices. Last but not least, a web page may contain a large number of event handlers that respond to user clicks. To perform a large-scale comprehensive study, we have to efficiently interact with all those event handlers.

To tackle the challenges mentioned above, we design our analysis framework by customizing an open-sourced Web browser. We first mediate all JavaScript accesses to hyperlinks in a web page in the browser's renderer. In this way, we can identify the *initiator* of the URL associated with each hyperlink. Second, we monitor the creation and execution of JavaScript objects so that we can track down the *provenance* of dynamic inline JavaScript code. Third, we monitor all event handlers registered on every HTML element and hook navigation-related JavaScript APIs. With this design, we can develop an automated approach to monitor the event handlers accordingly, and determine if an event handler might be used to hijack user clicks. Last but not least, we derive the navigation URL without really firing the navigation that is initiated by a user click. This allows us to interact with all the click event handlers in an efficient way. It also helps us understand the reason why a particular user click is of the interest of a script.

In this work, we developed OBSERVER, a prototype of the aforementioned analysis framework by customizing and extending the Chromium browser. Using this framework, we performed a large-scale data crawling on the Alexa top 250K websites. We discovered that 437 third-party scripts exhibited the activities of intercepting user clicks on 613 websites. They combined receive 43 million visits on a daily basis. In particular, we observed that some scripts tricked users into clicking their carefully crafted contents, which were usually disguised as first-party contents, or intentionally implemented as barely visible elements covering first-party elements. In addition, we revealed that these third-party scripts intercepted user clicks in order to monetize user clicks, which is a new practice we observe as committing ad click frauds. It is worth noting that we will make our implementation publicly available.

In summary, this paper makes the following contributions.

- We design and develop OBSERVER, a framework for studying click interception practices. This facilitates our capability in automatically detecting a wide range of click interception cases on various websites.
- We perform a large-scale measurement study to explore and understand how attackers manipulate web pages in the wild and thus intercept user clicks.
- We characterize the activities of click interceptions on top Alexa websites and discover the intents and consequences hidden behind the activities of click interception.

2 Related Work

In this section, we introduce existing studies about how attackers intercept user clicks or generate fake clicks, and how to detect and prevent such attempts. We also explain other studies analyzing how JavaScript libraries are included and what their behaviors are.

Clickjacking. Clickjacking, also known as UI redressing, is a popular attack designed to trick a victim into doing some tasks on another website the user has logged in, bypassing the same-origin policy. It is one type of *inter-page* click interception in which a malicious *first-party website* tricks a victim into clicking components in another website loaded in an *iframe*. For example, a malicious website could load a specific page of a target website via an invisible iframe, and place it on top of a crafted object that looks benign and independent to the target page. The malicious website then can trick a victim into unintentionally clicking the target page via the crafted object to activate some operations defined in that page. Framebusting [29–31] is a well-known defense to prevent clickjacking by disallowing untrusted websites to load specified pages via an iframe. However, framebusting is incompatible with third-party mashup or other techniques that demand cross-origin framing [15]. Rather, other studies

including ClickIDS [3] and InContext [10] rely on human perception to verify whether a click was intended by a user. Akhawe *et al.* [1], however, identified that such mechanisms are not comprehensive or suffer from an unacceptable usability cost.

Our research complements these studies by investigating new practices of *intra-page* click interception by *third-party scripts*, which intercept a victim's clicks on components (including iframes) within the same page/frame. Further, we demonstrate that the scripts can use hyperlinks, event listeners, and visual deceptions, to intercept user clicks.

Link Hijacking. Link hijacking is an attack to modify the destination of links on websites. Nikiforakis *et al.* [24] investigated ad-based URL shortening services and discovered link hijacking by an embedded third-party iframe on a "waiting page" through automatic tab redirects, which the new Chrome browser can prevent [8]. Our research demonstrates a new form of link hijacking that modifies all first-party hyperlinks before the user even clicks them, and shows our system can automatically detect them.

Visual Deception. Prior works have studied how visual deceptive contents can be used to intercept user clicks. Duman *et al.* [7] studied trick banners (*e.g.*, download buttons) that look similar to first-party contents, and further proposed a defense based on a supervised classifier. Rafique *et al.* [28] discovered overlay ads and invisible banners in free live-streaming services. Note that our research does not focus on a specific category of visual deceptive contents or services. Moreover, OBSERVER is able to distinguish deceptive contents created by *different scripts* because of its provenance tracking capability, allowing us to detect the real culprits.

Click Fraud and Click Spam. Click fraud and click spam are attempts to raise revenue by submitting fake ad clicks to an ad network. In traditional click fraud, attackers usually operate a botnet to fabricate a large number of ad clicks automatically to an ad network. For example, Pearce *et al.* [27] estimated that the ZeroAccess click-fraud botnet incurred advertising losses on the order of $100,000 per day. In click spam, unethical content publishers or ad injection attackers [32, 37] either trick the users into clicking ads, or use malware to click ads on behalf of the users. Click spams could even lead victim users to malicious ads [16, 37, 39]. Defenses against click fraud and click spam mostly aim to distinguish fake clicks from real clicks by analyzing their patterns [5, 6, 12, 17, 21, 22, 38]. Thus, attackers try to make their click traffic look as benign as possible. For example, some attacks hijack real human clicks through rogue DNS servers and redirect them to ad networks [2]. We discover that the click interception techniques we identify have already been used by attackers for generating realistic click traffic in the wild.

JavaScript Inclusion and Behavior Analysis. Numerous researchers have analyzed the behavior of third-party JavaScript libraries and how they are included. Nikiforakis *et al.* [23] investigated the Alexa Top 10K websites to discover how many remote JavaScript libraries they include and from which library hosting servers they include the scripts. They also assessed the security of those hosting servers to infer whether they could serve malicious JavaScript code. Lauinger *et al.* [14] and Retire.js [25] studied the semantics of JavaScript libraries, by considering whether hosted JavaScript libraries are outdated or have known vulnerabilities. Systems like EvilSeed [11] and Revolver [13] focus on detecting malicious web pages using content or code similarities. Also, ScriptInspector [40] inspects API calls from third-party scripts to study how they interact with critical resources, such as the DOM, local storage and network. It is able to detect suspicious third-party scripts that violate some access policies. These studies, however, rely on the origin of a JavaScript script to determine whether it is a first-party or third-party script. This implies that they cannot properly handle the situation where a website includes JavaScript libraries from their subdomains or other domains, and from other CDNs (§4.2). Furthermore, unlike ScriptInspector, OBSERVER can track the dynamic creation of JavaScript objects and DOM elements such that it can accurately attribute hyperlink modifications and event listener registrations.

3 Overview of OBSERVER

In this section, we present OBSERVER, an analysis framework that is designed to comprehensively log all potential click-interception-related events performed by JavaScript code in a best-effort manner. OBSERVER focuses on three fundamental actions that JavaScript code might rely on to intercept clicks: 1) *modifying* an existing hyperlink in a page; 2) *creating* a new hyperlink in a page; and 3) *registering* an event handler to an HTML element to hook a user click. Whenever OBSERVER identifies any of such actions, it *tags* the corresponding element with the unique identifier of the script that *initiates* the action. Further, OBSERVER logs the reaction (*i.e.*, *navigation*) of a page after it *intentionally* clicks a hyperlink or an element associated with an event handler in the page, to know the URLs to which a click interceptor aims to lead a user.

In the following, we first demonstrate our threat model (§3.1). We then describe how OBSERVER monitors the JavaScript accesses to HTML anchor elements (§3.2), and how it tracks the dynamic creation of HTML anchor elements and HTML script elements (§3.3). Further, we show how OBSERVER hooks several APIs to catch navigation-related JavaScript event listeners (§3.4). Finally, we detail our prototype implementation based on the Chromium browser (§3.5).

3.1 Threat Model

In our threat model, we consider only click interception activities performed by *third-party* scripts as malicious. Although first-party websites might exhibit similar activities to intercept

user clicks, we do not consider them as malicious, because they have the full privilege to control their own applications. Nevertheless, OBSERVER can comprehensively collect all data related to click interception.

3.2 Recording Accesses to HTML Anchor Elements

Modifying a hyperlink in a web page is one of the most explicit methods to intercept and navigate a user click into a different URL rather than the original one. OBSERVER aims to record any accesses to all hyperlinks in a web page to detect any such attempts. In HTML, a hyperlink is defined with an anchor element (*i.e.*, an <a> tag), and its `href` attribute specifies the associated destination URL. Thus, by monitoring and recording which script modifies the `href` attribute of an <a> tag, OBSERVER is able to recognize a script's potential click interception.

JavaScript can modify the `href` attribute through DOM APIs in several ways. We use the keyword `a` to represent an HTML Anchor Element object and the keyword `url` to represent a URL string in the following examples. First, a script can directly assign a new value to the attribute as in `a.href = url;`, or in `a.attributes["href"] = url;`. Second, it may also call the `setAttribute()` API as in `a.setAttribute("href", url)` to perform a similar operation. Note that developers may leverage APIs defined in some third-party JavaScript libraries, *e.g.*, jQuery, to change the attribute. OBSERVER can cover all these wrapper libraries because they would still need to call the above APIs defined in the DOM standard, which is implemented by all browsers to ensure cross-browser compatibility.

OBSERVER hooks all these DOM APIs to monitor modifications to the `href` attribute of <a> tags in the DOM. Specifically, it intercepts any call to such an API. Once intercepted, it inspects the current JavaScript call stack to reason about the origin of API invocation. It locates the bottom JavaScript frame in the call stack to find the JavaScript function that initiates the API call.

Script Identification. To attribute the API access to a specific script, we need to obtain the identity of the accessing JavaScript code. OBSERVER assigns a `scriptID` to each script object to uniquely identify it in the JavaScript runtime. In HTML, JavaScript code is usually enclosed between <script> and </script> tags as an inline script, or stored in an external JavaScript file and loaded with <script> tags as an external script. Each <script> tag is compiled into an individual JavaScript object in the JavaScript engine. There are also other types of inline JavaScript code. For example, JavaScript code can be written as the on-event listener attributes of HTML elements. This kind of inline scripts that are not wrapped within a <script> tag are also compiled into separate JavaScript objects, which are identified by the unique scriptIDs.

OBSERVER associates the scriptID of a script with its `sourceURL`, which is the URL the browser uses to load the remote JavaScript code. The `sourceURL` of an inline script, however, is empty. Instead, we use the URL of the embedding frame, *i.e.*, the URL that the browser uses to load the HTML document into the embedding frame, as the `sourceURL` of *static* inline scripts. However, inline scripts can also be created on-the-fly by JavaScript. We will discuss how we attribute a DOM access to a dynamic inline script in §3.3.2. Besides the scriptID, we also record the row number, column number, and name of the function in the accessing script in a *shadow* data store associated with the element. It is worth noting that JavaScript code cannot modify the shadow data store because it is a C++ data structure that is not writable on the JavaScript side.

3.3 Tracking Dynamic Element Creation

Dynamically creating a new hyperlink in a web page is another method to intercept a user click. In short, OBSERVER considers direct and indirect approaches that a script can exploit to achieve this goal: 1) creating a hyperlink and 2) creating a script that creates a hyperlink.

3.3.1 HTML Anchor Elements

JavaScript code can dynamically create any HTML elements, including an anchor element, in a web page. Specifically, JavaScript can insert a new <a> tag into the DOM tree of a web page through APIs such as `document.write("<a>...</a>")` and `document.createElement("a")`. A script can even replace the entire element with a new element by changing the `outerHTML` attribute of it, *e.g.*, `a.outerHTML = '<a href="' + url + '">...</a>'`. These techniques could be exploited by scripts as another way to intercept user clicks instead of modifying existing hyperlinks. Thus, OBSERVER needs to track the dynamic creation of <a> tags in the browser.

OBSERVER attaches a *shadow* `initiator` attribute to each anchor element in the DOM tree to represent the creator of the object. The initiator attribute is the scriptID of the script that creates the corresponding element. OBSERVER assigns a special initiator value—`0`, which represents the owner of a document—to all static elements that are built by the browser parser. The static <a> tags are the *first-party* hyperlinks. OBSERVER intercepts all the element creation APIs in the web browser to find the initiating JavaScript frame in the call stack. The scriptID of the initiating script is used as the initiator of the dynamically created elements (hyperlinks). OBSERVER would also record any accesses to the `href` attribute of the dynamically created anchor elements.

3.3.2 JavaScript

JavaScript code can also be dynamically generated in web applications, just like HTML elements. Specifically, as one class

of HTML elements, new <script> elements can be dynamically created by JavaScript using the same APIs for creating elements. OBSERVER aims to assign unique identifies to all of such dynamically created scripts. If an external script file is loaded from a remote host into a dynamically inserted <script> element, getting its identity is not different from getting the sourceURL of one static <script> element. Some strings can also be dynamically parsed as inline JavaScript code if they are defined as inline event handlers or passed in the call of APIs like `window.eval("...")`.

However, it is not straightforward to tell the identity of a dynamically generated inline script because its sourceURL is blank. To overcome this difficulty, OBSERVER hooks the APIs that are used to generate dynamic scripts. It saves the sourceURL of the JavaScript code that calls the script generation API as the sourceURL of the newly generated inline script. To distinguish the dynamically generated script, or the *child* script (either an inline script or an external script), from the generating script, or the *parent* script (the one that generates the script), OBSERVER records the scriptID of the parent script as the `parentScriptID` attribute of the child script. The parentScriptID of all scripts that are initially statically embedded by the document owner is set to `0`. This allows us to construct a script dependency graph in the analysis.

OBSERVER also logs all accesses to any inline on-event handlers of any DOM object as it does with the `href` attribute of <a> elements. It finds the last script that sets an inline on-event handler as its parent script and derives the sourceURL from it. If no such an entry can be found, OBSERVER sets the script that creates the receiver object as its parent script.

3.4 Monitoring JavaScript Event Listeners

Instead of modifying or creating hyperlinks, a script can register an event listener or handler to an HTML element. The event handler is asynchronously executed whenever there is a user click on the element. In particular, a script may open an arbitrary URL in a new browser window/tab, or send an HTTP request in the background, when a user clicks any element it listens for. Therefore, OBSERVER aims to monitor all event listeners registered by JavaScript code in a page to identify whether they will navigate a user to a different URL according to a user click.

OBSERVER first monitors event listener registration by hooking the `addEventListener()` API and monitoring accesses to the on-event listeners, to identify the scripts that are interested in user interactions. It then intercepts any click-related user events (*e.g.*, *click* and *mousedown*) when they are fired in the web browser and detects the event target element in the DOM tree. Since a script may not necessarily initiate a page navigation in its event handler (*e.g.*, an analytic script), OBSERVER filters those scripts by hooking several APIs that can be used for starting a navigation, *e.g.*, `window.open('...')`, `window.location = '...';`, *etc.* OBSERVER detects the bottom frame in the JavaScript call stack and further constructs and logs the navigation URL in these APIs in the shadow data store of the target element.

One challenge we met in our design is that one event handler can be activated multiple times. In the DOM, the events are propagated in three phases: capturing, target, and bubbling. For example, in the capturing phase, an event is propagated from the root node in the DOM tree—the <html> node, then through any intermediate parent nodes, before finally reaching the target node. An event handler registered in the capturing phase at the <html> tag will always be triggered whenever any of its child elements is clicked[1]. To avoid activating such event listeners multiple times, OBSERVER would skip calling an event listener at a node if the `Event.currentTarget` object (*i.e.*, the current node) is different from the `Event.target` object in event propagation. We further set a flag in OBSERVER to abort all page navigations, including those caused by clicking the <a> tags, after the navigation URLs are saved in the logs. This enables us to efficiently interact with all elements in a web page without really visiting the linked URLs.

3.5 Implementation

We implement a prototype of OBSERVER in the Chromium browser (version 64.0.3282.186). We will release our prototype implementation as an open source software. We implement OBSERVER in a full-fledged browser to escape any artificial result that might be caused by using a simpler and uncommon user agent. We add several custom attributes (*e.g.*, initiator, accessLog, scriptID, parentScriptID, sourceURL) to the Node[2] objects to save the monitoring data. All these custom attributes can be read but not written by JavaScript for further analysis. For performance concerns, we implement a lazy update mechanism for setting the above attributes. The values of these attributes are kept in the hidden attribute members of the modified C++ classes. They are updated in the DOM tree only when the attributes are first accessed by JavaScript.

We hook the above DOM APIs by inserting custom monitoring code in the C++ implementation of the V8 binding layer between the V8 JavaScript engine and the DOM implementation in WebKit. The custom monitoring code identifies the JavaScript caller by fetching the scriptID of the bottom frame in the JavaScript call stack. It appends the logs of accesses to the `href` attribute and the inline on-event handlers to the hidden `accessLog` attribute of the corresponding DOM object. The code sets the `initiator` attribute of an anchor element when it is created by either JavaScript code or the browser parser. Furthermore, the sourceURL and parentScrip-

[1] An event handler registered in the bubbling phase at a parent node may not be activated because the event propagation can be stopped by some other event handler registered at its child node.

[2] Node is the base class of HTML elements in WebKit.

tID of all scripts are stored with a <script> object. We further store the scriptID in the sourceURL dictionary at the global Document object.

The prototype of OBSERVER can comprehensively log all click-interception-related events. In the browser, a click-driven navigation can be started by the built-in default event handler of anchor elements (hyperlinks) and the developer-defined event handlers, which we have introduced in §3.2 and §3.4. OBSERVER ensures complete mediation of element accesses and event handler registrations in the C++ implementation of the corresponding DOM APIs (including the built-in default event handler), which cannot be bypassed by any JavaScript code. In other words, the browser must go through the underlying C++ APIs and our monitoring code when JavaScript code accesses any hyperlink or registers an EventListener to any HTML element.

4 Methodology

In order to study the click interception problems in the wild, we perform a large-scale data crawling of the Alexa top 250K websites. We describe our data collection method in §4.1, how we determine the owner and privilege of JavaScript code as well as HTML elements in §4.2, and finally how we detect three classes of click interception in §4.3.

4.1 Data Collection

We use the OBSERVER prototype to collect data for investigating the click interception problem. In particular, we aim to identify all hyperlinks and scripts that react on user clicks, and the destination URLs that the browser would visit after the clicks. We leverage the Selenium WebDriver Python binding to automatically drive OBSERVER and interact with the web page it renders. To this end, we run our analysis framework on a 64 core CPU Linux server and collect data from the Alexa top 250K websites.

We collect data in two phases for each web page: 1) collecting *default data* right after page rendering; and 2) collecting *reaction data* by interacting with a rendered page. In each page navigation, we first asks OBSERVER to wait for a page to be completely rendered by the browser for up to 45 seconds. After that, we insert a script into the page to traverse the DOM tree in pre-order to collect all the data OBSERVER has logged with each element. In addition, we log for each element several display properties (*e.g.*, width, height, position, opacity, *etc.*) to study additional tricks that may be used to intercept user clicks (*e.g.*, some third-party contents overlap with or appear similar to first-party contents). We then save a snapshot of the current DOM tree into an external HTML file as well as a full-page screenshot for further analysis.

Next, we interact with a rendered page to collect data about how the page reacts to our clicks, such as navigation and DOM modification. We disable the navigation flag in OBSERVER to deactivate real navigations that may be caused by event handlers or hyperlinks. We then automatically click all elements in the DOM tree through Selenium to trigger the click event listeners and hyperlink navigations to collect navigation logs. For each navigation triggered by a click, we log the information regarding the navigation URL, the clicked element, and, if exist, the corresponding event listeners and scripts that initiate the navigation. In addition, we traverse the DOM tree again, as we do in the first phase, to identify whether scripts update the DOM elements due to user clicks.

4.2 Third-party Content Detection

In this section, we explain our techniques to distinguish first-party scripts/contents from third-party scripts/contents, which is necessary to detect click interceptions driven by third-party scripts. A naïve technique that merely relies on the exact origin of scripts is not enough because a website frequently loads its own scripts from its subdomains, its different domains, and domains operated by others such as content delivery network (CDN) services. For example, the main page of https://www.google.com/ includes scripts from its subdomain apis.google.com and its CDN domain gstatic.com. If we use only origin information, we may misidentify these scripts as third-party scripts. We aim to solve this problem using *domain substring matching* and *DNS record matching*.

Domain substring matching is a heuristic technique to infer that a remote script is a first-party script if the remote script's domain name is similar to the current page's domain name. It first checks whether the main domain names of a remote script and the current page are the same while excluding domain suffixes. For example, a script loaded from https://apis.google.com/ on https://www.google.co.jp/ is determined as a first-party script because its main domain name excluding the suffix *com* is *google*, which is identical to that of the current page excluding the suffix *co.jp*. Second, it tests whether the proper subdomain name of a remote script consists of the main domain name of the current page without suffixes, to come up with CDN practices that maintain custom subdomain names for individual websites. For example, a script loaded from https://static-global-s-msn-com.akamaized.net/ on https://www.msn.com/ are inferred as a first-party script because the proper subdomain name *static-global-s-msn-com* contains the main domain name *msn*. We do realize that our technique has limitations, which we will discuss in §6.

DNS record matching leverages several DNS records to decide whether two distinct domains are operated by the same organization. Specifically, we inspect the DNS SOA records [36] and the DNS NS records [34] of the two hostnames (domain names). An SOA record includes the email address used to register the domain. Many organizations would use the same email address to register multiple domains. For instance, the SOA email addresses of google.com and gstatic.com are both *dns-admin@google.com*. However, there are also exceptions.

Different organizations may use the same Managed DNS providers [35] to register domains. Accordingly, their SOA same email addresses are identical. For example, both dropbox.com and bitbucket.org use *awsdns-hostmaster@amazon.com* as their SOA email address.

We address this limitation by further examining if the name server (NS) records of a script/URL and the first-party web page have an intersection. Specifically, we use the *domain name* instead of the full hostname of a NS, because one domain may use several NSs from a large pool. If the first-party domain name is found in a common NS, we mark the external script as a first-party script. For instance, both gstatic.com and google.com use NSs *nsX.google.com*, where *X* is a numeric value. Therefore, we determine the two domains belong to the same organization because they have a common NS domain name—*google.com*, and an identical SOA email address. Note that we exclude all common NSs that are operated by any known managed or dynamic DNS providers.

Dynamic Element. Recognizing the sources of dynamic elements is also important to identify cross-party accesses. We classify dynamic elements into two groups based on which parties their initiating scripts belong to. This allows us to distinguish first-party contents from third-party contents.

4.3 Click Interception Detection

Normally, a user may explicitly click a hyperlink to navigate to another web page, or click some components such as images or buttons to interact with the current web page. However, some scripts may deliberately intercept a user's clicks to override the default action that the user may expect. Furthermore, a user could also be fooled by a script into clicking some components she/he would not click. We designate such undesired click manipulation caused by privilege abuse as **click interception** in web applications. As discussed earlier, we do not consider click interceptions exhibited by first-party scripts as malicious.

Based on how a user click could be manipulated, we categorize click interception into three classes—interception by hyperlinks, interception by event handlers, and interception by visual deception. In particular, a script can intercept user click by 1) using an existing hyperlink or creating a new hyperlink; 2) registering a click event handler with an element; and 3) manipulating the UI to deceive a user into clicking elements controlled by the script.

In the following, we explain the methods to detect the three classes of click interception. Specifically, we leverage the *navigation URL* and the *navigation APIs*[3] (§3.4), and the display properties of the element (§4.1).

[3]The default event handler of <a> tags is also considered as one API.

4.3.1 Interception by Hyperlinks

In general, a script can intercept user clicks with hyperlinks in two ways: modifying one existing (first-party) hyperlink, and adding one hyperlink to a huge element.

Modifying Existing Hyperlinks. A *third-party* script can intercept a user's click through a *first-party* hyperlink by overwriting the *href* attribute. A *third-party* script might also employ a similar approach to intercept a user's click on *another third-party* hyperlink. Therefore, we search in the *href* attribute log of an anchor element the last script that modifies its value. If a (different[4]) third-party script is found, the script is marked as one click interception script. We use the technique in §4.2 to determine if the script and the anchor element belong to the same organization. A third-party script might also intercept a user's click through attaching an event listener to a first-party hyperlink, which we discuss in the following section. Note that although a *first-party* script may modify a *third-party* hyperlink, we think this is legitimate because the first party as the owner of the web page is entitled to include or remove any third-party contents.

Creating Huge Hyperlinks. A script can trick users into clicking its hyperlink by enclosing a huge clickable element. In particular, it can enclose a significant part of its web page within one <a> tag such that a click on any of the enclosed contents would result in a page navigation that is controlled by it. Therefore, we also check the size of an anchor element relative to the browser window[5]. Specifically, we use 75% as the threshold to detect the suspicious huge hyperlinks that can be used to intercept user clicks. According to our knowledge, most (but not all) links on the web are relatively small compared to the browser window. Therefore, we think 75% is a reasonably large threshold to help quickly identify the suspicious ones. Further, we exclude any hyperlinks pointing to a first party navigation URL, because the first party has the right to use huge hyperlinks in its own pages.

4.3.2 Interception by Event Handlers

The event handlers are the second technique that a script can use to intercept user clicks. However, a script listening for user click may not necessarily navigate the user to another URL. For instance, an analytic script may observe user clicks to determine and log only user engagement within the current page. We leverage the navigation-related APIs to solve this problem.

To start a new navigation, a developer needs to either call the `window.open()` API or change the `location` of the current frame. The two JavaScript DOM APIs are implemented by the C++ methods `LocalDOMWindow::open()` and `Location::SetLocation()` in WebKit, respectively. For each element, we

[4]We use the term a *different script* to represent a script of a different organization in the rest of the paper.

[5]We used 1024px x 768px as the browser window size in our experiments.

examine if the two C++ methods are (indirectly) called upon a click on the element. We then extract the navigation URLs from the associated logs.

Third-party Interception Scripts using Event Handlers. We determine a *third-party* script as a click interception script if it (indirectly) calls either one of the above two C++ methods in its click event listener that is added to a *first-party* element. We name such a click event listener as a *navigation event listener*. Similarly, if such a navigation event handler is added to a *third-party* element created by the script of a different organization, the third-party script implementing the event handler is also determined as a click interception script.

Intercepting Huge Elements with Event Handlers. We use the same 75% relative size threshold to detect suspicious huge elements that are registered with a *third-party navigation event handler* and can be used to intercept user clicks. We also filter the elements that are associated with a first-party navigation URL.

4.3.3 Interception by Visual Deception

Third party scripts can also intercept a user's clicks through visual implementation tricks to deceive a user. In particular, the third-party contents are designed in some way such that a user is likely to click. We do not consider first-party contents with similar characteristics malicious because the first-party websites have the complete freedom to design their contents.

This last click interception category could be controversial in our opinion, as some third-party developers may argue that they *do not intend to deceive the end users*. Nevertheless, we still classify such practices as click interception (but not necessarily malicious) because the users *can be deceived through the visual tricks*.

We have identified two possible visual deceptions—mimicry, and transparent overlay. We detect these visual deceptive tricks for each *group of third-party elements*, which are the largest sub DOM tree that consists of only elements of the same third-party script (organization).

Mimicry. Some third-party script would deliberately decorate its elements such that they are almost visually indistinguishable from first-party contents. A user might consequently click these mimic elements. However, the imitating elements are usually not exact copies of some first-party elements. As a result, we cannot use pixel-wise comparison to detect such mimic elements.

We utilize the structural information as well as the display properties of a third-party element group to detect mimicry. Specifically, we compute the relative size of media contents, *e.g.*, images, videos, and iframes, in *a group of third-party elements*, as well as the size of the largest container of them. We then compute the same metrics for any *group of first-party elements* whose root node is a sibling (neighbor) to that of the third-party element group. Next, we calculate a similarity score between the two groups of elements using: 1) the CSS class names of the two root nodes, which are primarily used to describe the representations of HTML elements; 2) the numbers of each kind of media tags, which indicate how media contents are implemented; and 3) the relative sizes of media contents in two groups and the sizes of the largest container nodes, which represent the visual layout of an element group.

We set a threshold learned from our training phase to keep only third-party element groups that are very similar to some first-party element groups. Note that we compute the similarity scores using the display property data before we click the elements to find the elements whose default representation is likely to fool a user. We do acknowledge that there are other features (*e.g.*, the DOM tree structure, color histogram) that may better determine the similarity. However, we find the ones that we select work well in our manual test over a small set of samples. We plan to leverage more sophisticated techniques (*e.g.*, image classification [7]) in our future work.

Transparent Overlay. A third-party script can inject contents that partially overlap with or completely cover first-party contents. In the case that some first-party contents are completely covered, the user might not notice their existence and treat the covering third-party contents as first-party ones. Further, a script can make some of its elements barely visible by setting a small value to their *opacity* style property. Subsequently, a user's click could be delivered to these "hidden" elements when the user is intending to click some other elements beneath them. We detect transparent overlay third-party contents in the following two steps.

First, for each group of third-party elements, we compute the *minimum* portion of a *first-party* element that it overlaps with. Specifically, we scroll the browser window virtually to compute all the possible overlapped regions with each *first-party* element. If the covered portion of a first-party element is always greater than a pre-defined threshold (*e.g.*, 25%), we label this group of third-party elements as overlay elements. Since some third-party scripts may implement components allowing a user to cancel out the overlay elements, we further exclude those that no longer significantly overlap with any first-party element after our automatic clicks, which must include a click on one of such cancel-out buttons if there are any. However, this method may not work well in some cases. For example, the covering elements could first be hidden by a click on a cross button, and later be revealed by another click on another button. We consider it as a limitation and plan to leverage knowledge in computer vision to develop a better automated testing method in our future work.

Next, we detect third-party transparent overlay element groups by comparing the *opacity* value collected in the display properties with a small threshold (*e.g.*, 0.1). A zero opacity value indicates complete transparency. We do not consider elements whose style is *visibility: hidden* or *display: none* because user clicks are not passed to these invisible elements. In addition, we keep only the transparent third-party element groups that are big enough to be easily clickable, *i.e.*, the

container size is greater than 1% of the browser window size.

5 Click Interception in the Wild

In this section, we first present our analysis on data collected in our web crawl (§5.1), then characterize click interception by demonstrating *how* different techniques (§5.2) are employed by *which* scripts (§5.3) to intercept user clicks, and finally explain *why* they do it and its consequences (§5.4).

5.1 Dataset

We crawled data from the main pages of Alexa top 250K websites in May 2018. Excluding those that timed out or crashed in our data collection process, we were able to gather valid data of 228,614 (91.45%) websites. We identified *third-party* navigation URLs (the first URL the browser would visit upon a user click) collected in a web page using the method described in §4.2. We obtained 2,065,977 unique third-party navigation URLs, which corresponded to 427,659 unique domains. On average, a web page contains 9.04 third-party navigation URLs, pointing to 1.87 domains.

We visited each of the 2M navigation URLs and recorded both the intermediate redirect URLs and the landing URL. We could not visit 39 URLs in our experiment because of various errors (*e.g.*, HTTP 404 status code, too many redirects, *etc.*). We managed to obtain 1,982,613 unique landing URLs.

We collected 413,075 intermediate redirect URLs (excluding the navigation URLs and the landing URLs) in this process. Specifically, we observed no redirection for 1,263,754 (61.17%) navigation URLs. We encountered at most 29 intermediate hops before we reached a final landing URL.

We detected 2,001,081 distinct third-party scripts that were loaded from 1,170,582 different domains. On each page, there are on average 8.75 third-party scripts.

5.2 Click Interception Techniques

In this section, we demonstrate how the different techniques that we identify in §4.3 are employed for click interception.

5.2.1 Interception by Hyperlinks

We identify three possible ways that a *third-party* script can intercept user clicks through hyperlinks (§4.3.1). In total, we observe that 4,178 hyperlinks on 221 websites were intercepted, which can lead a user to 2,695 distinct *third-party* URLs. We present in Table 1 the breakdown of the 4,178 links and the total number of daily visits to the affected websites[6].

Hyperlink Modifications. Surprisingly, the *href* attribute of 4,027 first-party <a> tags on 100 websites were directly tampered by a third-party script. For instance, the ad URL shortening script https://cdn.adf.ly/js/link-converter.js modified the *href* attribute of one anchor element to http://ay.gy/2155800/... on the website http://magazinweb.net/. Similarly, the third-party script https://cpm4link.com/js/full-page-script.js modified hyperlinks on the website https://www.lnmta.com/ to https://cpm4link.com/full/?api=.... They are obviously privilege abuses. In addition, we find that 31 third-party hyperlinks on 2 websites were modified by a different third-party script. For example, the script https://s7.addthis.com/js/300/addthis_widget.js modified 11 third-party hyperlinks on the website https://www.crazy-net.com/ to https://plus.google.com/110631064773293614230; the script http://media1.admicro.vn/core/log_cafef.js modified 20 third-party hyperlinks on the website http://cafef.vn/ to http://lg1.logging.admicro.vn/nd?nid=.... This indicates that those third-party scripts indiscriminately modify anchor elements to intercept user clicks.

Huge Hyperlinks. We observe 120 huge *third-party* <a> tags on 119 websites. These anchor elements enclose contents whose size is at least 75% of the browser window size. As a result, a visitor has a very high chance to click such an anchor element. For example, on the website http://torrents73.ru/, the third-party script http://gynax.com/js/MjgxMw==.js created a large anchor, which encloses a huge background image. Users would be directed to another page https://wheel.grand-casino48.com/ upon a click. We also identify that 135 websites used 148 huge *first-party* <a> tags, which we currently consider as legitimate as we discussed in §3.1.

Table 1: Categorization of Click Interception Techniques

Technique	#Cases	#Websites	%Cases	#Visits/day
Hyperlinks	4,178	221	89.52	12,686,591
Modifying 1st-party links	4,027	100	86.29	2,496,620
Modifying 3rd-party links	31	2	0.66	638,247
Inserting huge 3rd-party links	120	119	2.57	9,551,724
Event Handlers	203	172	4.35	5,455,821
On 1st-party nodes	189	161	4.05	4,636,145
On 3rd-party nodes	14	12	0.30	819,676
On huge 3rd-party nodes	0	0	0	0
Visual Deceptions	286	231	6.13	25,269,314
Mimicry	140	87	3.00	16,604,258
Transparent Overlay	146	144	3.13	8,665,056

5.2.2 Interception by Event Handlers

We analyze how event handlers are exploited to intercept user clicks. Overall, we find 203 elements across 172 websites were attached with *navigation event handlers*, which would drive a user to a *third-party* URL upon click.

We observe that 189 *first-party* elements of 161 websites were added at least one *third-party* navigation event handler. For example, the third-party script https://smashseek.com/rq/4949 intercepted user clicks on the website https://www1.mydownloadtube.com by adding a navigation event listener to the <html> element. The user's browser would open a new URL (the specific URL changes upon each user click) when

[6] We get the statistics using the SimilarWeb API.

a user clicks any element on this page[7]. Another example is detected on the page http://azasianow.com/, where the third-party script http://fullspeeddownload.com/rq/4297 registered an event handler on the <body> element. We also consider such practices as a type of privilege abuse, as they force a user to visit a URL when the user interacts only with first-party contents. What is worse, even an experienced user with some technical background cannot easily find out that the navigation is actually controlled by a third-party script rather than the website she/he directly visits.

Interestingly, we find on 12 websites that 14 *third-party* elements were attached with navigation event handlers by a third-party script of a different organization. For example, the website https://www.mlbstream.io/ included the third-party script https://amadagasca.com/rgCQwi5INUm04AxMu/5457, which registered an event handler on an <img> element. The user would be directed to https://jackettrain.com/imp/5457/?scontext_r=... upon clicking on that image and finally land at a random website. One possible reason is that the attaching scripts were loaded after the other third-party scripts had inserted those elements, so that they mistakenly attached event handlers to the other third-party elements.

We do not find any third-party script intercepting user clicks by registering navigation event handlers with huge third-party elements. On the other hand, we discover 2 websites added navigation event handlers to their own huge elements. In particular, the websites http://www.force-download.net/ and http://www.force-download.es/ both registered a navigation event handler to the <html> node to intercept user clicks, just as the above-mentioned third-party scripts. Nevertheless, we do not consider them as malicious.

5.2.3 Interception by Visual Deception

We analyze how the two visual deception techniques, *mimicry* and *transparent overlay* (§4.3.3) are used in the wild.

Mimicry. We discover 140 *mimic* third-party element groups on 87 websites. These *third-party* contents are carefully designed to resemble nearby *first-party* contents. Hence, unwary users are very likely to be fooled and consequently click them.

Figure 1(a) shows an example of such a *mimicry* trick that we detect on the website https://www.bintang.com. The contents enclosed within the yellow rectangle were inserted by the third-party script https://securepubads.g.doubleclick.net/gpt/pubads_impl_207.js, whereas those in the red rectangles were the organic first-party contents. Without scrutiny, they just look like each other. The only visual hint for discriminating them is the text *Sponsored*, which was displayed in a very small font size just as the first-party sub captions in the red rectangles. Even though a user may notice this small text, she/he may still decide to click the third-party elements as they appear to be provided directly by the first-party website which she/he trusts. However, such trust would be abused in this case because those contents were generated solely by a third-party script the user does not know. In particular, the navigation URL was under the full control of this unknown third-party script and could take the user to any (potentially unsafe) page. We will discuss more about the security implication in §5.4.



(a) Mimicry.



(b) Transparent overlay.

Figure 1: Examples of visual deceptive third-party contents.

Transparent Overlay. We detect 146 *transparent overlay* third-party element groups on 144 websites. Specially, they covered a significant portion (at least 25%) of first-party elements regardless of mouse scroll. We could not cancel them out by automatically clicking elements in those websites. Further, they were either completely transparent or translucent with a very low *opacity* style value. What is worse, many of them contained NO user-perceivable content (*e.g.*, texts or images), hence being *transparent*. As a result, they were almost—if not absolutely—invisible and thus difficult to be noticed.

Figure 1(b) demonstrates an example of such a visual trick that we identify on the website http://jgsdf.ucoz.com. The yellow rectangle includes the *third-party* contents that overlapped with the underlying *first-party* contents, which are enclosed by the cyan rectangles. The script that created these third-party contents is http://pl14318198.puserving.com/a2/49/14/a2491467a19ffc3f9fe0dbe66e54bae0.js. Although the overlay third-party contents were not visible in this case, they constantly covered about 50% of the first-party contents in the cyan rectangles no matter how a user scrolled this page. As a result, this script could intercept any click on the covered first-party elements, because the click would be first passed to the overlay third-party elements. When a user clicked within the area of yellow rectangle, an ad link was opened in a new window.

Although third-party scripts can deceive a user with different tricks, the effectiveness can vary dramatically depending on their implementation and the end user's technical background. In general, we think they are less effective compared with the other two direct techniques we have discussed above. In particular, whether the *mimic* contents are deceptive is really subjective. We leave it for our future work to examine

[7] This is not true for elements with other click event listeners that stop the event propagation.

how effective the visual deceptions are on real users.

5.2.4 Evasion of Detection

We also detect a few cases that third-party scripts *selectively* intercepted user clicks. In particular, they would limit the rate at which they intercept the clicks to avoid a user's suspicion. For instance, some scripts would activate the page navigation code in their event handlers only when a user *first* visits a page. This can be easily implemented by dropping a cookie in a user's browser. They might clear this flag after some time (*e.g.*, a day) to reactivate the click interception code. However, we do not have enough data to learn the timeouts they use. We discuss next such a detection evasion example.

The script https://pndelfast.com/riYfAyTH5nYD/4869—included by the website https://torrentcounter.to/—selectively intercepted the user clicks on the background of the website. We observed the interception only when we visited the page with a clean cookie, which suggests the script used a cookie to log click interception status. Interestingly, we find the script was obfuscated to prevent a normal user from analyzing it. We deobfuscate the script (Listing 1), and search for the keyword *cookie*. As expected, we find several functions that are used to control the rate of click interception. Lines 8, 13, and 16 define the functions `"setCookie"`, `"removeCookie"`, and `"getCookie"`, respectively. Line 6 defines the `"timeout"` variable that we suspect to control the interception timeout or interval. It sets the cookie in Line 28, if the return value of the function `init` defined in Line 20 is not true. The cookie is deleted in Line 33. This script also defines several variables, *e.g.*, `"certain_click"`, `"every_x_click"`, `"delay_before_start_clicks"`, `"click_num"`, `"interval_between_ads_clicks"`, which we believe to be used to control click interception. As is limited by the space, we do not discuss in more details how the script works. It would be an interesting research topic to investigate how these scripts cloak their malicious activities to avoid detection.

Summary. We confirm that various click interception techniques have been used in the wild. Third-party scripts intentionally intercepted user clicks using event listeners, and manipulate user clicks through visual deceptions. They also leveraged huge anchor elements to deliberately intercept user clicks. Further, many *third-party* scripts even modified *first-party* hyperlinks to intercept user clicks.

5.3 Click Interception Scripts

In this section, we characterize click interception based on the third-party scripts that intercept user clicks. Further, we investigate how they were embedded to intercept user clicks.

```
1  var _0x3e0d = ["...", "certain_click", "every_x_click"
       , "delay_before_start_clicks", "click_num", "
       interval_between_ads_clicks", "has_adblock", "...
       "];
2  var build = function() {
3    var target = {
4      "data" : {
5        "key" : "cookie",
6        "value" : "timeout"
7      },
8      "setCookie" : function(value, name, path, headers)
         {
9        var cookie = name + "=" + path;
10       headers["cookie"] = cookie;
11     },
12     "removeCookie" : function() {
13       return "dev";
14     },
15     "getCookie" : function(match, href) {
16       var v = match(new RegExp("(?:^|; )" + href["
         replace"](/([.$?*|{}()[]\/+^])/g, "$1") + "
         =([^;]*)"));
17       return v ? decodeURIComponent(v[1]) : undefined;
18     }
19   };
20   var init = function() {
21     var test = new RegExp("\\w+ *\\(\\) *{\\w+
         *['|\"].+['|\"];? *}");
22     return test["test"](target["removeCookie"]["
         toString"]());
23   };
24   target["updateCookie"] = init;
25   var array = "";
26   var _0x418128 = target["updateCookie"]();
27   if (!_0x418128) {
28     target["setCookie"](["*"], "counter", 1);
29   } else {
30     if (_0x418128) {
31       array = target["getCookie"](null, "counter");
32     } else {
33       target["removeCookie"]();
34     }
35   }
36 };
```

Listing 1: A simplified click interception script from https://pndelfast.com.

5.3.1 Third-party Scripts Characterization

Our results in §5.2 demonstrate that third-party scripts leverage all the three techniques to intercept user clicks. We present the statistics of these scripts—the *unique* number of script URLs, origins, and domains in Table 2.

Huge Hyperlinks. We detect 86 unique *third-party* scripts that injected huge <a> tags into their embedding pages. We show the top 5 origins of such scripts in Table 3. The noticeable scripts are those loaded from http://gynax.com. They were found to create one huge <a> element on each of 47 websites they were included. Each <a> tag was enclosed within a <noindex> element, which further contained a full-page image. All the hyperlinks would finally reach https://wheel.28grand-casino.com/, which is an online gambling game website.

Hyperlink Modifications. We detect 57 unique *third-party* scripts that directly intercepted user clicks by modifying *first-party* hyperlinks. We show the top 10 origins of such scripts in Table 4. The top script https://cdn.adf.ly/js/link-converter.js

Table 2: Statistics of unique click interception scripts.

Technique	#URLs	#Origins	#Domains
Hyperlinks	145	76	63
Modifying 1st-party links	57	41	35
Modifying 3rd-party links	2	2	2
Inserting huge 3rd-party links	86	33	26
Event Handlers	106	72	58
On 1st-party nodes	103	69	55
On 3rd-party nodes	7	7	7
On huge 3rd-party nodes	0	0	0
Visual Deceptions	197	173	95
Mimicry	78	60	54
Transparent Overlay	119	114	42

Table 3: Top 3rd-party script origins injecting huge anchors.

Script	#Websites	#Elements
http://gynax.com	47	47
https://securepubads.g.doubleclick.net	7	7
https://yastatic.net	7	7
http://bgrndi.com	6	6
http://js883.guangzizai.com	5	5

was found on 18 websites. *Adf.ly* is a short URL service that helps websites monetize their links. As its name suggests, this script converts *every first-party* hyperlinks to a *third-party* hyperlink. If a user clicks any converted hyperlink, the user would be taken to an intermediary page of *adf.ly* hosted on http://clearload.bid/. This page displayed an advertisement as shown in Figure 2. The user can click the *SKIP AD* button on the right top corner to continue to visit the original *first-party* hyperlink. Many other top scripts in Table 4, *e.g.*, https://linkshrink.net/fp.js, https://api.getsurl.com/js/get_auto.js and https://adshort.co/js/full-page-script.js, worked in a very similar way. This is definitely very distracting to users. However, as we will demonstrate next in §5.4, the first-party websites explicitly included these click interception scripts to monetize their websites.

Event Handlers and Visual Deceptions. We find 103 unique third-party scripts which listened for clicks on first-party elements to intercept user clicks. We also discover 78 and 119 unique *third-party* scripts that injected mimic and transparent overlay contents, respectively, into the embedding websites. We discuss next that how these click interception third-party scripts were included in those "victim" websites.

5.3.2 Click Interception Script Inclusion

While we discover that third-party scripts deliberately intercepted clicks via several tricks, it is not clear if they were intentionally included by the first-party websites. To this end, we analyze the script dependency data to figure out the inclusion relationship between third-party scripts and first-party websites. In particular, we aim to determine if a click interception third-party script was *directly included by the website itself*, or *indirectly included by another third-party script*.

Table 4: Top 3rd-party script origins modifying first-party links.

Script	#Websites	#Elements
https://cdn.adf.ly	18	583
https://cdn.shopify.com	11	245
https://static.v2.paysites.czechcash.com	9	640
https://www.sc.pages02.net	7	82
https://linkshrink.net	7	190
https://api.getsurl.com	5	384
https://static-js.sixshop.co.kr	4	59
http://cdn.adf.ly	2	190
http://shinkme.com	2	38
https://adshort.co	2	28

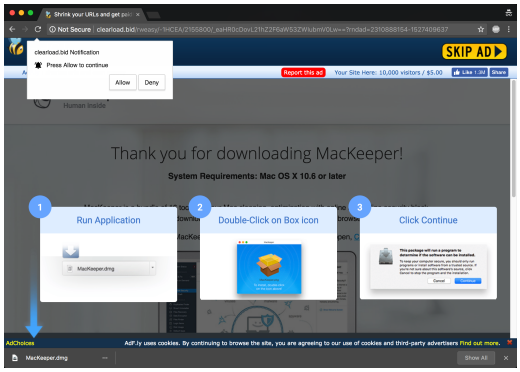


Figure 2: A drive-by download page visited via click interception.

We categorize how a remote third-party script can be included into three classes. First, a third-party script is *statically included by the first-party website*, if the corresponding <script> tag is statically defined in the original web page HTML source. Next, a third-party script is *dynamically included by the first-party website*, if it is loaded through a <script> tag that is dynamically created by a first-party script, including those first-party scripts hosted on a different domain. Finally, a third-party script is *dynamically included by another third-party script*, if it is loaded through a <script> tag that is dynamically created by another third-party script. We summarize the results in Table 5.

Static Inclusion. We find that the majority of these third-party scripts, *i.e.*, 280 unique scripts (64.07%) out of 437 third-party click interception scripts, were statically included by 397 websites. This indicates that these websites deliberately included the click interception scripts, even though they may not intercept user clicks by themselves. In particular, the short URL monetization script https://cdn.adf.ly/js/link-converter.js was found to be statically included by those 18 websites. The script https://wchat.freshchat.com/js/widget.js was statically included by 17 websites. These websites *explicitly allowed such scripts to intercept their users' clicks in exchange for payments*.

Dynamic Inclusion. We discover that 103 unique third-party scripts (23.57%) were dynamically included by first-party

Table 5: How third-party click interception scripts are included.

Inclusion Type	#Websites	#Scripts
Statically included by 1st-party website	397	280
Dynamically included by 1st-party website	112	103
Included by another 3rd-party script	104	63

Table 6: Advertising click interception navigation URLs.

Technique	#URLs	#Ad URLs	%Ad URLs
Hyperlinks	2,695	1,088	40.37
Event Handlers	186	21	11.29
Visual Deceptions	380	74	19.47

websites. For instance, the scripts script=http://gynax.com/j/w.php and http://bgrndi.com/js/NTQw.js were dynamically included by 5 and 4 first-party websites, respectively. In other words, these websites used JavaScript to dynamically create <script> tags to include those scripts. Such websites would be responsible for the privilege abuses by those click interception scripts even if they do not intercept user clicks. They either *did not scrutinize the scripts before including them*, or *deliberately allowed them to intercept user clicks*.

Indirect Inclusion. On the other hand, we discover that only 63 third-party click interception scripts (14.42%) were indirectly included by other third-party scripts. One such a top script is https://tags.bkrtx.com/js/bk-coretag.js, which was included by other third-party scripts on 6 websites. For example, it was included by the script https://s.accesstrade.net/js/atd/bluekai/atd_bluekai.js?id=... on the website https://haken-mikata.com. The latter script was also indirectly included by another script https://s.accesstrade.net/js/atd/satd.js?pt=824F2E4C4077D97ECC014C7A3DE07136725853, which was statically included by the first-party website. In such cases, we cannot blame the first-party websites for indulging those suspicious scripts. Click interception caused by these scripts could be prevented if the websites configure a proper Content Security Policy (CSP) [33] that disallows the browser to load scripts from unknown sources. However, in practice it is difficult and even infeasible to use CSP because many websites need to allow dynamic inclusion of advertising scripts that may be loaded from arbitrary sources due to ad syndication. Therefore, a finer-grained security policy that limits the privilege of included scripts would be more desirable in preventing such privilege abuses.

Summary. We discover that 437 third-party scripts attempted to intercept user clicks on a total of 613 websites. Several top third-party scripts deliberately intercepted user clicks on all their embedding websites. Surprisingly, many of them were included directly by the first-party websites, to monetize the hyperlinks, or more accurately, the user clicks, of those websites.

5.4 Click Interception Reasons and Consequences

We have demonstrated that some third-party scripts intercepted user clicks through various tricks. In this section, we seek to understand the motivations and consequences of such undesired activities.

5.4.1 Monetization

As we have demonstrated in §5.3.1, many *third-party* scripts offer monetization services by converting *first-party* hyperlinks into *third-party* ad links. They force a user to view an advertisement before navigating to the original destination page when the user clicks any hijacked link. As a result, both the third-party click interception script and the first-party website can earn some commission from those participating advertisers. Similarly, we find many other cases where a click was intercepted by a third-party script to visit an advertiser's landing page.

Identifying Advertising URLs. To understand if monetization via advertising is really a common reason for click interception, we compare the navigation URLs in the click interception cases with all the other navigation URLs in our dataset. Specifically, we leveraged the Ghostery extension to determine if one navigation URL is *advertising-related* by testing if it matches the URL pattern of any known advertising company. A navigation URL is marked as an advertising URL, if a positive match is found for any of its intermediate redirect URLs (if any) and the landing URL. We also manually labeled the URLs generated by those short URL monetization scripts as ad URLs because they are not known to the extension.

Surprisingly, we find that 1,183 (36.39%) out of the 3,251 unique click interception navigation URLs are advertising URLs (Table 6), which is a *18.7 times higher* rate than that of normal third-party navigation URLs[8]. In total, only 40,278 (1.95%) out of the 2,065,977 third-party navigation URLs are identified as advertising URLs.

Potential Click Fraud. These click interception websites and scripts have a "good" reason to trick users into clicking those advertising URLs. In online display advertising, the publishers and the ad networks are paid by an advertiser when a user clicks the advertiser's ad under the pay-per-click billing mode. Although they can also earn some commission for an ad impression in the pay-per-view billing mode, the money is much less than what they can get paid when the ad is clicked. However, the ad click-through rate is usually very low—around 2% (in a business-to-consumer banner ad case [18]). To boost ad revenue, the straightforward and effective approach is to leverage real user clicks, as modern ad networks can accurately detect bot-based click frauds [2, 6, 9, 38]. On the other hand, the third-party scripts also have the incen-

[8] We exclude all first-party navigation URLs in our analysis.

tive to cheat advertisers for higher income because many of them are also ad networks. This well explains why the short URL monetization scripts, which also operate as ad networks, have been helping websites intercept user clicks.

In our research, we observe that third-party scripts have leveraged various click interception techniques to monetize user clicks. Further, our results demonstrate that click interception has become an emerging way for generating realistic click traffic to commit ad click fraud.

5.4.2 Distributing Malicious Content

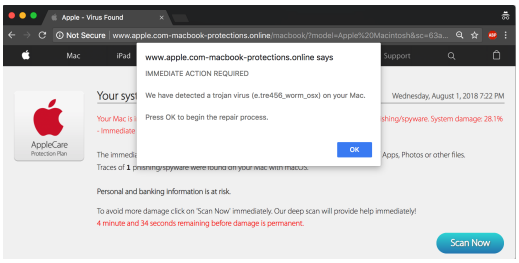
Besides monetization, we find that click interception can lead a user to visit malicious contents. In particular, we were directed to some fake anti-virus (AV) software and drive-by download pages when we manually examined some of the click interception URLs.

For instance, we were forced to visit an ad click URL by the script https://pndelfast.com/riYfAyTH5nYD/4869 on the website https://torrentcounter.to/. Since the navigation URL is an ad click URL, the landing URL is random each time we visit. Nonetheless, one landing URL we visited is a fake AV website, as shown in Figure 3(a). This website showed some fake warnings about virus infection with alarm to fool the user into clicking the *Scan Now* button. After that, it displayed some scanning animation and finally generated a fake scan report to trick the user into installing the fake AV software, as shown in Figure 3(b). The Google search results of the domain *1bcde.com* also suggest it is a malicious redirect website.

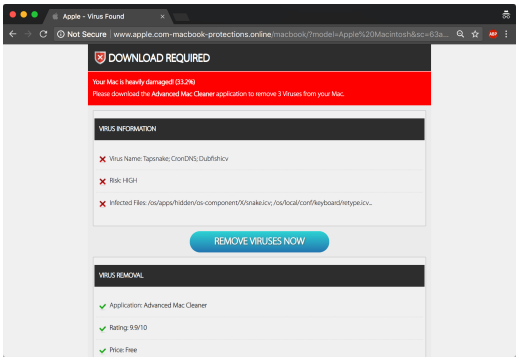
We also find that the script http://cdn.adf.ly/js/link-converter.js converted one link of the website http://magazinweb.net/ into http://ay.gy/2155800/..., which is an advertising link. It once took our browser to a drive-by download page, as shown in Figure 2. When we visited the page, our browser automatically started downloading the MacKeeper installer, which is considered as scamware [20]. The page even shows detailed instruction to trick the user into installing this scamware.

These are just two of many malicious examples we have encountered in our manual investigation. We think that there were much more malicious cases that we have yet to discover. Unfortunately, manually verifying all the 2 million URLs in our dataset is infeasible. We plan to leverage automated URL scanning techniques to automatically detect the malicious URLs associated with click interception in the future.

Summary. We identify that many third-party scripts intercept user clicks to monetize user clicks. In particular, they intercept real user clicks to fabricate ad clicks as a new form of committing ad click fraud. Further, the landing URLs that they trick the users into visiting can be malicious.



(a)



(b)

Figure 3: A fake AV website visited because of click interception.

6 Discussion and Future Work

We discuss the limitations of our work, the possible mitigation of the click interception threat, and our future work.

Third-party Script Detection. Our methodology for distinguishing first-party scripts from third-party scripts is not 100% accurate. First, the domain substring matching can be problematic if an adversary can create victim-specific subdomains. For example, a third-party can intentionally generate a subdomain xyz.third-party.org by adding a new entry in its name server. Our technique would mislabel this subdomain as a first-party URL if it is included by xyz.com. Second, an organization may use distinct email addresses for its subsidiaries. For instance, the SOA email address of https://www.instagram.com/ is *awsdns-hostmaster@amazon.com*, whereas that of https://www.facebook.net/ is *dns@facebook.com*. We classify scripts loaded directly from Facebook on Instagram as third-party scripts even though Instagram is owned by Facebook. Although our approach to determining the relationship between two hosts is not complete, it is good enough for achieving our goal and provides better results compared with a similar approach using only whois records [4].

Measurement Scope. We visited only the main pages of Alexa top 250K websites, so we could miss scripts that are loaded only in their sub pages. However, our goal is to have a preliminary understanding of the click interception problem. We do not intend to and are not able to cover all pages and scripts that can be found on these websites. In the future, we

will consider sub pages of these websites to investigate the differences between the main pages and the sub pages.

Artificial Interaction with Web Pages. OBSERVER applies an artificial way to interact with websites, *i.e.*, using a script to click all the elements on a page, in order to automate the analysis. This could be different from the normal behavior of a real human being. Nevertheless, our goal is to collect as much click-related data as possible in each page visit. It would be an interesting research topic to study if developers would write code to distinguish authentic clicks from automatically generated ones[9].

Generating Security Warnings. Click interception can direct a user to an unknown URL by modifying first-party hyperlinks or hijacking user clicks on first-party elements. It exploits the fact that the user cannot determine the provenance of the URL that he or she is about to visit (unintentionally). To protect a user from visiting potentially attacker-controlled URLs, a possible defense is to provide the user the provenance information regarding each hyperlink and click. In particular, the browser can display a message alongside each hyperlink about its provenance, *e.g.*, if the associated URL is provided by the first-party website or a third party. The additional message needs to be unforgeable and tamper-proof from JavaScript code, such that the adversary cannot manipulate such security-related data. One potential implementation is to utilize the browser UI that is usually not accessible to JavaScript. For example, we can display the message in the status bar when the user hovers the mouse over a link. Similarly, to defend against event-listener interception, we can display an unforgeable warning message if the user hovers over an element that is potentially intercepted by a third-party script. However, this may cause a lot of false positives as an event handler may not necessarily initiate a navigation upon user click. Therefore, it might be better to show such warning when the user actually performs the click, as [10] does. According to our experiment, OBSERVER introduces negligible performance overhead on navigation. It is thus suitable to be extended as a real-time detection tool for the end users. We plan to extend OBSERVER by incorporating these defenses, and conduct a user study to evaluate their effectiveness.

Ensuring Link and Click Integrity. The above defenses require a user to make security decisions, which might not be very effective in practice. Alternatively, we can let the browser automatically enforce integrity policies for hyperlinks and click event handlers. For example, an integrity policy can specify that all first-party hyperlinks shall not be modifiable by third-party JavaScript code. One may further specify that third-party scripts are not allowed to control frame navigations, although listening for user click is still permitted. Enforcing all such policies would effectively prevent click-interception by hyperlinks and event handlers. However, it might also break the functionalities of some third-party components. To give the user and the website administrator better control, the polices can specify the permissions for each script, matched by an absolute URL, a domain name, a wild card, or a secret token, mimicking the Content Security Policy [33]. We plan to develop and evaluate such an integrity protection mechanism as our future work.

[9] The clicks in our experiment were generated through Selenium and are different from those generated using JavaScript, which can be easily detected.

7 Conclusion

We have investigated the click interception problem on the Web with a custom analysis framework developed based on the Chromium browser. We collected data from the Alexa top 250K websites and identified several techniques that can be employed to intercept user clicks. We detected that 437 third-party scripts intercepted user clicks using hyperlinks, event handlers and visual deceptions on 613 websites. We further revealed that many third-party scripts intercept user clicks for monetization via committing ad click fraud. In addition, we demonstrated that click interception can lead victim users to malicious contents. Our research sheds light on an emerging client side threat, and highlights the need to restrict the privilege of third-party JavaScript code.

8 Acknowledgments

The authors thank the anonymous reviewers and our shepherd, Franziska Roesner, for their helpful suggestions and feedback to improve the paper. This material is based on research supported by CUHK under grant 4055081. The views, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily represent the views of CUHK.

References

[1] Devdatta Akhawe, Warren He, Zhiwei Li, Reza Moazzezi, and Dawn Song. Clickjacking Revisited: A Perceptual View of UI Security. In *Proceedings of the 6th USENIX Workshop on Offensive Technologies (WOOT)*, 2014.

[2] Sumayah Alrwais, Christopher Dunn, Minaxi Gupta, Alexandre Gerber, Oliver Spatscheck, and Eric Osterweil. Dissecting Ghost Clicks: A Tale of Ad Fraud Via Misdirected Human Clicks. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2012.

[3] Marco Balduzzi, Manuel Egele, Engin Kirda, Davide Balzarotti, and Christopher Kruegel. A Solution for the Automated Detection of Clickjacking Attacks. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Beijing, China, April 2010.

[4] Frank Cangialosi, Taejoong Chung, David Choffnes, Dave Levin, Bruce M. Maggs, Alan Mislove, and Christo Wilson. Measurement and Analysis of Private Key Sharing in the HTTPS Ecosystem. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, October 2016.

[5] Vacha Dave, Saikat Guha, and Yin Zhang. Measuring and Fingerprinting Click-Spam in Ad Networks. In *Proceedings of the 2012 ACM SIGCOMM*, Helsinki, Finland, August 2012.

[6] Vacha Dave, Saikat Guha, and Yin Zhang. Viceroi: Catching Click-spam in Search Ad Networks. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, October 2013.

[7] Sevtap Duman, Kaan Onarlioglu, Ali Osman Ulusoy, William Robertson, and Engin Kirda. TrueClick: Automatically Distinguishing Trick Banners from Genuine Download Links. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2014.

[8] Google. Expanding user protections on the web. https://blog.chromium.org/2017/11/expanding-user-protections-on-web.html.

[9] Google. Google Ad Traffic Quality. https://www.google.com/ads/adtrafficquality/.

[10] Lin-Shung Huang, Alexander Moshchuk, Helen J Wang, Stuart Schecter, and Collin Jackson. Clickjacking: Attacks and Defenses. In *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, August 2012.

[11] Luca Invernizzi, Stefano Benvenuti, Marco Cova, Paolo Milani Comparetti, Christopher Kruegel, and Giovanni Vigna. EvilSeed: A Guided Approach to Finding Malicious Web Pages. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[12] Ari Juels, Sid Stamm, and Markus Jakobsson. Combating Click Fraud via Premium Clicks. In *Proceedings of the 16th USENIX Security Symposium (Security)*, Boston, MA, August 2007.

[13] Alexandros Kapravelos, Yan Shoshitaishvili, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Revolver: An Automated Approach to the Detection of Evasive Web-based Malware. In *Proceedings of the 22nd USENIX Security Symposium (Security)*, Washington, DC, August 2013.

[14] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February–March 2017.

[15] Sebastian Lekies, Mario Heiderich, Dennis Appelt, Thorsten Holz, and Martin Johns. On the Fragility and Limitations of Current Browser-Provided Clickjacking Protection Schemes. In *Proceedings of the 6th USENIX Workshop on Offensive Technologies (WOOT)*, 2012.

[16] Zhou Li, Kehuan Zhang, Yinglian Xie, Fang Yu, and XiaoFeng Wang. Knowing Your Enemy: Understanding and Detecting Malicious Web Advertising. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, October 2012.

[17] Bin Liu, Suman Nath, Ramesh Govindan, and Jie Liu. DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Seattle, WA, March 2014.

[18] Ritu Lohtia, Naveen Donthu, and Edmund K Hershberger. The Impact of Content and Design Elements on Banner Advertising Click-through Rates. *Journal of Advertising Research*, 43(4):410–418, 2003.

[19] Malwaretips. How to remove Web Browser Redirect Virus (Windows Help Guide). https://malwaretips.com/blogs/remove-browser-redirect-virus/.

[20] Mike Matthews. What MacKeeper is and why you should remove it from your Mac, 2018. https://www.imore.com/removing-mackeeper-your-mac.

[21] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. DETECTIVES: DETEcting Coalition hiT Inflation attacks in adVertising nEtworks Streams. In *Proceedings of the 16th International Conference on World Wide Web (WWW)*, 2007.

[22] Brad Miller, Paul Pearce, Chris Grier, Christian Kreibich, and Vern Paxson. What's Clicking What? Techniques and Innovations of Today's Clickbots. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2011.

[23] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, October 2012.

[24] Nick Nikiforakis, Federico Maggi, Gianluca Stringhini, M Zubair Rafique, Wouter Joosen, Christopher Kruegel, Frank Piessens, Giovanni Vigna, and Stefano Zanero. Stranger Danger: Exploring the Ecosystem of Ad-based URL Shortening Services. In *Proceedings of the 21st International World Wide Web Conference (WWW)*, Seoul, Korea, April 2011.

[25] Erlend Oftedal. Retire.js: What your require you must also retire. https://retirejs.github.io/retire.js/.

[26] OWASP. Clickjacking. https://www.owasp.org/index.php/Clickjacking.

[27] Paul Pearce, Vacha Dave, Chris Grier, Kirill Levchenko, Saikat Guha, Damon McCoy, Vern Paxson, Stefan Savage, and Geoffrey M. Voelker. Characterizing Large-Scale Click Fraud in ZeroAccess. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, Scottsdale, Arizona, November 2014.

[28] M. Zubair Rafique, Tom Van Goethem, Wouter Joosen, Christophe Huygens, and Nick Nikiforakis. It's Free for a Reason: Exploring the Ecosystem of Free Live Streaming Services. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[29] David Ross and Tobias Gondrom. HTTP Header Field X-Frame-Options. Technical report, 2013.

[30] Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson. Busting Frame Busting: a Study of Clickjacking Vulnerabilities at Popular Sites. In *Proceedings of the IEEE Web 2.0 Security and Privacy (W2SP)*, 2010.

[31] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the Web with Content Security Policy. In *Proceedings of the 19th International World Wide Web Conference (WWW)*, Raleigh, NC, April 2010.

[32] Kurt Thomas, Elie Bursztein, Chris Grier, Grant Ho, Nav Jagpal, Alexandros Kapravelos, Damon McCoy, Antonio Nappa, Vern Paxson, Paul Pearce, Niels Provos, and Moheeb Abu Rajab. Ad Injection at Scale: Assessing Deceptive Advertisement Modifications. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[33] W3C. Content Security Policy Level 3. https://www.w3.org/TR/CSP3/.

[34] Wikipedia. List of DNS record types. https://en.wikipedia.org/wiki/List_of_DNS_record_types#NS.

[35] Wikipedia. List of managed DNS providers. https://en.wikipedia.org/wiki/List_of_managed_DNS_providers.

[36] Wikipedia. SOA record. https://en.wikipedia.org/wiki/SOA_record.

[37] Xinyu Xing, Wei Meng, Byoungyoung Lee, Udi Weinsberg, Anmol Sheth, Roberto Perdisci, and Wenke Lee. Understanding Malvertising Through Ad-Injecting Browser Extensions. In *Proceedings of the 24th International World Wide Web Conference (WWW)*, Florence, Italy, May 2015.

[38] Haitao Xu, Daiping Liu, Aaron Koehl, Haining Wang, and Angelos Stavrou. Click Fraud Detection on the Advertiser Side. In *Proceedings of the 19th European Symposium on Research in Computer Security (ESORICS)*, Wroclaw, Poland, September 2014.

[39] Apostolis Zarras, Alexandros Kapravelos, Gianluca Stringhini, Thorsten Holz, Christopher Kruegel, and Giovanni Vigna. The Dark Alleys of Madison Avenue: Understanding Malicious Advertisements. In *Proceedings of the 2014 Conference on Internet Measurement Conference (IMC)*, 2014.

[40] Yuchen Zhou and David Evans. Understanding and Monitoring Embedded Web Scripts. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

What Are You Searching For?
A Remote Keylogging Attack on Search Engine Autocomplete

John V. Monaco
Naval Postgraduate School, Monterey, CA

Abstract

Many search engines have an autocomplete feature that presents a list of suggested queries to the user as they type. Autocomplete induces network traffic from the client upon changes to the query in a web page. We describe a remote keylogging attack on search engine autocomplete. The attack integrates information leaked by three independent sources: the timing of keystrokes manifested in packet inter-arrival times, percent-encoded Space characters in a URL, and the static Huffman code used in HTTP2 header compression. While each source is a relatively weak predictor in its own right, combined, and by leveraging the relatively low entropy of English language, up to 15% of search queries are identified among a list of 50 hypothesis queries generated from a dictionary with over 12k words. The attack succeeds despite network traffic being encrypted. We demonstrate the attack on two popular search engines and discuss some countermeasures to mitigate attack success.

1 Introduction

Search queries contain sensitive information about individuals, such as political preferences, medical conditions, and personally identifiable information [7, 25]. They can reveal user demographics, hobbies, and interests, and are routinely used for targeted advertising [4, 24]. To protect user privacy, all major search engines now encrypt search query traffic.

Autocomplete is a feature that provides suggested queries to the user as they type based on the partially completed query, trending topics, and the user's search history [2]. Intended to enable the user to find information faster, autocomplete requires the user's client to communicate with the server as keyboard input events are detected. As a result, the user's keystrokes manifest in network traffic.

We present a remote keylogging attack on websites that implement autocomplete. The attack detects keystrokes in encrypted network traffic and identifies search queries using information from three independent sources: keystroke timings manifested in packet inter-arrival times, percent-encoding of Space characters in a URL, and the static Huffman code used in HTTP2 header compression.

The attack we developed, called KREEP (Keystroke Recognition and Entropy Elimination Program), consists of five stages: keystroke detection, which separates packets that correspond to keystrokes from background traffic; tokenization to delineate words in the packet sequence; dictionary pruning, which uses an HTTP2 header compression side channel to eliminate words from a large dictionary; word identification, performed by a neural network that predicts word probabilities from packet inter-arrival times; and a beam search, which generates hypothesis queries using a language model. KREEP is a remote passive attack that operates entirely on encrypted network traffic.

Autocomplete has been incorporated into almost every major search engine. We demonstrate our attack on two popular search engines and evaluate its performing using a collected dataset of 16k search queries. Using a dictionary with over 12k words, KREEP identifies 15% of queries and recovers up to 60% of the query text among a list of 50 hypothesis queries. The attack is robust to packet delay variation (PDV). We simulate up to ±32ms of network noise and find relatively little loss in performance with moderate levels of PDV. However, the attack is not robust to padding and we propose a simple padding defense that mitigates both the HTTP2 header compression side channel and ability to delineate words.

To summarize, the main contributions of this work include:

1) *A method to detect packets induced by autocomplete and delineate words in a query.* The pattern of autocomplete packet sizes from each search engine is characterized by a deterministic finite automaton (DFA). We generalize the longest increasing subsequence problem, which has an efficient dynamic programming solution, to that of finding the longest subsequence accepted by the DFA. This approach can detect keystrokes in network traffic with near-perfect accuracy and delineate words with greater than 90% accuracy.

2) *A side channel attack that leverages the static Huffman code used in HPACK, the HTTP2 header compression for-*

mat. Previously, it was shown that HPACK leaked relatively little information through compressed size [50]. However, with autocomplete, a search query is built up incrementally one character at a time and then recompressed. Due to this incremental compression, the information leaked is more than previously thought. We describe a method to leverage this information leakage to prune a dictionary, which increases the accuracy of our remote keylogging attack.

3) *A neural network that identifies words from keystroke timings.* We define a neural network architecture that takes into account the preceding and succeeding context of each observed timing and a method to identify words from a dictionary containing over 12k entries. The network is trained on keystrokes recorded from 83k typists, and words are correctly identified with 19% accuracy.

4) *The integration of a language model and keystroke timing attack to leverage the relatively low entropy of English language.* Previous keystroke timing attacks have noted the relatively low entropy of natural language compared to password input [47]. We introduce a method that combines a keystroke timing attack with a language model to generate hypothesis search queries. The use of a language model significantly improves performance.

In the next section, we provide background information on keylogging side channels and autocomplete. The attack workflow and threat model are described in Section 3, followed by keystroke detection and tokenization in Section 4. Dictionary pruning and the HTTP2 header compression side channel are described in Section 5. Word identification from timings and the language model are described in Section 6. Sections 7 and 8 contain results and discussion, respectively, and Section 9 concludes.

2 Background

2.1 Keylogging side channels

A keylogging side channel attack aims to recover the keystrokes of a victim through unintended information leakage. Such attacks have been demonstrated for a wide range of modalities such as acoustics [5], seismic activity [31], hand motion [54], and spikes in CPU load [46]. These generally fall into two different categories: spatial attacks, which utilize a channel that leaks spatial information about where a key is located on the keyboard, and temporal attacks, which utilize a channel that leaks only the timing of the keyboard events [34]. Our attack leverages both spatial and temporal information leaked through network traffic generated by a website with autocomplete.

Temporal keylogging attacks attempt to recognize which keys a user typed based only on the key press and release timings. This is possible because different key sequences can result in characteristic time intervals, such as typing the key sequence “th” quicker than “aq”. Consequently, the exposure of keyboard event timings is a threat to user privacy. Remote keystroke timing attacks may target applications in which a keystroke induces network traffic from the victim’s host, such as SSH [47] or a search engine with autocomplete functionality [51]. Packet inter-arrival times, when observed remotely, reveal the time between successive keystrokes. Keyboard input events can also be detected from within a sandboxed environment on the host [46] or on a multi-user system [59].

Keylogging attacks can be characterized by the type of input that occurs. For password input, an attack may assume that each key has an equal probability of occurrence, i.e., maximum entropy, whereas for natural language it is often assumed that the user typed a word contained in a dictionary [29]. For the purpose of identifying search queries, we assume natural language input which enables KREEP to leverage a language model in generating hypothesis queries.

Two main problems arise when trying to determine keystrokes from timings. The first is keystroke detection: given a sequence of events, such as network packets, spikes in CPU load, or memory accesses, determine which events correspond to keystrokes and which do not. This is a binary classification problem. In our attack, we consider a sequence of network packets emitted by the victim which includes background traffic in addition to the HTTP requests induced by autocomplete. The second problem is key identification: given that a key press has occurred, the attacker must determine which key it was. This is a multi-class classification problem. In our attack, we assume that each key is either an English alphabetic character (A-Z) or the Space key, for a total of 27 keys.

We address the problems of keystroke detection and key identification separately. KREEP detects keystrokes by finding a subsequence of packet sizes that are characteristic of autocomplete requests. For key identification, KREEP leverages both packet size and packet inter-arrival timings, which faithfully preserve key-press latency.

2.2 Web search autocomplete

Many websites have autocomplete functionality. With this feature, a list of suggested search queries is presented to the user as they enter text into a search form. The list of suggested queries is determined by an algorithm based on the user’s search history, current trending topics, and geographic location [2]. Because the suggestions are automated, this can sometimes result in unfavorable associations implied between search terms which has made autocomplete the focus of several legal disputes [27].

As changes to the query are detected, the client sends an HTTP GET request to the server and the server responds with a list of suggested search queries [26]. This results in a series of HTTP requests following keyboard events, such as those shown in Figure 1. The request contains the partially completed query in addition to other parameters, such as an

Google	Size	URL
	163	?q=**t**&cp=1&...
	164	?q=**th**&cp=2&...
	164	?q=**the**&cp=3&...
	166	?q=**the%20**&cp=4&...
	167	?q=**the%20l**&cp=5&...
	168	?q=**the%20la**&cp=6&...
	169	?q=**the%20laz**&cp=7&...
	170	?q=**the%20lazy**&cp=8&...
	172	?q=**the%20lazy%20**&cp=9&...
	173	?q=**the%20lazy%20d**&cp=10&...
	173	?q=**the%20lazy%20do**&cp=11&...
	174	?q=**the%20lazy%20dog**&cp=12&...

Baidu	Size	URL
	661	?wd=**t**&csor=1&...
	668	?wd=**th**&csor=2&pwd=t&...
	670	?wd=**the**&csor=3&pwd=th&...
	674	?wd=**the%20**&csor=4&pwd=the&...
	678	?wd=**the%20l**&csor=5&pwd=the%20&...
	680	?wd=**the%20la**&csor=6&pwd=the%20l&...
	682	?wd=**the%20laz**&csor=7&pwd=the%20la&...
	684	?wd=**the%20lazy**&csor=8&pwd=the%20laz&...
	688	?wd=**the%20lazy%20**&csor=9&pwd=the%20lazy&...
	693	?wd=**the%20lazy%20d**&csor=10&pwd=the%20lazy%20&...
	695	?wd=**the%20lazy%20do**&csor=11&pwd=the%20lazy%20d&...
	697	?wd=**the%20lazy%20dog**&csor=12&pwd=the%20lazy%20do&...

Figure 1: Autocomplete requests for the query "the lazy dog" in Google (left) and Baidu (right). After each key press, the client sends an HTTP GET request that contains the partially completed query in the URL (shown in bold). Packet size is in bytes.

authentication token and page load options, which generally do not change between successive requests. As a result, each request changes by only a single character, and the size of each packet increases by about 1 byte over the previous.

There are primarily two methods to implement autocomplete [35]. The first is a polling model in which a web page periodically checks the contents of the query input field at fixed intervals. When a change is detected, an autocomplete request is sent to the server to retrieve the query suggestions. Depending on the polling rate and the speed of the typist, an autocomplete request may not immediately follow every keystroke. If two keystrokes occur before the polling timer expires, then they will both be included in the next autocomplete request. In this situation when the typing rate exceeds the polling rate, the keyboard input event times are not faithfully preserved in packet inter-arrival times due to multiple keys being merged into a single request.

The second method of implementing autocomplete is a callback model in which the requests are triggered by HTML DOM `keydown` or `keyup` input events. In this approach, each autocomplete request immediately follows each input event such that the packet inter-arrival times faithfully preserve the time between keyboard events. Non-printable characters, such as Shift, Ctrl, and Alt, are ignored since these alone do not result in visible changes to the query.

We focus only on search engines that implement autocomplete requests triggered by `keydown` events. This results in packet inter-arrival times that are highly correlated with key-press latencies, i.e., time between successive `keydown` events. Previously, we determined that Bing implements a polling model with 100 ms timer, DuckDuckGo implements a callback model triggered by `keyup` events, and Baidu, Google, and Yandex implement a callback model triggered by `keydown` events [35]. Because Yandex is not vulnerable to the method of tokenization described in Section 4, we consider only search engines Google and Baidu. As of January, 2019, Google search comprises over 90% of worldwide market share [49], and Baidu comprises over 70% of the market share within China [48].

3 Attack overview

In this section, we define the threat model and describe the attack workflow. We then summarize the performance metrics used to evaluate each component of KREEP separately as well as overall attack success.

3.1 Threat model

We assume a remote passive adversary who can capture encrypted network traffic emitted by a victim using a search engine with autocomplete. We do not make any assumptions about background traffic or the ability to detect when a web page loaded; KREEP is able to isolate the subsequence of packets that contain autocomplete requests.

We assume the victim types only alphabetic keys and the Space key (27 keys total) to form a query made of lowercase English words with each word separated by a Space. This excludes queries that were copied and pasted, the use of Backspace and Delete keys, and any other input that might cause the cursor to change position, such as arrow keys. The victim might select an autocomplete suggestion before typing a complete query; KREEP can identify the query up to the point a selection was made.

The query must contain words in a large English dictionary known to the attacker. We use a dictionary of over 12k words comprised of the 10k most common English words [32] together with English words that appear in the Enron email corpus and English gigaword newswire corpus [19] (used to simulate search queries, see Section 7.1 for dataset details). KREEP does not require any labeled data from the victim for the keystroke timing attack; the neural network that performs

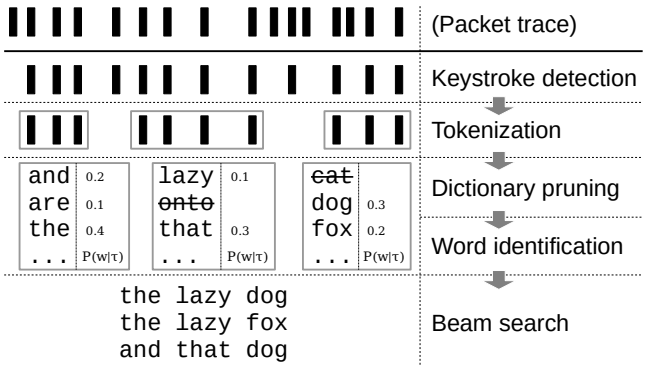


Figure 2: Attack workflow. Input to KREEP is a packet trace containing autocomplete and background traffic; output is a list of hypothesis search queries. Each component provides input to the next. See text for component definitions.

word identification is trained on an independent dataset. We assume that the attacker has access to this dataset.

3.2 Workflow

Our attack consists of five stages applied in a pipeline architecture shown in Figure 2 and summarized below.

Keystroke detection: packets that correspond to keyboard events are first detected from the full packet trace. This is a binary classification problem where each packet is labeled as either key-press or non-key-press. Each autocomplete request contains the query typed up to that point, so the sequence of autocomplete packet sizes has approximate linear growth over time. This makes it possible to separate keystrokes from background traffic, described in Section 4.2.

Tokenization: from the detected subsequence of packets, words are delineated based on packet size differences. Tokenization is also a binary classification problem where each packet is labeled as either Space or non-Space. Space characters in a URL are encoded by a three-byte escape sequence whereas other characters occupy a single byte. This behavior enables tokenization, described in Section 4.3.

Dictionary pruning: packet size differences are compared to a dictionary to eliminate words that could not have resulted in the observed sequence. This effectively prunes the hypothesis query search space. Dictionary pruning is possible due to the static Huffman code in HTTP2 header compression. This side channel is described in Section 5.

Word identification: the probability of each word remaining in the dictionary is determined from the observed packet inter-arrival times, which faithfully preserve key-press latencies. Word identification is performed by a neural network described in Section 6.1.

Beam search: word probabilities are combined with a language model in a beam search that generates hypothesis queries. The number of hypothesis queries is controlled by the beam width. The beam search is described in Section 6.2.

3.3 Performance metrics

We measure the performance of each component of the attack separately as well as overall attack success.

Both keystroke detection and tokenization are binary classification problems. For keystroke detection, a false positive occurs when a packet is incorrectly labeled as an autocomplete request, and a false negative occurs when an autocomplete request packet is missed. Likewise, a tokenization false positive occurs when a letter is incorrectly labeled as a Space, and false negative occurs when a Space is missed. Let fp, fn be the number of false positives and false negatives, and let tn, tp be the number of true negatives and true positives, respectively. We measure the performance of both keystroke detection and tokenization by the F-score,

$$F_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{1}$$

where

$$\text{Precision} = \frac{tp}{tp+fp}, \quad \text{Recall} = \frac{tp}{tp+fn}. \tag{2}$$

The F-score varies between 0, for missing all positives, and 1, for perfect precision and recall. Both keystroke detection and tokenization provide input to later stages of the attack, the success of which critically depends on performing well at both these tasks. As demonstrated in Section 7.2, making these tasks more difficult significantly reduces overall performance.

The utility of dictionary pruning is measured by the information gain due to incremental HTTP2 header compression. We compare this to the information gain in a classical compression side channel where only the total compressed size of the query is known.

For word identification, we report the word classification accuracy from packet timings, assuming perfect detection and tokenization. This evaluates word identification separately from the other components.

We consider two metrics to measure overall attack success. First is the rate at which a query is correctly identified among the list of hypothesis queries. Using a beam width of 50, this corresponds to a top-50 classification accuracy. Since the hypotheses may contain queries that are close, but do not exactly match, the true query, we also consider the Levenshtein edit distance between the true and hypothesis queries. Edit distance is used instead of character or word classification accuracy since failures in keystroke detection can result in a predicted query that is either shorter or longer than the original query. This metric is thought to better reflect the overall performance of a keylogging attack in such cases [16]. We report the minimum edit distance among the hypotheses to the true query, which roughly corresponds to the maximum proportion of keys that are correctly identified.

4 Keystroke detection and tokenization

In this section, we characterize the network traffic emitted by autocomplete in two different search engines. We then describe the first two stages of attack: a method to detect packets that contain autocomplete requests and a method to delineate words in the query. Both stages leverage characteristics of autocomplete packet sizes.

4.1 Autocomplete packet sizes

The problem of keystroke detection involves deciding whether each captured packet was induced by a keyboard event or not. As the user types a query into a search engine with autocomplete, the client emits HTTP requests that contain the partially completed query, such as those shown in Figure 1. However, these are mixed together with requests to load page assets, such as HTML and CSS files, AJAX requests supporting dynamic web content, and other background traffic. We found that typing a query with 12 characters on Google search induces 95 outgoing packets with payload greater than 0 bytes (436 packets including those with empty payloads), only 12 of which correspond to autocomplete requests.

Each autocomplete request contains a new character appended to the URL path. As a result, the sequence of packet sizes is monotonically increasing, shown in Figure 1. We perform keystroke detection by isolating a subsequence of packets that exhibit this pattern, taking into account the particular behavior of each search engine described below.

The behavior of each search engine is characterized by the sequence of size differences between successive autocomplete request packets. That is, let s_i be the size in bytes of the ith autocomplete request and s_0 the size of the first request. Packet size differences are given by $d_i = s_i - s_{i-1}$ for $i > 0$. This sequence reflects packet size growth as a function of query length, invariant to the size of other parameters contained in the request which vary across hosts due to different sized identifiers, authentication tokens, and page load options. However, these parameters typically remained unchanged in successive autocomplete requests from a single host.

Figure 3 shows the distribution of d_i as a function of query length for both Google and Baidu. From this figure and a manual inspection of several HTTP request packets, we make several observations about the behavior of each search engine. We then use these observations to build a DFA that accepts a sequence of autocomplete packet size differences.

Google autocomplete emits packets that typically increase by between 0 and 3 bytes. As each new character is appended to the "q=" parameter of the URL, the size increases by about a byte. The 2 and 3 byte increases correspond to the addition of percent-encoded characters in the URL, described in Section 4.3. The 0 byte increases are an artifact of HTTP2 header compression, described in Section 5.1. A larger increase of approximately 20 bytes occurs after about 12 requests. At

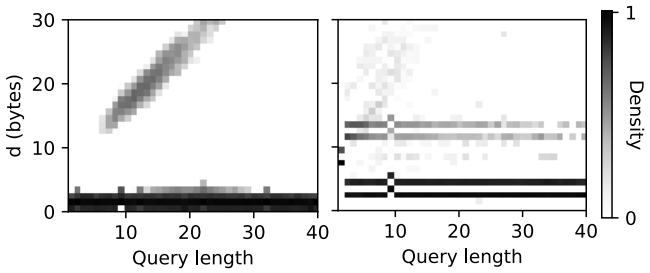


Figure 3: Density of packet size difference between successive autocomplete requests for Google (left) and Baidu (right).

this point, an additional "gs_mss" parameter with the partially completed query is added to the URL. This results in a sudden increase of about 20 bytes: 8 bytes for "&gs_mss=" and 12 bytes for the query. The request then continues to increase by about 1 byte per character thereafter.

The autocomplete packet sizes of Baidu typically increase by either 2 or 4 bytes per character, with a larger increase of 7 or 9 bytes at the beginning of the sequence. After the first request, an additional parameter "pwd=" referring to the previous query is appended to the URL. For example, if the user types "th", the first request will contain "wd=t" followed by "wd=th&pwd=t", resulting in a 7 byte increase (6 bytes for "&pwd=t" and 1 byte for "h"). A 4 byte increase corresponds to the addition of escaped characters in the URL, which occupy 3 bytes. Baidu requests also occasionally include a new cookie not present in previous requests, resulting in a larger increase of either 11 or 13 bytes.

Both search engines include a parameter that keeps track of the request number. In Google, this parameter is "cp=", where "cp" increments with each request (see Figure 1), and Baidu uses the "csor=" parameter. On the 10th request, "cp=9" becomes "cp=10", resulting in an additional 1 byte increase.

4.2 Keystroke detection

Since autocomplete request size is monotonically increasing, keystroke detection could be performed by finding the longest increasing subsequence (LIS) of packet sizes which has an efficient solution through dynamic programming [44]. However, the LIS fails to capture the fact that packets typically increase by a fixed amount and that two successive packets may be the same size due to HTTP2 header compression. To that end, we generalize the LIS problem to that of finding the longest subsequence accepted by a sequence detector DFA based on observations in the previous section.

We define a DFA that accepts a sequence of packet size differences generated by the autocomplete of each search engine. The DFA for Google autocomplete packets is shown in Figure 4, where edges denote a constraint on d that must be met to traverse to the next state. States a and b correspond to

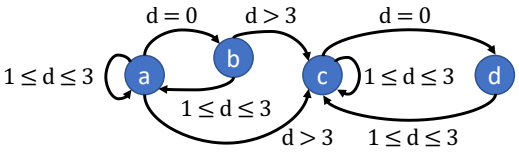


Figure 4: DFA that accepts a sequence of packet size differences generated by autocomplete in Google search.

increases of between 0 and 3 bytes prior to the large increase from the addition of the "gs_mss" parameter, and states c and d are reached after the large increase. The absence of a recurrent connection on states b and d indicate that two consecutive non-increases cannot occur. This DFA takes as input a sequence of packet size differences, and if at any point an unreachable state is met, it rejects the sequence.

Let the longest automaton subsequence (LAS) be the longest subsequence accepted by the DFA. Keystrokes are detected by finding the LAS in the sequence of packet sizes. The LAS is determined efficiently through dynamic programming in a similar manner to that of the LIS problem. Let F be an acceptor DFA and L_i the longest subsequence accepted by F ending in the ith packet. Assume the LAS ending in element L_i must necessarily be part of the solution if it contains L_i (optimal substructure). Then L_i need only be computed once and may be considered as the prefix to any other subsequence L_j where $j > i$ (overlapping subproblems). We then need only check if the DFA that accepted the sequence ending in packet i can transition to packet j. Note that in general, these assumptions may not hold and thus the dynamic programming solution might be suboptimal; however, we found this method to work well in practice and leave for future work a formal treatment of the LAS problem.

4.3 Tokenization

Tokenization is the process of delineating words in the sequence of autocomplete requests. Since we assume the search query to be made of English words separated by a Space, this enables the following stages of attack (dictionary pruning, word identification, and beam search) to be conducted at the word level. Like detection, tokenization is a binary classification problem since each packet may be labeled as either a delimiter or part of a word. We consider the Space character as the only delimiter between words.

Percent-encoding is an escape sequence used to represent a character in a URL that is outside the set of allowable characters [8]. A percent-encoded sequence consists of three ASCII characters, "%" followed by two hexadecimal digits. The Space character (ASCII=32) in a URL has percent-encoding "%20". When the user types a Space into the search query field, this escape sequence is appended to the URL causing the uncompressed request packet to increase by 3 bytes.

Google autocomplete packets increase by 2 bytes when the Space key is pressed as a result of HTTP2 header compression. The Huffman code for characters "%", "2", and "0" have bit lengths 6, 5, and 5 respectively, and the sequence "%20" has a total compressed bit length of 16 bits. Tokenization is performed by marking packets that increase by 2 bytes as word boundaries.

Baidu does not use HTTP2, so the escape sequence "%20" occupies 3 bytes. However, since the previous query is included in each request, when a Space is pressed the packet size increases by 4 bytes: 1 for the new character appended to the "pwd" parameter, and 3 for "%20" appended to the "wd" parameter. This also occurs twice in a row since when another letter key is pressed following the Space, "%20" is then appended to the "pwd" parameter. For example, see the URL and sizes of the third and fourth packets in Figure 1 (right), which demonstrate two consecutive 4 byte increases. Tokenization of Baidu queries is achieved by detecting the first of any two consecutive 4 byte increases.

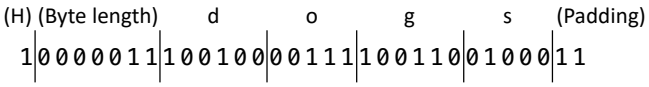


Figure 5: Huffman encoded string literal "dogs" in HPACK.

5 Dictionary pruning

We describe a side channel that leverages the static Huffman code used in HTTP2 header compression. This enables pruning the dictionary, but is only applicable to Google which supports HTTP2. Baidu does not currently support HTTP2.

5.1 Incremental compression side channel

HPACK is the HTTP2 header compression format, which uses a static Huffman code to encode string literals [39]. A Huffman code is a near-optimal lossless compression scheme. Symbols are encoded by bit string with length based on the frequency of the symbol. Huffman codes are prefix-free, such that the code for a symbol is not the prefix to any other. The encoded string becomes the concatenation of all encoded symbols, avoiding the need for symbol delimiters. In HPACK, the encoded string is padded with between 0 and 7 bits to align with the nearest octet boundary.

The static Huffman code in HPACK was determined using a large sample of HTTP headers, and all HPACK implementations must use the same Huffman code defined in the specification [39]. The sizes of lowercase letters range from 5 bits for frequently used characters, such as "e" and "t", to 7 bits for infrequent characters, such as "j" and "z". As an example, the compressed string literal "dogs" is shown in Figure 5. The encoded symbols occupy 6+5+6+5=22 bits, which is then padded with 2 bits for a total size of 3 bytes.

It was previously determined that size alone does not leak a considerable amount of information in HPACK [50]. Let h_i be the bit length of the ith symbol in a string as specified by the static Huffman code and $b = \sum h_i$ the total bit length of the compressed string. The size b reveals only that the string must be some linear combination of encoded symbols to achieve the same compressed size. For example, the string "fish" has compressed length 6+5+5+6=22 bits, exactly the same compressed size as "dogs" in Figure 5.

Less than 0.05 bits per character are revealed in this way, making an HTTP2 compression side channel impractical [50]. This estimate is actually an upper bound since compressed string literals in HPACK are padded to the nearest octet. Instead of b, an adversary observes byte size $B = \frac{p+\sum h_i}{8}$ where $0 \leq p \leq 7$ is an unknown amount of padding to align the compressed bit string with the nearest octet.

However, the query in a sequence of autocomplete requests grows incrementally. Each request contains a single new character appended to the URL path, which then passes through header compression before being sent to the server. We refer to this as *incremental compression*. As a result, instead of total size B, an adversary observes the sequence of cumulative byte sizes $B_1, \ldots, B_n$ of the compressed query after each new character is appended. Due to differences in the size of each symbol, different words grow at different rates and the cumulative byte size sequence can reveal the query.

To leverage the information leaked through incremental compression, we compare the observed sequence of cumulative byte sizes to the cumulative sizes of every word contained in the dictionary. The sizes of words in the dictionary are precomputed for each possible amount of padding, which is unknown to the attacker. Words for which the observed sequence never occurs can be eliminated from the dictionary.

Let d_i be the observed size increase in bytes of the ith request packet and let $B_j = \sum_{i \leq j} d_i$ for $1 \leq j \leq n$ be the observed cumulative size up to the nth request. The sequence $B_1, \ldots, B_n$ characterizes the size growth of a word after undergoing incremental compression. The cumulative byte size sequence $B_1^{w,p_0}, \ldots, B_n^{w,p_0}$ is computed for each word w in the dictionary, given by

$$B_j^{w,p_0} = \left\lfloor \frac{p_0 + \sum_{i \leq j} h_i}{8} \right\rfloor \tag{3}$$

where $0 \leq p_0 \leq 7$ is an unknown amount of padding applied to the compressed URL *prior* to the request containing the first character of the word. The observed size sequence $B_1, \ldots, B_n$ is compared to every sequence $B_1^{w,p_0}, \ldots, B_n^{w,p_0}$ in the dictionary to discover potential matches and eliminate words that could not have been typed by the user.

An example is shown in Figure 6 where the user typed a 4 letter word with cumulative byte size $[1,2,3,3]$. Comparing this sequence to the dictionary, there are two potential matches: the observed sequence $[1,2,3,3]$ appears for the word "dogs" with padding $p_0 = 0$ and for the word "guns" with $p_0 = 0$ or $p_0 = 1$. It's therefore possible that the query contains either the word "dogs" or "guns". However, the user definitely did not search for "cats" since the sequence $[1,2,3,3]$ is not attainable for the word "cats" under any padding p_0. The total size alone does not reveal this much information since all words in the dictionary could have the same total compressed size as the query (3 bytes) given some unknown amount of padding.

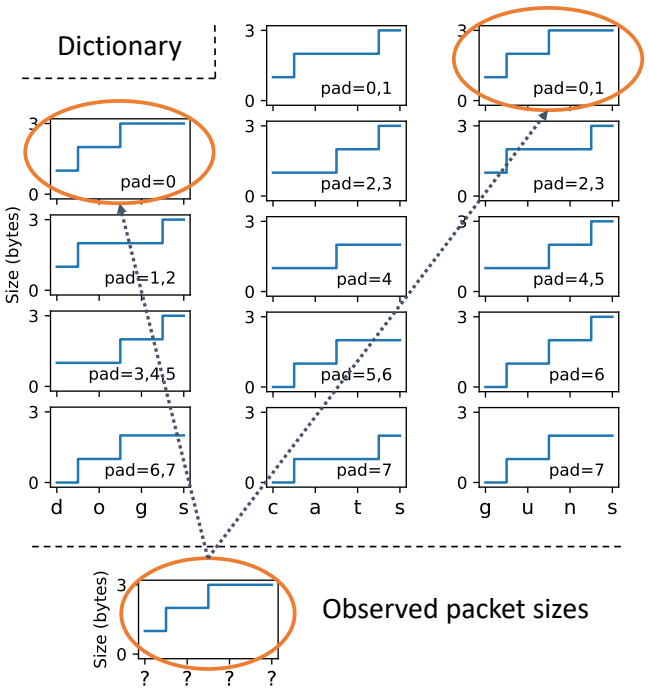


Figure 6: Dictionary pruning. The dictionary contains every possible sequence of cumulative packet size, determined for each word in the dictionary under each unknown prior padding amount (0 to 7 bytes). The cumulative size of an observed query is compared to each sequence in the dictionary. Words that don't have any matches to the observed sequence are eliminated. The observed sequence $[1,2,3,3]$ matches "dogs" with no padding and "guns" with 0 or 1 byte padding; "cats" has no matches and can be safely eliminated.

Note that in general, if $h_i \leq p_{i-1}$, then $d_i = 0$, where p_i is the padding applied after the ith character. That is, when the bit length of a new character is equal to or less than the previous amount of padding used, the packet size will remain the same. Since the lengths of lowercase ASCII characters range from 5 to 7 bits, an increase of at least 1 byte is guaranteed when $p_{i-1} < 5$. It is also never the case that $d_i = 0$ and $d_{i+1} = 0$, i.e., every two consecutive requests must increase by at least one byte.

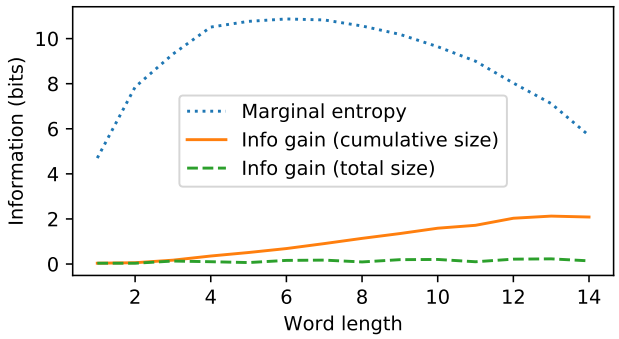


Figure 7: Information gain from an incremental compression side channel, where the cumulative size of a string is exposed, compared to a conventional compression side channel, where only the total size is exposed.

5.2 Pruning and information gain

To measure the impact of this side channel, we determined the expected information gain using a dictionary of 12k common English words and compare this to the information gained from total size alone. Given observed cumulative byte size sequence $\mathbf{B} = B_1, \ldots, B_n$, the probability of each word in the dictionary may be computed by Bayes' formula,

$$P(w|\mathbf{B}) = \frac{P(\mathbf{B}|w)\,P(w)}{P(\mathbf{B})} \tag{4}$$

where $P(\mathbf{B}|w)$ is the probability of sequence $\mathbf{B}$ given word w, $P(w)$ is the marginal probability of word w, and $P(\mathbf{B})$ is the marginal probability the sequence $\mathbf{B}$. Note that multiple byte size sequences could be observed for a particular word depending on the amount of padding used. For example in Figure 6, $P([1,2,3,3]\,|\text{"guns"}) = \frac{2}{8}$ since the sequence $[1,2,3,3]$ is possible for the word "guns" with paddings of 0 and 1 out of 8 possible padding amounts. In the same example, the marginal $P([1,2,3,3]) = \frac{3}{24}$ since the sequence $[1,2,3,3]$ appears 3 times in the dictionary with 24 precomputed sequences (3 words $\times$ 8 padding amounts). Words for which $P(w|\mathbf{B}) > 0$ are retained in the dictionary in the later stages of the attack and words for which $P(w|\mathbf{B}) = 0$ are eliminated.

From $P(w|\mathbf{B})$, the conditional entropy $H(w^n|\mathbf{B})$ is determined for words of length n. Information gain is given by $I(w^n;\mathbf{B}) = H(w^n) - H(w^n|\mathbf{B})$, where $H(w^n)$ is the marginal entropy of words of length n. We assume each word has an equal probability of occurrence, i.e., $H(w^n)$ has maximum entropy. The information gain is shown in Figure 7. Note that information gain from total byte size B is negligible as previously reported [50]. However, the information gain from cumulative size increases for longer words due to the "uniqueness" of the cumulative byte sizes revealed through incremental compression. These gains lead to more accurate query identification.

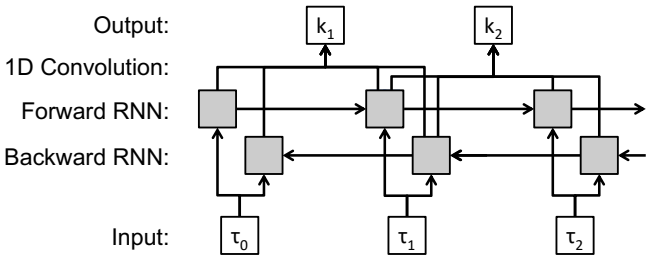


Figure 8: Neural network architecture that predicts n keys from $n+1$ packet inter-arrival times.

6 Word identification and beam search

In the last stages of KREEP, packet inter-arrival timings are used to predict which words the user typed. Word probabilities are determined for the remaining words in the dictionary after pruning, and these probabilities are combined with a language model in a beam search to generate hypothesis queries.

6.1 Word identification from timings

Since each autocomplete request is triggered by a key-press event, packet inter-arrival times faithfully preserve key-press latencies. These latencies are used to predict which keys the user pressed. Unlike previous work which considered either each latency in isolation [47], or words in a limited dictionary [29], we define a model that predicts key probabilities considering their surrounding context and also able to recognize words not seen during training.

We use a three-layer neural network to predict key probabilities. Generally, each word of length n has $n+1$ packet inter-arrival times since a Space precedes the first character and follows the last character. The model takes as input the sequence of latencies τ_i for $0 \le i \le n$ and predicts $P(k_i)$, the probability of each key k_i for $1 \le i \le n$.

The first layer of the network is a bidirectional recurrent neural network (RNN) with gated recurrent units (GRU) that takes as input the sequence of $n+1$ time intervals. The second layer is a 1-dimensional convolutional layer with kernel size 2 and no padding. The convolutional layer reduces the size of the output from $n+1$ to n. The last layer is a dense layer with softmax activation that predicts the probability of each key (26 classes) at each time step. This architecture is shown in Figure 8.

The network architecture was motivated by several factors. The use of a bidirectional RNN ensures that the predictions at key i are made within the context of latencies preceding and following i. The convolutional layer with kernel size 2 combines the latency immediately before and after key i, reducing the size of the sequence from $n+1$ (number of latencies) to n (number of keys). Note that while generally a word of length n has $n+1$ latencies, the first and last words in the query each have n latencies due to missing the leading Space and trailing

Space, respectively. We augment the missing intervals with the mean latency obtained over the entire training dataset.

Word probabilities are determined from the sequence of key probabilities output by the network. The probability of word w is the joint probability of all keys in that word,

$$P(w|\tau) = \prod_{k_i \in w} P(k_i) \tag{5}$$

where τ is the sequence of observed latencies. Making predictions at the key-level and then calculating word probability by the joint key probability has several advantages. First, the number of output classes in the network remains small (26 keys) compared to the number of possible words (over 12k). Second, the probability of any word can be determined whether or not it was contained in the dataset used to train the model. In this way, the dictionary used to generate hypothesis queries is independent of the key identification model.

Finally, learned features may be shared across words. For example, if a particular pattern of latencies is indicative of the sequence "th", the model can learn to recognize "th" in different words such as "the", "there", "beneath", and so on. If instead predictions were made at the word level, these features would have to be learned separately for each word.

6.2 Language model and beam search

In the last stage, word probabilities are combined with a language model to generate hypothesis queries in a beam search.

We assume the query to be a sequence of N words w_i for $1 \leq i \leq N$ and take advantage of the fact that some words are more likely to follow others in natural language. As an example, consider trying to predict an 8-letter word that follows the sequence "recovering from a _". The probability of words such as "sprained" and "fractured" should be relatively higher than other words such as "purchase" and "position".

The use of a language model enables constraints of English language to be leveraged in conjunction with word probabilities from packet timings. A language model estimates the probability of a word given the words that preceded it, denoted by $P(w_i|w_1 \ldots w_{i-1})$. We combine the language model with the keystroke timing model to determine the probability of an entire query $\mathbf{w} = [w_1, \ldots, w_N]$, given by

$$P(\mathbf{w}) = \prod_{w_i \in \mathbf{w}} P(w_i|\tau) P(w_i|w_1 \ldots w_{i-1})^{\alpha} \tag{6}$$

where α is a parameter that controls the weight of the language model. Smaller α places more weight in the packet inter-arrival timings, while larger α places more weight on the language model. In this work, we found α in the range of 0.2 to 0.5 work well and we use $\alpha = 0.2$. The language model is a 5-gram model with Kneser-Ney smoothing [21] trained on the Billion Word corpus [11].

Determining the sequence with maximum *a posterior* probability (MAP) is NP-hard due to the exponential growth of the search space. It is also unlikely that the MAP sequence itself exactly matches the true query. Instead, KREEP generates a list of hypothesis queries using a beam search. Beam search is a breadth-first greedy search algorithm that maintains a list of top candidates (the "beam") as it progresses the search tree.

For each token, all the words in the dictionary are appended to each hypothesis in the beam, which starts with the empty string. This results in a list of $W \times D$ candidates, where W is the beam width and D is the size of the dictionary. The W sequences with highest likelihood are retained, and the rest discarded. This repeats until the last token is reached, at which point the search returns a list of W hypothesis queries. We use a beam width of 50. To measure the performance of KREEP, we determine the rate at which the query is correctly identified among the 50 hypotheses as well as the minimum edit distance in the list of 50 hypotheses to the true query.

7 Results

In this section, we describe our data collection setup and evaluate attack performance. KREEP is first tested under ideal conditions. We then evaluate performance with increasing levels of simulated network noise and propose a simple padding defense to mitigate attack success.

7.1 Data collection

We built a system that captures network traffic while a query is typed into a search engine with autocomplete. The measurement setup consists of a keystroke dataset previously collected from human subjects, browser automation with Selenium WebDriver, and a process to replay keystrokes by writing keyboard events to `/dev/uinput` in real time.

To train the neural network, we used a subset of a publicly available keystroke dataset collected from over 100k users typing excerpts from the Enron email corpus and English gigaword newswire corpus [15]. From this dataset, we retained 83k users with US English locale on either desktop or laptop keyboards and QWERTY keyboard layout.

To simulate search queries, we randomly selected 4k phrases between 1 and 20 words in length containing only letters and the Space key. This selection contains a wide variety of typing speeds, ranging from 1.5 to 22 keys per second. Of the 4k phrases, 3k are unique. They contain a total of 1717 unique words ranging from 1 to 14 characters with an average word size of 6 characters. None of the users in the evaluation data appeared in the dataset used to train the neural network.

Each capture proceeded as follows. The web browser was opened and cookies cleared before starting the capture process (tshark). One second after the capture began, the website was loaded using Selenium. There was then a two second delay before replaying the keystrokes. The keystroke sequence was replayed by writing the sequence of key events to the

	Google		Baidu	
	Chrome	Firefox	Chrome	Firefox
Detect F-score	99.99	99.96	99.62	99.98
Perfect detect rate	99.72	98.70	96.35	99.52
Token F-score	97.26	95.45	96.85	97.33
Perfect token rate	81.12	74.89	86.70	88.30

Table 1: Keystroke detection and tokenization F-scores (%) and rates (%) of achieving perfect accuracy (F-score=100%).

`uinput` device with delays between each event that correspond to the original keystroke sequence. The data collection was performed on an Ubuntu Linux desktop machine with kernel version 4.15 compiled with the CONFIG_NO_HZ=y option, which omits scheduling clock ticks when the CPU is idle [1]. This ensures keyboard event times are replayed with high fidelity and not quantized due to the presence of a global system timer.

We captured 4k unique queries on search engines Google and Baidu, both of which default to an HTTPS connection and generate autocomplete requests upon key-press events. All results were obtained on the encrypted traffic: TLSv1.3 for Google and TLSv1.2 for Baidu. Both sites leak information through the size of the TLS records, which includes the size of the payload plus a fixed amount for the authentication code (GMAC). Thus, TLS preserves differences in payload length, although TLSv1.3 does contain a provision for record padding to hide length [40].

To understand how the browser itself might affect network timings, the data collect was performed in both Chrome (v.71, with QUIC disabled) and Firefox (v.64). The captured dataset contains a total of 16k queries (4k queries $\times$ 2 search engines $\times$ 2 web browsers), obtained over approximately 7 days. During this time, we did not experience any rate limiting. However, a small number of captures did miss some of the outgoing traffic (< 1%). The unsuccessful captures were repeated until success.

7.2 Attack performance

The first step of the attack is to detect keystrokes. Keystroke detection accuracy is reported separately for each website in each browser in Table 1. In both websites and browsers, keystrokes are detected with near perfect accuracy with a high rate of achieving perfect detection. Tokenization F-scores are also shown in Table 1. The rates of achieving perfect tokenization are strictly lower than that of detection since tokenization is applied after detection.

We examined the cases in which tokenization failed. We found that false positives in Google were due mainly to rollover of the `String Length` field in the HPACK header, which specifies the size in bytes of a compressed string. In HPACK, the string length starts as a 7-bit integer (see Figure 5). When the number of compressed bytes exceeds $2^7 - 1$, an additional byte is allocated for the string length, resulting in an overall increase of 2 bytes (+1 from the String Length increase and +1 from the new character in the query). Since it is generally not known where this rollover occurs, we cannot distinguish whether the 2 byte increase was due to `String Length` rollover or the addition of a percent-encoded Space.

Google		Google (no prune)		Baidu	
Chrome	Firefox	Chrome	Firefox	Chrome	Firefox
15.83	15.13	14.20	13.55	12.85	12.63

Table 2: Top-50 classification accuracy: % of queries that are correctly identified among the 50 hypothesis queries.

False negatives in both Google and Baidu were due mainly to larger changes in packet size coinciding with a Space. In Google, this occurs when the "gs_mss" parameter is added to the query in the same request as a Space, and in Baidu, from the inclusion of a cookie that was not previously present. These larger changes (> 10 bytes) mask the change in size due to the Space key (2 or 4 bytes).

Following detection and tokenization, the dictionary is pruned, word probabilities from packet inter-arrival timings are determined, and hypothesis phrases are generated in a beam search. Attack success critically depends on accurate keystroke detection and tokenization. This is because the later stages of the attack assume that word lengths have been correctly identified. If the wrong word lengths have been determined, due either to a failure in detection or tokenization, then the correct query cannot be identified.

This behavior is shown qualitatively in Figure 9. In this example, perfect detection and tokenization result in hypothesis queries that have the correct word lengths and low edit distance to the true query. When either a false negative or false positive detection error occurs, the hypothesis queries will have a different length than the true query. In Figure 9 (middle), the 7th packet (containing the 1st "r" in "recovering") is incorrectly labeled as non-keystroke. As a result, the third word in the hypothesis has 9 letters instead of 10. This results in sequences that have relatively high edit distance to the true query. Tokenization errors have a similar effect in that word lengths in the hypothesis will not match the query. In Figure 9 (right) the 11th packet (containing the 2nd "e" in "recovering") is incorrectly labeled as a Space. The hypothesis queries have the same total length as the true query but differ in word lengths, resulting in relatively high edit distance.

The proportion of attacks in which the true query is identified among the hypotheses queries, analogous to a top-50 classification accuracy, is shown Table 2. We also determined the minimum edit distance for each search engine as a function of query length and compare this to a baseline attack in which the timing and language model probabilities are ignored. Baseline performance is obtained by generating 50 ran-

Perfect detection/tokenization		Keystroke detection false negative		Tokenization false positive	
he is recovering from a sprained	0	to be president from a position	18	is to learn from such a position	23
he is recovering from a strained	1	to be president from a business	17	is to learn from such a purchase	23
he is recovering from a fracture	7	to be president just a fraction	22	is to learn more from a position	20
he is recovering from a position	7	to be president from a possible	18	is to learn from such a pressing	22
he is recovering from a possible	7	to be president from a southern	18	is to learn from such a practice	21

Figure 9: Query hypotheses in three different scenarios: perfect detection and tokenization (left), false negative keystroke detection (center, the 7th packet is missed), and false positive tokenization (right, the 11th packet is labeled as a Space). The edit distance to the true query "he is recovering from a sprained", is shown to the right of each hypothesis.

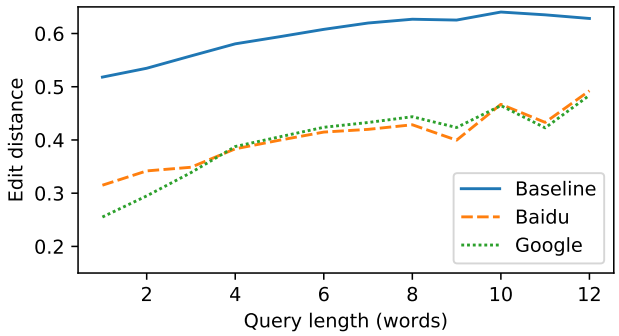


Figure 10: Minimum edit distance (the closest query among 50 hypotheses to the true query) vs query length.

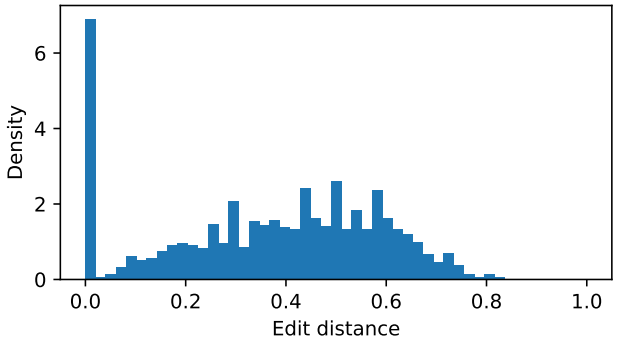


Figure 11: Minimum edit distance distribution. Two modes indicate that KREEP either exactly identifies a query (0 edit distance) or performs near the baseline (0.55 edit distance).

dom hypotheses, choosing dictionary words the same length as the detected tokens. Note that this baseline still uses information gained through keystroke detection and tokenization. These results are shown in Figure 10.

Generally, the difficulty in identifying the query increases with query length. The hypotheses have an average minimum edit distance of 0.37 to the true query. Note that edit distance reduces to Hamming distance for strings of equal length, and perfect detection (F-score of 100%) is achieved in about 98% of queries. Therefore, 0.37 edit distance is roughly a 63% key identification accuracy. We did not find any significant difference in performance across browsers, but did achieve overall higher query identification rates on Google due information leaked through incremental compression.

We found the example in Figure 9 to be representative of attack success which generally had polarized outcomes: the hypotheses were either very similar to or very different from the true query. This behavior is revealed in the distribution of minimum edit distances shown in Figure 11, which has two modes: one occurring near the baseline (0.55, achieved by guessing random words) and the other at 0.

7.3 Information sources

To better understand the relative contribution of each component, we evaluate attack performance ignoring the packet timings, language model probabilities, or header compression. Considering only queries in Google (Baidu does not support HTTP2), performance is evaluated for three scenarios: using only the packet timings (TM only), using timings and the language model (TM+LM), and using both with dictionary pruning applied (TM+LM+Pruning).

These results are shown in Figure 12 with baseline performance as described in the previous section. The largest gains are achieved with the use of packet timings and language model. The neural network alone identifies words with 19.1% accuracy. Incremental gains are then achieved when the dictionary is pruned.

7.4 Effects of network noise

We tested the robustness of the attack to network noise. Since key identification uses packet inter-arrival times, packet delay variation (PDV) can potentially reduce attack success. PDV corresponds to changes in network latency, which can obfuscate the key-press timings in packet inter-arrival times. In this regard, variations in routing delay potentially provide a natural defense to remote keystroke timing attack.

The Laplace distribution has previously been proposed as a model for PDV [60]. We simulate PDV by drawing samples from a Laplace distribution parameterized by the mean absolute deviation (MAD). The simulated PDV is added to

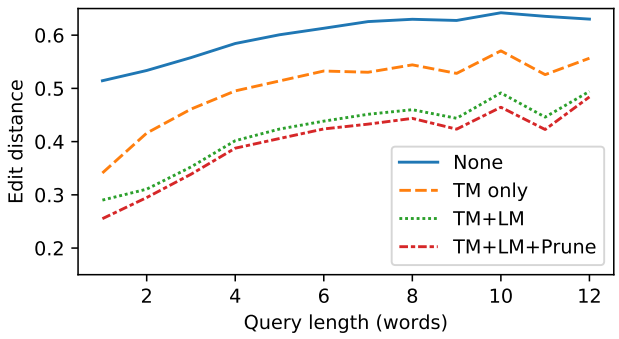


Figure 12: Performance with/without the use of the timing model (TM), language model (LM) and dictionary pruning.

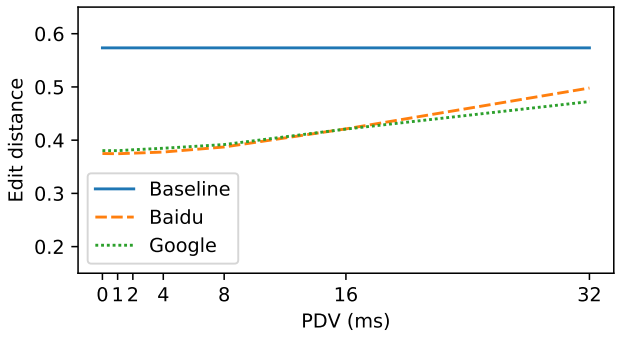


Figure 13: Effects of packet delay variation. Baseline ignores packet timing, uses only packet size to generate hypotheses.

the captured packet times before attempting to identify the query with KREEP. Performance as a function of increasing PDV is shown in Figure 13. The attack is relatively robust to PDV less then 8 ms, but approaches baseline performance with PDV in excess of 32 ms.

7.5 Effects of padding

With the attack being robust to low levels of network noise, we explored other means of mitigating attack success. Query identification critically depends on accurate detection and tokenization, and chances of attack success can be greatly reduced with a simple padding scheme.

We simulate random padding by modifying the captured packet sizes. The size of each autocomplete packet is increased by 1 byte with probability 0.5. The sizes of other packets in the trace remain unchanged such that the padding defense could be implemented entirely in the client side autocomplete logic.

Detect F-score		Token F-score		Min edit distance	
Original	Padded	Original	Padded	Original	Padded
99.89	94.46	96.72	51.23	37.76	61.32

Table 3: Effects of randomly padding packets with 0 or 1 byte.

The effects of this defense nearly double the minimum edit distance, shown in Table 3. While this scheme does not greatly reduce the ability to detect keystrokes, it makes tokenization difficult which poisons later stages of the attack. Note that tokenization could also be made more difficult by encoding the Space key as a single character, such as "+" instead of the 3 byte sequence "%20". Search engines Yandex and DuckDuckGo both use this strategy. However, this does not exclude the possibility of tokenization through other means such as timings, an item we leave for future work.

8 Discussion

Search engines with autocomplete are part of a larger class of applications in which the manifestation of human-computer interactions in network traffic can lead to a remote side channel attack. This includes VoIP: as utterances are compressed and transmitted in real time, spoken phrases can be identified in encrypted network traffic [55, 56]; SSH: single characters are transmitted to and echoed back by the server, exposing the timing of key presses [47]; HTTP: unencrypted network traces contain a user's web browsing activity [36, 57]; and HTTPS: in dynamic web applications, server response size can reveal interactions with specific elements on a web page [12].

8.1 Related work

Keystroke timing attacks Keystroke timing attacks were introduced in [47], which considered the identification of key pairs (bigrams) from key-press latencies to aid in password inference. Such an attack is generally possible because of the non-zero mutual information between keys and keystroke timings, e.g., keys far apart are usually pressed in quicker succession than keys that are close together [43]. This behavior generalizes across subjects, similar to other phenomena in human-computer interaction (HCI) such as Fitts' Law [17]. There has been some debate whether a remote keystroke timing attack poses a credible threat [3, 22]. Evidence suggests that while information gain is generally possible, attack success is user-dependent with some users being more vulnerable than others [33, 34].

In [47], a hidden Markov model and generalization of the Viterbi algorithm were used to generate candidate passwords from timings. The key-press latencies used to train the model were recorded in isolation, wherein subjects pressed a key pair as opposed to typing a full password. In addition, the keystrokes were recorded on the host under the assumption that the key-press latencies would be faithfully preserved in the network traffic. Our work confirms that assumption by using timings obtained from actual network traffic and users typing complete phrases instead of isolated bigrams.

There have been numerous works focused on the *detection* of keyboard events (which enables a timing attack), such as through spikes in CPU load [45], cache and memory usage [41], and the `proc` filesystem [23]. Few works have considered remote keylogging attacks [12,58]. In [51], the authors examine the extent to which autocomplete exposes key-press latencies in network traffic and found that multiple observations were required to recover the true latency. In a recent work, we characterized the autocomplete network traffic of five major search engines and measured the correlation between key-press latencies on the host and packet inter-arrival times observed remotely [35], finding search engines Google and Baidu to leak the most information. The findings in [35] partly motivated the development of KREEP.

Since the work [47], several studies have examined timing attacks on password [6, 59] and PIN [29, 30] input. We depart from prior work, which has focused on sequences with maximum prior entropy, by targeting natural language input, which is more susceptible to keystroke timing attack due to a relatively lower prior entropy (roughly 1 bit per char, as noted in [47]). We introduced a method to combine language model probabilities with information leaked through keystroke timings, inspired by the use of language models in conjunction with acoustic models in automatic speech recognition [20]. In addition, our attack combines multiple independent sources of information leakage beyond keystroke timings, including URL escape sequences and HTTP2 header compression.

Compression side channels A compression side channel leverages information leaked through the compression of a plaintext prior to encryption [28]. Because different strings compress to different sizes, compressed size can reveal information about the plaintext. HTTPS exposes the length of an encrypted payload, making it vulnerable to attack when the payload is compressed. There have been several attacks on HTTPS based on this principle.

The CRIME attack exploits compression in TLS and in the now deprecated SPDY protocol [42]. This attack requires a man-in-the-middle vantage in which an attacker inserts a guess for a secret, e.g., an HTTP cookie or a CSRF token, into a message and observes the compressed size. The DEFLATE compression algorithm in SPDY uses redundancy to compress a string [14] such that the compressed size of a packet containing the correct guess will be smaller than an incorrect guess. The BREACH attack leveraged a similar principle for server responses, targeting compression at the HTTP level (e.g., gzip) [18], and the TIME attack used server response time as a proxy to measure response size [9].

HEIST lowered the bar for attack, enabling CRIME-like attacks to be deployed remotely within a victim's web browser [52]. The size of a compressed server response is determined at the application level by examining whether the response time spans multiple round trips, an indication that the entire response exceeded the TCP congestion window. This general technique can be used in a variety of side channel attacks beside guessing secrets, such as determining whether a user is logged into a particular site [52].

DEFLATE, the compression algorithm used in gzip, uses a combination of LZ77 and Huffman coding [14]. To date, all compression attacks against HTTPS have exploited the LZ77 component of DEFLATE, which builds a dictionary from the redundant parts of a string. The Huffman code in DEFLATE has been treated as noise, typically dealt with by making guesses in pairs. For example, to find out whether a secret starts with "p", an attacker guesses "secret=p_" and "secret=_p": if the sizes are the same, then only Huffman coding is used and the guess is wrong; otherwise, if the sizes are different, the LZ77 component was invoked based on redundancy between the first guess and the secret, and only Huffman coding was invoked in the second guess.

HPACK, the header compression format in HTTP2, was designed to be resistant to CRIME-like attacks targeting LZ77 compression, although HTTP2 borrowed many concepts from SPDY [39]. Commonly used header fields are compressed with a dictionary lookup, and string literals are compressed using a static Huffman code, which was previously determined to leak relatively little information [50]. But unlike previous attacks, KREEP leverages the static Huffman code in HPACK rather than an LZ77 dictionary. We found considerably more information is leaked due to several contributing factors:

1. *HTTPS exposes payload size*. HTTPS was previously shown to leak information by exposing the length of an encrypted payload. The HTTPS Bicycle attack uses the size differences between HTTP requests to infer the size of an unknown secret [53]. An attacker simply subtracts the size of all known parts of the request, leaving only the size of the secret. Our attack relies on a similar principle, taking the difference in size between successive autocomplete requests.

2. *Characters are independently compressed*. The size difference between two compressed payloads that differ only by the insertion of a single character reveals the compressed size of that character. However, Huffman encoded strings in HPACK are padded to the nearest octet, mitigating the amount of information that would otherwise be leaked without padding. Since byte, and not bit, size differences are observed, the symbol size is known only to within a margin of error that depends on an unknown amount of padding.

3) *The Huffman code is standard*. Every HPACK implementation uses the same Huffman code, which is publicly available [39]. An attacker needs only to map dictionary words to their cumulative compressed sizes, taking into account the unknown amount of padding applied beforehand. Potential matches to a secret are revealed by comparing its cumulative compressed size to every word in the dictionary.

Search query identification Previous work on identifying search queries has utilized features obtained primarily through traffic analysis. In [37], keywords in search queries are identi-

fied over Tor using both inbound and outbound autocomplete traffic. Keystrokes were replayed in a data collection setup similar to ours described in Section 7.1. Packet inter-arrival times were not considered since the replayed keystrokes used random, and not human, timings. Instead, each search query is characterized by packet counts and sizes, inbound and outbound Tor cell counts, and other features specific to Tor traffic. The work of [37] did not attempt keystroke detection but instead focused on the identification of queries that contain a particular keyword from a set of target keywords. With this approach, a query containing any one of 300 target keywords could be identified with 85% accuracy, and individual keywords with 48% accuracy.

We instead aim to reconstruct an entire query rather than identify the presence of some target words, and we leverage information leaked through packet size, which is obfuscated by cell size in Tor traffic. While keystroke detection may be possible in traffic over Tor, for example by detecting traffic that has "keystroke-like" packet inter-arrival times, tokenization and dictionary pruning cannot be applied since the autocomplete packet sizes are masked behind Tor cell sizes. An attack that uses *only* packet inter-arrival times might be feasible in Tor, but would require a different approach than our attack.

While previous work has shown HTTP response size to leak a considerable amount of information about a user's query when autocomplete suggestions are provided [12], we chose to focus only on HTTP requests. In [12], an attacker guesses a victim's query one letter at a time by trying all combinations and matching the server response size. This assumes the attacker can submit queries that induce the same suggestions as the victim received. In practice, this is difficult because autocomplete suggestions depend on the victim's search history and location, among other factors [2]. To our knowledge, KREEP is the first attack targeting autocomplete traffic from the client independent of these factors, relying only on packet inter-arrival times and packet size differences.

8.2 Countermeasures

Keylogging attacks require successful keystroke detection *and* key identification. Therefore, it is sufficient to prevent keystroke detection *or* key identification to counter the attack. We consider the tradeoffs of several countermeasures and how they affect each source of information leakage.

Padding Padding could be applied in two different ways: pad each request by a random amount, or pad to ensure all requests are the same size. To increase keystroke detection false negatives, the pad amounts must be sufficiently large to disguise autocomplete traffic with other background traffic, the size of which is generally not known a priori. Therefore, padding may not be effective to mitigate keystroke detection and does not provide any protection against a timing attack. However, we have confirmed that padding by a small random amount (1 byte with probability 0.5) does effectively mitigate tokenization and incremental compression. Note that padding in this way should be applied only to alphabetic characters and not to the addition of a Space; otherwise, some packets with a Space will increase by 3 bytes (2 bytes + 1 padding byte), while all other packets increase by no more than 2 bytes. The pad amounts should be chosen such that the *observed* packet size differences closely follow a uniform distribution.

Dummy traffic While padding aims to increase detection false negatives, generating dummy traffic aims to increase false positives. A false positive occurs when background traffic is labeled as a keystroke. Generating dummy autocomplete requests with approximately the same size as the actual request would make keystroke detection a difficult task. With each autocomplete request, the client could send a burst of several packets with similar size (within several bytes), randomly ordering the actual request within the dummy request. While an attacker might still be able to perform detection with a low false negative rate, this comes at a cost of an increased false positive rate. This method mitigates tokenization, compression, and timing attacks. The background traffic would overwhelm the actual requests, similar to the generation of dummy keyboard events in KeyDrown [45]. This approach has the cost of increased bandwidth, a tradeoff reminiscent of the anonymity trilemma [13], and requires some cooperation from the server to ignore the dummy requests.

Merge requests Most search engines make an autocomplete request immediately following each new character appended to the input field [35]. Instead, combining multiple characters into a single request would mitigate our attack in two different ways. First, with multiple characters merged into a single request, the number of false negative detection errors must increase since a packet contains multiple keystrokes. This conceals the timing information of all but the last character in the merged request, reducing information leakage through a keystroke timing attack.

Additionally, merged requests effectively eliminate the incremental compression side channel since the increase in packet size corresponds not to a single character but to multiple characters. The compressed size of the merged characters must be some linear combination of symbols in the Huffman code, and as string length increases, the number of combinations grows exponentially [50].

Combining requests could be achieved in several ways: 1) update the list of autocomplete suggestions after every other, or every *n*th, key (similar to Nagle's algorithm, except at the application level); 2) use a polling model with polling rate slower than the user's typing speed (Bing performs polling with 100ms interval, making this attack impractical for fast typists); or 3) trigger callbacks on `keyup` events instead of `keydown` events (DuckDuckGo does this), which merges requests when consecutive keystrokes overlap [35], a typing

phenomenon referred to as *rollover* [15]. The drawback in all cases is that merging requests could adversely affect usability since the suggested queries are delayed to the user.

8.3 Limitations and future work

We point out several limitations of our attack, emphasizing the conditions under which it succeeds, and identify ways in which KREEP could be extended or improved.

Other websites In this work, KREEP has only been tested on search engines Google and Baidu. Keystroke detection and tokenization are both application-specific, based on the packet size pattern each search engine emits. Extensions to other search engines or websites would require modification to these components. For websites that aren't vulnerable to tokenization, delimiters might be identified based on packet inter-arrival times (e.g., larger intervals indicate Space, smaller intervals indicate letters).

Other modalities Since autocomplete requests are induced by keyboard events, KREEP is applicable only up to the point when a user stops typing or selects a suggested query. We assumed that no deletions or corrections were made and that the user did not press any non-printable keys, e.g., arrow keys, that cause the caret to change position. However, selecting a query from the provided suggestions does not preclude the possibility of other attacks that incorporate the timing of both autocomplete requests and server responses. It may be the case that the way the user interacts with the autocomplete suggestions also leaks course-grained information, such as user identity or the type of query (navigational, informational, or transactional) [10]. One might also consider the timing of mouse clicks that induce network traffic as a source of information leakage by leveraging a general model that governs click behavior, such as Fitts' Law [17].

Targeted attacks Finally, while we made an effort to evaluate our attack on phrases that are representative of natural language, the content of actual search queries is quite different and varies between users as evidenced by the AOL search dataset [7, 38]. Some strings that have a low probability of occurrence in natural language, such as "www", tend to occur frequently in search queries. This affects the prior probability of each symbol, which must be properly accounted for in the language model. We verified this difference by comparing the frequency of characters in the AOL search dataset to the keystroke dataset we used to evaluate KREEP (which itself borrowed phrases from the Enron email corpus [15]). These are shown in Figure 14. Notably, the frequencies of "w" and "c" in search queries are about twice that of natural language, likely due to the presence of navigational queries to a specific URL, such as "www.example.com". Likewise, Space characters in search are about half as frequent compared to natural language. In a targeted attack, the language model in KREEP could be tailored towards a particular victim, leveraging information such as the victim's native language, geographic location, and public blog entries.

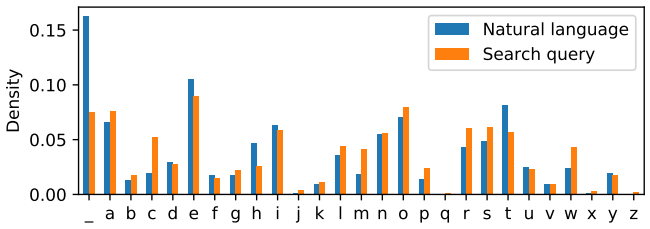


Figure 14: Character frequency in natural language (Enron corpus) compared to search queries (AOL search dataset).

9 Conclusion

KREEP leverages multiple independent sources of leaked information to identify search queries in encrypted network traffic. Autocomplete request packets are detected based on packet size; queries are tokenized by detecting the presence of URL-escaped characters; keys are identified based on packet inter-arrival times; and impossible words are eliminated from a dictionary based on incremental compression. Despite many moving pieces, the attack obtains a reasonable success rate, recovering more than half the characters in a query on average. But more importantly, the pieces that contribute to this attack present some starting points for future research.

The static Huffman code used in HTTP2 header compression leaks more information than previously thought [50] when incremental changes are made to a string in the header. This kind of attack is not limited to search engines with autocomplete but could apply to any website with dynamic content that updates incrementally. It will be beneficial to identify other web applications that exhibit incremental compression. Besides websites that provide search suggestions, this could include mapping services, which modify the geographic coordinates in a URL as the user drags the map center location, or websites that autosave the contents of a text field.

Likewise, websites that generate network traffic in response to user input events may be vulnerable to timing attack. Sites that support remote document editing, such as Google Docs, frequently transmit the document state from the client to the server. When this process is event driven, i.e., triggered by `keydown` events, the network traffic can leak information about the user's actions or document content. Similarly, chat applications that aim to provide real-time updates about a conversation partner's activity, e.g., by displaying a notification that "X is typing", also risk exposing keystroke timings in network traffic if those notifications are directly driven by the conversation partner's keystrokes.

Availability

KREEP is available at https://github.com/vmonaco/kreep. The keystroke dataset is publicly available [15].

Acknowledgements

We thank the anonymous reviewers and our shepherd for valuable feedback during the review process. The manuscript was much improved based on insightful discussions with colleagues Justin Rohrer and Robert Beverly, who also provided comments on an early draft.

References

[1] NO_HZ: Reducing Scheduling-Clock Ticks. http://web.archive.org/web/20190208124417/https://www.kernel.org/doc/Documentation/timers/NO_HZ.txt. Accessed: 2019-02-08.

[2] Search using autocomplete. http://web.archive.org/web/20190209193857/https://support.google.com/websearch/answer/106230?hl=en. Accessed: 2019-02-09.

[3] Timing analysis is not a real-life threat to ssh secure shell users. http://web.archive.org/web/20010831024537/http://www.ssh.com/products/ssh/timing_analysis.cfm. Accessed: 2019-02-09.

[4] Eytan Adar. User 4xxxxx9: Anonymizing query logs. In *Proc of Query Log Analysis Workshop, International Conference on World Wide Web*, 2007.

[5] Dmitri Asonov and Rakesh Agrawal. Keyboard acoustic emanations. In *Proc. IEEE Symp. on Security & Privacy (SP)*, pages 3–11. IEEE, 2004.

[6] Kiran S. Balagani, Mauro Conti, Paolo Gasti, Martin Georgiev, Tristan Gurtler, Daniele Lain, Charissa Miller, Kendall Molas, Nikita Samarin, Eugen Saraci, Gene Tsudik, and Lynn Wu. SILK-TV: Secret information leakage from keystroke timing videos. In *Computer Security*, pages 263–280. Springer International Publishing, 2018.

[7] Michael Barbaro, Tom Zeller, and Saul Hansell. A face is exposed for aol searcher no. 4417749. *New York Times*, 9(2008):8, 2006.

[8] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifier (URI): Generic syntax. Technical report, jan 2005.

[9] Tal Be'ery and Amichai Shulman. A perfect crime? only time will tell. *Black Hat Europe*, 2013, 2013.

[10] Andrei Broder. A taxonomy of web search. In *ACM Sigir forum*, volume 36, pages 3–10. ACM, 2002.

[11] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. One billion word benchmark for measuring progress in statistical language modeling. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[12] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *Proc. IEEE Symp. on Security & Privacy (SP)*, pages 191–206. IEEE, 2010.

[13] Debajyoti Das, Sebastian Meiser, Esfandiar Mohammadi, and Aniket Kate. Anonymity trilemma: Strong anonymity, low bandwidth overhead, low latencychoose two. In *Proc. IEEE Symp. on Security & Privacy (SP)*. IEEE, 2018.

[14] P. Deutsch. DEFLATE compressed data format specification version 1.3. Technical report, may 1996.

[15] Vivek Dhakal, Anna Maria Feit, Per Ola Kristensson, and Antti Oulasvirta. Observations on typing from 136 million keystrokes. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI 18*. ACM Press, 2018.

[16] Tobias Fiebig, Janis Danisevskis, and Marta Piekarska. A metric for the evaluation and comparison of keylogger performance. In *Proc. 7th Usenix Conf. on Cyber Security Experimentation and Test*, pages 7–7. USENIX Association, 2014.

[17] Paul M Fitts. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of experimental psychology*, 47(6):381, 1954.

[18] Yoel Gluck, Neal Harris, and Angelo Prado. Breach: reviving the crime attack. 2013.

[19] David Graff, Junbo Kong, Ke Chen, and Kazuaki Maeda. English gigaword. *Linguistic Data Consortium, Philadelphia*, 4(1):34, 2003.

[20] Alex Graves and Navdeep Jaitly. Towards end-to-end speech recognition with recurrent neural networks. In *International conference on machine learning*, pages 1764–1772, 2014.

[21] Kenneth Heafield, Ivan Pouzyrevsky, Jonathan H. Clark, and Philipp Koehn. Scalable modified Kneser-Ney language model estimation. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, pages 690–696, Sofia, Bulgaria, August 2013.

[22] Michael Augustus Hogye, Christopher Thaddeus Hughes, Joshua Michael Sarfaty, and Joseph David Wolf. Analysis of the feasibility of keystroke timing attacks over ssh connections. *Research Project at University of Virginia*, 2001.

[23] Suman Jana and Vitaly Shmatikov. Memento: Learning secrets from process footprints. In *Proc. IEEE Symp. on Security & Privacy (SP)*, pages 143–157. IEEE, 2012.

[24] Bernard J. Jansen, Amanda Spink, and Tefko Saracevic. Real life, real users, and real needs: a study and analysis of user queries on the web. *Information Processing & Management*, 36(2):207–227, mar 2000.

[25] Rosie Jones, Ravi Kumar, Bo Pang, Andrew Tomkins, Andrew Tomkins, and Andrew Tomkins. I know what you did last summer: query logs and user privacy. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pages 909–914. ACM, 2007.

[26] Sepandar D Kamvar et al. Anticipated query generation and processing in a search engine, 2004.

[27] Stavroula Karapapa and Maurizio Borghi. Search engine liability for autocomplete suggestions: personality, privacy and the power of the algorithm. *International Journal of Law and Information Technology*, 23(3):261–289, jul 2015.

[28] John Kelsey. Compression and information leakage of plaintext. In *Fast Software Encryption*, pages 263–276. Springer Berlin Heidelberg, 2002.

[29] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical keystroke timing attacks in sandboxed javascript. In *Proc. 22nd European Symp. on Research in Computer Security*, 2017.

[30] Ximing Liu, Yingjiu Li, Robert H. Deng, Shujun Li, and Bing Chang. When human cognitive modeling meets PINs: User-independent inter-keystroke timing attacks. *Computers & Security*, sep 2018.

[31] Philip Marquardt, Arunabh Verma, Henry Carter, and Patrick Traynor. (sp) iphone: Decoding vibrations from nearby keyboards using mobile phone accelerometers. In *Proc. 18th ACM Conf. on Computer and Communications Security (CCS)*, pages 551–562. ACM, 2011.

[32] Jean-Baptiste Michel, Yuan Kui Shen, Aviva Presser Aiden, Adrian Veres, Matthew K Gray, Joseph P Pickett, Dale Hoiberg, Dan Clancy, Peter Norvig, Jon Orwant, et al. Quantitative analysis of culture using millions of digitized books. *science*, 331(6014):176–182, 2011.

[33] John V Monaco. Poster: The side channel menagerie. In *Proc. IEEE Symp. on Security & Privacy (SP)*. IEEE, 2018.

[34] John V Monaco. Sok: Keylogging side channels. In *Proc. IEEE Symp. on Security & Privacy (SP)*. IEEE, 2018.

[35] John V Monaco. Feasibility of a keystroke timing attackon search engines with autocomplete. In *2019 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2019.

[36] Christopher Neasbitt, Roberto Perdisci, Kang Li, and Terry Nelms. ClickMiner. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS '14*. ACM Press, 2014.

[37] Se Eun Oh, Shuai Li, and Nicholas Hopper. Fingerprinting keywords in search queries over tor. *Proceedings on Privacy Enhancing Technologies*, 2017(4):251–270, oct 2017.

[38] Greg Pass, Abdur Chowdhury, and Cayley Torgeson. A picture of search. In *InfoScale*, volume 152, page 1, 2006.

[39] R. Peon and H. Ruellan. HPACK: Header compression for HTTP/2. Technical report, may 2015.

[40] E. Rescorla. The transport layer security (tls) protocol version 1.3. Technical report, aug 2018.

[41] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proc. 16th ACM Conf. on Computer and Communications Security (CCS)*, pages 199–212. ACM, 2009.

[42] Juliano Rizzo and Thai Duong. The crime attack. In *Ekoparty Security Conference*, 2012.

[43] Timothy A Salthouse. Perceptual, cognitive, and motoric aspects of transcription typing. *Psychological bulletin*, 99(3):303, 1986.

[44] C. Schensted. Longest increasing and decreasing subsequences. *Canadian Journal of Mathematics*, 13:179–191, 1961.

[45] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. Keydrown: Eliminating keystroke timing side-channel attacks. In *Proc. Network and Distributed System Security Symp (NDSS)*, 2018.

[46] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript. In *Proc. 21st Intl. Conf. on Financial*

Cryptography and Data Security (FC), page 11. IFCA, 2017.

[47] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on ssh. In *Proc. Usenix Security Symp.*, 2001.

[48] Statcounter. Search engine market share china. http://web.archive.org/web/20190209193125/http://gs.statcounter.com/search-engine-market-share/all/china. Accessed: 2019-02-09.

[49] Statcounter. Search engine market share worldwide. http://web.archive.org/web/20190209193145/http://gs.statcounter.com/search-engine-market-share. Accessed: 2019-02-09.

[50] Jiaqi Tan and Jayvardhan Nahata. Petal: Preset encoding table information leakage. Technical report, 2013.

[51] Chee Meng Tey, Payas Gupta, Debin Gao, and Yan Zhang. Keystroke timing analysis of on-the-fly web apps. In *Proc. Intl. Conf. on Applied Cryptography and Network Security*, pages 405–413. Springer, 2013.

[52] Mathy Vanhoef and Tom Van Goethem. Heist: Http encrypted information can be stolen through tcp-windows. *Black Hat USA 2016*, page 1, 2016.

[53] Guido Vranken. Https bicycle attack. Technical report, dec 2015. Accessed: 2019-05-10.

[54] He Wang, Ted Tsung-Te Lai, and Romit Roy Choudhury. Mole: Motion leaks through smartwatch sensors. In *Proc. 21st Annual Intl. Conf. on Mobile Computing and Networking (MobiCom)*, pages 155–166. ACM, 2015.

[55] Andrew M. White, Austin R. Matthews, Kevin Z. Snow, and Fabian Monrose. Phonotactic reconstruction of encrypted VoIP conversations: Hookt on fon-iks. In *2011 IEEE Symposium on Security and Privacy*. IEEE, may 2011.

[56] Charles V. Wright, Lucas Ballard, Scott E. Coull, Fabian Monrose, and Gerald M. Masson. Spot me if you can: Uncovering spoken phrases in encrypted VoIP conversations. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, may 2008.

[57] Guowu Xie, Marios Iliofotou, Thomas Karagiannis, Michalis Faloutsos, and Yaohui Jin. Resurf: Reconstructing web-surfing activity from network traffic. In *IFIP Networking Conference, 2013*, pages 1–9. IEEE, 2013.

[58] Ge Zhang and Simone Fischer-Hübner. Timing attacks on pin input in voip networks (short paper). In *Proc. Intl. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 75–84. Springer, 2011.

[59] Kehuan Zhang and XiaoFeng Wang. Peeping tom in the neighborhood: Keystroke eavesdropping on multi-user systems. *analysis*, 20:23, 2009.

[60] Li Zheng, Liren Zhang, and Dong Xu. Characteristics of network delay and delay jitter and its effect on voice over IP (VoIP). In *ICC 2001. IEEE International Conference on Communications. Conference Record (Cat. No.01CH37240)*. IEEE.

Iframes/Popups Are Dangerous in Mobile WebView: Studying and Mitigating Differential Context Vulnerabilities

GuangLiang Yang, Jeff Huang, Guofei Gu
Texas A&M University
{ygl, jeffhuang, guofei}@tamu.edu

Abstract

In this paper, we present a novel class of Android WebView vulnerabilities (called *Differential Context Vulnerabilities* or *DCVs*) associated with web iframe/popup behaviors. To demonstrate the security implications of DCVs, we devise several novel concrete attacks. We show an untrusted web iframe/popup inside WebView becomes dangerous that it can launch these attacks to open holes on existing defense solutions, and obtain risky privileges and abilities, such as breaking web messaging integrity, stealthily accessing sensitive mobile functionalities, and performing phishing attacks.

Then, we study and assess the security impacts of DCVs on real-world apps. For this purpose, we develop a novel technique, *DCV-Hunter*, that can automatically vet Android apps against DCVs. By applying DCV-Hunter on a large number of most popular apps, we find DCVs are prevalent. Many high-profile apps are verified to be impacted, such as Facebook, Instagram, Facebook Messenger, Google News, Skype, Uber, Yelp, and U.S. Bank. To mitigate DCVs, we design a multi-level solution that enhances the security of WebView. Our evaluation on real-world apps shows the mitigation solution is effective and scalable, with negligible overhead.

1 Introduction

Nowadays, mobile app developers enjoy the benefits of the amalgamation of web and mobile techniques. They can easily and smoothly integrate all sorts of web services in their apps (*hybrid* apps) by embedding the browser-like UI component "WebView". WebView is as powerful as regular web browsers (e.g., desktop browsers), and well supports web features, including the utilization of *iframes/popups*.

In the web platform, iframes/popups are frequently used, but also often the root cause of several critical security issues (e.g., frame hijacking [11] and clickjacking [23, 43]). In past years, in regular browsers, their behaviors have been well studied, and a variety of mature iframe/popup protection solutions (e.g., Same Origin Policy (SOP) [6], HTML5 iframe sandbox [4], and navigation policies [11]) have been deployed.

Inconsistencies Between Browsers and WebView. However, in WebView, a totally different working environment is provided for iframes/popups, due to WebView's own programming and UI features. Although these features improve app performance and user experience, they extensively impact iframe/popup behaviors and introduce security concerns. In particular, WebView enables several programming APIs (Figure 1) to help developers customize iframe/popup behaviors. For example, the setting APIs allow developers to configure their WebView instances. In the customized web environment (WebView), it is unclear whether existing iframe/popup protection solutions are still effective.

Furthermore, WebView UI is designed in a simple style (Figure 2) that only one UI area for rendering web content is provided. Due to the lack of the address bar, it is difficult for users to learn what web content is being loaded; due to the lack of the tab bar, it is unknown how multiple WebView UI instances (WUIs) are managed. Therefore, if an iframe/popup has abilities to secretly navigate the main frame (the top frame) or put their own WUI to the foremost position for overlaying the original WUI, phishing attacks occur and may cause serious consequences. Consider the scenario shown in Figure 3 and 4. The Huntington banking app (one million+ downloads) uses WebView to help users reset passwords (Figure 3-a,b). Inside WebView, the main frame contains an iframe for isolatedly loading untrusted third-party tracking content (Figure 4). However, if the untrusted web content inside the iframe obtains the ability of stealthily redirecting the main frame to a fake website (Figure 3-c), serious security risks are posed. For example, users' personal (e.g., SSN info and Tax ID) and bank account information may be stolen, and further financial losses may also be caused.

***Differential Context Vulnerability* (*DCV*)**. Motivated by above security concerns, we conduct the first security study of iframe/popup behaviors in the *context* of Android WebView. In this paper, we use the term "context" to refer to a web environment that includes GUI elements (e.g., the address and tab bars), corresponding web management APIs (e.g., the setting APIs in WebView), and security policies (e.g., SOP

Table 1: A Summary of Differential Context Vulnerabilities (DCVs)

Critical Features & Behaviors	Different Contexts		Attacks	Explanations	Consequences
	Browsers	WebView			
Main-Frame Creation	Address Bar	Java APIs	Origin Hiding Attack	Special common origins (e.g., null) Of Main-Frame	Sensitive functionalities behind postMessage and JavaScript Bridges can be leveraged, which may cause the leakage of sensitive information (e.g., location), and risky access on Hardware (e.g., camera and microphone)
Management of new popups	Tab Bar	Android Frameworks	WUI overlap attack	No protection on the WUI rendering sequence	Phishing attacks
			WUI closure attack		
Main-Frame Navigation	Address Bar	Java APIs	Traditional navigation based attack	Permissive navigation policies	
			Privileged navigation attack	Harmful conflict between WebView Customizations and web APIs	

and navigation policies).

As a consequence, our study uncovers a novel class of vulnerabilities and design flaws in WebView. These vulnerabilities are rooted in the inconsistencies between different contexts of regular browsers and WebView. As summarized in Table 1, several critical web features and behaviors (i.e., main-frame creation, popup creation, and main-frame navigation) are involved (see more details in Section 3). These features and behaviors are harmless or even safe in the context of regular browsers, but become risky and dangerous in the context of WebView. To demonstrate their security implications, we devise several concrete attacks. We show through these attacks, remote adversaries (e.g., web or network attackers on iframes/popups) can obtain several unexpected and risky privileges and abilities:

1) *Origin-Hiding*: hiding the origin when
   - breaking the integrity of web messaging (i.e., postMessage) [8], which allows the communication between mutually distrusted web frames; and
   - secretly accessing web-mobile bridges [21], which link the web layer with the *mobile* or *native* layer (e.g., Java for Android) (Figure 1);

   Existing work has shown that postMessage's message receivers [44, 47] and web-mobile bridges [21, 49, 53] often carry sensitive functionalities. Thus, these functionalities can be further stealthily accessed by the untrusted iframe/popup through the attack. As a result, sensitive information (e.g., GPS location) may be stolen, and important hardware (e.g., microphone) may be unauthorizedly accessed.
2) *WebView UI Redressing*: performing phishing attacks by overlapping the foremost benign WUI with an untrusted WUI;
3) *(Privileged) Main-Frame Navigation*: freely redirecting the main frame to a fake website.

Moreover, we examine the effectiveness of existing protection solutions, which include not only the solutions designed for regular browsers (inherited by WebView), but also the solutions proposed for Android UI and WebView. We find that these solutions are ineffective to defend against the above attacks:

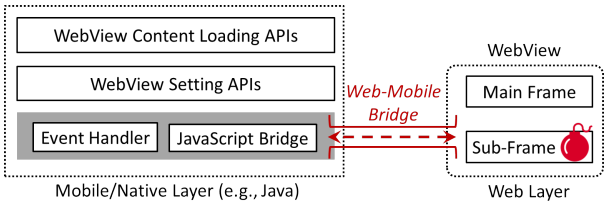
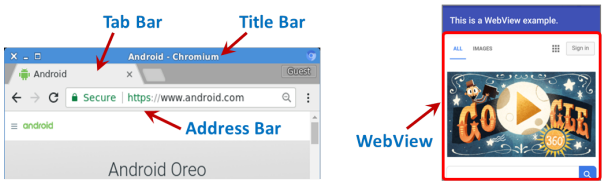


Figure 1: WebView Programming Features

1) For origin-hiding attacks, existing defense solutions for postMessage [11, 44, 47, 52] and web-mobile bridges [18, 21, 38, 45, 49] usually provide security enforcement relying on origin validation. However, unfortunately, the key origin information of the untrusted iframe/popup can be hidden during attacks, which leads to the bypass of the security enforcement.
2) For WUI redressing attacks, they are similar to Android UI redressing attacks [15, 20, 35]. However, the associated Android UI protection solutions (e.g., [13, 41]) are circumscribed to prevent WUI addressing attacks. This is mainly because that these protections work by monitoring exceptional Android UI state changes between different apps, while the WUI state change occurs within an app during attacks.
3) For main-frame navigation attacks, one related solution is the iframe sandbox security mechanism, which can effectively limit the navigation capability of an arbitrary iframe. However, through DCV attacks, an untrusted iframe can still break the above limitation and cause privilege escalation.

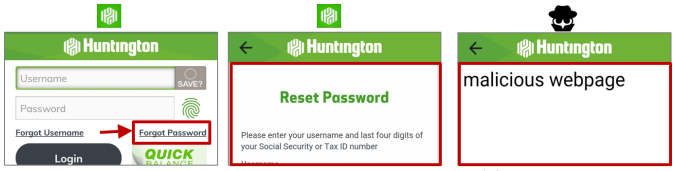
More details about the vulnerabilities and the weakness of existing defense solutions are presented in Section 3. For convenience, considering the root reason of this new type of vulnerability (i.e., the inconsistencies between the contexts of regular browsers and WebView), we refer to the vulnerabilities as *Differential Context Vulnerabilities* or *DCVs*, and the associated attacks as DCV attacks.

DCV-Hunter & Findings. We next study and assess the security impact of DCVs on real-world hybrid apps. To achieve the goal, we develop a novel static vulnerability detection technique, *DCV-Hunter*, to automatically vet given apps against



(a) Regular Browser UI (b) WebView UI

Figure 2: UI Comparison



(a) Click "Forgot password" (b) Run WebView (c) Secretly Navigate the Main Frame

Figure 3: Attacking the Huntington Bank App

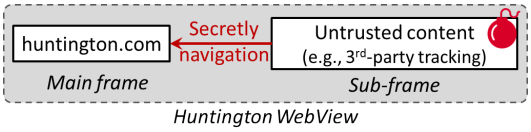
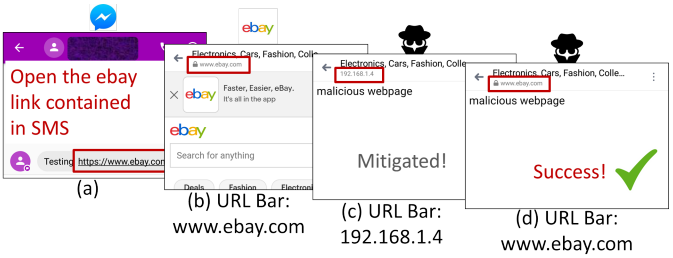


Figure 4: Attack Scenario



(a) (b) URL Bar: www.ebay.com (c) URL Bar: 192.168.1.4 (d) URL Bar: www.ebay.com

Figure 5: Attacking Facebook Messenger

DCVs. Then, by applying DCV-Hunter on a number of most popular apps, we show that DCVs are prevalent. More specifically, we find 38.4% of 11,341 hybrid apps are potentially vulnerable, including 13,384 potentially vulnerable WebView instances and 27,754 potential vulnerabilities. Up to now, the potentially impacted apps have been downloaded more than 19.5 Billion times in total. Furthermore, our evaluation shows DCV-Hunter is scalable and effective, and has relatively low false positives (~1.5%).

We also manually verify that many high-profile apps are vulnerable (a list of video demos of our attacks can be found online [2]), including Facebook, Instagram, Facebook Messenger, Google News, Skype, Uber, Yelp, WeChat, Kayak, ESPN, McDonald's, Kakao Talk, and Samsung Mobile Print. Several popular third-party development libraries, such as Facebook Mobile Browser and Facebook React Native, are also vulnerable and they influence hundreds of apps. Several special sensitive categories of apps are affected including leading password management apps (such as dashlane, lastpass, and 1password), and popular banking apps (such as U.S. bank, Huntington bank, and Chime mobile bank).

In our analysis, we also find that some apps implement their own URL address and title bars, which reduce the inconsistencies between regular browsers and WebView. However, these home-brewed URL bars hardly eliminate DCVs due to several limitations. One major limitation is that their implementation is often error-prone. For example, Facebook Messenger (Figure 5, one billion+ downloads) is equipped with the library "Facebook Mobile Browser" to handle URLs contained in messages (e.g., SMS). The browser library implements its own address bar (Figure 5-b) to reflect the change of web content (Figure 5-c) and mitigate DCV attacks (e.g., the WUI overlap attack). However, this address bar contains a design flaw (race condition). By combining a couple of DCV attacks, untrusted iframes/popups can still launch phishing attacks (Figure 5-d). Due to the inclusion of the vulnerable library, many high-profile apps are impacted, such as Facebook and Instagram. In addition to the vulnerable library, we find this design flaw is shared by many other popular apps that are not equipped with that library, such as Kakao Talk (100 million+ downloads).

We have reported our findings to the Android security team and many app developers. Up to now, a number of them (e.g., the Android and Facebook security teams) have confirmed our findings.

DCV Mitigation. DCVs are not caused by programming mistakes. It is extremely difficult for developers to eliminate the DCV security issues, especially considering the existence of the limitations in WebView (Section 3.6). To mitigate the problem, we propose a multi-level protection solution by enhancing the security of WebView programming and UI features. Our defense solution is implemented by instrumenting WebView's independent library, but without touching the source code of Android frameworks. Our solution is easy to use, and can simply work after developers involve our instrumented library, and provide a list of trusted domains. Our evaluation on real-world apps shows that our solution is effective and scalable, and introduces negligible overhead. Furthermore, considering the Android version fragmentation issue, we also test the compatibility of our solution. The result shows our solution is available in many major popular Android versions (5.0+), and covers almost 90% of Android devices in use.

Contributions. In sum, we make the following contributions:

- We investigate the security of iframe/popup in Android WebView, and discover several novel and fundamental design flaws and vulnerabilities in WebView (i.e., DCVs).
- We design a novel automatic vulnerability detection tool "DCV-Hunter" to quantify the prevalence of DCVs.
- We apply DCV-Hunter on a set of popular apps, and confirm that DCVs have severe security impacts.
- We further propose a multi-level solution to mitigate DCV attacks.

2 Background and Threat Model

Before we dive into our study of iframe/popup security, we first introduce necessary background information and our threat model.

2.1 Iframes/Popups and Related Protections

Iframes/popups are frequently used in web apps, for example, to view files in various formats (e.g., images, videos and PDFs), or load third-party untrusted web content (e.g., ads). They are easy to use. To create an iframe, developers can 1) either use the HTML element <iframe>; 2) or run JavaScript code to dynamically build an iframe DOM node.

Furthermore, to enable a popup, developers can use the following HTML code to generate a link:

```
<a href="URL" target="_blank|_top|frame_name|...".
```

When users click the link, "URL" will be opened in the frame that is determined by the "*target*" attribute. If *target* is "_blank", a new popup window will be opened to show "URL". Moreover, if *target* is "_top" or a specific frame name, "URL" will be loaded in the main frame or the specific frame determined by "frame_name". Developers can also use JavaScript code to open or close a web window:

```
window.open(URL, <target>, ...)  or window.close().
```

Similar to the usage of the HTML element <a>, "*window.open()*" can also determine where to open popup content.

Related Protections. Up to now, several practical protection solutions were designed and deployed in regular browsers:

- *Same origin policy (SOP)*: SOP isolates web frames whose origins are different. Note that SOP causes side effects that different origins are not allowed to communicate with each other. To mitigate the problem, the postMessage mechanism is designed in HTML5.
- *Built-in security policies*: Several built-in policies are available. For example, remote web code is not allowed to create a new sub-frame for loading local files, and the main frame is not allowed to load the data scheme URL.
- *HTML5 iframe sandbox*: The iframe sandbox mechanism can limit iframes' abilities, mainly including the enablement of JavaScript, main-frame navigation ("<a>" or "*window.open()*"), and popup-creation. Since the security of the popup behavior is one of our research objectives, we assume the popup-creation ability is allowed in iframe sandbox. Thus, in this paper, we mainly consider the abilities related to JavaScript enablement and main-frame navigation.
- *Navigation policies*: As studied in existing work [11], in regular browsers, the main frame is often exempt from strict navigation policies, which means any sub-frame can directly navigate the main frame by using "<a>" or "*window.open()*". There are several reasons for such a design. First, this type of navigation is frequently used by benign web apps, for example, for preventing framing attacks [43]. Second, even though the main frame is navigated, the consequence is quite limited in consideration of the stealthiness: any navigation can be explicitly reflected by URL indicators (e.g., the URL address bar).

2.2 WebView and Related Protections

WebView is an embedded, browser-like UI component. Android WebView is equipped with the newest kernel of the regular browser "Chrome/Chromium", and performs as powerful as regular browsers.

As discussed in Section 1, there are several inconsistencies between regular browsers and WebView. First, WebView UI is like a small and compacted version of a regular browser. It does not contain several common UI elements, including the address, tab, title and status bars.

Second, WebView UI is a case of view group, a collection of multiple Android UI components. More than that, it can also be added to an existing view group. A view group may consist of a set of WUIs with the same size. It manages multiple WUIs with a *rendering queue*, and only rendering the foremost WUI to users.

Third, the manners of initializing web content are different. Compared to regular browsers, which allow users to manually type the address of a website, WebView initializes web content through programming APIs (Figure 1), including

- *loadUrl(URL/file/JS)*: loading content in the main frame;
- *loadData(HTML, ...)*: loading code with the "null" origin;
- *loadDataWithBaseURL(origin,HTML,...)*: loading HTML code with a specified origin.

Last, as shown in Figure 1, developers can customize a WebView instance through several programming features, such as settings, and web-mobile bridges. Settings can manage WebView configurations, while Web-mobile bridges can link the web and mobile layers together. Generally, the bridges include 1) event handlers, which let mobile code handle web events that occur inside WebView; and 2) JavaScript bridges, which can allow JavaScript code to directly access mobile methods.

Furthermore, as shown in Table 2, several programming features can impact iframe/popup behaviors. To enable the creation of a popup, the setting *SupportMultipleWindows* should be set as true, and the event handler *onCreateWindow()* is also required to be implemented and return true. This event handler should create or open a WUI for rendering this popup, and also return the WUI to Android. Otherwise, the popup-creation operation will be ignored. This also means that *different popup windows are rendered by different WUIs at one time*. Besides, to support the closure of a WUI, the event handler *onCloseWindow()* should be also implemented. Note that when any web frame, including the main frame, loads content, the content should be approved by the event handler *"shouldOverrideUrlLoading()"*.

Summary of Related Protections. In past years, WebView security, especially the security of web-mobile bridges, has drawn more and more attention [12, 16, 21, 27, 30, 33, 34, 50, 53–55]. Several defense solutions [18, 21, 38, 45, 49, 50]

Table 2: Iframe/Popup-Related Programming Features

Features	Content	Explanation
Settings	*OpenWindowsAutomatically*	Enable "*window.open()*"
	SupportMultipleWindows	Enable the event handler "*onCreateWindow()*"
Event Handlers	*onCreateWindow()*	Handle window-creation
	onCloseWindow()	Handle window-closure
	shouldOverrideUrlLoading()	Handle URL-loading

were proposed to enhance the security of WebView by providing the security enforcement and access control mechanisms. However, we find they are ineffective against our new attacks. Section 7 provides a review of these existing work.

2.3 Threat Model

In this paper, we mainly focus on the hybrid app whose WebView contains an untrusted sub-frame. In our threat model, we assume the native or mobile code (e.g., Java code), and the main frame loaded in its WebView are secure and trusted. The main frame usually loads web content from the *first-party* benign domains (e.g., developer.com). For the embedded untrusted sub-frames, we mainly consider two *possible* attack scenarios:

Network attacks. When the sub-frames use HTTP network, attackers may perform man-in-the-middle (MITM) attacks to inject attack code into the sub-frames, and then launch DCV attacks. Although HTTPS have been widely adopted in modern web apps, there is still much legacy code using HTTP.

This scenario is feasible, especially considering many public unsafe WiFi hotspots are available [24]. Consider a possible scenario: attackers may set up a free WiFi hotspot in a crowded place. Nearby smartphone users may use this WiFi. If these users open vulnerable apps (e.g.,Facebook and skype) and click web links, apps' WebView may load these links. If the loaded web content embeds iframes/popups using unsafe network channels (e.g., HTTP), attackers may inject malicious code into the iframes/popups and launch attacks.

Web attacks. The inclusion of *third-party* content usually introduces security implications [26, 36]. Hence, we assume web attackers may be the owner of a third-party domain (e.g., ads.com) severing an embedded untrusted iframe/popup. Our empirical study on a set of popular hybrid apps and mobile websites shows iframes/popups are frequently used to load third-party content, especially third-party advertising and tracking content. Existing work has demonstrated that third-party advertising [28, 56] and tracking [14, 32, 37, 42, 46] services often causes serious security concerns. More than that, as figured out by existing work [39, 48], a third-party iframe may even directly work as a malicious entry point for malware.

This scenario is also possible in practice. For example, as demonstrated in prior work (e.g., [36]), some domains may expire, which still commonly occurs in recent years. Attackers may register and get the control of these domains. If these domains are embedded by some websites in iframes/popups, attackers may broadcast these websites to lure users to access them using corresponding vulnerable apps (e.g., Facebook or Facebook Messenger). In the vulnerable apps, WebView may be started, and also access the domains controlled by attackers. Thus, attackers obtain chances to inject malicious code and launch attacks.

Furthermore, as discussed in Section 2.1, considering the security of the popup behavior is one of our research objectives, we assume the popup-creation ability of an iframe/popup is enabled in its sandbox attribute.

3 Differential Context Vulnerabilities

In this section, we mainly focus on DCVs, and also explain why existing defense solutions are ineffective to prevent DCV attacks. We first show the overview of our security study, and then present the details of each vulnerability. Last, we discuss the advantages of DCV attacks over existing attacks, also with the analysis of the root causes of DCVs.

3.1 Study Overview

Guided by the inconsistencies between regular browsers and WebView (Section 2.2), our security study of iframe/popup behaviors is mainly concerned with the following three dimensions:

The application of common origins. As introduced in Section 2.2, WebView content initialization APIs may create the main frame with common origins, such as "file://" and "null". For example, the invocation

```
WebView.loadurl('file:///android_asset/index.html')
```

can load a local file with the origin "file://", while *WebView.loadData()* and *WebView.loadDataWithBaseURL()* may create a main frame to load web data with the "null" origin.

However, these common origins are not unique for the main frame, and may be reproduced by untrusted iframes/popups in their inside sub-frames for launching attacks. More specifically, if an untrusted sub-frame can generate a new nested sub-frame "F_{nested}" with above common origins, the untrusted sub-frame may place its essential attack code inside F_{nested} to make risky operations, which are aimed to attack all potential objectives, including the main frame, other sub-frames, or WebView itself. In the attack process, the victims may validate the operations by checking the corresponding origins. However, the origin information they can obtain is F_{nested}'s origin, rather than the real origin (i.e., the origin of the untrusted sub-frame). Considering F_{nested} have the same origin as the main frame, the origin validation process fails. Finally, the victims may treat untrusted operations as benign operations and handled them as usual.

Our study confirms that a sub-frame is not allowed to generate a new sub-frame with the "file://" origin, due to built-in security policies (Section 2.1). However, a nested sub-frame with a "null" origin can still be generated by using the data scheme URL (e.g., *<iframe src="data://...">*), which is frequently used to load simple HTML code (such as images)

in the web platform. Although SOP can prevent cross-frame scripting between two "null" origins (e.g., the main frame and F_{nested}), untrusted sub-frames can still leverage the "null" origin to make several nefarious actions (Section 3.2).

Concise WebView UI design. As discussed in Section 1, WebView's UI design causes security risks that untrusted iframes/popups may perform phishing attacks, if they have the abilities of 1) manipulating the rendering order of multiple WUIs; 2) navigating the main frame. To verify the former potential ability, we first conduct an empirical study on a set of popular hybrid apps. This study is aimed to understand how WUIs are managed in practice. We find Android takes the responsibility of managing multiple WUIs. Our study also shows when a popup is created, Android place its WUI behind current WUI *at default*.

This WUI management strategy seems safe. However, it does not meet app development requirements. Instead, some apps manage WUIs by themselves, which is yet error-prone due to the design flaws of the WebView event handler system (Section 3.6). As a result, the crucial ability of manipulating the WUI rendering order is exposed (Section 3.3.1). Thus, an untrusted iframe/popup can get the ability of overlapping begin WUIs with its own WUI. Our study also shows that even when Android's default WUI management strategy is adopted, it is still possible for untrusted iframes/popups to change the WUI rendering order by combining WUI creation and closure operations (Section 3.3.2).

Second, to confirm the latter potential navigation ability, we study the navigation policies of WebView. We find WebView inherits permissive navigation policies from Chrome/Chromium. These navigation policies have been well investigated in the context of regular browsers (Section 2.1), but rarely scrutinized in the context of WebView. These navigation policies allow an untrusted sub-frame to navigate the main frame. Due to the lack of the address bar, the navigation based attack is stealthier and more powerful in the context of WebView (Section 3.4.1).

Note that the above navigation can be disabled by iframe sandbox (Section 2.1). But considering iframe sandbox is hardly used in practice, the attack is still prevalent and has negative security impacts in real-world hybrid apps. This is also verified in our evaluation (Section 5.2).

WebView programming features. As discussed in Section 1, WebView's programming features may impact the effectiveness of existing defense solutions. To verify it, we extensively test these protection solutions' performance, when different programming features are enabled. Consequently, we identify a critical conflict between WebView programming features and web popup-creation manners. By leveraging this conflict, untrusted iframes/popups can perform **privileged** main-frame navigation attacks, even when this sub-frame's navigation capability is disabled by iframe sandbox (Section 3.4.2).

DCVs and DCV attacks are summarized in Table 1. More details are discussed below.

3.2 Origin Hiding Attacks

As introduced in Section 3.1, in the context of WebView, security risks are introduced that untrusted iframes/popups may leverage the "null" origin (created through the data scheme URL) to hide their own origins while making stealthy risky actions. In this section, we introduce two extended attacks: attacking web messaging integrity (Section 3.2.1) and stealthily accessing web-mobile bridges (Section 3.2.2).

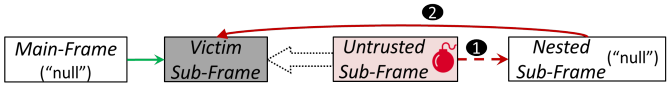


Figure 6: Attacking Web Messaging

3.2.1 Attacking Web Messaging

Figure 6 shows an attack scenario for web messaging. Assume the main frame whose origin is "null" sends web messages to a benign victim sub-frame. Meanwhile, the main frame also contains an untrusted sub-frame. If the untrusted sub-frame spawns a new nested sub-frame F_{nested} with the "null" origin, and let F_{nested} send a fake message to the victim sub-frame, the victim sub-frame may be fooled.

As shown in Listing 1, the victim sub-frame may validate the origin of the received message to ensure the message is from an authorized frame. However, this may not still recognize the fake message because the fake message has the same origin as the main frame. As a result, the victim sub-frame may handle the message as normal. If the victim sub-frame carries sensitive functionalities, these functionalities may be leveraged, and serious consequences may be caused.

```
1 // Message Handler
2 onmessage = function (e) {
3     // Validating the message source origin
4     if (e.origin == "null") { // From main frame?
5         // Making sensitive actions here
6     }
```

Listing 1: Validating the Message Origin in the Victim Sub-frame

In addition to the above origin validation based protection, the above attack cannot also be prevented by other defense solutions, such as [11,44,47,52], because it is challenging for them to distinguish between the main frame and F_{nested}.

3.2.2 Accessing Web-Mobile Bridges

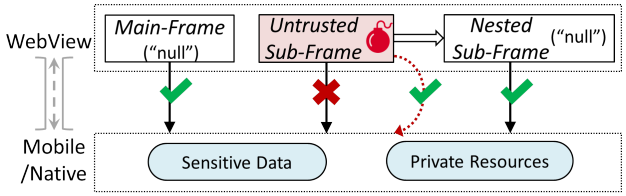


Figure 7: Freely Accessing Web-Mobile Bridges

As shown in Figure 7, the security risks are also posed that untrusted iframes/popups can also secretly access web-mobile bridges by leveraging the "null" origin (Listing 2), but

without being blocked by existing defense solutions. This is because existing defense solutions are coarse-grained, and the origin they can obtain is F_{nested}'s (i.e., "null"), rather than the origin of the untrusted iframes/popups. Hence, they would approve the untrusted operation.

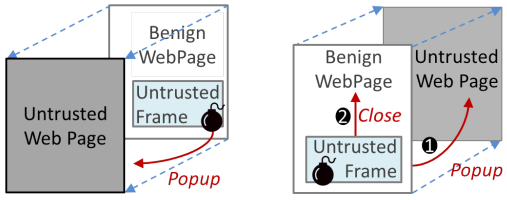
To verify the attacks, we develop two proof-of-concept (POC) apps that can launch the attacks. Then, we test their performance when the-state-of-the-art protection solution "NoFrak" [21] and "Draco" [49] are enforced respectively. NoFrak extends SOP to the mobile layer of a third-party development framework, while Draco implements the access control in WebView. In the first POC app, we integrate the popular third-party hybrid development framework "Apache Cordova" and instrument its plugin manager to implement NoFrak. In the second POC app, we use our instrumented WebView library, which implements Draco's prototype system [49]. In both POC apps, we find that untrusted accesses by DCV attacks on web-mobile bridges, especially JavaScript bridges, cannot be prevented.

```
1 // Creating a nested sub-frame with the data scheme URL
2 var ifrm = document.createElement('iframe');
3 // Triggering onJsAlert()
4 ifrm.setAttribute('src', 'data:text/html;charset=UTF-8,<
      html>...<script>alert(\I am the main frame\', \'*\')</
      ' + 'script>');...
5 document.body.appendChild(ifrm);
```

Listing 2: Accessing the Event Handler onJsAlert() in the Untrusted Iframe/Popup

3.3 WebView UI Redressing Attacks

The root cause of the attacks is that there is no protection on the WUI rendering order and WebView UI integrity. Hence, the security risks exist that untrusted iframes/popups can freely manipulate it and perform phishing attacks. In this section, we illustrate two extended attacks: the WUI overlap attack (Figure 8-a), and the WUI closure attack (Figure 8-b). We next describe them in detail.



(a) WUI Overlap Attack (b) WUI Closure Attack

Figure 8: WebView UI Redressing Attacks

3.3.1 WebView UI Overlap Attack

```
1  // Customizing onCreateWindow() to enable popup-creation
2  boolean onCreateWindow(WebView view, ...) {
3      // Creating a new WebView UI
4      WebView myNewWebView = new WebView(getContext());
5      // Initializing the new WebView UI
6
7      // Putting the new WebView UI before current WebView UI
8      view.addView(myNewWebView);
9
10     // Providing the new WebView UI to Android
11     ...
```

Listing 3: Vulnerable *onCreateWindow()*

Listing 3 shows a representative but vulnerable implementation of the event handler *"onCreateWindow()"*. When a popup is created, the event handler is triggered and may select to put the new WUI in the front of current benign WUI by calling "*ViewGroup.addView(new WebView)*" (Line 8). Thus, the new WUI is presented to users. However, this ability of changing the WUI rendering order can also be obtained by untrusted web code. This is mainly because the event handler *onCreateWindow()* cannot distinguish between benign and untrusted requests, due to its design flaws (Section 3.6).

As a result, untrusted iframes/popups obtain the ability of performing phishing attacks by simply triggering a popup-creation event, and letting the created WUI load fake web content and overlap the benign WUI. Due to the lack of the address and tab bars, this risky popup-creation operation may be hardly noticed by users. As shown in Listing 4, the overlap attack can be easily set up in practice.

```
1 // Using HTML Code
2 <a href="https://attacker.com" target="_blank" ...
3 // or Calling JavaScript code
4 window.open("https://attacker.com", "_blank" ...)
```

Listing 4: Exploit Code of the WUI overlap attack and the privileged navigation attack

We note that the key API name "addView" also appears in existing work on Android UI redressing attacks such as [35]. However, these APIs are totally different. In existing work, "addView" means "WindowManager.addView()", which is used to change UI layout between different apps. In this paper, "addView" means "ViewGroup.addView()", which is used to change a specific UI layout inside an app. To our knowledge, we are the first to discuss the security risk of the latter API.

3.3.2 WebView UI Closure Attack

When apps use the default Android WUI management strategy, it is still possible for an untrusted iframe/popup to change the WUI rendering order (Section 3.1). As shown in Figure 8-b, the untrusted iframe/popup may first create a new popup window, whose corresponding WUI is placed behind current benign WUI. Then, the untrusted code triggers the window-closure event, which is handled by the event handler "onCloseWindow()". If the event handler is vulnerable and removes the foremost benign WUI (Line 8 in Listing 5) from the WUI rendering order, the former untrusted WUI appears instead and phishing attacks may occur. Similar to the WUI overlap attack, due to the lack of the address and tab bars, such attacks are stealthy, and can be easily launched in practice (e.g., using the code in Listing 6).

```
1 // Customizing onCloseWindow() to enable WebView UI closure
2 public void onCloseWindow(WebView window) {
3     super.onCloseWindow(window);
4     // Destroying the WebView UI being closed
5     ...
6
7     // Removing the WebView UI being closed from current
          view layout
8     myRootWebViewLayout.removeView(window);
9 }
```

Listing 5: Vulnerable *onCloseWindow()*

```
1 // Creating a new WebView UI
2 window.open("https://attacker.com", "_blank" ...)
3 // Closing current WebView UI
4 window.close()
```

Listing 6: Exploit Code of the WUI Closure Attack

We note that as introduced in Section 1, WebView UI redressing attacks cannot be defended by existing Android UI protection solutions. These two UI redressing attacks are different. Android UI redressing is performed between different apps, while WebView UI redressing occurs within one app.

3.4 Main-Frame Navigation Attacks

3.4.1 Traditional Navigation Attack

Untrusted iframes/popups can leverage traditional navigation policies (Section 2.1) to launch phishing attacks (e.g., using the code in Listing 7 to perform phishing attacks), when their navigation capabilities are not disabled. Due to the lack of URL indicators (e.g., the address bar), the attack is stealthier and may be hardly noticed by users.

```
// Using HTML Code
<a href="https://attacker.com" target="_top" ...
// Or Calling JavaScript code
window.open("https://attacker.com", _top, ...
```

Listing 7: Leveraging Traditional Navigation Policies

3.4.2 Privileged Navigation Attack

Even when the navigation capability is disabled by iframe sandbox (which prevents the above traditional navigation-based attack directly), it is still possible for untrusted iframes/popups to launch privilege escalation attacks and obtain the ability of performing navigation attacks. This is mainly caused by the inconsistencies between the WebView programming features and web regular navigation actions. When web popup creation code (e.g., <*a*> and *window.open()*) is executed in a sub-frame, Android always tries to select a WUI to show the popup content. Note that the WUI selection *always* occurs, even when popup-creation is disabled in the mobile layer (e.g., the setting *SupportMultipleWindows* is false). However, when popup-creation is not allowed, there is not a new WUI for rendering. Instead, Android selects current WUI for showing the popup content, which means the main frame is navigated to the popup. Thus, phishing attacks may occur.

In practice, the privileged navigation attack can be easily launched by using the exploit code shown in Listing 4. Note that this code is also used for launching the WUI overlap attack. When popup-creation is disabled (*by default*), the code may launch the navigation attack. Otherwise, the WUI redressing attack may be available.

3.5 Advantages of DCV Attacks

Compared to existing Android attacks (such as Trojan attacks [5]), DCV attacks do not require declaring permissions, or carrying payload. Compared to other WebView-based attacks (e.g., [21, 25, 30, 51]), which require JavaScript or JavaScript-bridges to be enabled, DCV attacks do not have these requirements and limitations. More importantly, DCV attacks are more powerful that attackers may obtain abilities to not only access web-mobile bridges, but also directly leverage critical web features.

Furthermore, different from existing MITM attacks on a sub-frame inside WebView, DCV attacks cannot be prevented by existing web protections (e.g., SOP). Unlike existing touch hijacking in WebView [31], DCV attacks do not need to control the mobile code, and craft the placement of multiple WebView components in Activity layout XML.

In addition, DCVs can be leveraged to boost other attacks. For example, event-oriented attacks [53] rely on triggering WebView event handlers, but it is difficult to trigger several critical event handlers (e.g., *onPageStarted()* and *onPageFinished()*). This problem can be well solved through exploiting DCVs, such as the privileged navigation attack (Section 3.4.2).

3.6 Root Causes of DCVs

DCVs are rooted in the inconsistencies between WebView and regular browsers in terms of UI and programming features (Section 1 and 3.1). We demonstrate several critical and frequently used web features and behaviors are harmless and safe in the context of regular browsers, but they become risky in the context of WebView.

In addition, we also find the design of the event handler features is also flawed. In theory, through event handlers, developers have chances to reject DCV attacks. However, unfortunately, the design flaws of event handlers make it extremely difficult to achieve the goal. For example, when the WUI overlap attack is performed, the event handler *''onCreateWindow(view,isDialog,isUserGesture,resultMsg)''* is always triggered. If the event handler could deny the creation of an untrusted WUI, attackers would fail to launch the WUI redressing attack. However, this is very difficult because the event handler *onCreateWindow()* does not provide the victim app any origin information about who is creating a popup and what content is being loaded in the popup. Thus, the victim app has to *blindly* allow or deny *all* popup-creation operations, no matter whether the operations are made by benign or untrusted code. In addition to *onCreateWindow()*, other event handlers such as *onCloseWindow()* face similar problems.

Another event handler *shouldOverrideUrlLoading(view,request)* (as introduced in Section 2.2) is always triggered when a URL loading event occurs. This event handler provides the information of the URL that is being accessed, which may be used as a complement of other event handlers to prevent DCV attacks (e.g., allow the victim app to deny untrusted URLs). However, the combination is hardly used in practice. Even when the associated URL is identified and denied, the new WUI is already created and still in the control of untrusted iframes/popups. Untrusted iframes/popups may still use the new WUI to consume the resources (such as CPU and memory) of the victim devices in background. Hence, to avoid this, it is required for the victim app to always explicitly destroy the new WUI.

In addition, *shouldOverrideUrlLoading()* often has its own implementation problems in origin validation. For example, our empirical study shows some hybrid apps do not even per-

form any check, and some of them only check the domain of the URL but ignore the scheme (e.g., "HTTP" or "HTTPS").

4 DCV-Hunter

There are several tools for analyzing hybrid apps [22, 53, 55], however, it is challenging to directly apply these tools to detect DCVs. On the one hand, existing static analysis tools are not designed for the analysis of iframe/popup behavior (e.g., [22, 55]), and they are often coarse-grained (e.g. [33]). More specifically, they can hardly extract and reconstruct the context information of each WebView instance. When there are multiple WebView instances in a hybrid app, which is common in practice, these tools can produce high false positives. On the other hand, existing dynamic analysis tools (e.g., [53]) have high false negatives, as it is very difficult to trigger a WebView instance at runtime. For example, as shown in Figure 5, to trigger WebView inside the Facebook Messenger app, the analysis tools need to automatically log in and open a URL link.

We propose a novel static detection tool, *DCV-Hunter*, that utilizes program analysis to automatically vet apps. As shown in Figure 9, DCV-Hunter's approach is four-fold. Given an app, DCV-Hunter first generates its complete call graph (CG). Next, DCV-Hunter leverages CG to reconstruct the context of each WebView instance. Then, DCV-Hunter verifies if untrusted sub-frames exists. Finally, DCV-Hunter determines if the given app is potentially vulnerable or not.

4.1 Complete Call Graph Construction

We leverage FlowDroid [10] to generate call graphs (CG) of the target app. However, we find FlowDroid faces challenges to analyze WebView related function invocations. This is mainly due to the missing of type information and semantics related to WebView (e.g., the semantics of WebView event handlers). To mitigate this issue, we patch the target app during CG construction by inserting extra instructions, which provide necessary type and semantic information of WebView. Thus, FlowDroid can generate necessary edges and construct complete CG.

4.2 WebView Context Reconstruction

In this phase, DCV-Hunter re-constructs the whole context for each WebView instance. First, DCV-Hunter identifies all WebView instances from CG. Then, DCV-Hunter separately reconstructs each WebView instance's own context, which includes 1) the URL or HTML code to be loaded; 2) settings (e.g., the enablement of popup creation); 3) implementation of event handlers (e.g., "*onCreateWindow()*" and "*onCloseWindow()*"). To reconstruct the WebView context, points-to analysis is applied [33]. For example, when an event handler class that contains the implementation of event handlers is configured through the API "*setWebChromeClient(...)*", DCV-Hunter can check the points-to information of the API's parameter, and retrieve the parameter's actual class name.

However, points-to analysis does not scale well, especially when the target app is complex. To mitigate the problem, we also apply the data flow tracking technique (also provided by FlowDroid) as a complement. For example, when an event handler class is instantiated, the corresponding instance is treated as source. Then, the event handler configuration APIs (e.g., "*setWebChromeClient(...)*") are treated as sink. Finally, if there is a flow between above source and sink, the event handler class should be a part of the context of the corresponding WebView instance.

In addition to an event handler class, several context-related objects (e.g., URL strings, WebView settings) can also be analyzed using data flow tracking. These objects and their corresponding APIs are treated as source and sink, respectively. More details are shown in Table 3. Note that different from WebView settings and event handlers, which are often class instances, the URL source may have several different formats, such as 1) HTML code or URL string; 2) Intent messages (inter-component communication in Android). Both formats are often used in real-world apps. For example, as shown in Figure 5, in Facebook Messenger, when a link is clicked, an Intent message that includes the link is sent out to an activity (Andrioid UI) to start WebView and show that link.

Table 3: Source and Sink APIs

Source	Sink
URLs	WebView content loading APIs
Settings	WebView Setting APIs
Event Handlers	setWebViewClient()
	setWebChromeClient()
WebView	WebView content loading APIs
	WebView Setting config APIs
	Event handler registration APIs

4.3 Untrusted Iframe/Popup Detection

In this phase, given a WebView instance, DCV-Hunter checks whether an untrusted iframe/popup is included in its loaded content. To achieve the goal, DCV-Hunter first extracts the URLs of the untrusted iframe/popup, and then examine the event handler "shouldOverrideUrlLoading()" (Section 2.2) through path constraint analysis to determine whether extracted URLs are approved.

4.3.1 Untrusted URL Extraction

Given a WebView instance, the web content loaded in WebView is analyzed based on its formats:

- *HTML code*: This format is usually used by the content loading APIs "*loadData()*" and "*loadDataWithBaseURL()*" (for origin-hiding attacks). Based on the patterns of iframes/popups (Section 2.1), all internal associated links can be extracted and then checked. On the one hand, if a link is unsafe, such as using HTTP, code injection surface should exist, and the link is untrusted. On the other hand, if a link uses HTTPS, it is difficult to determine if the link is third-party, considering the main frame does not have an explicit domain (i.e., the "null"

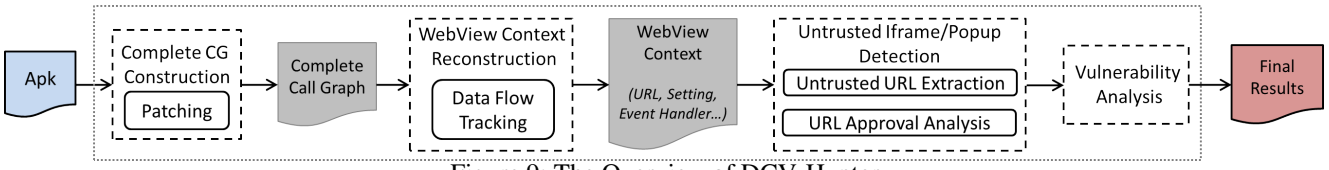


Figure 9: The Overview of DCV-Hunter

origin).

To mitigate the problem (i.e., determine the first-party URLs), we leverage several heuristics: 1) inside the target app, WebView class name and its internal package names are usually related with developers' website. Hence, we reverse them as first-party URLs. Please also note that the reversed class and package names should not be related to third-party URLs (e.g., [3]). 2) We also check the app information that is provided by developers in Google Play. This information includes the links of developers' home page, email and "privacy policy". Finally, these links are also treated as first-party URLs, since they are likely trusted by developers.

- *URL links*: DCV-Hunter handles URL links, based on their formats. If a URL is a network link, we build a crawler based on Selenium [7] to automatically collect the webpages (the mobile version) that can be navigated to from the URL within three depth levels. For each collected web page, its sub-frame is checked based on our threat model (Section 2.3).

  If URL is a local file link (e.g., "file://..."), DCV-Hunter first dumps the corresponding local file from the target app, and then handles it like above regular HTML code. This is mainly because the file scheme link is similar with the null origin and does not provide any first-party domain information.
- *Intent*: Our empirical study on a set of popular hybrid apps shows that the values of the links saved in an intent message may be arbitrary. Hence, to avoid potential false negatives, DCV-Hunter assumes that this format of web content contains untrusted iframes/popups.

4.3.2 URL Approval Analysis

To determine whether an extracted untrusted URL is approved by the event handler "shouldOverrideUrlLoading()" or not, we perform a path-sensitive constraint analysis on the event handler code. The key observation behind the idea is that based on the specification of the event handler [9], when untrusted iframes/popups are opened or created, the event handler is triggered, and should return false (Please note returning true is usually used for denying the link or other purposes [53]).

Below is our solution. We construct the conditions (constraints over strings) of the paths to "returning false", and check whether the extracted URL can satisfy the conditions. More specifically, based on the CG and control-flow graph of the event handler, we first find all the possible paths to the key instruction "returning false". Then, starting from each key instruction, we perform a fast backward slicing along each path

Table 4: APIs for the Analysis of WUI redressing problems

Attacks	Sensitive APIs
Overlap	*ViewGroup.addView()*
Closure	*ViewGroup.RemoveView()*
	WebView.setVisibility()
	...

to construct the path constraints. The unknown variables in the constraints are all over the string parameters (i.e., URL or request) of "shouldOverrideUrlLoading()". After that, based on our threat model and the content of extracted URLs, we add more constraints to the collected constraints, including

```
    1) <parameter>.scheme == "HTTP"
or  2)  <extracted_URL>.domain == <parameter>.domain.
```

The first constraint is aimed to check if attackers can freely inject code into the sub-frame through MITM attacks. The second constraint is used to verify if the domain of the extracted URL is approved. Finally, we use an SMT solver (i.e., z3 [19]) to solve all constraints. If path constraints can be satisfied, it indicates that the extracted URL should be approved.

Our path constraint analysis is implemented by embedding and extending the symbolic execution module of our previous work "EOEDroid" [53]. Please also note we model several frequently used Java classes (e.g., WebResourceRequest, URL, and String) to support the related operations.

4.4 Vulnerability Analysis

To determine each vulnerability, DCV-Hunter checks its conditions respectively:

- *Origin-hiding*: DCV-Hunter first verifies whether the origin of the main frame is "null". This is done by checking the corresponding WebView content loading APIs and their associated parameters. Then, for convenience, the valuable attack targets are also checked, such as web messaging or web-mobile bridges.
- *WUI redressing*: DCV-Hunter first verifies WebView's settings and event handlers to check whether WUI creation and closure are enabled. Then, DCV-Hunter checks whether the corresponding event handlers *onCreateWindow()* or *onCloseWindow()* are vulnerable or not. This is done by checking the existence of the sensitive APIs listed in Table 4. Based on the analysis of the design flaws of these event handlers (Section 3.6), which have to blindly approve or deny all requests, these simple checks can obtain high accuracy.
- *Main-frame navigation*: For the traditional navigation based problem, iframe sandbox is checked. If iframe sandbox is used, DCV-Hunter then verifies if the navigation capability is disabled. For the privileged navigation attack,

DCV-Hunter checks whether multiple window mode is disabled, which is done by directly checking associated settings.

5 Security Impact Assessment

To assess DCVs's security impacts on real-world popular apps, we collected 17K most popular free apps from Google Play. They are gathered from 32 categories, and each category contains 540 most popular apps. By applying DCV-Hunter on these collected apps, we found 11,341 apps contained at least one path from their entry points to WebView content loading APIs. Among them, 4,358 apps (38.4%) were potentially vulnerable, including 13,384 potentially vulnerable WebView instances and 27,754 potential vulnerabilities (Table 5). This indicates DCVs widely impact real-world apps.

We evaluated the accuracy of DCV-Hunter by measuring its false positives. We randomly selected 400 apps from the apps flagged as "potentially vulnerable" by DCV-Hunter, and manually checked them (see more details in Section 5.1). We find 6 of them (1.5%) are false positives. Our further inspection revealed in four of these apps, during the reconstruction of the URL loaded by WebView (Section 4.2), some unrelated URLs were accounted, due to the imprecise taint analysis (i.e., overtaint). For the remaining two apps, "URL Approval Analysis" (Section 4.3.2) on untrusted iframe/popup links faced difficulty in handling constraints that contained string regular expressions. We leave addressing these weaknesses as our future work.

All experiments were run on a high-performance computer. We ran DCV-Hunter with 100 processes in parallel and each process was assigned with limited resources (two regular computing cores and 8GB memory). Our time cost showed each process needed 144 seconds for each app.

5.1 Manual Verification

To manually verify target apps, we firstly modify Android source code (version 6) to let it print necessary WebView related information. Next, we install the modified Android system in a real device (Nexus 5). Then, we test target apps. For each app, when internal WebView instances are started, we inject attack code to target iframes/popups. Last, based on the web content shown in WebView and the logs printed by Android, we determine if the attack code works and the app is vulnerable.

Please note that different from prior work, we do not use proxy for code injection. We find proxy has several shortcomings. For example, it is time consuming and inefficient to locate the target iframes/popups for code injection. Instead, we leverage Chrome's USB debug interfaces to ease our test. Since we run test in a real device, we connect the device with PC using USB. Then, we open Chrome in PC to inject code to target WebView instances. For example, we select a WebView instance and then open console (in Chrome) to run extra attack code for code injection. But please always keep in mind that before executing any code, we must select a (target) sub-frame as the code execution environment in console.

Table 5: Potential Vulnerability Details

Potential Attacks	Impacted WebView	Impacted Apps	App Downloads
Origin-Hiding	1,737	1,238	3.5 Billion
WUI Overlap	138	89	8 Billion
WUI Closure	5	5	13 Million
Traditional Navigation	13,384	4,358	19.5 Billion
Privileged Navigation	12,490	4,161	17.8 Billion
Total	13,384	4,358	19.5 Billion

5.2 Findings

Many high-profile apps are impacted by DCVs. DCVs widely exist in hybrid apps. Up to now, the potentially vulnerable apps have been downloaded more than 19.5 Billion times (the fourth column of Table 5). Furthermore, these also include many manually verified popular apps (some examples are shown in Table 6) such as Facebook, Instagram, Facebook Messenger, Google News, Skype, Uber, Yelp, U.S. Bank.

Almost all categories of apps are affected. Figure 10 shows the related distribution data. The light blue line and the bars respectively represent the distribution of potentially vulnerable apps and each potential vulnerability in each category. Almost all categories of apps are impacted, including several sensitive categories (e.g., password management and banking apps). This indicates DCVs are common.

We observe some categories are more subject to DCV attacks than others, such as news, dating, and food-drink. We manually analyze a set of apps in these categories, and find these categories of apps use WebView more often to load third-party untrusted content in iframes/popups. For example, the Google News app (one billion+ downloads) provides the news collections to users. It allows any website to be loaded in its WebView. We manually check several news links and find it is common for these news web pages to embed third-party content, especially ads and tracking services.

We also find in some apps, their loaded web pages are safe, and do not include any untrusted content. However, after the web pages are fully loaded, these apps run extra JavaScript code through the API "*WebView.evaluateJavascript()*" to created and embedded new iframes/popups for loading ads content, which introduces security risks.

Furthermore, we find the events and news apps are more likely to suffer from WUI redressing attacks. This is mainly because these apps tend to manage WUIs by themselves. For example, in some news apps, when a user scrolls down to the bottom of the web page, the apps will directly append and show more content, without letting the user click a "next page" button. When the user clicks a concrete news link, a new WUI is created and placed in the front of current WUI to show that link. When the user finishes that web page, developers can close current WUI and show previous WUI. In this way, the state of previous WUI is not changed, and the dynamically appended content is also kept. This rendering

Table 6: Summary of Example (Manually Verified) Vulnerable Apps/Libraries
(* can be any domain, while OH, WO, WC, TN, PN, and BA respectively mean Origin-Hiding, WUI Overlap, WUI Closure, Traditional Navigation, Privileged Navigation, and Blended attacks.)

Apps/Libraries	Possible Attack Scenarios		Vulnerabilities						Downloads
	Main-Frame	Untrusted Sub-frame	OH	WO	WC	TN	PN	BA	
Facebook	*	*		✓		✓		✓	1 Billion+
Instagram	*	*		✓		✓		✓	1 Billion+
Facebook Messenger	*	*		✓		✓		✓	1 Billion+
Kakao Talk	*	*		✓		✓		✓	1 Billion+
Google News	*	*				✓	✓		1 Billion+
Skype	*	*				✓	✓		1 Billion+
WeChat	*	*				✓	✓		100 Million+
Yelp	*	*				✓	✓		10 Million+
Kayak	*	*		✓		✓			10 Million+
Uber	uber.com	third-party tracking		✓		✓			100 Million+
ESPN	espn.com	third-party tracking		✓		✓			10 Million+
McDonald's	mcdonalds.com	third-party tracking				✓	✓		10 Million+
Samsung Mobile Print	*	*				✓	✓		5 Million+
lastpass	*	*				✓			5 Million+
dashlane	*	*				✓	✓		1 Million+
1password	*	*				✓	✓		1 Million+
U.S. bank	*	*				✓	✓		1 Million+
Huntington bank	huntington.com	third-party tracking				✓	✓		1 Million+
Chime mobile bank	*	*				✓	✓		1 Million+
Facebook Mobile Browser Library	*	*		✓		✓		✓	
Facebook React Native Library	*	*				✓	✓		

strategy improves user experience. However, as described in Section 3.6, due to the design flaws of the event handler system, such a WUI management strategy is also exposed to untrusted iframes/popups, and cause security issues.

Traditional and privileged navigation attacks impact more apps than other DCV attacks. As summarized in the second and third columns of Table 5, navigation based attacks are more popular than the other vulnerabilities. It is mainly because the security assumptions of these two attacks are more easily satisfied. For example, many WebView instances prefer using the default configuration (e.g., disabling popup-creation), and suffer from privileged navigation attacks.

The traditional navigation based attack causes more serious consequences in the context of WebView. This type attack almost affects all potentially vulnerable apps. One important reason is that the effective defense solution "iframe sandbox" is hardly used in practice. There are several reasons. First, it may be difficult to add the sandbox attribute to an iframe, especially considering developers have to find the corresponding web code of that frame from a large amount of web files and code. Second, it is difficult to manage the sandbox configurations for each iframe. Each iframe has its own specific security configurations, including disabling JavaScript or navigation. When the iframe number rapidly rises, the configuration management may become quite difficult. Third, iframe sandbox is not flexible. Its configurations are often bound with iframes, rather than origins. If an iframe is navigated to a different origin, it is hard for developers to update the sandbox restriction policies.

5.3 Case Studies

We have successfully manually launched DCV attacks in many popular apps (some examples are shown in Table 6). Readers can find also several video demos at [2] (the website is anonymized). In this section, we present two example apps (Skype and Kayak) in detail, and also briefly discuss other examples listed in Table 6.

5.3.1 Skype

This is a very popular communication app (one billion+ downloads). Our study shows it suffers from traditional and privileged main-frame navigation attacks. A possible attack scenario is shown in Figure 11. An attacker sends the victim user a message containing a benign but vulnerable link (e.g., ebay.com). When the user clicks the link, a WebView instance is started to render that link (Figure 11-b). However, the loaded web page includes third-party untrusted tracking web content (e.g., double-click) in iframes. The embedded untrusted content has the ability to secretly navigate the main frame through traditional or privileged navigation attacks, which may result in stealthy phishing attacks (Figure 11-d).

We also observe when a web page is opened, its URL (e.g., ebay.com) is shown in the top of the app. This is relatively helpful to mitigate DCV attacks. However, after the web content is fully loaded by WebView (Figure 11-c), we find the URL is replaced by the title of the loaded web page. After that, the URL will not be shown again, even when a naviga-

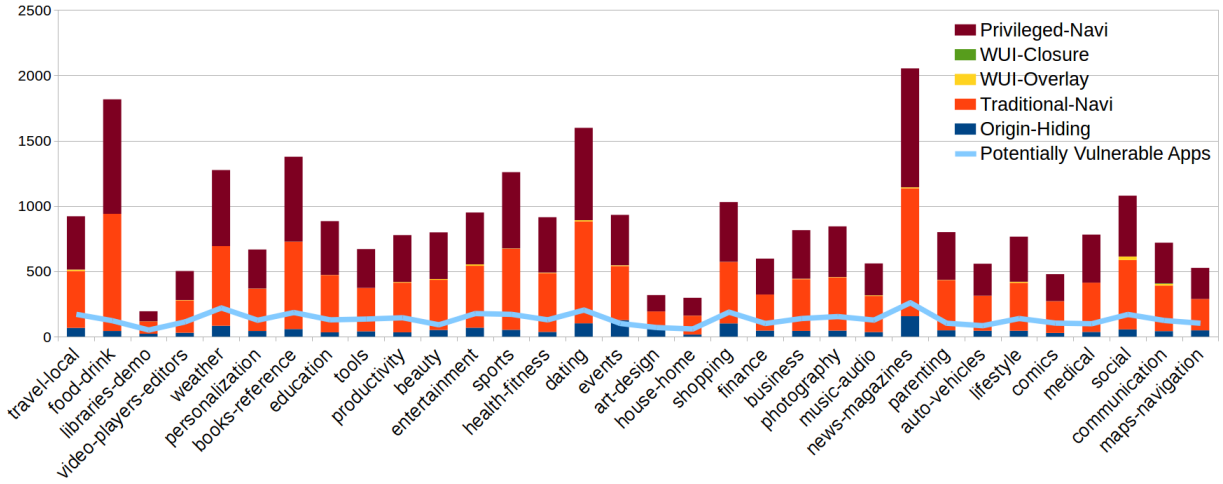


Figure 10: Distribution of Potentially Vulnerable Apps and Potential Vulnerabilities

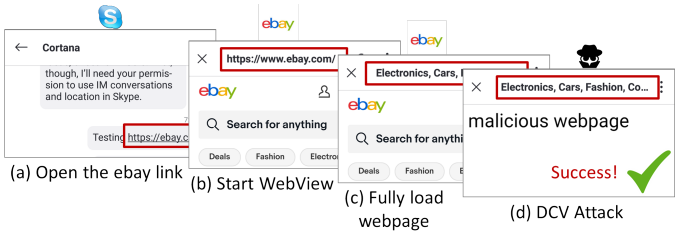


Figure 11: Attacking Skype

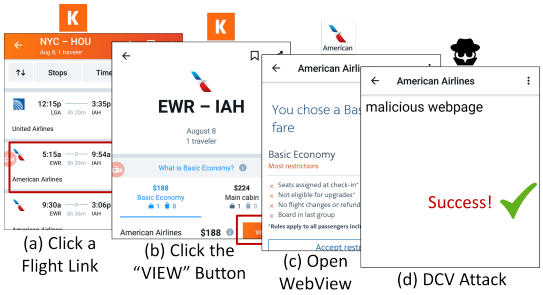


Figure 12: Attacking Kayak

tion event occurs. Hence, when the phishing attack occurs, the victim user may hardly be aware of it.

5.3.2 Kayak

It is a leading app (ten million+ downloads) for providing traveling-relevant searching services, which are aimed to help users find better prices of flights, hotels, rental cars, and so on. However, as shown in Figure 12, it suffers from WebView UI redressing attacks, which may cause account information leakage and financial losses. Consider a possible scenario that a user is searching a flight. The user clicks one of the searching results (Figure 12-a), such as the AA flight, and then clicks the "View" button to get more details (Figure 12-b).

Next, a customized WebView instance is triggered to show more flight details from "aa.com" (Figure 12-c). However, in the AA web page, an extra iframe is embedded to load third-party tracking content (tag management). In the Kayak app, the untrusted iframe obtains the ability of performing phishing attacks by leveraging the WUI overlap issue (Figure 12-d).

In addition, similar with the Skype app, the Kayak app also provides a title bar to reduce the UI inconsistencies. However, this is limited to defend against DCV attacks, since the opened fake web pages often have the same title content.

5.3.3 More Examples

In addition to Skype and Kayak, more examples listed in Table 6 are discussed below.

- *Facebook Mobile Browser, Facebook, Instagram, and Facebook Messenger*: The Facebook Mobile Browser library is frequently used in Android apps, such as Facebook, Instagram, and Facebook Messenger. In our study, the traditional navigation and WUI overlap vulnerabilities exist. As shown in Section 1 and Figure 5, an address bar is provided in the library and is helpful to mitigate DCV attacks. However, as discussed in Section 5.4, the address bar may face pixel and race condition flaws. By leveraging these flaws, untrusted sub-frames can still obtain the ability of launching phishing attacks.
- *Kakao Talk*: Kakao Talk is a popular instant messaging app. Although Kakao Talk is not equipped with the Facebook Mobile Browser library, it is also impacted by the above race condition flaw (Section 5.4).
- *Google News*: As introduced in Section 5.2, the Google News app can show any news websites. When there is an untrusted sub-frame in the rendered news web page, which is common in practice, the untrusted sub-frame can perform traditional or privileged navigation attacks.
- *WeChat*: WeChat is another popular instant messaging app. Similar with Skype (Section 5.3.1), WeChat also faces traditional and privileged navigation vulnerabilities.
- *Yelp*: The Yelp app are also impacted by traditional and privileged navigation vulnerabilities. Different with Skype and WeChat, Yelp's WebView is triggered by clicking the

homepage link of a restaurant or a store. When the opened "homepage" web page contains an untrusted sub-frame, the untrusted sub-frame can launch traditional or privileged navigation attacks.

- *Uber*: Uber's WebView can be started to show "Terms and Conditions" from its own website by sequentially clicking the buttons "menu", "legal" and "terms&conditions". Our analysis shows the term and condition webpage contains an untrusted iframe for loading third-party tracking content (market analyst). The untrusted iframe can launch traditional or privileged navigation attacks.
- *ESPN*: The ESPN app shows news from its own website. However, its web pages load third-party tracking content from Google in an iframe. Hence, the untrusted sub-frame can also do phishing attacks by leveraging traditional navigation and WUI overlap vulnerabilities.
- *McDonald's*: In the app, several events are listed. When an event link (such as "trick n' treat") is clicked, WebView is started to show more details from its own website. However, an untrusted sub-frame is also contained that it may exploit traditional or privileged navigation vulnerabilities.
- *Samsung Mobile Print, lastpass, dashlane*: These apps provide an internal web browser to improve user experience. These internal browsers suffer from main-frame navigation attacks. Although they also offer address bars, unfortunately, the length of their address bars is much short than the average length "29 letters" (Section 5.4. For example, in the same environment (Nexus 5), Samsung Mobile Print only shows 23 letters, and lastpass only display 18 letters.
- *1password*: DCV-Hunter finds several paths to WebView content loading APIs. Because we do not have an account to login, this app is not fully tested. However, when we click its discount link, we still find a vulnerable WebView instance is launched. The WebView instance can show any content, and suffers from traditional or privileged navigation attacks.
- *The U.S., Huntington and Chime Mobile Bank apps*: These bank apps provide WebView to load content from their websites. Note that some of their WebView can be navigated to any websites. The loaded content can include third-party (tracking) content, which can launch traditional or privileged navigation attacks.
- *The Facebook React Native library*: This library is designed to help JavaScript developers implement cross-platform mobile apps. In its WebView, the related default configurations are applied. It suffers from traditional and privileged navigation vulnerabilities.

5.4 Security Impacts of Home-Brewed URL Address Bars

Our study shows that some hybrid apps implement their own URL address and title bars (such as those in our case studies), which could reduce the UI inconsistencies between WebView and regular browsers. To better evaluate the security impacts, we conducted an empirical study of 100 apps that contain home-brewed address bars. These apps are collected by filtering the DCV-Hunter analysis results (by checking if there is a path or flow from WebView's real-time URLs (such as the API "*WebView.getUrl()*" and the second parameter of the event handler "*onPageFinished(view, url)*") to UI components' updating APIs such as "*TextView.setText()*").

We find that the home-brewed address bars are ineffective to prevent DCV attacks, for two main reasons: limited address bar lengths, and implementation errors.

Limited Address Bar Lengths. In our study on a real phone (Nexus 5), which has the representative screen width, we find that typical address bars averagely show 29 letters. When domains, including sub-domains, being accessed exceed that length, security risks could be caused, even when some existing solutions such as showing the rightmost/leftmost of origin/URL are in use (e.g., Chrome/Chromium). This is also partially verified by existing work (e.g., [29]).

Implementation Errors. Some apps/libraries, such as "Facebook Mobile Browser", use very small fonts to show origins (Figure 5). This mitigates the above length limitation problem. As Figure 5-c shows, this address bar can effectively mitigate a DCV attack, such as the WUI overlap attack, since the address bar can show the origin of the fake web page in real time. However, it also has several flaws. First, due to the small font, it faces the pixel problem. Attackers may build a fake and confusing URL by replacing few letters of the benign URL with confusing letters (such as replacing the letter "O" with the number "0"). The fake URL may still spoof users.

Moreover, in these apps, our analysis finds a race condition flaw, which can be utilized to show fake web content in WebView, while still presenting the benign URL (e.g., ebay.com) in the address bar (Figure 5-d). This issue is rooted in the design flaw that several WUIs share only one address bar, while all these WUIs have abilities to update the content of the address bar. Hence, attackers can still perform phishing attacks by combining a couple of DCV attacks. For example, in the Facebook Mobile Browser library, which suffers from the WUI overlap attack, attackers may open a WUI to load fake content, and then immediately update the overlapped benign WUI in background. As a result, the address bar only show attackers' URL in a very short time and is quickly updated to display the benign URL. In our test, we find sometimes the bad URL may not even appear (see our online demo [2]). This indicates the blended attack is stealthy. In practice, the blended attack can be easily launched by using the code shown in Listing 8.

```
1 // Opening a fake web page (WUI overlap attack)
2 window.open("https://attacker.com", "_blank")
3 // Refreshing the address bar (Traditional navigation attack)
4 window.open("https://eaby.com", "_top")
```

Listing 8: Exploit Code of Blended Attacks

6 Vulnerability Mitigation

6.1 Mitigation Solution

To mitigate DCV attacks, we propose a multi-level solution that enhances the security of WebView. First, we enhance the security of event handlers by addressing their design flaws (Section 3.6). For example, in *onCreateWindow()*, necessary information is provided, including the operator origin who is creating a popup, and the URL the created popup is going to load. Thus, based on the provided information, developers can reject an unauthorized request. To ease the deployment of our solution, we also provide security enforcement. If developers provide the list of trusted URLs in a configuration file inside their apps (located in the app folder "assets"), the untrusted requests can be automatically denied.

Second, we also mitigate the UI inconsistencies by providing floating URL indicators. For example, when the main frame is navigated to a different domain by an iframe/popup, the URL indicator can provide users an alert. Furthermore, when users longly press a WebView instance, the origin of the main frame being loaded by the WebView instance is presented.

Note this URL indicator is locally bound with a WUI, which is helpful to avoid the race condition flaw (Section 5.4). When there are multiple WUIs available, only the foremost WUI's URL indicator is visible.

Third, to mitigate origin-hiding attacks, in critical operations (e.g., accessing web-mobile bridges), we replace the "null" origin with the origin who creates the "null" origin. This makes existing defense solutions effective again, since they can enforce security checks or policies on the new origin.

Fourth, to counter the WebView UI redressing problem, changes of the WUI rendering order are monitored. When a change is performed by an iframe/popup, an alert is offered. Last, to limit the navigation based attacks, we introduce same origin restrictions into navigation, and also fix the conflict.

6.2 Mitigation Solution Implementation

Our implementation is mainly done by instrumenting the WebView library, without modifying the source code of Android frameworks.

6.2.1 Enhanced Event Handlers

To achieve the goal, event handlers related implementation is instrumented. Take the event handler *onCreateWindow()* as the example. To obtain the origin who is creating a popup, the call site is scanned to locate the last popup-creation operation. Next, the corresponding operator's web frame information (e.g., origin) is retrieved. However, if the web frame's origin is "null", DCV-Hunter checks the web frame tree to get the real frame who create the "null" frame. Then, to learn the URL the created popup is going to load, the parameter of the related API (e.g., *window.open()*) is also extracted. Furthermore, to implement the security enforcement of denying untrusted requests, the default implement of *onCreateWindow()* is also instrumented. When the configuration file (providing the list of trusted domains) exists, the trusted URLs are extracted and also used to match the URLs that trigger popup-creation requests.

6.2.2 URL Indicators

To present current origin loaded in a WebView instance, the long-click event of the WebView instance is handled. When the event occurs, the origin of the main frame is presented as a notification. However, the long-click event may also be used by developers. To avoid potential conflicts, we create an event handler wrapper, which first shows the origin information, and then calls the essential event handler registered by developers.

To monitor the main-frame navigation, the event handler "shouldOverrideUrlLoading()" is leveraged. When the event handler is triggered, the URL is checked. If the main frame is redirected to a different domain by a sub-frame, an alert can be given. Furthermore, considering WebView is also a view group (Section 2.2), we make the indicator *local*: we temporary add a text view to WebView as the indicator.

6.2.3 Replacing the "null" Origin

Since the "null" origin is meaningless, we replace it with the origin who creates the "null" origin. To achieve the goal, we scan the frame tree from bottom to top, and get the root frame, or the last frame whose origin is not "null". Then, the corresponding origin *O* is extracted for the replacement.

Next, to replace the "null" origin with *O* in postMessage, we instrument the associated methods of the class "*WebDOMMessageEvent*" and "*MessageEvent*". If the source origin is specified as "null", it will be replaced. Then, the security of web-mobile bridges is enhanced as follows. Take the event handler *onJsAlert(view, url, ...)* as the example. We instrument the event handler's relevant caller (i.e., "*AwJavaScriptDialogManager::RunJavaScriptDialog*") inside WebView. In the caller, if *url* is the data scheme URL, it will be replaced by *O*.

6.2.4 Popup Indicator

To mitigate the WebView UI redressing problem, all associated key APIs are monitored, such as *addView()*. When the WUI rendering order is changing by a sub-frame, an alert will be offered (implemented in the associated enhanced event handlers).

6.2.5 Safe Navigation

To avoid traditional navigation problem, we narrow down the navigation policy that navigation occurs only when two frames have the same origins. To achieve the goal, we instrument the key method "*LocalDOMWindow::open()*" to add the origin checks.

Furthermore, to fix privileged navigation problem, the conflict between WebView features and web APIs is handled. More specifically, in the key method "*RenderFrameHostImpl::CreateNewWindow*", we add more security restrictions. When the setting "*SupportMultipleWindows*" is false,

the popup behavior will be ignored.

6.3 Mitigation Evaluation

In our evaluation, we first test the usability of our defense solution, especially about how easy to deploy and apply our solution in practice. To do that, we select 10 real-world vulnerable apps for testing. We find our solution can simply work, if developers involve our own WebView header files, including the declarations of new function prototypes (e.g., *onCreateWindow()*), and also provide the configuration file with the list of third-party domains. Please note that because these real apps lack source code, we repackage them to involve necessary files.

Next, we verify the correctness of our mitigation solution by testing above ten apps. We test them in stock (vulnerable) WebView and the WebView that implements our mitigation solution, respectively. We find that 1) there are no errors introduced by our mitigation solution. Apps work well as usual; 2) DCV attacks are mitigated.

Then, we measure the overhead to check if our mitigation solution impacts user experience. We create a vulnerable app for testing. In the app, we call the WebView API *loadUrl()* to run associate HTML/JavaScript code to trigger all vulnerabilities. Meanwhile, all time costs are recorded. Similarly, we run the app in stock (vulnerable) WebView and the WebView that implements our mitigation solution. By comparing time costs, we find our mitigation solution only introduces tiny overhead: 2ms on average.

Last, considering the Android version fragmentation issue, we also test the compatibility of our mitigation solution by installing our own WebView library and running above the created app in major Android versions. The result shows our solution is available in many major popular Android versions (5.0+), and covers 89.3% of Android devices in use (based on the Android version distribution data of May 2019 [1]).

7 Related Work

Iframe/popup Security. In web apps, iframes/popups are often the cause of security issues, such as frame hijacking [11], clickjacking [43], and double-click clickjacking [23]. In past years, in the context of regular browsers, iframe/popup behaviors and these security issues were well studied. Many defense solutions were proposed. For example, the HTTP header "X-Frame-Options" and the frame busting [43] solution can prevent being framed. In this work, we mainly focus on the exploration of the abilities of untrusted iframes/popups. The more related security mechanisms, such as SOP, and navigation policies, are discussed in Section 2.1. As shown in Section 1 and 3, existing solutions are circumscribed to prevent DCV attacks.

WebView security. WebView security has attracted more and more attention. [17, 30, 33] generically studied WebView security. [21, 25, 27, 40, 49, 53] explored the security of web-mobile bridges, and also discovered several extended attacks. In Section 3.5, we compare DCV attacks with several related attacks, and show DCV attacks may have a set of advantages.

Several static analysis based approaches [22, 55] were proposed to vet hybrid apps. However, they were limited to analyze iframe/popup behaviors and event handlers (also see our discussion in Section 4). Several defense solutions were designed to provide protection for WebView and web-mobile bridges, such as NoFrak [21], Draco [49], MobileIFC [45], WIREframe [18], and HybridGuard [38]. NoFrak and MobileIFC extended SOP into the mobile layer, while other solutions provided security enforcement on web-mobile bridges. However, as discussed in Section 1 and 3, they were quite limited to prevent DCV attacks.

In addition, many solutions [13, 41] are also designed to mitigate the Android UI deception problems [15, 20, 35]. However, as discussed in Section 1 and 3.3, they cannot monitor the state change of WebView UI, and circumscribed to prevent WUI redressing attacks.

8 Discussion

Research scope. In this work, we mainly focus on Android, which is currently the most popular mobile OS. However, there are also other WebView formats in other platforms (e.g., WKWebView for iOS). The research on other platforms would be complementary to our work, and we leave this as our future work.

False negatives. DCV-Hunter faces false negatives in some situations. For example, in mobile apps, some URLs loaded in WebView are encrypted, some URL related data goes through implicit flows, and some WebView related code is dynamically loaded. Some of these issues can be simply partially mitigated. For example, apps can be dynamically tested for collecting and downloading dynamically loaded code. We leave the improvement of our tool to reduce all false negatives as our future work.

9 Conclusion

Iframes/popups are often the root cause of several critical web security issues, and have been well studied in regular browsers. However, their behaviors are rarely understood and scrutinized in WebView, which has a totally new working environment. In this paper, we fill the gap and identify several fundamental design flaws and vulnerabilities, named differential context vulnerabilities (DCVs). We find that by exploiting DCVs, an untrusted iframe/popup becomes very dangerous in Android WebView. We have designed a novel detection technique, DCV-Hunter, to assess the security impacts of DCVs on real-world apps. Our measurement on a large number of popular apps shows that DCVs are prevalent. We have also presented a multi-level protection solution to mitigate DCVs, which is shown to be scalable and effective.

Acknowledgments

We want to thank our shepherd Yinzhi Cao and the anonymous reviewers for their valuable comments. This material is based upon work supported in part by the National Science Foundation (NSF) under Grant no. 1642129 and 1700544. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF. We also thank Cong Zheng and Yuchen Zhou for the helpful discussions about our threat model and the design of DCV-Hunter.

References

[1] Android version distribution dashboard. https://developer.android.com/about/dashboards.

[2] Dcv-attacks. https://sites.google.com/view/dcv-attacks.

[3] Easyprivacy tracking protection list. https://easylist.to/tag/tracking-protection-lists.html.

[4] iframe - html standard. https://html.spec.whatwg.org/dev/iframe-embed-object.html#attr-iframe-sandbox.

[5] Mcafee mobile threat report. https://www.mcafee.com/us/resources/reports/rp-mobile-threat-report-2016.pdf.

[6] Same origin policy. https://en.wikipedia.org/wiki/Same-origin_policy.

[7] Selenium - web browser automation. https://www.seleniumhq.org.

[8] Web messaging standard. https://html.spec.whatwg.org/multipage/web-messaging.html.

[9] Webview client. https://developer.android.com/reference/android/webkit/WebViewClient.html.

[10] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI*, 2014.

[11] A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. In *USENIX Security*, 2009.

[12] A. B. Bhavani. Cross-site Scripting Attacks on Android WebView. *IJCSN International Journal of Computer Science and Network*, 2(2):1–5, 2013.

[13] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. What the app is that? deception and countermeasures in the android user interface. In *IEEE Symposium on Security and Privacy*, 2015.

[14] T. Bujlow, V. Carela-Español, J. Solé-Pareta, and P. Barlet-Ros. A survey on web tracking: Mechanisms, implications, and defenses. *Proceedings of the IEEE*, 2017.

[15] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In *USENIX Security*, 2014.

[16] E. Chin and D. Wagner. Bifocals: Analyzing webview vulnerabilities in android applications. In *International Workshop on Information Security Applications*, 2013.

[17] E. Chin and D. Wagner. Bifocals: Analyzing webview vulnerabilities in android applications. In *WISA*. 2013.

[18] D. Davidson, Y. Chen, F. George, L. Lu, and S. Jha. Secure integration of web content and applications on commodity mobile operating systems. In *ASIA CCS*, 2017.

[19] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS/ETAPS, pages 337–340. Springer-Verlag, 2008.

[20] Y. Fratantonio, C. Qian, S. P. Chung, and W. Lee. Cloak and dagger: from two permissions to complete control of the ui feedback loop. In *IEEE Symposium on Security and Privacy*, 2017.

[21] M. Georgiev, S. Jana, and V. Shmatikov. Breaking and fixing origin-based access control in hybrid web/mobile application frameworks. In *NDSS*, 2014.

[22] B. Hassanshahi, Y. Jia, R. H. C. Yap, P. Saxena, and Z. Liang. Web-to-application injection attacks on android: Characterization and detection. In *ESORICS*, 2015.

[23] L. Huang, A. Moshchuk, H. J. Wang, S. Schecter, and C. Jackson. Clickjacking: Attacks and defenses. In *USENIX Security*, 2012.

[24] InfoSecurity. Public wifi hotspots ripe for mitm attacks. https://www.infosecurity-magazine.com/news/public-wifi-hotspots-ripe-for-mitm-attacks/.

[25] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In *CCS*, 2014.

[26] A. Lerner, T. Kohno, and F. Roesner. Rewriting history: Changing the archived web from the present. CCS, 2017.

[27] T. Li, X. Wang, M. Zha, K. Chen, X. Wang, L. Xing, X. Bai, N. Zhang, and X. Han. Unleashing the walking dead: Understanding cross-app remote infections on mobile webviews. In *CCS*, 2017.

[28] Z. Li, K. Zhang, Y. Xie, F. Yu, and X. Wang. Knowing your enemy: Understanding and detecting malicious web advertising. In *CCS*, 2012.

[29] M. Luo, O. Starov, N. Honarmand, and N. Nikiforakis. Hindsight: Understanding the evolution of ui vulnerabilities in mobile browsers. CCS, 2017.

[30] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin. Attacks on webview in the android system. In *ACSAC*, 2011.

[31] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du. Touchjacking attacks on web in android, iOS, and windows phone. In *Foundations and Practice of Security*. 2013.

[32] J. R. Mayer and J. C. Mitchell. Third-party web tracking: Policy and technology. In *IEEE Symposium on Security and Privacy*, 2012.

[33] P. Mutchler, A. DoupÃ, J. Mitchell, C. Kruegel, G. Vigna, A. Doup, J. Mitchell, C. Kruegel, and G. Vigna. A Large-Scale Study of Mobile Web App Security. In *MoST*, 2015.

[34] M. Neugschwandtner, M. Lindorfer, and C. Platzer. A view to a kill: Webview exploitation. In *LEET*, 2013.

[35] M. Niemietz and J. Schwenk. Ui redressing attacks on android devices. *Black Hat*, 2012.

[36] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: Large-scale evaluation of remote javascript inclusions. CCS, 2012.

[37] X. Pan, Y. Cao, and Y. Chen. I do not know what you visited last summer - protecting users from third-party web tracking with trackingfree browser. In *NDSS*, 2015.

[38] P. H. Phung, A. Mohanty, R. Rachapalli, and M. Sridhar. Hybridguard: A principal-based permission and fine-grained policy enforcement framework for web-based mobile applications. In *MoST*, 2017.

[39] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iframes point to us. Usenix Security, 2008.

[40] V. Rastogi, R. Shao, Y. Chen, X. Pan, S. Zou, and R. Riley. Are these Ads Safe: Detecting Hidden Attacks through the Mobile App-Web Interfaces. *NDSS*, 2016.

[41] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu. Towards discovering and understanding task hijacking in android. In *USENIX Security*, 2015.

[42] F. Roesner, T. Kohno, and D. Wetherall. Detecting and defending against third-party tracking on the web. In *NSDI)*, 2012.

[43] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In *IEEE Oakland Web 2.0 Security and Privacy*, 2010.

[44] P. Saxena, S. Hanna, P. Poosankam, and D. Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *NDSS*, 2010.

[45] K. Singh. Practical context-aware permission control for hybrid mobile applications. In *RAID*. 2013.

[46] D. F. Somé, N. Bielova, and T. Rezk. Control what you include! - server-side protection against third party web tracking. In *Engineering Secure Software and Systems*, 2017.

[47] S. Son and V. Shmatikov. The postman always rings twice: Attacking and defending postmessage in html5 websites. In *NDSS*, 2013.

[48] K. Tian, Z. Li, K. D Bowers, and D. Yao. Framehanger: Evaluating and classifying iframe injection at large scale. In *SecureComm*, 2018.

[49] G. S. Tuncay, S. Demetriou, and C. A. Gunter. Draco: A system for uniform and fine-grained access control for web code on android. In *CCS*, 2016.

[50] R. Wang, L. Xing, X. Wang, and S. Chen. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *CCS*, 2013.

[51] T. Wei, Y. Zhang, H. Xue, M. Zheng, C. Ren, and D. Song. Sidewinder targeted attack against android in the golden age of ad libraries. In *Black Hat*. 2014.

[52] M. Weissbacher, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. Zigzag: Automatically hardening web applications against client-side validation vulnerabilities. In *USENIX Security*, 2015.

[53] G. Yang, J. Huang, and G. Gu. Automated generation of event-oriented exploits in android hybrid apps. In *NDSS*, 2018.

[54] G. Yang, J. Huang, G. Gu, and A. Mendoza. Study and mitigation of origin stripping vulnerabilities in hybrid-postmessage enabled mobile applications. In *IEEE Symposium on Security and Privacy*, 2018.

[55] G. Yang, A. Mendoza, J. Zhang, and G. Gu. Precisely and scalably vetting javascript bridge in android hybrid apps. In *RAID*, 2017.

[56] A. Zarras, A. Kapravelos, G. Stringhini, T. Holz, C. Kruegel, and G. Vigna. The dark alleys of madison avenue: Understanding malicious advertisements. In *IMC*, 2014.

Small World with High Risks: A Study of Security Threats in the npm Ecosystem

Markus Zimmermann
Department of Computer Science
TU Darmstadt

Cristian-Alexandru Staicu
Department of Computer Science
TU Darmstadt

Cam Tenny
r2c

Michael Pradel
Department of Computer Science
TU Darmstadt

Abstract

The popularity of JavaScript has lead to a large ecosystem of third-party packages available via the npm software package registry. The open nature of npm has boosted its growth, providing over 800,000 free and reusable software packages. Unfortunately, this open nature also causes security risks, as evidenced by recent incidents of single packages that broke or attacked software running on millions of computers. This paper studies security risks for users of npm by systematically analyzing dependencies between packages, the maintainers responsible for these packages, and publicly reported security issues. Studying the potential for running vulnerable or malicious code due to third-party dependencies, we find that individual packages could impact large parts of the entire ecosystem. Moreover, a very small number of maintainer accounts could be used to inject malicious code into the majority of all packages, a problem that has been increasing over time. Studying the potential for accidentally using vulnerable code, we find that lack of maintenance causes many packages to depend on vulnerable code, even years after a vulnerability has become public. Our results provide evidence that npm suffers from single points of failure and that unmaintained packages threaten large code bases. We discuss several mitigation techniques, such as trusted maintainers and total first-party security, and analyze their potential effectiveness.

1 Introduction

JavaScript has become one of the most widely used programming languages. To support JavaScript developers with third-party code, the *node package manager*, or short *npm*, provides hundreds of thousands of free and reusable code packages. The npm platform consists of an online database for searching packages suitable for given tasks and a package manager, which resolves and automatically installs dependencies. Since its inception in 2010, npm has steadily grown into a collection of over 800,000 packages, as of February 2019, and will likely grow beyond this number. As the primary source of third-party JavaScript packages for the client-side, server-side, and other platforms, npm is the centerpiece of a large and important software ecosystem.

The npm ecosystem is open by design, allowing arbitrary users to freely share and reuse code. Reusing a package is as simple as invoking a single command, which will download and install the package and all its transitive dependencies. Sharing a package with the community is similarly easy, making code available to all others without any restrictions or checks. The openness of npm has enabled its growth, providing packages for any situation imaginable, ranging from small utility packages to complex web server frameworks and user interface libraries.

Perhaps unsurprisingly, npm's openness comes with security risks, as evidenced by several recent incidents that broke or attacked software running on millions of computers. In March 2016, the removal of a small utility package called *left-pad* caused a large percentage of all packages to become unavailable because they directly or indirectly depended on left-pad.[1] In July 2018, compromising the credentials of the maintainer of the popular *eslint-scope* package enabled an attacker to release a malicious version of the package, which tried to send local files to a remote server.[2]

Are these incidents unfortunate individual cases or first evidence of a more general problem? Given the popularity of npm, better understanding its weak points is an important step toward securing this software ecosystem. In this paper, we systematically study security risks in the npm ecosystem by analyzing package dependencies, maintainers of packages, and publicly reported security issues. In particular, we study the potential of individual packages and maintainers to impact the security of large parts of the ecosystem, as well as the ability of the ecosystem to handle security issues. Our analysis is based on a set of metrics defined on the package dependency graph and its evolution over time. Overall, our study involves 5,386,239 versions of packages, 199,327 maintainers, and

[1] https://www.infoworld.com/article/3047177/javascript/how-one-yanked-javascript-package-wreaked-havoc.html

[2] https://github.com/eslint/eslint-scope/issues/39

609 publicly known security issues.

The overall finding is that the densely connected nature of the npm ecosystem introduces several weak spots. Specifically, our results include:

- Installing an average npm package introduces an implicit trust on 79 third-party packages and 39 maintainers, creating a surprisingly large attack surface.
- Highly popular packages directly or indirectly influence many other packages (often more than 100,000) and are thus potential targets for injecting malware.
- Some maintainers have an impact on hundreds of thousands of packages. As a result, a very small number of compromised maintainer accounts suffices to inject malware into the majority of all packages.
- The influence of individual packages and maintainers has been continuously growing over the past few years, aggravating the risk of malware injection attacks.
- A significant percentage (up to 40%) of all packages depend on code with at least one publicly known vulnerability.

Overall, these findings are a call-to-arms for mitigating security risks on the npm ecosystem. As a first step, we discuss several mitigation strategies and analyze their potential effectiveness. One strategy would be a vetting process that yields trusted maintainers. We show that about 140 of such maintainers (out of a total of more than 150,000) could halve the risk imposed by compromised maintainers. Another strategy we discuss is to vet the code of new releases of certain packages. We show that this strategy reduces the security risk slightly slower than trusting the involved maintainers, but it still scales reasonably well, i.e., trusting the top 300 packages reduces the risk by half. If a given package passes the vetting process for maintainers and code, we say it has "perfect first-party security". If all its transitive dependencies pass the vetting processes we say that it has "perfect third-party security". If both conditions are met, we consider it a "fully secured package". While achieving this property for all the packages in the ecosystem is infeasible, packages that are very often downloaded or that have several dependents should aim to achieve it.

2 Security Risks in the npm Ecosystem

To set the stage for our study, we describe some security-relevant particularities of the npm ecosystem and introduce several threat models.

2.1 Particularities of npm

Locked Dependencies In npm, dependencies are declared in a configuration file called package.json, which specifies the name of the dependent package and a version constraint. The version constraint either gives a specific version, i.e., the dependency is *locked*, or specifies a range of compatible versions, e.g., newer than version X. Each time an npm package is installed, all its dependencies are resolved to a specific version, which is automatically downloaded and installed.

Therefore, the same package installed on two different machines or at two different times may download different versions of a dependency. To solve this problem, npm introduced package-lock.json, which developers can use to lock their transitive dependencies to a specific version until a new lock file is generated. That is, each package in the dependency tree is locked to a specific version. In this way, users ensure uniform installation of their packages and coarse grained update of their dependencies. However, a major shortcoming of this approach is that if a vulnerability is fixed for a given dependency, the patched version is not installed until the package-lock.json file is regenerated. In other words, developers have a choice between uniform distribution of their code and up-to-date dependencies. Often they choose the later, which leads to a technical lag [12] between the latest available version of a package and the one used by dependents.

Heavy Reuse Recent work [11, 18] provides preliminary evidence that code reuse in npm differs significantly from other ecosystems. One of the main characteristic of the npm ecosystem is the high number of transitive dependencies. For example, when using the core of the popular Spring web framework in Java, a developer transitively depends on ten other packages. In contrast, the Express.js web framework transitively depends on 47 other packages.

Micropackages Related to the reuse culture, another interesting characteristic of npm is the heavy reliance on packages that consist of only few lines of source code, which we call *micropackages*. Related work documents this trend and warns about its dangers [1, 19]. These packages are an important part of the ecosystem, yet they increase the surface for certain attacks as much as functionality heavy packages. This excessive fragmentation of the npm codebase can thus lead to very high number of dependencies.

No Privilege Separation In contrast to, e.g., the Java security model in which a SecurityManager[3] can restrict the access to sensitive APIs, JavaScript does not provide any kind of privilege separation between code loaded from different packages. That is, any third-party package has the full privileges of the entire application. This situation is compounded by the fact that many npm packages run outside of a browser, in particular on the Node.js platform, which does not provide any kind of sandbox. Instead, any third-party package can access, e.g., the file system and the network.

[3] https://docs.oracle.com/javase/6/docs/api/java/lang/SecurityManager.html

No Systematic Vetting The process of discovering vulnerabilities in npm packages is still in its infancy. There currently is no systematic vetting process for code published on npm. Instead, known vulnerabilities are mostly reported by individuals, who find them through manual analysis or in recent research work, e.g., injection vulnerabilities [30], regular expression denial of service [9, 29], path traversals [16], binding layer bugs [6].

Publishing Model In order to publish a package, a developer needs to first create an account on the npm website. Once this prerequisite is met, adding a new package to the repository is as simple as running the "npm publish" command in a folder containing a package.json file. The user who first published the package is automatically added to the maintainers set and hence she can release future versions of that package. She can also decide to add additional npm users as maintainers. What is interesting to notice about this model is that it does not require a link to a public version control system, e.g., GitHub, hosting the code of the package. Nor does it require that persons who develop the code on such external repositories also have publishing rights on npm. This disconnect between the two platforms has led to confusion[4] in the past and to stealthy attacks that target npm accounts without changes to the versioning system.

2.2 Threat Models

The idiosyncratic security properties of npm, as described above, enable several scenarios for attacking users of npm packages. The following discusses threat models that either correspond to attacks that have already occurred or that we consider to be possible in the future.

Malicious Packages (TM-mal) Adversaries may publish packages containing malicious code on npm and hence trick other users into installing or depending on such packages. In 2018, the eslint-scope incident mentioned earlier has been an example of this threat. The package deployed its payload at installation time through an automatically executed post-installation script. Other, perhaps more stealthy methods for hiding the malicious behavior could be envisioned, such as downloading and executing payloads only at runtime under certain conditions.

Strongly related to malicious packages are packages that violate the user's privacy by sending usage data to third parties, e.g., insight[5] or analytics-node[6]. While these libraries are legitimate under specific conditions, some users may not want to be tracked in this way. Even though the creators of these packages clearly document the tracking functionality, transitive dependents may not be aware that one of their dependencies deploys tracking code.

Exploiting Unmaintained Legacy Code (TM-leg) As with any larger code base, npm contains vulnerable code, some of which is documented in public vulnerability databases such as npm security advisories[7] or Snyk vulnerability DB[8]. As long as a vulnerable package remains unfixed, an attacker can exploit it in applications that transitively depend on the vulnerable code. Because packages may become abandoned due to developers inactivity [8] and because npm does not offer a forking mechanism, some packages may never be fixed. Even worse, the common practice of locking dependencies may prevent applications from using fixed versions even when they are available.

Package Takeover (TM-pkg) An adversary may convince the current maintainers of a package to add her as a maintainer. For example, in the recent *event-stream* incident[9], the attacker employed social engineering to obtain publishing rights on the target package. The attacker then removed the original maintainer and hence became the sole owner of the package. A variant of this attack is when an attacker injects code into the source base of the target package. For example, such code injection may happen through a pull request, via compromised development tools, or even due to the fact that the attacker has commit rights on the repository of the package, but not npm publishing rights. Once vulnerable or malicious code is injected, the legitimate maintainer would publish the package on npm, unaware of its security problems. Another takeover-like attack is typosquatting, where an adversary publishes malicious code under a package name similar to the name of a legitimate, popular package. Whenever a user accidentally mistypes a package name during installation, or a developer mistypes the name of a package to depend on, the malicious code will be installed. Previous work shows that typosquatting attacks are easy to deploy and effective in practice [31].

Account Takeover (TM-acc) The security of a package depends on the security of its maintainer accounts. An attacker may compromise the credentials of a maintainer to deploy insecure code under the maintainer's name. At least one recent incident (eslint-scope) is based on account takeover. While we are not aware of how the account was hijacked in this case, there are various paths toward account takeover, e.g., weak passwords, social engineering, reuse of compromised passwords, and data breaches on npm.

Collusion Attack (TM-coll) The above scenarios all assume a single point of failure. In addition, the npm ecosystem may get attacked via multiple instances of the above threats. Such a collusion attack may happen when multiple maintainers decide to conspire and to cause intentional harm, or when multiple packages or maintainers are taken over by an attacker.

[4] http://www.cs.tufts.edu/comp/116/archive/spring2018/etolhurst.pdf

[5] https://www.npmjs.com/package/insight

[6] https://www.npmjs.com/package/analytics-node

[7] https://www.npmjs.com/advisories

[8] https://snyk.io/vuln/?type=npm

[9] https://github.com/dominictarr/event-stream/issues/116

3 Methodology

To analyze how realistic the above threats are, we systematically study package dependencies, maintainers, and known security vulnerabilities in npm. The following explains the data and metrics we use for this study.

3.1 Data Used for the Study

Packages and Their Dependencies To understand the impact of security problems across the ecosystem, we analyze the dependencies between packages and their evolution.

Definition 3.1 *Let t be a specific point in time, P_t be a set of npm package names, and $E_t = \{(p_i, p_j) | p_i \neq p_j \in P_t\}$ a set of directed edges between packages, where p_i has a regular dependency on p_j. We call $G_t = (P_t, E_t)$ the **npm dependency graph** at a given time t.*

We denote the universe of all packages ever published on npm with $\mathcal{P}$. By aggregating the meta information about packages, we can easily construct the dependency graph without the need to download or install every package. Npm offers an API endpoint for downloading this metadata for all the releases of all packages ever published. In total we consider 676,539 nodes and 4,543,473 edges.

To analyze the evolution of packages we gather data about all their releases. As a convention, for any time interval t, such as years or months, we denote with t the snapshot at the beginning of that time interval. For example, G_{2015} refers to the dependency graph at the beginning of the year 2015. In total we analyze 5,386,239 releases, therefore an average of almost eight versions per package. Our observation period ends in April 2018.

Maintainers Every package has one or more developers responsible for publishing updates to the package.

Definition 3.2 *For every $p \in P_t$, the set of **maintainers** $M(p)$ contains all users that have publishing rights for p.*

Note that a specific user may appear as the maintainer of multiple packages and that the union of all maintainers in the ecosystem is denoted with $\mathcal{M}$.

Vulnerabilities The npm community issues advisories or public reports about vulnerabilities in specific npm packages. These advisories specify if there is a patch available and which releases of the package are affected by the vulnerability.

Definition 3.3 *We say that a given package $p \in \mathcal{P}$ is **vulnerable at a moment** t if there exists a public advisory for that package and if no patch was released for the described vulnerability at an earlier moment $t' < t$.*

We denote the set of vulnerable packages with $\mathcal{V} \subset \mathcal{P}$. In total, we consider 609 advisories affecting 600 packages. We extract the data from the publicly available npm advisories[10].

[10] https://www.npmjs.com/advisories

3.2 Metrics

We introduce a set of metrics for studying the risk of attacks on the npm ecosystem.

Packages and Their Dependencies The following measures the influence of a given package on other packages in the ecosystem.

Definition 3.4 *For every $p \in P_t$, the **package reach** $\mathrm{PR}(p)$ represents the set of all the packages that have a transitive dependency on p in G_t.*

Note that the package itself is not included in this set. The reach $\mathrm{PR}(p)$ contains names of packages in the ecosystem. Therefore, the size of the set is bounded by the following values $0 \leq |\mathrm{PR}(p)| < |P_t|$.

Since $|\mathrm{PR}(p)|$ does not account for the ecosystem changes, the metric may grow simply because the ecosystem grows. To address this, we also consider the average package reach:

$$\overline{\mathrm{PR}_t} = \frac{\sum_{\forall p \in P_t} |\mathrm{PR}(p)|}{|P_t|} \tag{1}$$

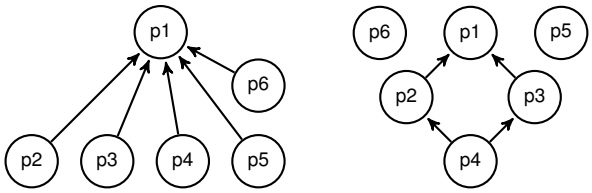
Using the bounds discussed before for $\mathrm{PR}(p)$, we can calculate the ones for its average $0 \leq \overline{\mathrm{PR}_t} < |P_t|$. The upper limit is obtained for a fully connected graph in which all packages can reach all the other packages and hence $|PR(p)| = |P_t| - 1, \forall p$. If $\overline{\mathrm{PR}_t}$ grows monotonously, we say that the ecosystem is getting more dense, and hence the average package influences an increasingly large number of packages.

The inverse of package reach is a metric to quantify how many packages are implicitly trusted when installing a particular package.

Definition 3.5 *For every $p \in P_t$, the set of **implicitly trusted packages** $\mathrm{ITP}(p)$ contains all the packages p_i for which $p \in \mathrm{PR}(p_i)$.*

Similarly to the previous case, we also consider the size of the set $|\mathrm{ITP}(p)|$ and the average number of implicitly trusted package $\overline{\mathrm{ITP}_t}$, having the same bounds as their package reach counterpart.

Even though the average metrics $\overline{\mathrm{ITP}_t}$ and $\overline{\mathrm{PR}_t}$ are equivalent for a given graph, the distinction between their non-averaged counterparts is very important from a security point of view. To see why, consider the example in Figure 1. The average $\overline{PR} = \overline{ITP}$ is 5/6 = 0.83 both on the right and on the left. However, on the left, a popular package $p1$ is dependent upon by many others. Hence, the package reach of $p1$ is five, and the number of implicitly trusted packages is one for each of the other packages. On the right, though, the number of implicitly trusted packages for $p4$ is three, as users of $p4$ implicitly trust packages $p1$, $p2$, and $p3$.



(a) Wide distribution of trust: $max(\text{PR}) = 5, max(\text{ITP}) = 1$ (b) Narrow distribution of trust: $max(\text{PR}) = 3, max(\text{ITP}) = 3$

Figure 1: Dependency graphs with different maximum package reaches (PR) and different maximum numbers of trusted packages (ITP).

Maintainers The number of implicitly trusted packages or the package reach are important metrics for reasoning about TM-pkg, but not about TM-acc. That is because users may decide to split their functionality across multiple micropackages for which they are the sole maintainers. To put it differently, a large attack surface for TM-pkg does not imply one for TM-acc.

Therefore, we define maintainer reach $\text{MR}_t(m)$ and implicitly trusted maintainers $\text{ITM}_t(p)$ for showing the influence of maintainers.

Definition 3.6 *Let m be an npm maintainer. The* ***maintainer reach*** $\text{MR}(m)$ *is the combined reach of all the maintainer's packages,* $\text{MR}(m) = \cup_{m \in M(p)} \text{PR}(p)$

Definition 3.7 *For every* $p \in P_t$*, the set of* ***implicitly trusted maintainers*** $\text{ITM}(p)$ *contains all the maintainers that have publishing rights on at least one implicitly trusted package,* $\text{ITM}(p) = \cup_{p_i \in \text{ITP}(p)} M(p_i)$.

The above metrics have the same bounds as their packages counterparts. Once again, the distinction between the package and the maintainer-level metrics is for shedding light on the security relevance of human actors in the ecosystem.

Furthermore, to approximate the maximum damage that colluding maintainers can incur on the ecosystem (TM-coll), we define an order in which the colluding maintainers are selected:

Definition 3.8 *We call an ordered set of maintainers* $L \subset \mathcal{M}$ *a* ***desirable collusion strategy*** *iff* $\forall m_i \in L$ *there is no* $m_k \neq m_i$ *for which* $\cup_{j<i} \text{MR}(m_j) \cup \text{MR}(m_i) < \cup_{j<i} \text{MR}(m_j) \cup \text{MR}(m_k)$.

Therefore, the desirable collusion strategy is a hill climbing algorithm in which at each step we choose the maintainer that provides the highest local increase in package reach at that point. We note that the problem of finding the set of n maintainers that cover the most packages is an NP-hard problem called *maximum coverage problem*. Hence, we believe that the proposed solution is a good enough approximation that shows how vulnerable the ecosystem is to a collusion attack, but that does not necessary yield the optimal solution.

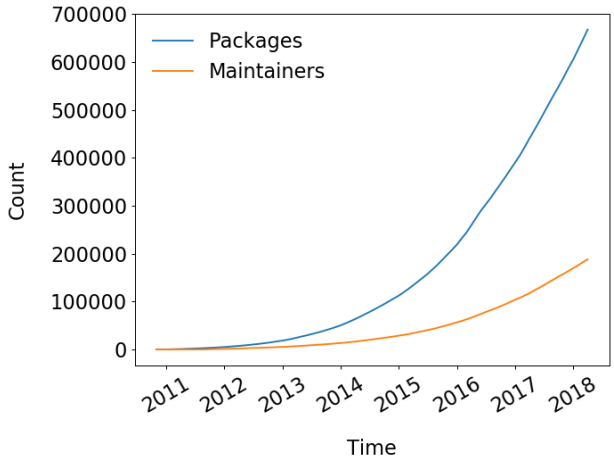


Figure 2: Evolution of number of packages and maintainers.

Vulnerabilities For reasoning about TM-leg, we need to estimate how much of the ecosystem depends on vulnerable code:

Definition 3.9 *Given all vulnerable packages* $p_i \in \mathcal{V}_t$ *at time t, we define the* ***reach of vulnerable code*** *at time t as* $\text{VR}_t = \cup_{p_i \in \mathcal{V}_t} PR(p_i)$.

Of course the actual reach of vulnerable code can not be fully calculated since it would rely on *all* vulnerabilities present in npm modules, not only on the published ones. However, since in TM-leg we are interested in publicly known vulnerabilities, we define our metric according to this scenario. In these conditions, the speed at which vulnerabilities are reported is an important factor to consider:

Definition 3.10 *Given all vulnerable packages* $p_i \in \mathcal{V}_t$ *at time t, we define the* ***vulnerability reporting rate*** VRR_t *at time t as* $\text{VRR}_t = \frac{|\mathcal{V}_t|}{|P_t|}$.

4 Results

We start by reporting the results on the nature of package level dependencies and their evolution over time (corresponding to TM-mal and TM-pkg). We then discuss the influence that maintainers have in the ecosystem (related to TM-acc and TM-coll). Finally, we explore the dangers of depending on unpatched security vulnerabilities (addressing TM-leg).

4.1 Dependencies in the Ecosystem

To set the stage for a thorough analysis of security risks entailed by the structure of the npm ecosystem, we start with a general analysis of npm and its evolution. Since its inception in 2010, the npm ecosystem has grown from a small collection of packages maintained by a few people to the world's largest software ecosystem. Figure 2 shows the evolution of the number of packages available on npm and the

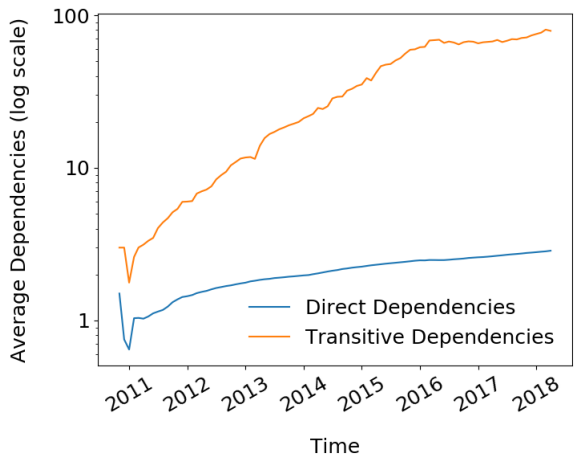


Figure 3: Evolution of direct package dependencies and its impact on transitive dependencies. Note the logarithmic scale on the y-axis.

number of maintainers responsible for these packages. Both numbers have been increasing super-linearly over the past eight years. At the end of our measurement range, there is a total of 676,539 packages, a number likely to exceed one million in the near future. These packages are taken care of by a total of 199,327 maintainers. The ratio of packages to maintainers is stable across our observation period (ranging between 2.81 and 3.51).

In many ways, this growth is good news for the JavaScript community, as it increases the code available for reuse. However, the availability of many packages may also cause developers to depend on more and more third-party code, which increases the attack surface for TM-pkg by giving individual packages the ability to impact the security of many other packages. The following analyzes how the direct and transitive dependencies of packages are evolving over time (Section 4.1.1) and how many other packages individual packages reach via dependencies (Section 4.1.2).

4.1.1 Direct and Transitive Dependencies

Figure 3 shows how many other packages an average npm package depends on directly and transitively. The number of direct dependencies has been increasing slightly from 1.3 in 2011 to 2.8 in 2018, which is perhaps unsurprising given the availability of an increasing code base to reuse. The less obvious observation is that a small, linear increase in direct dependencies leads to a significant, super-linear increase in transitive dependencies. As shown by the upper line in Figure 3, the number of transitive dependencies of an average package has increased to a staggering 80 in 2018 (note the logarithmic scale).

From a security perspective, it is important to note that each directly or transitively depended on package becomes part of the implicitly trusted code base. When installing a package, each depended upon package runs its post-installation scripts on the user's machine – code executed with the user's operating system-level permissions. When using the package, calls into third-party modules may execute any of the code shipped with the depended upon packages.

> When installing an average npm package, a user implicitly trusts around 80 other packages due to transitive dependencies.

One can observe in Figure 3 a chilling effect on the number of dependencies around the year 2016 which will become more apparent in the following graphs. Decan et al. [14] hypothesize that this effect is due to the left-pad incident. In order to confirm that this is not simply due to removal of more than a hundred packages belonging to the left-pad's owner, we remove all the packages owned by this maintainer. We see no significant difference for the trend in Figure 3 when removing these packages, hence we conclude that indeed there is a significant change in the structure of transitive dependencies in the ecosystem around 2016.

4.1.2 Package Reach

The above analysis focuses on depended upon packages. We now study the inverse phenomenon: packages impacted by individual packages, i.e., package reach as defined in Section 3. Figure 4 shows how many other packages a single package reaches via direct or indirect dependencies. The graph at the top is for an average package, showing that it impacts about 230 other packages in 2018, a number that has been growing since the creation of npm. The graph at the bottom shows the package reach of the top-5 packages (top in terms of their package reach, as of 2018). In 2018, these packages each reach between 134,774 and 166,086 other packages, making them an extremely attractive target for attackers.

To better understand how the reach of packages evolves over time, Figure 5 shows the distribution of reached packages for multiple years. For example, the red line shows that in 2018, about 24,500 packages have reached at least 10 other packages, whereas only about 9,500 packages were so influential in 2015. Overall, the figure shows that more and more packages are reaching a significant number of other packages, increasing the attractiveness of attacks that rely on dependencies.

> Some highly popular packages reach more than 100,000 other packages, making them a prime target for attacks. This problem has been aggravating over the past few years.

The high reach of a package amplifies the effect of both vulnerabilities (TM-leg) and of malicious code (TM-mal). As an example for the latter, consider the event-stream incident discussed when introducing TM-acc in Section 2.2. By

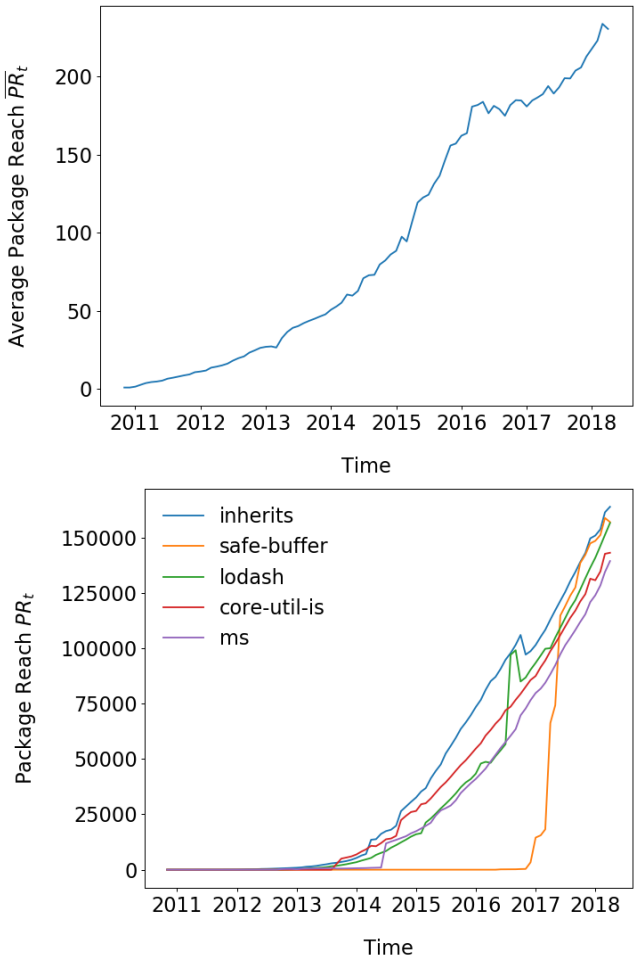


Figure 4: Evolution of package reach for an average package (top) and the top-5 packages (bottom).

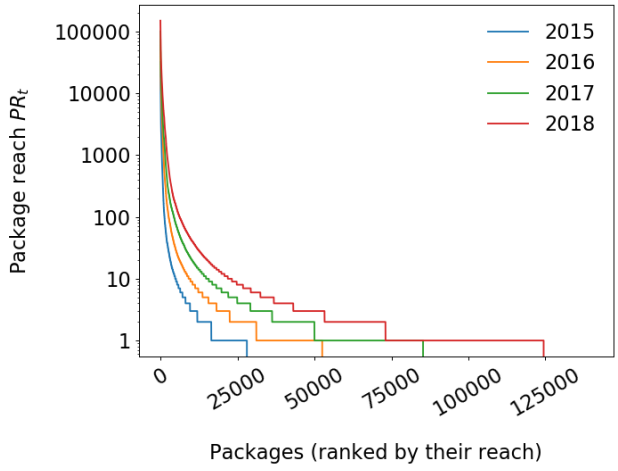


Figure 5: Distribution of package reach by individual packages, and how it changes over time. Note the log scale on the vertical axis.

computing event-stream's reach and comparing it with other packages, we see that this package is just one of many possible targets. As of April 1, 2018 (the end of our measurement period), event-stream has a reach of 5,466. That is, the targeted package is relatively popular, but still far from being the topmost attractive package to compromise. In fact, 1,165 other packages have a greater or equal reach than event-stream.

> Variants of the event-stream attack could easily be repeated with other packages.

In order to perform a similar analysis for the eslint-scope security incident, we need to use a slightly modified version of package reach. This attack targeted a development tool, namely eslint, hence, to fully estimate the attack surface we need to consider dev dependencies in our definition of reach. We do not normally consider this type of dependencies in our measurements because they are not automatically installed with a package, unlike regular dependencies. They are instead used only by the developers of the packages. Therefore the modified version of package reach considers both transitive regular dependencies and direct dev dependencies.

We observe that eslint-scope has a modified reach of more than 100,000 packages at the last observation point in the data set. However, there are 347 other packages that have a higher reach, showing that even more serious attacks may occur in the future.

> The attack on eslint-scope has targeted a package with an influence not larger than that of hundreds of other packages. It is likely that similar, or perhaps even worse, attacks will happen and succeed in the future.

4.2 Analysis of Maintainers

We remind the reader that there is a significant difference between npm maintainers and repository contributors, as discussed in Section 2.1. Even though contributors also have a lot of control over the code that will eventually end up in an npm package, they can not release a new version on npm, only the maintainers have this capability. Hence, the discussion that follows, about the security risks associated with maintainers, should be considered a lower bound for the overall attack surface.

Attacks corresponding to TM-acc in which maintainers are targeted are not purely hypothetical as the infamous eslint-scope incident discussed earlier shows. In this attack, a malicious actor hijacked the account of an influential maintainer and then published a version of eslint-scope containing malicious code. This incident is a warning for how vulnerable the ecosystem is to targeted attacks and how maintainers influence can be used to deploy malware at scale. We further discuss the relation between packages and maintainers.

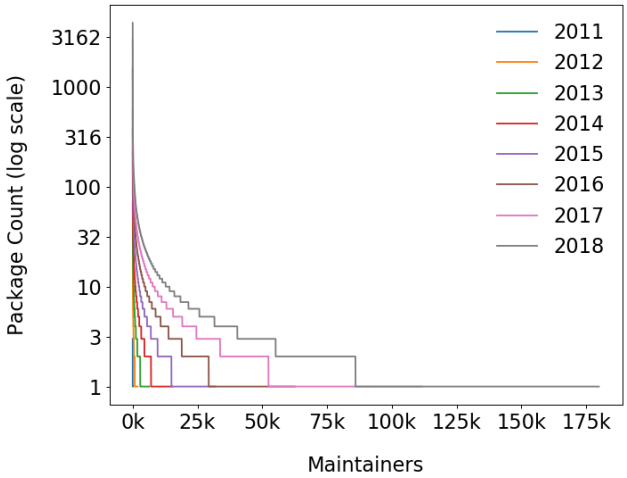


Figure 6: Evolution of maintainers sorted by package count per year.

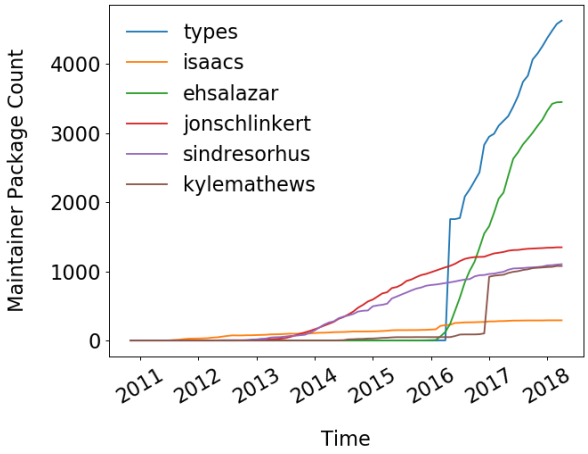


Figure 7: Evolution of package count for six popular maintainers.

4.2.1 Packages per Maintainer

Even though the ecosystem grows super-linearly as discussed in Section 4.1, one would expect that this is caused mainly by new developers joining the ecosystem. However, we observe that the number of packages per maintainer also grows suggesting that the current members of the platform are actively publishing new packages. The average number of packages controlled by a maintainer raises from 2.5 in 2012 to 3.5 in 2013 and almost 4.5 in 2018. Conversely, there are on average 1.35 maintainers in the lifetime of a package. The top 5,000 most popular packages have an average number of 2.83 maintainers. This is not unexpected, since multiple people are involved in developing the most popular packages, while for the majority of new packages there is only one developer.

Next, we study in more detail the evolution of the number of packages a maintainer controls. Figure 6 shows the maintainer package count plotted versus the number of maintainers having such a package count. Every line represents a year. The scale is logarithmic to base 10. It shows that the majority of maintainers maintain few packages, yet some maintainers maintain over 100 packages. Over the years, the package count for the maintainers increased consistently. In 2015, only slightly more than 25,000 maintainers maintained more than one package, whereas this number has more than tripled by 2018.

We further analyze five different maintainers in top 20 according to number of packages and plot the evolution of their package count over the years in Figure 7. *types* is the largest maintainer of type definitions for TypeScript, most likely a username shared by multiple developers at Microsoft, *ehsalazar* maintains many security placeholder packages, *jonschlinkert* and *sindresorhus* are maintaining many micropackages and *isaacs* is the npm founder. From Figure 7 we can see that for two of these maintainers the increase is super-linear or even near exponential: *types* and *kylemathews* have sudden spikes where they added many packages in a short time. We explain this by the tremendous increase in popularity for TypeScript in the recent years and by the community effort to prevent typosquatting attacks by reserving multiple placeholder. The graph of the other maintainers is more linear, but surprisingly it shows a continuous growth for all the six maintainers.

> The number of packages that both the influential and the average maintainers control increased continuously over the years.

4.2.2 Implicitly Trusted Maintainers

One may argue that the fact that maintainers publish new packages is a sign of a healthy ecosystem and that it only mimics its overall growth. However, we show that while that may be true, we also see an increase in the general influence of maintainers. That is, on average every package tends to transitively rely on more and more maintainers over time.

In Figure 8 we show the evolution of $\overline{ITM_t}$, the average number of implicitly trusted maintainers. As can be seen, $\overline{ITM_t}$ almost doubled in the last three years for the average npm package, despite the plateau of the curve reached in 2016 which we again speculate it is caused by the left-pad incident. This is a worrisome development since compromising any of the maintainer accounts a package trusts may seriously impact the security of that package, as discussed in TM-acc. The positive aspect of the data in Figure 8 is that the growth in the number of implicitly trusted maintainers seems to be less steep for the top 10,000 packages compared to the whole ecosystem. We hypothesize that the developers of popular packages are aware of this problem and actively try to limit the $\overline{ITM_t}$. However, a value over 20 for the average popular package is still high enough to be problematic.

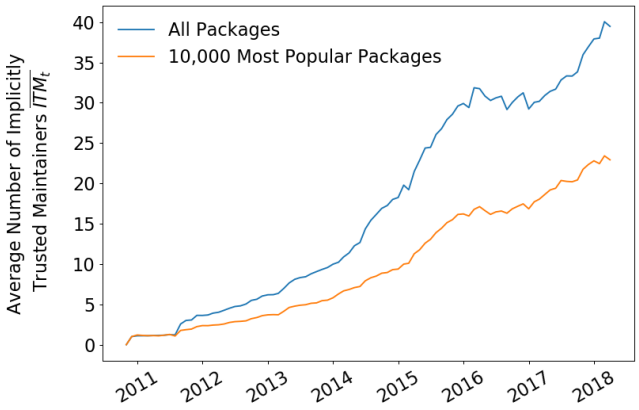


Figure 8: Evolution of average number of implicitly trusted maintainers over years in all packages and in the most popular ones.

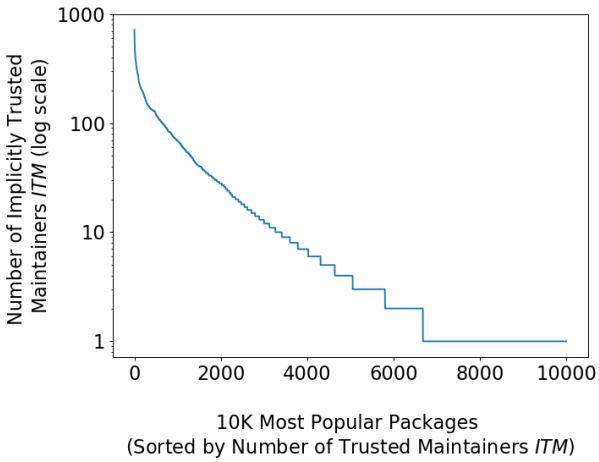


Figure 9: Number of implicitly trusted maintainers for top 10,000 most popular packages.

> The average npm package transitively relies on code published by 40 maintainers. Popular packages rely on only 20.

When breaking the average $\overline{ITM_t}$ discussed earlier into individual points in Figure 9, one can observe that the majority of these packages can be influenced by more than one maintainer. This is surprising since most of the popular packages are micropackages such as "inherits" or "left-pad" or libraries with no dependencies like "moment" or "lodash". However, only around 30% of these top packages have a maintainer cost higher than 10. Out of these, though, there are 643 packages influenced by more than a hundred maintainers.

> More than 600 highly popular npm packages rely on code published by at least 100 maintainers.

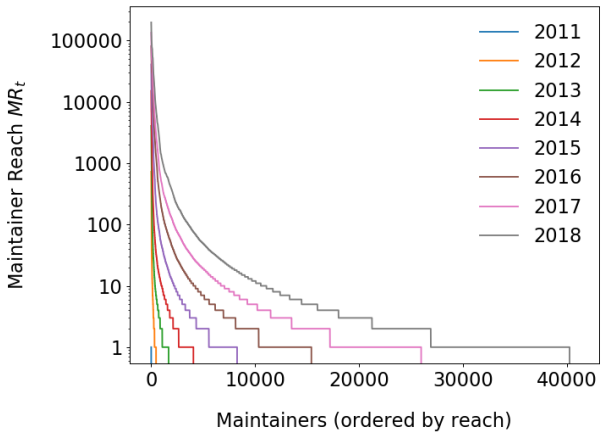


Figure 10: Distribution of maintainers reach in different years.

4.2.3 Maintainers Reach

In Figure 10, we plot the reach MR_t of the maintainers in the npm ecosystem. The reach has increased over the years at all levels. For example, in 2015 there were 2,152 maintainers that could affect more than 10 packages, and this number increased to 4,041 in 2016, 6,680 in 2017 and finally reaching an astonishingly high 10,534 in 2018. At the other end of the distribution, there were 59 maintainers that could affect more than 10,000 packages in 2015, 163 in 2016, 249 in 2017 and finally 391 in 2018. The speed of growth for MR_t is worrisome, showing that more and more developers have control over thousands of packages. If an attacker manages to compromise the account of any of the 391 most influential maintainers, the community will experience a serious security incident, reaching twice as many packages as in the event-stream attack.

> 391 highly influential maintainers affect more than 10,000 packages, making them prime targets for attacks. The problem has been aggravating over the past years.

Finally, we look at the scenario in which multiple popular maintainers collude, according to the desirable collusion strategy introduced in Section 3.2, to perform a large-scale attack on the ecosystem, i.e., TM-col. In Figure 11 we show that 20 maintainers can reach more than half of the ecosystem. Past that point every new maintainer joining does not increase significantly the attack's performance.

4.3 Security Advisories Evolution

Next, we study how often vulnerabilities are reported and fixed in the npm ecosystem (TM-leg). Figure 13 shows the number of reported vulnerabilities in the lifetime of the ecosystem. The curve seems to resemble the evolution of number of packages presented in Figure 2, with a steep increase in the last two years. To explore this relation further we plot in Figure 14 the evolution of the number of advisories

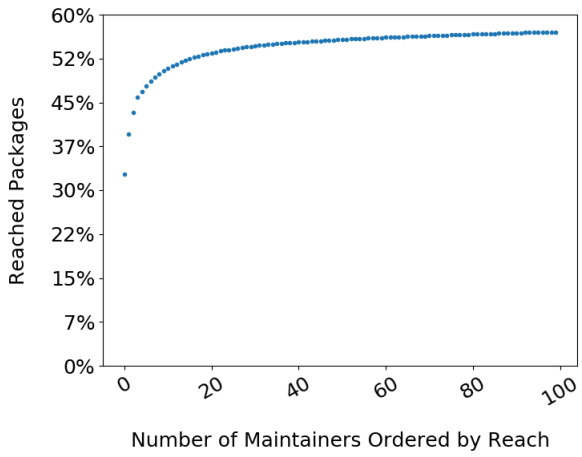


Figure 11: Combined reach of 100 influential maintainers.

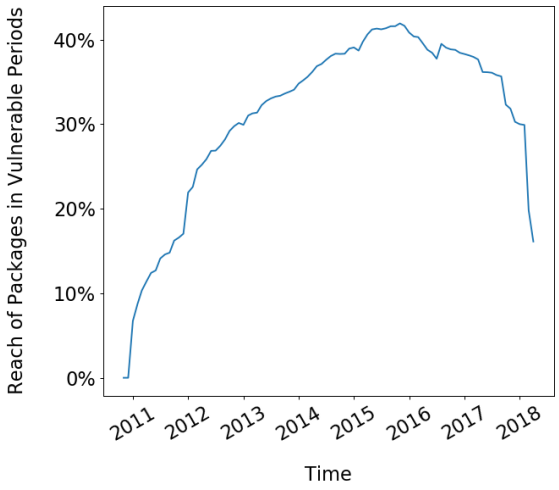


Figure 12: Total reach of packages for which there is at least one unpatched advisory (vulnerability reach VR_t).

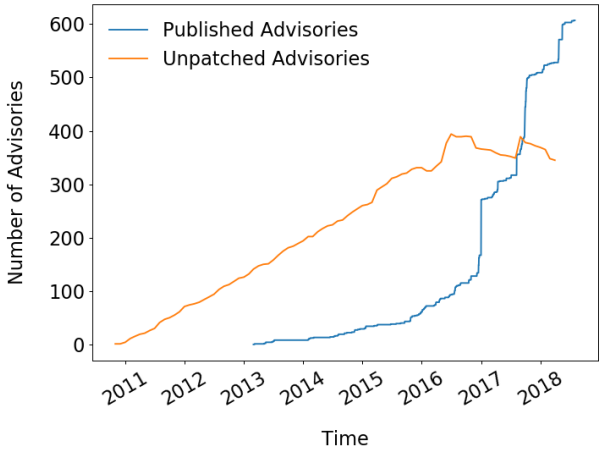


Figure 13: Evolution of the total and unpatched number of advisories.

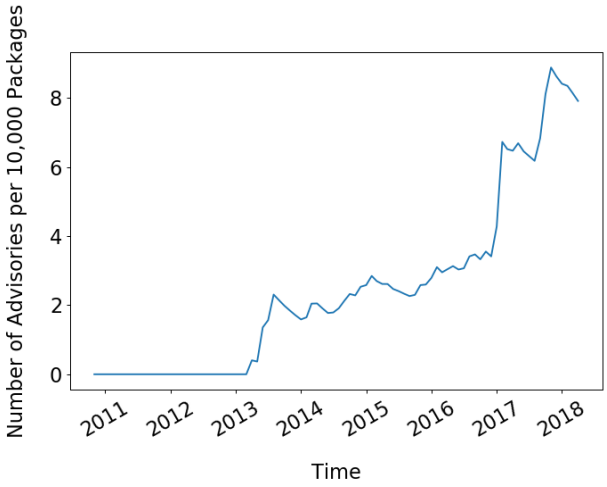


Figure 14: Evolution of VRR_t, the rate of published vulnerabilities per 10,000 packages.

reported per 10,000 packages and we observe that it grows from two in 2013 to almost eight in 2018. This is a sign of a healthy security community that reports vulnerabilities at a very good pace, keeping up with the growth of the ecosystem.

When analyzing the type of reported vulnerabilities in details, we observe that almost half of the advisories come from two large-scale campaigns and not a broader community effort: First, there are 141 advisories published in January 2017 involving npm packages that download resources over HTTP, instead of HTTPs. Second, there are 120 directory traversal vulnerabilities reported as part of the research efforts of Liang Gong [16]. Nevertheless, this shows the feasibility of large-scale vulnerability detection and reporting on npm.

Publishing an advisory helps raise awareness of a security problem in an npm package, but in order to keep the users secure, there needs to be a patch available for a given advisory. In Figure 13 we show the evolution of the number of unpatched security vulnerabilities in npm, as defined in Section 3. This trend is alarming, suggesting that two out of three advisories are still unpatched, leaving the users at risk. When manually inspecting some of the unpatched advisories we notice that a large percentage of unpatched vulnerabilities are actually advisories against malicious typosquatting packages for which no fix can be available.

To better understand the real impact of the unpatched vulnerabilities we analyze how much of the ecosystem they impact, i.e., vulnerability reach as introduced in Section 3.2. To that end, we compute the reach of unpatched packages at every point in time in Figure 12. At a first sight, this data shows a much less grim picture than expected, suggesting that the reach of vulnerable packages is dropping over time. However, we notice that the effect of vulnerabilities tends to be retroactive. That is, a vulnerability published in 2015 affects multiple versions of a package released prior to that date, hence influencing the data points corresponding to the years 2011-2014 in Figure 12. Therefore, the vulnerabilities

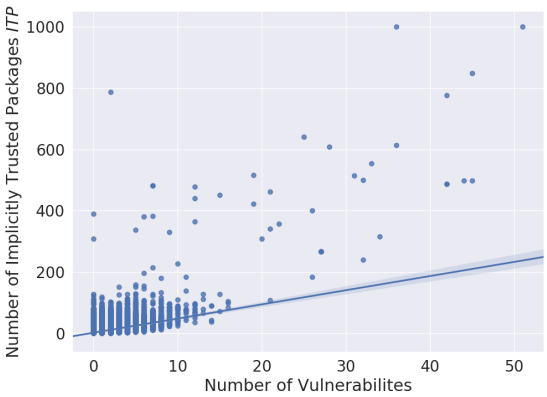


Figure 15: Correlation between number of vulnerabilities and number of dependencies.

that will be reported in the next couple of years may correct for the downwards trend we see on the graph. Independent of the downwards trend, the fact that for the majority of the time the reach of vulnerable unpatched code is between 30% and 40% is alarming.

> Up to 40% of all packages rely on code known to be vulnerable.

5 Potential Mitigations

The following section discusses ideas for mitigating some of the security threats in the npm ecosystem. We here do not provide fully developed solutions, but instead outline ideas for future research, along with an initial assessment of their potential and challenges involved in implementing them.

5.1 Raising Developer Awareness

One line of defense against the attacks described in this paper is to make developers who use third-party packages more aware of the risks entailed by depending on a particular package. Currently, npm shows for each package the number of downloads, dependencies, dependents, and open issues in the associated repository. However, the site does not show any information about the transitive dependencies or about the number of maintainers that may influence a package, i.e., our ITP and ITM metrics. As initial evidence that including such metrics indeed predicts the risk of security issues, Figure 15 shows the number of implicitly trusted packages versus the number of vulnerabilities a package is affected by. We find that the two values are correlated (Pearson correlation coefficient of 0.495), which is not totally unexpected since adding more dependencies increases the chance of depending on vulnerable code. Showing such information, e.g., the ITP metric, could help developers make more informed decisions about which third-party packages to rely on.

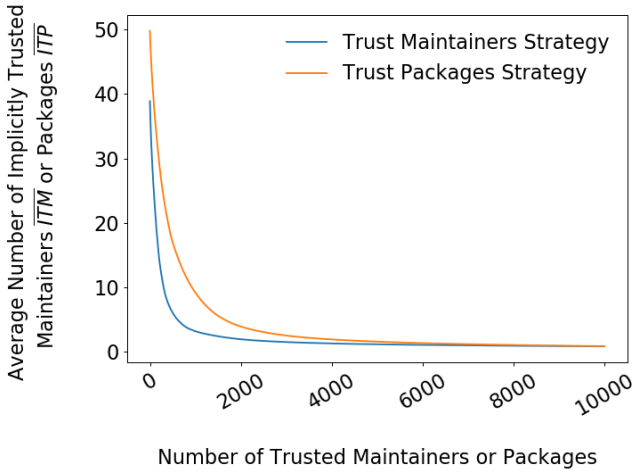


Figure 16: Decrease in average number of implicitly trusted maintainers and packages as the set of trusted maintainers or packages increases.

5.2 Warning about Vulnerable Packages

To warn developers about unpatched vulnerabilities in their dependencies, the `npm audit` tool has been introduced. It compares all directly depended upon packages against a database of known vulnerabilities, and warns a developer when depending upon a vulnerable version of a package. While being a valuable step forward, the tool currently suffers from at least three limitations. First, it only considers direct dependencies but ignores any vulnerabilities in transitive dependencies. Second, the tool is limited to known vulnerabilities, and hence its effectiveness depends on how fast advisories are published. Finally, this defense is insufficient against malware attacks.

5.3 Code Vetting

A proactive way of defending against both vulnerable and malicious code is code vetting. Similar to other ecosystems, such as mobile app stores, whenever a new release of a vetted package is published, npm could analyze its code. If and only if the analysis validates the new release, it is made available to users. Since the vetting process may involve semi-automatic or even manual steps, we believe that it is realistic to assume that it will be deployed step by step in the ecosystem, starting with the most popular packages. Figure 16 (orange curve) illustrates the effect that such code vetting could have on the ecosystem. The figure shows how the average number of implicitly trusted packages, $\overline{\text{ITP}}$, reduces with an increasing number of vetted and therefore trusted packages. For example, vetting the most dependent upon 1,500 packages would reduce the $\overline{\text{ITP}}$ ten fold, and vetting 4,000 packages would reduce it by a factor of 25.

An obvious question is how to implement such large-scale code vetting, in particular, given that new versions of packages are released regularly. To estimate the cost of vetting new releases, Figure 17 shows the average number of lines of code

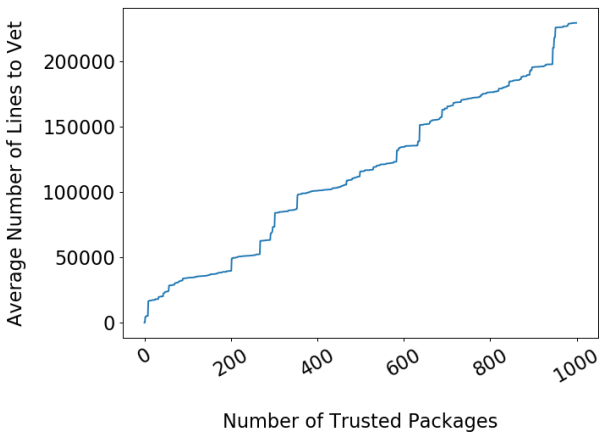


Figure 17: Number of lines of code that need to be vetted for achieving a certain number of trusted packages.

that are changed per release of a package, and would need to be vetted to maintain a specific number of trusted packages. For example, vetting the changes made in a single new release of the top 400 most popular packages requires to analyze over 100,000 changed lines of code. One way to scale code vetting to this amount of code could be automated code analysis tools. Recently, there have been several efforts for improving the state of the art of security auditing for npm, both from academia, e.g., Synode [30], BreakApp [32], NodeSec [16], NoRegrets [25], Node.cure [10], and from industry practitioners, e.g., Semmle[11], r2c[12], and DeepScan[13]. Orthogonal to automated code analysis tools, the npm community could establish crowd-sourced package vetting, e.g., in a hierarchically organized code distribution model similar to the Debian ecosystem.

Another challenge for code vetting is that npm packages, in contrast to apps in mobile app stores, are used across different platforms with different security models. For example, XSS vulnerabilities are relevant only when a package is used on the client-side, whereas command injection via the `exec` API [30] is a concern only on the server-side. A code vetting process could address this challenge by assigned platform-specific labels, e.g., "vetted for client-side" and "vetted for server-side", depending on which potential problems the vetting reveals.

5.4 Training and Vetting Maintainers

Another line of proactive defense could be to systematically train and vet highly influential maintainers. For example, this process could validate the identity of maintainers, support maintainers in understanding basic security principles, and ensure that their accounts are protected by state-of-the-art techniques, such as two-factor authentication. To assess the effect that such a process would have, we simulate how training and vetting a particular number of *trusted maintainers* influences the average number of implicitly trusted maintainers, $\overline{\text{ITM}}$. The simulation assumes that the most influential maintainers are vetted first, and that once a maintainer is vetted she is ignored in the computation of the $\overline{\text{ITM}}$. The results of this simulation (Figure 16) show a similar effect as for vetting packages: Because some maintainers are highly influential, vetting a relatively small number of maintainers can significantly reduce security risks. For example, vetting around 140 maintainers cuts down the $\overline{\text{ITM}}$ in half, and vetting around 600 could even reduce $\overline{\text{ITM}}$ to less than five. These results show that this mechanism scales reasonably well, but that hundreds of maintainers need to be vetted to bring the average number of implicitly trusted maintainers to a reasonable level. Moreover, two-factor authentication has its own risks, e.g., when developers handle authentication tokens in an insecure way[14] or when attackers attempt to steal such tokens, as in the eslint-scope incident.

6 Related Work

In this section we discuss the closest related work contained mainly in two distinct research areas: JavaScript security and software ecosystem studies. While some of this work studies the npm ecosystem, to the best of our knowledge, we are the first to analyze in depth the role maintainers play in the ecosystem and the impact of different types of attacks, as well as the potential impact of vetting code.

Server-side JavaScript Security There are many studies that investigate problems with dependency management for the JavaScript or other ecosystems. Abdalkareem et al. [2] investigate reasons why developers would use trivial packages. They find that developers think that these packages are well implemented and tested and that they increase productivity as the developer does not need to implement such small features herself. Another empirical study on micropackages by Kula et al. [19] has similar results. They show that micropackages have long dependency chains, something we also discovered in some case studies of package reach. We also show that these packages have a high potential of being a target of an attack as they are dependent on by a lot of packages. Another previously studied topic is breaking changes introduced by dependencies. Bogart et al. [5] perform a case study interviewing developers about breaking changes in three different ecosystems. They find that npm's community values a fast approach to new releases compared to the other ecosystems. Developers of npm are more willing to adopt breaking changes to fight technical debt. Furthermore, they find that the semantic versioning rules are enforced more overtime than in the beginning. Similarly, Decan et al. [11] analyze three package

[11] https://semmle.com/
[12] https://r2c.dev/
[13] https://deepscan.io/

[14] https://blog.npmjs.org/post/182015409750/automated-token-revocation-for-when-you

ecosystems, including npm, and evaluate whether dependency constraints and semantic versioning are effective measures for avoiding breaking changes. They find that both these measures are not perfect and that there is a need for better tooling. One such tool can be the testing technique by Mezzetti et al. [25] which automatically detects whether an update of a package contains a breaking change in the API. With this method, they can identify type-related breaking changes between two versions. They identify 26 breaking changes in 167 updates of important npm packages. Pfretzschner et al. [27] describe four possible dependency-based attacks that exploit weaknesses such as global variables or monkeypatching in Node.js. They implement a detection of such attacks, but they do not find any real-world exploits. One way to mitigate these attacks is implemented by Vasilakis et al. [32] in *BreakApp*, a tool that creates automatic compartments for each dependency to enforce security policies. This increases security when using untrusted third-party packages. Furthermore, third-party packages can have security vulnerabilities that can impact all the dependents. Davis et al. [9] and Staicu et al. [29] find denial of service vulnerabilities in regular expressions in the npm ecosystem. In another study, Staicu et al. [30] find several injection vulnerabilities due to the *child_process* module or the *eval* function. Brown et al. [6] discuss bugs in the binding layers of both server-side and client-side JavaScript platforms, while Wang et al. [33] analyze concurrency bugs in Node.js Finally, Gong [16] presents a dynamic analysis system for identifying vulnerable and malicious code in npm. He reports more than 300 previously unknown vulnerabilities, some of which are clearly visible on the figures in Section 4.3. Furthermore, there are studies that look at how frequent security vulnerabilities are in the npm ecosystem, how fast packages fix these and how fast dependent packages upgrade to a non-vulnerable version. Chatzidimitriou et al. [7] build an infrastructure to measure the quality of the npm ecosystem and to detect publicly disclosed vulnerabilities in package dependencies. Decan et al. [13] perform a similar study but they investigate the evolution of vulnerabilities over time. They find that more than half of the dependent packages are still affected by a vulnerability after the fix is released. However, we show that the problem is even more serious because for more than half of the npm packages there is no available patch.

Client-Side (JavaScript) Security Client-side security is a vast and mature research area and it is out scope to extensively survey it here. Instead, we focus on those studies that analyze dependencies in client-side code. Nikiforakis et al. [26] present a study of remote inclusion of JavaScript libraries in the most popular 10,000 websites. They show that an average website in their data set adds between 1.5 and 2 new dependencies per year. Similar to our work, they then discuss several threat models and attacks that can occur in this tightly connected ecosystem. Lauinger et al. [20] study the inclusion of libraries with known vulnerabilities in both popular and average websites. They show that 37% of the websites in their data set include at least one vulnerable library. This number is suprisingly close to the reach we observe in npm for the vulnerable code. However, one should take both these results with a grain of salt since inclusion of vulnerable libraries does not necessary lead to a security problem if the library is used in a safe way. Libert et al. [22] perform a HTTP-level analysis of third-party resource inclusions, i.e., dependencies. They conclude that nine in ten websites leak data to third-parties and that six in ten spwan third-party cookies.

Studies of Software Ecosystems Software ecosystem research has been rapidly growing in the last year. Manikas [23] surveys the related work and observes a maturing field at the intersection of multiple other research areas. Nevertheless, he identifies a set of challenges, for example, the problem of generalizing specific ecosystem research to other ecosystems or the lack of theories specific to software ecosystems. Serebrenik et al. [28] perform a meta-analysis of the difficult tasks in software ecosystem research and identify six types of challenges. For example, how to scale the analysis to the massive amount of data, how to research the quality and evolution of the ecosystem and how to dedicate more attention to comparative studies. Mens [24] further looks at the socio-technical view on software maintenance and evolution. He argues that future research needs to study both the technical and the social dimension of the ecosystem. Our study follows this recommendation as it not only looks at the influence of a package on the npm ecosystem, but also at the influence of the maintainers. Several related work advocates metrics borrowed from other fields. For example, Lertwittayatrai et al. [21] use network analysis techniques to study the topology of the JavaScript package ecosystem and to extract insights about dependencies and their relations. Another study by Kabbedijk et al. [17] looks at the social aspect of the Ruby software ecosystem by identifying different roles maintainers have in the ecosystem, depending on the number of developers they cooperate with and on the popularity of their packages. Overall, the research field is rising with a lot of studied software ecosystems in addition to the very popular ones such as JavaScript which is the focus of our study.

Ecosystem Evolution Studying the evolution of an ecosystem shows how fast it grows and whether developers still contribute to it. Wittern et al. [34] study the whole JavaScript ecosystem, including GitHub and npm until September 2015. They focus on dependencies, the popularity of packages and version numbering. They find that the ecosystem is steadily growing and exhibiting a similar effect to a power law distribution as only a quarter of packages is dependent upon. Comparing these numbers with our results, we see a continuous near-exponential growth in the number of released packages and that only 20% of all packages are dependent upon. A similar study that includes the JavaScript ecosystem by Kikas et al. [18] collects data until May 2016 and focuses on the evolution of dependencies and the vulnerability of the

dependency network. They confirm the same general growth as the previous study. Furthermore, they find packages that have a high impact with up to 30% of other packages and applications affected. Our study gives an update on these studies and additionally looks at the evolution of maintainers as they are a possible vulnerability in the ecosystem. The dependency network evolution was also studied for other ecosystems. Decan et al. [14] compare the evolution of seven different package managers focusing on the dependency network. Npm is the largest ecosystem in their comparison and they discover that dependencies are frequently used in all these ecosystems with similar connectedness between packages. Bloemen et al. [4] look at software package dependencies of the Linux distribution *Gentoo* where they use cluster analysis to explore different categories of software. German et al. [15] study the dependency network of the *R* language and the community around its user-contributed packages. Bavota et al. [3] analyze the large Apache ecosystem of Java libraries where they find that while the number of projects grows linearly, the number of dependencies between them grows exponentially. Comparing this to the npm ecosystem, we find the number of packages to grow super-linearly while the average number of dependencies between them grows linearly.

7 Conclusions

We present a large-scale study of security threats resulting from the densely connected structure of npm packages and maintainers. The overall conclusion is that npm is a small world with high risks. It is "small" in the sense that packages are densely connected via dependencies. The security risk are "high" in the sense that vulnerable or malicious code in a single package may affect thousands of others, and that a single misbehaving maintainer, e.g., due to a compromised account, may have a huge negative impact. These findings show that recent security incidents in the npm ecosystem are likely to be the first signs of a larger problem, and not only unfortunate individual cases. To mitigate the risks imposed by the current situation, we analyze the potential effectiveness of several mitigation strategies. We find that trusted maintainers and a code vetting process for selected packages could significantly reduce current risks.

Acknowledgments

This work was supported by the German Federal Ministry of Education and Research and by the Hessian Ministry of Science and the Arts within CRISP, by the German Research Foundation within the ConcSys and Perf4JS projects. The authors would also like to thank the team at r2c for their engineering support in obtaining the data for this work.

References

[1] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. Why do developers use trivial packages? an empirical case study on npm. In *FSE*, 2017.

[2] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. Why do developers use trivial packages? an empirical case study on npm. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 385–395, 2017.

[3] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. The evolution of project inter-dependencies in a software ecosystem: The case of apache. In *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*, pages 280–289, 2013.

[4] Remco Bloemen, Chintan Amrit, Stefan Kuhlmann, and Gonzalo Ordóñez-Matamoros. Gentoo package dependencies over time. In *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, pages 404–407, 2014.

[5] Christopher Bogart, Christian Kästner, James D. Herbsleb, and Ferdian Thung. How to break an API: cost negotiation and community values in three software ecosystems. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 109–120, 2016.

[6] Fraser Brown, Shravan Narayan, Riad S. Wahby, Dawson R. Engler, Ranjit Jhala, and Deian Stefan. Finding and preventing bugs in javascript bindings. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 559–578, 2017.

[7] Kyriakos C. Chatzidimitriou, Michail D. Papamichail, Themistoklis G. Diamantopoulos, Michail Tsapanos, and Andreas L. Symeonidis. npm-miner: an infrastructure for measuring the quality of the npm registry. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 42–45, 2018.

[8] Eleni Constantinou and Tom Mens. An empirical comparison of developer retention in the rubygems and npm software ecosystems. *ISSE*, 13(2-3):101–115, 2017.

[9] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. The impact of regular expression denial of service (redos) in practice: an empirical study at the ecosystem scale. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 246–256, 2018.

[10] James C. Davis, Eric R. Williamson, and Dongyoon Lee. A sense of time for javascript and node.js: First-class timeouts as a cure for event handler poisoning. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, pages 343–359, 2018.

[11] Alexandre Decan, Tom Mens, and Maëlick Claes. An empirical comparison of dependency issues in OSS packaging ecosystems. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, pages 2–12, 2017.

[12] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the evolution of technical lag in the npm package dependency network. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*, pages 404–414, 2018.

[13] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 181–191, 2018.

[14] Alexandre Decan, Tom Mens, and Philippe Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *CoRR*, abs/1710.04936, 2017.

[15] Daniel M. Germán, Bram Adams, and Ahmed E. Hassan. The evolution of the R software ecosystem. In *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013*, pages 243–252, 2013.

[16] Liang Gong. *Dynamic Analysis for JavaScript Code*. PhD thesis, University of California, Berkeley, 2018.

[17] Jaap Kabbedijk and Slinger Jansen. Steering insight: An exploration of the ruby software ecosystem. In *Software Business - Second International Conference, ICSOB 2011, Brussels, Belgium, June 8-10, 2011. Proceedings*, pages 44–55, 2011.

[18] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. Structure and evolution of package dependency networks. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 102–112, 2017.

[19] Raula Gaikovina Kula, Ali Ouni, Daniel M. Germán, and Katsuro Inoue. On the impact of micro-packages: An empirical study of the npm javascript ecosystem. *CoRR*, abs/1709.04638, 2017.

[20] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. In *NDSS*, 2017.

[21] Nuttapon Lertwittayatrai, Raula Gaikovina Kula, Saya Onoue, Hideaki Hata, Arnon Rungsawang, Pattara Leelaprute, and Kenichi Matsumoto. Extracting insights from the topology of the javascript package ecosystem. In *24th Asia-Pacific Software Engineering Conference, APSEC 2017, Nanjing, China, December 4-8, 2017*, pages 298–307, 2017.

[22] Timothy Libert. Exposing the hidden web: An analysis of third-party HTTP requests on 1 million websites. *CoRR*, abs/1511.00619, 2015.

[23] Konstantinos Manikas. Revisiting software ecosystems research: A longitudinal literature study. *Journal of Systems and Software*, 117:84–103, 2016.

[24] Tom Mens. An ecosystemic and socio-technical view on software maintenance and evolution. In *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*, pages 1–8, 2016.

[25] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. Type regression testing to detect breaking changes in node.js libraries. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, pages 7:1–7:24, 2018.

[26] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: large-scale evaluation of remote JavaScript inclusions. In *CCS*, pages 736–747, 2012.

[27] Brian Pfretzschner and Lotfi Ben Othmane. Identification of dependency-based attacks on node.js. In *Proceedings of the 12th International Conference on Availability, Reliability and Security, Reggio Calabria, Italy, August 29 - September 01, 2017*, pages 68:1–68:6, 2017.

[28] Alexander Serebrenik and Tom Mens. Challenges in software ecosystems research. In *Proceedings of the 2015 European Conference on Software Architecture Workshops, Dubrovnik/Cavtat, Croatia, September 7-11, 2015*, pages 40:1–40:6, 2015.

[29] Cristian-Alexandru Staicu and Michael Pradel. Freezing the web: A study of redos vulnerabilities in javascript-based web servers. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, pages 361–376, 2018.

[30] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. SYNODE: understanding and automatically preventing injection attacks on NODE.JS. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[31] Nikolai Philipp Tschacher. *Typosquatting in programming language package managers*. PhD thesis, Universität Hamburg, Fachbereich Informatik, 2016.

[32] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. Breakapp: Automated, flexible application compartmentalization. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[33] Jie Wang, Wensheng Dou, Yu Gao, Chushu Gao, Feng Qin, Kang Yin, and Jun Wei. A comprehensive study on real world concurrency bugs in node.js. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 520–531, 2017.

[34] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. A look at the dynamics of the javascript package ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, pages 351–361, 2016.

"Johnny, you are fired!" – Spoofing OpenPGP and S/MIME Signatures in Emails

Jens Müller[1], Marcus Brinkmann[1], Damian Poddebniak[2], Hanno Böck, Sebastian Schinzel[2], Juraj Somorovsky[1], and Jörg Schwenk[1]

[1]*Ruhr University Bochum*
[2]*Münster University of Applied Sciences*

Abstract

OpenPGP and S/MIME are the two major standards to encrypt and digitally sign emails. Digital signatures are supposed to guarantee authenticity and integrity of messages. In this work we show practical forgery attacks against various implementations of OpenPGP and S/MIME email signature verification in five attack classes: (1) We analyze edge cases in S/MIME's container format. (2) We exploit in-band signaling in the GnuPG API, the most widely used OpenPGP implementation. (3) We apply MIME wrapping attacks that abuse the email clients' handling of partially signed messages. (4) We analyze weaknesses in the binding of signed messages to the sender identity. (5) We systematically test email clients for UI redressing attacks.

Our attacks allow the spoofing of digital signatures for arbitrary messages in 14 out of 20 tested OpenPGP-capable email clients and 15 out of 22 email clients supporting S/MIME signatures. While the attacks do not target the underlying cryptographic primitives of digital signatures, they raise concerns about the actual security of OpenPGP and S/MIME email applications. Finally, we propose mitigation strategies to counter these attacks.

1 Introduction

Email is still the most important communication medium on the Internet and predates the World Wide Web by roughly one decade. At that time, message authenticity was not a major concern, so the early SMTP and email [1, 2] standards did not address confidentiality or the authenticity of messages.

Email Authenticity As the ARPANET evolved into the Internet, email usage changed. Email is now used for sensitive communication in business environments, by the military, politicians and journalists. While technologies such as SPF, DKIM, and DMARC can be used to authenticate the domain of the sending SMTP server, these are merely helpful in mitigating spam and phishing attacks [3]. These technologoies, however, do not extend to authenticating the sending *person*, which is necessary to provide message authenticity.

Two competing email security standards, OpenPGP [4] and S/MIME [5], offer end-to-end authenticity of messages by digital signatures and are supported by many email clients since the late 1990s. Digital signatures provide assurance that a message was written by a specific person (i.e., authentication and non-repudiation) and that it was not changed since then (i.e., integrity of messages). Adoption is still low[1] due to severe usability issues, but both technologies have a large footprint either in the industry (S/MIME) or with high-risk roles such as journalists, lawyers, and freedom activists (OpenPGP). One example: Debian (a volunteer group with over 1000 members) relies on the authenticity of signed emails for voting on project leaders and proposals. We thus ask: *Is it possible to spoof a signed email such that it is indistinguishable from a valid one even by an attentive user?*

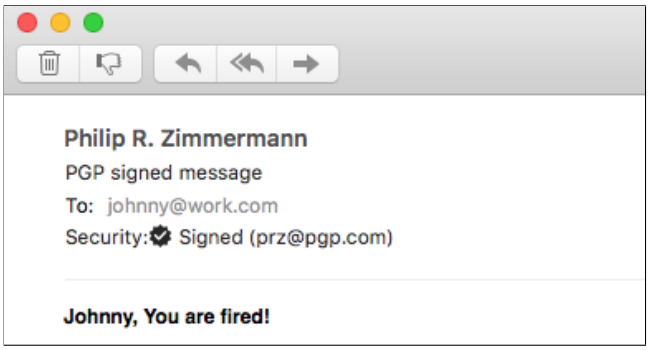


Figure 1: Screenshot of a spoofed PGP signature in Apple Mail, based on wrapping a signed email published by Phil Zimmermann.

Spoofing Valid Signatures Signature verification in the context of email is complex. For example, emails can be signed by an entity other than the sender or signed emails may be forwarded (resulting in partly signed messages).

[1]We examined OpenPGP keyservers and measured over 20k key uploads per month consistently over the last 4 years. We also found that Thunderbird reports 150k daily users (on work days) for the PGP-plugin Enigmail.

Also, signatures are an optional feature of email and therefore generally not enforced. Most importantly, the result of the verification must be presented to the user such that there is no room for misinterpretation. Any failure to do so may lead to a signature spoofing attack as shown in Figure 1. This paper describes the results of our analysis of the most widely used email clients supporting PGP or S/MIME signatures.

Attack Classes Our attacks do not break the cryptography in digital signatures, but rather exploit weaknesses in the way PGP and S/MIME signatures are verified by email clients, and how the verification outcome is presented to the user.

We define the following five attack classes:

1. **CMS attacks.** Cryptographic Message Syntax (CMS) is a versatile standard for signed and encrypted messages within the X.509 public-key infrastructure. We found flaws in the handling of emails with contradicting or unusual data structures (such as multiple signers) and in the presentation of issues in the X.509 trust chain.
2. **GPG API attacks.** GnuPG is the most widely used OpenPGP implementation, but it only offers a very restricted command line interface for validating signatures. This interface is vulnerable to injection attacks.
3. **MIME attacks.** The body of an email is conceptually a MIME *tree*, but typically the tree has only one leaf which is signed. We construct non-standard MIME trees that trick clients into showing an unsigned text while verifying an unrelated signature in another part.
4. **ID attacks.** The goal of this attack class is to display a valid signature from the identity (ID) of a trusted communication partner located in the mail header, although the crafted email is actually signed by the attacker.
5. **UI attacks.** Email clients indicate a valid signature by showing some security indicators in the user interface (UI), for example, a letter with a seal. However, several clients allow the mimicking of important UI elements by using HTML, CSS, and other embedded content.

Contributions We make the following contributions:

- We present the results of our structured analysis of OpenPGP and S/MIME signature verification in 25 widely used email clients.
- We present three new attack classes: attacks based on CMS evaluation, on the syntax of the GnuPG machine interface, and on wrapping signed content within invisible subtrees of the MIME tree.
- We adapt two attack classes of web security to the email context, namely UI redressing and ID spoofing.
- We show that our attacks bypass signature validation in about 70% of the tested email clients, including Outlook, Thunderbird, Apple Mail, and iOS Mail.

Coordinated Disclosure We reported all our attacks and additional findings to the affected vendors and gave advice on appropriate countermeasures.

2 Background

2.1 End-to-End Email Authenticity

The digital signature parts of the OpenPGP and S/MIME standards provide end-to-end authenticity for email messages. First and foremost, both technologies are configured on the endpoints and technically-versed users can therefore choose to use them independently of the email server configuration. In both standards, the keys are bound to users and thus authenticate users independently of the transport.

OpenPGP Email Signing Phil Zimmerman invented PGP (Pretty Good Privacy) in 1991 and due to its popularity, PGP was standardized as *OpenPGP* by the IETF [6]. The most popular implementation is GnuPG.[2] There are two common ways to use OpenPGP in emails. With *Inline PGP*, the email body directly contains the OpenPGP data. The MIME multipart standard is not used and the MIME type is `text/plain`. With *PGP/MIME*, the email has a `multipart/signed` MIME structure, where the signed message is the first part and the detached signature is the second part. Some email clients support PGP natively, but most (in particular Thunderbird, Apple Mail, and Outlook) need a plugin, which provides an intermediate layer between the mail client and a PGP implementation like GnuPG.

S/MIME Email Signing In 1999, the IETF published S/MIME (Secure/Multipurpose Internet Mail Extension) version 3 as an extension to the MIME standard with certificate-based cryptography [5]. S/MIME is the result of a long history of secure email protocols and can be seen as the first Internet standards-based framework to digitally sign, authenticate, or encrypt emails. S/MIME uses the *Cryptographic Message Syntax (CMS)* as its underlying container format. The signatures themselves are always CMS encoded, but the signed message can either be included in the CMS (*opaque* signature) or be transmitted as the first part of a `multipart/signed` message (*detached* signature).

2.2 Trust and Validity

Verifying the cryptographic integrity of a signature is often not sufficient. In addition to this verification, the public key

[2] W. Koch, *GNU Privacy Guard*, `https://gnupg.org/`.

that generated the signature must be connected to some actual person or entity, such as an email address, by a certificate. S/MIME certificates are issued by certificate authorities which are trusted by the email clients. It is easy for users to order S/MIME certificates and sign messages which are accepted by all clients. PGP as a product of the cypherpunk movement distrusts central authorities, so user IDs in PGP are only self-signed by default. This does not provide any protection against spoofing and puts responsibility for trust management into the hands of users and applications. In fact, no version of the OpenPGP standard defines a trust model for user ID binding signatures. Historically, users of PGP were encouraged to participate in the Web of Trust, a decentralized network of peers signing each other's user IDs, paired with a scoring system to establish *trust paths* between two peers that want to communicate. Signatures and keys are exchanged through a network of public keyservers. This approach has been found difficult to use, privacy invasive, and hard to scale, so some email clients implement their own trust model (e.g., OpenKeychain, R2Mail2, and Horde/IMP).

3 Attacker Model and Methodology

In our scenario we assume two trustworthy communication partners, Alice and Bob, who have securely exchanged their public PGP keys or S/MIME certificates. The goal of our attacker Eve is to create and send an email with arbitrary content to Bob whose email client falsely indicates that the email has been digitally signed by Alice.

Attacker Model We assume that Eve is able to create and send arbitrary emails to Bob. The email's sender is spoofed to Alice's address, for example, by spoofing the `FROM` header, a known impersonation technique which should be prevented by digital signatures. This is our default attacker model with the weakest prerequisites. It is sufficient for the UI attack class, and some CMS and GPG API attacks.

For the MIME attack class and some CMS attacks, we also assume that the attacker has a single valid S/MIME or OpenPGP signature from Alice which may have been obtained from previous email correspondence, public mailing lists, signed software packages, signed GitHub commits, or other sources. This is a weak requirement as well, because digital signatures are usually not kept secret. In fact, in many cases digital signatures are used explicitly to give public proof that some content was created by the signer.

For the ID attack class and one attack in the GPG API class, we assume that Bob trusts Eve's signatures. For S/MIME this condition always holds because Eve can easily obtain a valid certificate from a trusted CA for her own email address. For OpenPGP, Bob must import Eve's public key and mark it as valid. This is a stronger condition, but holds if Eve is a legitimate communication partner of Bob.

An overview of the attack classes and attacker models is given in Table 1. Each attack class is described in section 4 and the subscript identifies the specific attack, i.e., M_1 identifies attack 1 in the MIME attack class.

	Attacker Model		
Attack class	**Mail only**	**Need signature**	**Key trusted**
CMS (4.1)	C_3, C_4	C_1, C_2	–
GPG (4.2)	G_1	–	G_2
MIME (4.3)	–	M_1, M_2, M_3, M_4	–
ID (4.4)	–	–	I_1, I_2, I_3
UI (4.5)	U_1	–	–

Table 1: Attacker's capabilities for all test cases in each attack class.

Our attacker model does not include any form of social engineering. The user opens and reads received emails as always, so awareness training does not help to mitigate the attacks.

Methodology We define that the *authenticity of a signed email is broken in the context of an email client UA* if the presentation of a crafted email in *UA* is indistinguishable from the presentation of a "valid" signed email (either as *perfect* or *partial* forgery). Furthermore, we document cases where we could forge *some, but not all* GUI elements required for indistinguishability (i.e., a *weak* forgery).

- **Perfect forgery** (●) If a presentation is identical at any number of user interactions, regardless of any additional actions the user takes within the application, we call the forgery "perfect" (e.g., Figure 1).
- **Partial forgery** (◐) If a presentation is only identical at the first user interaction (i.e., when an email is opened and the standard GUI features are visible), we call the forgery "partial" (e.g., Figure 14).
- **Weak forgery** (○) If a presentation contains contradicting GUI elements at the first user interaction, with some *but not all* elements indicating a valid signature, we call this forgery "weak" (e.g., Figure 15).

We suspect that partial forgeries already go unnoticed by unwitting users, so we classify perfect and partial forgeries as successful attacks. Weak forgeries show signs of spoofing at the first glance. As part of our evaluation, we provide screenshots of interesting cases to illustrate the differences.

Selection of the Clients We evaluate our attacks against 25 widely-used email clients given in Table 2 and Table 3. Of these, 20 support PGP and 22 are capable of S/MIME signature verification. They were selected from a comprehensive list of over 50 email clients assembled from public software directories for all major platforms (i.e., Windows, Linux, macOS, Android, iOS, and web). Email clients were

excluded when they did not support PGP or S/MIME signatures, were not updated for several years, or the cost to obtain them would be prohibitive (e.g., appliances). All clients were tested in the default settings with an additional PGP or S/MIME plugin installed, where required.

4 Attacks

4.1 CMS Attack Class

The Cryptographic Message Syntax (CMS, the container format used by S/MIME) is a versatile standard for signed and encrypted emails. It not only supports a broad range of use cases (e.g., multiple signers), but also copes with legacy problems like lack of software support and misbehaving gateways. This made the standard more complex; several values in a CMS object are optional, or may contain zero or more values.[3] Furthermore, two different signature formats are defined. This makes it difficult for developers to test all possible combinations (either plausible or implausible).

Opaque and Detached Signatures The CMS and S/MIME standards define two forms of signed messages: *opaque* and *detached* signatures [7] (also called *embedded* and *external* signatures). The signature is always a CMS object, but the corresponding message can either be embedded into this object or transmitted by other means.

When signing in opaque mode, the to-be-signed content (i.e., "the message text") is embedded into the binary CMS signature object via a so called `eContent` (or "embedded content") field (see Figure 2a).

In detached signatures, the `eContent` must be absent in order to signal that the content will be provided by other means. This is what the multipart/signed structure does; the email is split into two MIME parts, the first one is the content and the second one is the CMS signature without the `eContent` field (see Figure 2b).

eContent Confusion (C_1) A confusing situation arises when the `eContent` field is present *even though* the multipart/signed mechanism is used. In this case, the client can choose which of the two contents (i.e., either the opaque or detached contents) to verify *and* which of the two contents to display. Clearly, it is a security issue when the verified content is not equal to the content which is displayed.

The "eContent Confusion" allows perfect forgeries of arbitrary signed emails for a person from which we already have a signed email. Because opaque signatures can be transformed into detached signatures and vice versa, any signed email will work for the attack.

[3] Here *optional* means potentially absent, which is different from present but empty.

```
From: Alice
To: Bob
Subject: Opaque signature
Content-Type: application/pkcs7-mime; smime-type=signed-data
Content-Transfer-Encoding: base64

<base64-encoded CMS object with eContent>
```

(a) A message with an opaque signature. The message is embedded in the CMS object and is not directly readable by a human.

```
From: Alice
To: Bob
Subject: Detached signature
Content-Type: multipart/signed; protocol="application/pkcs7-
     signature"; boundary="XXX"

--XXX
Content-Type: text/plain

Hello, World! This text is signed.
--XXX
Content-Type: application/pkcs7-signature;
Content-Transfer-Encoding: base64

<base64-encoded CMS object without eContent>
--XXX--
```

(b) A message with a detached signature. The message is in the first MIME part and directly readable by a human. Legacy software tended to damage line endings or encoding in such emails, which broke the signature verification.

Figure 2: Opaque and detached signatures as used in S/MIME.

Attack Refinement. Although the `eContent` is not shown in the email client, it is easily revealed under forensic analysis that an old email was reused for this attack. Interestingly, the attack can be refined to remove the original content entirely without affecting the outcome of the signature verification.

Similarly to opaque and detached signatures, where an absent `eContent` signals that the content is provided somewhere else, CMS supports so called "signed attributes", whose absence or presence signals what was signed. If a `signedAttrs` field is present, the signature covers the exact byte sequence of the `signedAttrs` field and not the content per se. If naively implemented, this would of course leave the content unauthenticated. Therefore, the `signedAttrs` field *must* contain a hash of the content [7]. If the `signedAttrs` field is absent, the signature covers the `eContent` directly.

This indirect signing allows the replacement of the original content with the exact byte sequence of the `signedAttrs` field without affecting the signature verification outcome. An email modified in this way will appear "empty" or contain seemingly "garbage" (because the `signedAttrs` is interpreted as ASCII). In either case, this can be used to hide where the old signature originated from. We consider this a noteworthy curiosity.

Multiple Signers (C_2) S/MIME and CMS allow multiple signers in parallel to sign the same content [7]. Obviously, the outcome of the verification may differ for each signer

and the user interface should make that clear. However, it is reasonable to show a simplified version. We consider it a forgery if an *invalid* signature is marked as "valid" due to the presence of an unrelated valid signature.

No Signers (C_3) A CMS signature object may contain zero or more signers. Although RFC 5652 gives limited advice regarding zero signers, it does not state explicitly what to do with "signed messages" without a signer.

Trust Issues (C_4) In contrast to OpenPGP, S/MIME is built upon trust hierarchies. Certificates are authentic as long as they can be traced back to a valid root certificate. In practice, this means that most S/MIME certificates (in the Internet PKI) are indirectly trusted. However, clients must check the validity of the certificate chain. We consider it a forgery if a client accepts invalid certificates, such as self-signed certificates or otherwise untrusted or non-conforming certificates.

4.2 GPG API Attack Class

GnuPG, a stand-alone OpenPGP implementation, provides a stateful text-based command-line interface with approximately 380 options and commands. The complexity of the API and the need for consumers to parse GnuPG's string messages provides a rich attack surface.

GnuPG Status Lines GnuPG provides a machine-readable interface by emitting *status lines* in a simple text-based format (see Figure 3), via the `--status-fd` option. Each status line starts with `[GNUPG:]` and one of approximately 100 possible keywords (such as `GOODSIG`), followed by additional text specific to the keyword.

Although some documentation exists, it does not cover all possible sequences of status lines and their significance for any given operation. In fact, due to streaming processing, the complexity of the API reflects the overall complexity of the OpenPGP message format. In particular, we note the following risk factors:

- The API contains text under the attacker's control (e.g., the `<user-id>` in `GOODSIG`), which must be escaped to prevent injection attacks.
- The number and order of status lines depend on the OpenPGP packet sequence, which is under the attacker's control. Applications must handle all combinations correctly that are allowed by GnuPG.
- The API is stateful, i.e., the semantics of a status line can depend on its position in the sequence. For example, the validity indicated by a `TRUST_*` line applies only to the signature started by the previous `NEWSIG`.

```
$ gpg --status-fd 2 --verify
[GNUPG:] NEWSIG
[GNUPG:] GOODSIG 88B08D5A57B62140 <alice@good.com>
[GNUPG:] VALIDSIG 3CB0E84416AD52F7E186541888B08D5A57B62140
     2018-07-05 1530779689 0 4 0 1 8 00 3
     CB0E84416AD52F7E186541888B08D5A57B62140
[GNUPG:] TRUST_FULLY 0 classic
```

(a) Example output for a single trusted signature (excerpt).

```
NEWSIG [<signers-user-id>]
GOODSIG <key-id> <user-id>
BADSIG <key-id> <user-id>
VALIDSIG <fingerprint> <date> <create-timestamp> <expire-
     timestamp> <version> <reserved> <public-key-algorithm> <
     hash-algorithm> <signature-class> [<primary-key-
     fingerprint>]
TRUST_NEVER <error-token>
TRUST_FULLY [0 [<validation-model>]]
PLAINTEXT <format> <timestamp> <filename>
PLAINTEXT_LENGTH <length>
```

(b) Important status lines for signature verification from GnuPG.

Figure 3: Status lines output by GnuPG as a side-effect of streaming message processing.

- The use of the API requires a good understanding of OpenPGP and trust models as implemented in GnuPG. The `GOODSIG`, `VALIDSIG` and `TRUST_*` lines have very specific technical meaning that is not always apparent from the inconsistent terminology in the interface.
- By default, GnuPG runs in the context of the user's home directory, using their configuration files and keyrings, which can influence the output of GnuPG within, and outside of, the status line API.

We focus our work on injection attacks and applications parsing the interface. First, we review the source code of GnuPG to identify places where an attacker could inject untrusted data at trusted positions in the API. Then we review all open source mail clients to identify exploitable mistakes in the API parser.

In-band Injection Attacks (G_1) There are various places in the GnuPG status line API that contain untrusted data under the attacker's control. For example, the `<user-id>` in a `GOODSIG` status line is an arbitrary string from the public key that can be crafted by an attacker. A naive idea is to append a newline character followed by a spoofed status line into the user ID of a public key. Normally, GnuPG protects against this naive attack by properly escaping special characters.

In addition to the status line API, we also review the logging messages for injection attacks. This is due to a common pattern, where applications using GnuPG conflate the status API and the logging messages by specifying the same data channel `stdout` for both (using the command line option `--status-fd 2`). Best practice requires separate channels to be used, but technical limitations can make this difficult for some plugins and cross-platform applications.

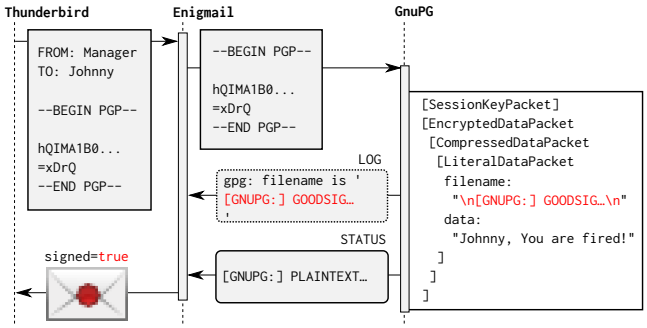


Figure 4: In-band injection of GnuPG status lines via log messages.

If an injection attack is successful, it can be very powerful, as the attacker can spoof the status lines entirely and provide arbitrary data to the application. Such spoofed status lines can include forged indications of a successful signature validation for arbitrary public keys. A valid PGP message containing the injection payload is showed in Figure 4.

Attacks on Email Clients Using GPG (G_2) The GnuPG interface provides only limited functionality; for example, it is not possible to validate a signature against a single public key in the keyring, but only against *all* public keys contained therein. Since this is insufficient for validating the sender of an email, GnuPG returns a string containing the user ID and the result of the validation. By manipulating this string, mail clients can be tricked into giving false validation results.

Applications using the GnuPG status line API have to parse the status lines and direct a state machine, keeping track of verification and decryption results under a wide range of different inputs, user configurations and GnuPG versions. Thus, application developers often use a common design pattern to deal with the complexity, such as: Iterating over all status line messages, parsing the details of those status lines that have information relevant to the task the application is interested in, and ignoring all other unknown or unsupported messages. This can lead to a number of serious vulnerabilities. For example, if an application is unprepared to handle multiple signatures, it might not reset the signature verification state at the `NEWSIG` status line, conflating the verification result of multiple signatures into a single result state. This might allow an attacker to add more signatures to an existing message to influence the result. Another example is the use of regular expressions that are not properly anchored to the status line API data format, thereby allowing an attacker to inject data that, although it is properly escaped by GnuPG, is then misinterpreted by the application.

4.3 MIME Attack Class

In this section we discuss attacks on how email clients handle partially signed messages in the case that the signed part is wrapped within the MIME tree of a multipart message. For this class of attacks, the attacker is already in possession of at least one message and a corresponding valid signature from the entity to be impersonated. The obtained message can be in *Inline PGP*, *PGP/MIME*, or *S/MIME* as all formats can be embedded as sub-parts within a multipart message.

Prepending Attacker's Text (M_1) Email clients may display a valid signature verification status even if only a single MIME part is correctly signed. In such a scenario of partially signed emails, the attacker can obfuscate the existence of the correctly signed original message within a multipart email. For example, this can be achieved by prepending the attacker's message to the originally signed part, separated by a lot of newlines, resulting in a weak forgery.

Hiding Signed Part with HTML (M_2) Another option is to completely hide the original part with HTML and/or CSS, resulting in a perfect forgery. There are several ways to do this. One way occurs if the email client renders the output of multiple MIME-parts within a single HTML document presented to the user, then the signed part can simply be commented out, for example, using HTML comments. Furthermore, it can be embedded in (and therefore hidden within) HTML tags, or wrapped into CSS properties like `display:none`. An example for such a MIME-wrapping attack based on a hidden signed part is shown in Figure 5.

```
From: manager@work.com
To: johnny@work.com
Subject: Signed part hidden with CSS/HTML
Content-Type: multipart/mixed; boundary="XXX"

--XXX
Content-Type: text/html

Johnny, you are fired!
<div style="display:none"><plaintext>

--XXX
---BEGIN PGP SIGNED MESSAGE---
Hash:  SHA512

Congratulations, you have been promoted!
---BEGIN PGP SIGNATURE---
iQE/BAEBAgApBQJbW1tqIhxCcnVjZSBXYXluZSA8YnJ1Y2V3YXluZTQ1...
---END PGP SIGNATURE---

--XXX--
```

Figure 5: Signature spoofing attack based on MIME wrapping. The signed part is hidden by HTML/CSS, while the message "Johnny, You are fired!" is displayed by the email client.

Hiding Signed Part in Related Content (M_3) Even if there is a strict isolation between multiple MIME parts, it can be broken using `cid:` references (see RFC 2392). This can be achieved by constructing a `multipart/related` message consisting of two parts. The first MIME part contains

```
From: Philip R. Zimmermann <prz@pgp.com>
To: johnny@work.com
Subject: PGP signed message
Content-Type: multipart/related; boundary="XXX"

--XXX
Content-Type: text/html

<b>Johnny, You are fired!</b>
<img src="cid:signed-part">

--XXX
Content-ID: signed-part

---BEGIN PGP SIGNED MESSAGE---
A note to PGP users: ...
---BEGIN PGP SIGNATURE---
iQA/AwUBOpDtWmPLaR3669X8EQLvOgCgs6zaYetj4JwkCiDSzQJZ1ugM...
---END PGP SIGNATURE---

--XXX--
```

Figure 6: Multipart email with a `cid:` reference to the signed part.

an attacker-controlled text and a `cid:` reference to the original signed part, which is placed into the second MIME part. An example email to demonstrate such an attack is given in Figure 6. It contains an HTML message part and a signed text which was written and published by Phil Zimmermann back in 2001.[4] The `cid:` reference enforces the signed (but invisible) part to be parsed by the mail client, which indicates a valid signature for the shown message (from the first part). This allows us to impersonate Phil Zimmermann[5] for arbitrary messages. A corresponding screenshot of Apple Mail (GPG Suite) is given in Figure 1 on the first page.

Hiding Signed Part in an Attachment (M_4) Even without using HTML, the originally signed part can be hidden by defining it as an attachment. This can be done by placing it into a sub-part with the additional header line shown below:

```
Content-Disposition: attachment; filename=signature.asc
```

4.4 ID Attack Class

In this section we discuss attacks on how email clients match a signed message to a sender's identity. These attacks are less powerful than those previously discussed, because indistinguishability is rarely given at all levels of user interaction, i.e., many clients allow the user to check the signature details, which may reveal signs of manipulation.

Not Checking If Sender = Signer (I_1) When dealing with digital signatures, the question *Signed by whom?* is important. If Bob's email client simply displayed "valid signature" for any PGP or S/MIME signed message, Eve could sign her message and send it to Bob with Alice set as the sender. This is due to a lack of binding between the user ID from the signature and the address given in the `FROM` header.

[4] P. Zimmermann, A (Inline PGP signed) note to PGP users, `https://philzimmermann.com/text/PRZ_leaves_NAI.txt`

[5] PGP key ID `17AFBAAF21064E513F037E6E63CB691DFAEBD5FC`

```
From: Alice <eve@evil.com>
```

(a) Display name and email address in `FROM` header.

```
From: alice@good.com <eve@evil.com>
```

(b) Display name set to email address `FROM` header.

```
From: alice@good.com
From: eve@evil.com
```

(c) Multiple `FROM` header.

```
Sender: alice@good.com
From: eve@evil.com
```

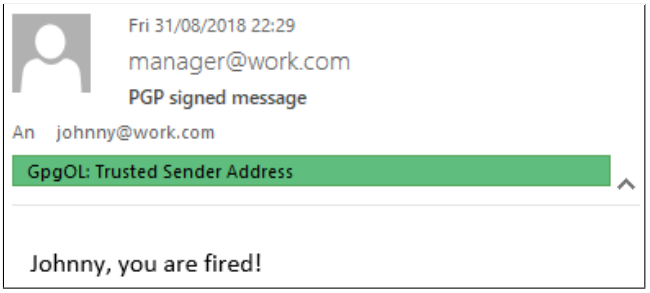
(d) `SENDER` and `FROM` headers.

Figure 7: Examples how to fool signature verification logic if the PGP user ID or the Internet mail address in the signer's S/MIME certificate is compared to sender address in the email header.

Display Name Shown as Signer (I_2) There are two options to handle this problem. First, a mail client can explicitly display the signer's identity somewhere in the UI and let the user compare it to the sender address. Second, the email client can check whether the signer's identity (i.e., email address) equals the sender's address, and show a warning if this is not the case. The second option gives a lot of room for attacks; RFC 2632 for S/MIME signed messages states that "receiving agents *must* check that the address in the From or Sender header of a mail message matches an Internet mail address in the signer's certificate". However, in practice email clients can only check if the `FROM` header *contains* the signer's email address because RFC 5322 allows additional display names to be associated with a mailbox (e.g., `Name <foo@bar>`). If an email client only shows the display name to the user, Eve can simply set *Alice* as display name for her own sender address (see Figure 7a). Also, in such a scenario the display name itself could be set to an email address such as *alice@good.com* that is presented to the user, see Figure 7b.

An example screenshot for Outlook, which required additional effort, is given in Figure 8a. The source code of this mail can be found in Figure 8b. While Outlook always shows the full sender address (*eve@evil.com*), it can simply be "pushed out of the display" by appending a lot of whitespaces to the display name (*manager@work.com*).

From/Sender Header Confusion (I_3) Another problem is how email clients deal with multiple `FROM` fields in the mail header—especially if PGP or S/MIME support is not implemented directly by the email client, but offered through a



(a) Screenshot of a spoofed PGP signed message in Outlook which was actually signed by the attacker (*eve@evil.com*).

```
From: manager@work.com [whitespaces] x <eve@evil.com>
To: johnny@work.com
Reply-to: manager@work.com
Subject: Signed by whom?
Content-Type: multipart/signed; boundary="XXX";
  protocol="application/pgp-signature"

--XXX

Johnny, you are fired!

--XXX
Content-Type: application/pgp-signature

[valid signature by eve@evil.com]
--XXX--
```

(b) Proof-of-concept email source code to forge PGP signatures.

Figure 8: PGP signature spoofing attack against Outlook/GpgOL, based on the RFC 5322 display name shown as the signer's identity.

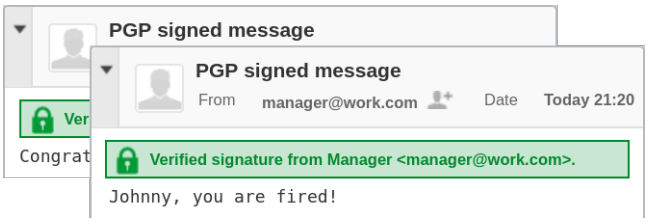
third-party plugin—as there may be different implementations on how to obtain the sender address of an email. For example, the email client could display the email address given in the first `FROM` header while the plugin would perform its checks against any occurrence of this header field (see Figure 7c). Also, the plugin could respect the `SENDER` header in its checks which "specifies the mailbox of the agent responsible for the actual transmission of the message" [8] while the mail client would display the email address taken from the `FROM` email header (see Figure 7d). Note that if Eve sets an additional `Reply-to: alice@good.com` header, which instructs the email client to reply to Alice, such attacks go unnoticed by Bob when replying to the email. They can, however, be detected in most clients by reviewing the signature details and spotting Eve as the real signer.

4.5 UI Attack Class (U_1)

In this section, we discuss UI redressing attacks that exploit the presentation of signature verification results to the user. The attacks are successful if the spoofed message is indistinguishable from a message with a real valid signature.

This attack class exploits that various email clients display the status of the signature within the email content itself. This part of the UI is under the control of the attacker. With HTML, CSS, or inline images it is easy to reproduce security-critical UI elements displaying a "valid signature".

Figure 9a shows a signed and a spoofed email in Roundcube. The spoofed email is based on displaying a valid signature indicator with HTML and CSS (an example of UI redressing). The HTML code is provided in Figure 9b.



(a) Two emails with visually indistinguishable signature indicators. One is PGP-signed, the other is a specially crafted HTML email.

```
From: manager@work.com
To: johnny@work.com
Subject: UI redressing
Content-Type: text/html

<div class="message-part">
<div id="enigma-message1" class="enigma-notice1" style="margin-
     left: -0.25em; margin-bottom: 5px; padding: 6px 12px 6px
     30px; font-weight: bold; background: url(enigma_icons.png
     ) 3px -171px no-repeat #c9e6d3; border: 1px solid #008a2e
     ; color: #008a2e">
      Verified signature from Manager <manager@work.com>.
</div><div class="pre">Johnny, You are fired!</div></div>
```

(b) Forging a PGP signature in Roundcube with UI redressing.

Figure 9: Security indicators in Roundcube. The indicator is in the attacker-controlled HTML area, which allows trivial spoofing.

5 Evaluation

Of the tested 20 clients with PGP support, 15 use *GnuPG* to verify signatures and call it either directly, or through some kind of plugin such as *Enigmail*[6] or *GPG Suite*,[7] or by using the *GPGME*[8] wrapper library. The remaining clients use *OpenPGP.js*,[9] *OpenKeychain*,[10] or a proprietary solution. Of the tested 22 clients with S/MIME support, only five require third party plugins. The results of signature spoofing attacks tested on the various email clients are shown in Table 2 for OpenPGP and in Table 3 for S/MIME.

The results of our evaluation show a poor performance of the overall PGP and S/MIME ecosystems when it comes to trustworthiness of digital signatures; for ten OpenPGP capable clients and seven clients supporting S/MIME we could spoof visually indistinguishable signatures on all UI levels (resulting in perfect forgeries). On four additional OpenPGP

[6] P. Brunschwig, *Enigmail*, `https://enigmail.net/`
[7] L. Pitschl *GPG Suite*, `https://gpgtools.org/`
[8] W. Koch, *GPGME*, `https://github.com/gpg/gpgme`
[9] ProtonMail, *OpenPGP.js* `https://openpgpjs.org/`
[10] Cotech, *OpenKeychain*, `https://openkeychain.org/`

capable clients and eight clients supporting S/MIME, we could spoof visually indistinguishable signatures on the first UI level (resulting in partial forgeries). While none of the attacks directly target the underlying cryptographic primitives, the high success rate raises concerns about the practical security of email applications. We discuss the results for each class of attack in this section. We also published proof-of-concept attack emails and screenshots of partial and weak forgeries in a public repository.[11]

5.1 CMS Attack Class

eContent Confusion (C_1) We found that Thunderbird, Postbox, MailMate, and iOS Mail are vulnerable to eContent confusion. Given a valid S/MIME signature for a specific user, this allows a perfect forgery for any message of this user. Note that opaque signed emails can be transformed into detached signed emails and vice versa. Thus, having *any* signed email from a target is enough to forge an arbitrary number of new messages. Mozilla assigned CVE-2018-18509 to this issue.

Multiple Signers (C_2) Evolution coerces multiple signers into one "valid signature" UI element (see Figure 10). However, the UI reveals the erroneous signature upon further inspection of the UI. By definition, this is a partial forgery.

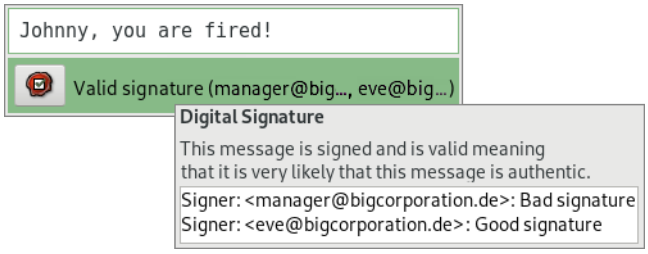


Figure 10: This email has two signers. The first signer did not sign this email. Evolution coerced both signers into one security indicator showing "valid signature". (Minor details were removed from the screenshot to make it smaller.)

No Signers (C_3) In the case of no signers, three email clients, namely Outlook, Mutt, and MailDroid, show some UI elements suggesting a valid signature and some UI elements doing the opposite. We consider this a weak forgery, because there is no clear indication that signature validation has succeeded/failed. In Outlook an otherwise very prominent red error bar is not shown and a seal indicates a valid signature (although it should show a warning sign), see Figure 11. Interestingly, clicking through the UI in Outlook may *strengthen* the illusion of a validly signed message because of the wording in the subdialogs.

[11] `https://github.com/RUB-NDS/Johnny-You-Are-Fired`

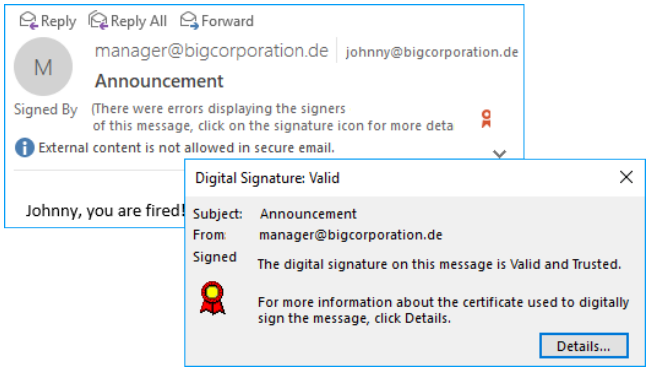


Figure 11: "Signed email" in Outlook with no signers at all.

Trust Issues (C_4) Although none of the clients accepted "extended certificates", i.e., certificates re-signed by a leaf certificate with no `CA=true` flag, we observed that Trojitá and Claws display conflicting UI elements on untrusted certificates (i.e., "success: bad signature"). Nine and MailDroid do not display information about the origin of a certificate on the first level of the UI. This means that, although there are security indicators suggesting a signed email, the origin may be completely untrustworthy, resulting in partial forgery.

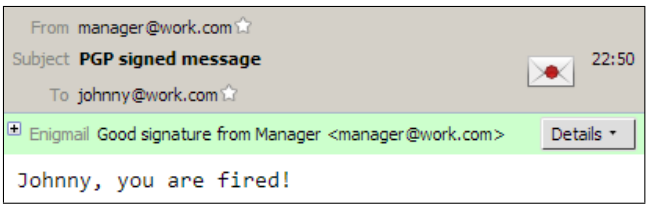
5.2 GPG API Attack Class

We found an injection attack for the "embedded filename" log message in GnuPG, as well as several issues in Enigmail's status line parser and state machine.

Status Line Injection through Embedded Filename (G1). OpenPGP Literal Data Packets, which contain the actual plaintext of a message, contain some metadata, in particular an *embedded filename* that is usually set by the sender to the original filename of a signed or encrypted file. We found that GnuPG's logging message for the embedded filename does not escape newline and other special characters. If an application combines the logging messages with the status line API in a single data channel, an attacker can use the embedded filename to inject arbitrary status lines (up to 255 bytes, which can be sufficient to spoof the `GOODSIG`, `VALIDSIG` and `TRUST_FULLY` lines for a single signature). Figure 12b shows the injected text highlighted. The successful attack is shown in Figure 12a.

Using this attack, we were able to spoof arbitrary signature verification results in Enigmail, GPG Suite, and Mailpile.[12] The attack only assumes our weakest attacker model, as all relevant status lines can be injected into the embedded filename. In fact, the message does not even need to be signed at all; the attack, however, has the additional requirement

[12] Manual signature verification using GnuPG on the command line is also affected; the embedded filename can contain arbitrary terminal escape sequences, allowing the attacker to overwrite any part of the terminal.



(a) Screenshot of a spoofed PGP signature in Thunderbird.

```
$ gpg --status-fd=2 --verbose message.gpg
gpg: original file name=''
[GNUPG:] GOODSIG 88B08D5A57B62140 Manager <manager@work.com>
[GNUPG:] VALIDSIG 3CB0E84416AD52F7E186541888B08D5A57B62140
      2018-07-05 1530779689 0 4 0 1 8 00
      3CB0E84416AD52F7E186541888B08D5A57B62140
[GNUPG:] TRUST_FULLY 0 classic
gpg:  ''
[GNUPG:] PLAINTEXT 62 1528297411 '%0A[GNUPG:]%20GOODSIG[...]
[GNUPG:] PLAINTEXT_LENGTH 56
```

(b) Example output for GnuPG status line injection (excerpt).

Figure 12: Status line injection attack on GnuPG.

that the user has enabled the `verbose` configuration option. This option is not enabled by default, but it is often used by experts and part of several recommended GnuPG settings.

The vulnerability is present in all versions of GnuPG until 2.2.8. Our finding is documented as CVE-2018-12020. Due to the severity of the attack, we also reviewed non-email applications for similar vulnerabilities, see subsection 7.5.

State Confusion and Regular Expressions in Enigmail (G2). We found two flaws in the way Enigmail handles status messages for multiple signatures:

- If the status of the *last* signature is `GOODSIG`, `EXPKEYSIG`, or `REVKEYSIG`, Enigmail will overwrite the signature details (e.g., fingerprint, creation time, algorithms) with those from the *first* `VALIDSIG`, confusing the metadata of two signatures. The attacker can change the state of a signature to good, expired, or revoked by adding a corresponding second signature.
- If *any* of the signatures is `TRUST_FULLY` or `TRUST_ULTIMATE`, and the last signature is good, expired, or revoked, then Enigmail will display the information from the first `VALIDSIG` as trusted.

We also found regular expressions that were not anchored to the beginning of status lines. This allows for injection of fake `VALIDSIG` and other status lines in malicious user ID strings. Combining this with the above, this allows an attacker to control the signature details completely. The attack involves two signatures on a single message, which Enigmail combines in a complex way to a single signature that is shown to the user. We assume that the attacker is trusted, and further assume that the attacker can poison the user's keyring with arbitrary untrusted keys. This is not a strong assumption because in PGP the local keyring is just an insecure cache of public keys. For example, the attacker might rely on automatic key retrieval from public keyservers or include the key as extra payload when sending her regular public key to Bob.

The *first signature* in the attack is by a key with a malicious user ID, crafted by the attacker, that injects a spoofed `VALIDSIG` status line. Because Enigmail processes this line twice using different parsers, some information needs to be duplicated to make the attack work. The *second signature* is a regular valid signature over the same message by the trusted attacker's key. This signature sets the global flag in Enigmail indicating that a signature is made by a trusted key, but otherwise it is completely ignored. Figure 13 shows which data in the attack is used by Enigmail. The fingerprint will be used to resolve further details such as the user ID of the (spoofed) signing key. The vulnerability is present in all versions of Enigmail until 2.0.7, affecting Thunderbird and Postbox. Our finding is documented as CVE-2018-12019.

```
[GNUPG:] NEWSIG
[GNUPG:] GOODSIG 8DA07D5E58B3A622 x 1527763815 x x x 1 10 x
      4F9F89F5505AC1D1A260631CDB1187B9DD5F693B VALIDSIG x x 0 4
      F9F89F5505AC1D1A260631CDB1187B9DD5F693B
[GNUPG:] TRUST_UNDEFINED
[GNUPG:] NEWSIG
[GNUPG:] GOODSIG 88B08D5A57B62140 <eve@evil.com>
[GNUPG:] TRUST_FULLY
```

Figure 13: The status line API as seen by Enigmail (abbreviated).

5.3 MIME Attack Class

On five PGP email clients, including popular products such as Thunderbird and Apple Mail, we could completely hide the original signed part (resulting in perfect forgeries) using multiple techniques, such as wrapping it into attacker-controlled HTML/CSS (M_1), referencing it as an "image" (M_2), or hiding it as an "attachment" (M_3). On another three clients we could only obfuscate the original signed message part (resulting in weak forgeries) by appending it and using a multitude of newlines to cover its existence (M_4). Our findings are documented as CVE-2017-17848, CVE-2018-15586, CVE-2018-15587, and CVE-2018-15588.

Our evaluation shows that S/MIME is less vulnerable to signature wrapping attacks by design; all but one tested email clients only show a valid signature verification if the whole email including all sub-parts is correctly signed (i.e., `Content-Type: multipart/signed` is the root element in the MIME tree). Unfortunately, it seems hard to get rid of partially signed emails or mark them as suspicious, as can be done for partially encrypted messages – a countermeasure Enigmail applied to the EFAIL attacks [9] – because in the PGP world, partially signed is quite common (e.g., mailing lists, forwards, etc.). There seems to be a different philosophy in the S/MIME world; for example, S/MIME forwarding

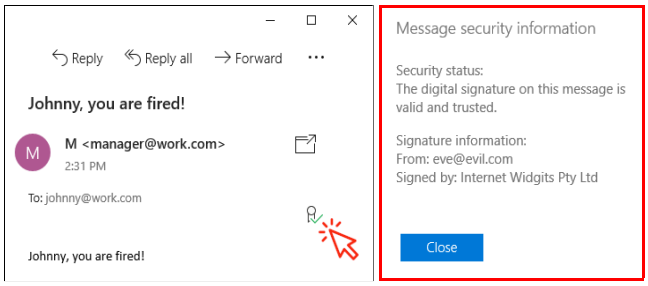


Figure 14: Partial forgery in the Windows 10 mail app. The email is actually signed by Eve which is only visible in the signature details.

intentionally breaks the signature because the forwarding entity should re-sign the message.

5.4 ID Class

Eleven PGP email clients and twelve S/MIME clients explicitly show the signer's identity (i.e., PGP user ID or Internet mail address in the signer's S/MIME certificate) when receiving a signed email. This can be considered safe because a user gets direct feedback regarding the *signed by whom?* question. The other clients do not show the signer's identity on the first level of the UI. Of those, two PGP email and three S/MIME mail clients, such as the Windows 10 mail app (see Figure 14), do not perform any correlation between the sender address and the signer's identity at all. They only show "signed" with no further information, making them easy targets for ID attacks (resulting in partial forgeries). The other seven PGP email clients and eight S/MIME clients compare the sender's address to the email address found in the public key or certificate matching the signature. This process is error-prone. For four PGP email clients, including GpgOL for Outlook, and eight S/MIME email clients, the correlation could be fully bypassed using the various techniques described in subsection 4.4. If Bob does not manually view the signature details, there is no indicator that the email was signed by Eve instead of Alice (resulting in partial forgery). For two of these clients (GpgOL for Outlook and Airmail) no signature details were available, resulting in perfect forgery.

5.5 UI Attack Class

Five tested PGP email clients and four S/MIME clients display the status of signatures within the email body, which is a UI component controlled by the attacker. This allowed us to create fake text or graphics implying a valid signature using HTML, CSS and inline images visually indistinguishable from a real signed message. Only further investigation of the email, such as viewing signature details, could reveal the attack (resulting in partial forgery). Three of these clients do not not even have an option for further signature details, resulting in perfect forgery. Spoofing signatures was especially easy for clients where the PGP or S/MIME plugin simply injects HTML code into the email body. We could just copy the original HTML snippet of a valid signature and re-send it within the body of our spoofed email.

Another seven PGP clients and nine S/MIME clients show the results of signature verification in, or very close to, the email body and could be attacked with limitations (causing weak forgeries). Some of these clients have additional indicators in other parts of the UI pointing out that the email is actually signed, but those indicators are missing in the case of spoofed signatures based on UI redressing (see Figure 15). Furthermore, in some of these clients the spoofed signature was not 100% visually indistinguishable.

6 Countermeasures

Similarly to other RFC documents, S/MIME [10] and OpenPGP [6] contain a section on *security considerations*. While these sections discuss cryptographic best practices (e.g., key sizes and cryptographic algorithms), they do not discuss how to design secure validation routines and interfaces. In this section, we discuss several possible countermeasures against the presented attacks in order to give guidance for implementing secure email clients.

6.1 CMS Attack Class

eContent Confusion (C_1) Both S/MIME signing variants are commonly used[13] and standard compliant clients are expected to support them. Thus, special care must be taken to only display the content which was subject to verification.

The relevant standards, RFC 5652 (CMS) and RFC 5751 (S/MIME), do not give any advice on how to handle the case were both variants are present. We recommend to display neither of them and show an error instead. In fact, Claws reports a "conflicting use".

[13] Outlook 2016 sends opaque signed messages by default and Thunderbird sends detached messages by default.

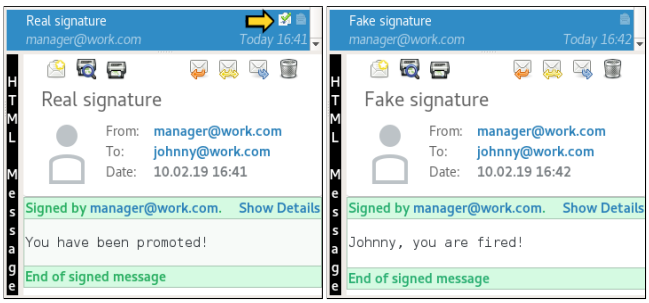


Figure 15: "Weak" forgery in KMail. The check mark UI indicatior in the upper right cannot be spoofed using simple UI redressing.

OS	Client	Plugin	GPG	MIME	ID	UI	Weaknesses
Windows	Thunderbird (52.5.2)	Enigmail (1.9.8)	●	●	–	○	G_1, G_2, M_2, M_3, U_1
	Outlook (16.0.4266)	GpgOL (2.0.1)	–	–	●	○	I_2, U_1
	The Bat! (8.2.0)	GnuPG (2.1.18)	–	○	○	–	M_1, I_1
	eM Client (7.1.31849)	native	–	–	–	◐	U_1
	Postbox (5.0.20)	Enigmail 1.2.3	●	–	–	–	G_1, G_2
Linux	KMail (5.2.3)	GPGME (1.2.0)	–	–	–	○	U_1
	Evolution (3.22.6)	GnuPG (2.1.18)	–	●	–	○	M_4, U_1
	Trojitá (0.7-278)	GPGME (1.2.0)	–	–	◐	●	I_2, I_3, U_1
	Claws (3.14.1)	GPG plugin (3.14.1)	–	○	–	–	M_1
	Mutt (1.7.2)	GPGME (1.2.0)	–	–	–	○	U_1
macOS	Apple Mail (11.2)	GPG Suite (2018.1)	●	●	–	○	G_1, M_1, M_2, M_3, U_1
	MailMate (1.10)	GPG Suite (2018.1)	–	●	○	●	M_1, M_2, M_3, I_2, U_1
	Airmail (3.5.3)	GPG-PGP (1.0-4)	–	●	●	–	M_3, I_2
Android	K-9 Mail (5.403)	OpenKeychain (5.2)	–	–	◐	–	I_2
	R2Mail2 (2.30)	native	–	–	◐	○	I_1, U_1
	MailDroid (4.81)	Flipdog (1.07)	–	○	–	◐	M_1, U_1
Web	Roundcube (1.3.4)	Enigma (git:48417c5)	–	–	–	●	U_1
	Horde/IMP (7.7.9)	GnuPG (2.1.18)	–	–	–	–	–
	Mailpile (1.0.0rc2)	GnuPG (2.1.18)	●	–	◐	–	G_1, I_1
	Mailfence (2.6.007)	OpenPGP.js (2.5.3)	–	–	–	–	–

● Indistinguishable signature on all UI levels (perfect forgery) ○ Signature can be spoofed with limitations (weak forgery)
◐ Indistinguishable signature on first UI level (partial forgery) – No vulnerabilities found

Table 2: Out of 20 tested email clients 14 were vulnerable to our OpenPGP signature spoofing attacks (perfect or partial forgery).

Disallow Multiple Signers (C_2) It is difficult to give good advice on the presentation of multiple signers, as different clients may implement different UI concepts. Furthermore, introducing UI elements might worsen the usability or introduce security problems on its own (e.g., UI redressing).

However, a simple solution is to not implement multiple signer support at all. Many up-to-date clients do not support multiple signers, e.g., Thunderbird and Outlook 2016. Additionally, we know of no client which is able to *produce* a message with multiple signers. Thus, it seems reasonable to us to not support this feature.

Error Out If No Signers (C_3) Messages with no signer should be treated as errorneous or as not signed. In either case there should be no UI element indicating a signed message. We recommend to not show the message and show an error instead. This is due to the possible application of signature stripping attacks as demonstrated by [11] and [9].

Redesign Trust Management and Workflow (C_4) Clients must validate the complete certification path and fail the signature verification on trust issues. Furthermore, certificates should be checked automatically. Clients must not accept self-signed certificates. If needed, a separate trust chain should be configured on the device or in the application.

6.2 GPG API Attack Class

GnuPG developers can improve the documentation and the API, but they have to consider backwards compatibility and extensibility. GnuPG must track attacker controlled data and always escape newlines and other special characters in all outputs. GnuPG should also validate the structure of OpenPGP messages and provide clear guidelines on how to achieve common tasks such as certificate pinning.

Frontend developers can harden the invocation of the backend (e.g. by using dedicated channels for log and status lines or adding `--no-verbose` to disable logging), their status line parsers (e.g., by anchoring all regular expressions), and the state machine aggregating the results (e.g., by keeping track of multiple signatures, as indicated by `NEWSIG`). However, applications that are too strict risk incompatibilities with future backend upgrades or unconventional user configurations.

The OpenPGP standard should be updated to provide a strict grammar for valid message composition, as the present flexibility (such as arbitrary nesting of encrypted, signed, and compressed messages) is unjustified in practice and puts the burden on implementations to define reasonable limits. Specifically, the OpenPGP standard should only allow one optional encryption layer, one optional compression layer, and one possibly signed literal data packet. More complex message composition (e.g., sign+encrypt+sign to allow for

OS	Client	Plugin	CMS	MIME	ID	UI	Weaknesses
Windows	Thunderbird (52.5.2)	native	●	–	○	–	C_1, I_3
	Outlook (16.0.4266)	native	○	–	–	○	C_3, U_1
	Win. 10 Mail (17.8730.21865)	native	–	–	◐	○	I_1, U_1
	Win. Live Mail (16.4.3528)	native	–	–	◐	○	I_2, U_1
	The Bat! (8.2.0)	native	–	–	◐	–	I_1
	eM Client (7.1.31849)	native	–	–	–	◐	U_1
	Postbox (5.0.20)	native	●	–	◐	–	C_1, I_3
Linux	KMail (5.2.3)	native	–	–	–	○	U_1
	Evolution (3.22.6)	native	◐	–	–	○	C_2, U_1
	Trojitá (0.7-278)	native	○	–	◐	●	C_4, I_2, I_3, U_1
	Claws (3.14.1)	GnuPG (gpgsm) (2.1.18)	○	–	–	–	C_4
	Mutt (1.7.2)	native	○	–	–	○	C_3, U_1
macOS	Apple Mail (11.2)	native	–	–	–	○	U_1
	MailMate (1.10)	native	●	●	○	○	$C_1, M_1, M_2, M_3, I_2, U_1$
	Airmail (3.5.3)	S/MIME (1.0-10)	–	–	●	–	I_2
Android	R2Mail2 (2.30)	native	–	–	◐	○	I_2, I_3, U_1
	MailDroid (4.81)	FlipDog (1.07)	◐	–	–	◐	C_3, C_4, U_1
	Nine (4.1.3a)	native	◐	–	◐	○	C_4, I_1, U_1
iOS	Mail App (12.01)	native	●	–	–	–	C_1
Web	Roundcube (1.3.4)	rc_smime (git:f294cde)	–	–	–	●	U_1
	Horde/IMP (6.2.21)	native	–	–	–	–	–
	Exchange/OWA (15.1.1034.32)	Control (4.0500.15.1)	–	–	–	○	U_1

●	Indistinguishable signature on all UI levels (perfect forgery)	○	Signature can be spoofed with limitations (weak forgery)
◐	Indistinguishable signature on first UI level (partial forgery)	–	No vulnerabilities found

Table 3: Out of 22 tested email clients 15 were vulnerable to our S/MIME signature spoofing attacks (perfect or partial forgery). Some clients with weak forgery, conflicting UI elements or unusual workflows are documented in more detail in the appendix.

early spam mitigation) may be desirable for certain applications in the future. These should then be covered in future standard revisions to ensure that they can be supported without introducing new risk factors. Until then, they can be supported either by nesting several distinct messages explicitly or as non-standard extensions that are disabled by default in compliant implementations.

6.3 MIME Attack Class

There are two approaches to counter signature spoofing attacks on partially signed messages which are hidden in the MIME tree. Either email clients should show exactly which part of the message was signed, for example, by using a green frame. However, note that this is hard to implement in a secure way because all edge cases and potential bypasses, such as internal `cid:` references, need to be considered, and it must be made sure that the frame cannot be drawn by the attacker using HTML/CSS. Or email clients should be conservative and only show a message as correctly signed if the whole message (i.e., the MIME root) was correctly signed. While this will break digital signatures in some mailing lists and in forwarded emails, such an *all-or-nothing* approach can be considered as more secure and is preferred by the authors of this paper. Furthermore, if the signature contains a timestamp, it should be shown to the user to draw suspicion on re-used and wrapped old signatures. For incoming new messages the timestamp could be compared to the current system time and an alert could be given if the signature contains a timestamp older than a certain threshold. This can also protect from attacks such as non-realtime surreptitious forwarding.

6.4 ID Attack Class

Approaches to encrypted and digitally signed email headers like Memory Hole [12] (a non-standard OpenPGP extension) or RFC 7508 (Securing Header Fields with S/MIME) aim to guarantee a cryptographic binding between the signature and the sender address. However, few email clients support these standards. Furthermore, clients supporting Memory Hole (Thunderbird/Enigmail, R2Mail2) still accept signed emails without header protection for backwards compatibility.

Even worse, Enigmail did not guarantee consistency between unprotected email headers and headers protected by Memory Hole in our tests. Clients remain vulnerable to ID

attacks unless further mitigations are applied. Hence, it can be considered a good practice to explicitly show the signer user IDs when displaying a PGP signed message. A comparison to the `FROM` or `SENDER` header fields may not be sufficient because—as our evaluation shows—that approach is error prone and hard to implement in a secure way.

6.5 UI Attack Class

The results of signature verification should not be shown in attacker-controlled parts of the UI, such as the message content itself, which may contain arbitrary graphics. In the context of webmail and native email clients using HTML5 for their UI, such as Electron[14] or XUL,[15] it must be ensured that there is a strict isolation (e.g., separate DOM) between the message content and the rest of the UI. Otherwise, it may be possible to influence the appearance of the UI, for example by injecting CSS properties into the message content. Following the example of browsers with regard to HTTPS, the trend is to avoid positive UI indicators and only show indicators if something is wrong. These systems aim to be secure by default. However, this is obviously infeasible for email signatures as long as most emails are unsigned.

7 Additional Findings

7.1 Crashes

We discovered multiple crashes during testing. For example, we found a nullpointer dereference in Mozilla's NSS library, which is also used in Evolution and Postbox. Although the security impact is rather low, sending a specifically crafted email does lead to a permanent denial of service, as email clients will cache this email and crash upon startup. We experienced a similar issue with iOS Mail, but did not evaluate the origin of the crash. Additionally, we observed crashes in MailMate, R2Mail2, Maildroid (Exception), Roundcube, and Windows 10 Mail.

7.2 Airmail Accepts Invalid PGP Signatures

We found that the Airmail GPG-PGP plugin does not properly validate OpenPGP signatures, accepting even invalid ones, irregardless of their status. This makes signature spoofing attacks trivial.

Also, Airmail does not correctly verify the validity of the signing key even for good signatures, allowing impersonation attacks by injecting public keys into the user's keyring with the email address that should be spoofed.

The vulnerability is present in all versions of Airmail GPG-PGP until "1.0 (9)". Our finding is documented as CVE-2019-8338.

[14] GitHub Inc., *Electron*, `https://electronjs.org/`

[15] Mozilla Foundation, *XUL*, `https://developer.mozilla.org/XUL`

7.3 OpenPGP Message Composition Attacks

OpenPGP messages and keys are sequences of (possibly nested) packets (see Fig. 16). This structure is under control of the attacker, so it must be validated by the implementation. According to the OpenPGP standard, any arbitrary nesting of encryption, compression, and signatures is allowed without restrictions. This flexibility is unjustified, and seems to be an oversight in the specification, as only a small number of combinations are actually meaningful in practice. It also opens PGP up to vulnerabilities such as decompression attacks [13]. In practice, implementations must enforce additional limits; for example, GnuPG allows up to 32 levels of nesting.

```
[SessionKeyPacket]
[EncryptedDataPacket
 [CompressedDataPacket
  [OnePassSignaturePacket]
  [LiteralDataPacket]
  [SignaturePacket]
 ]
]
```

(a) Example structure of an OpenPGP message that is signed with one signing key, compressed, and encrypted to one recipient key.

```
message :- encrypted | signed | compressed | literal.
encrypted :- SessionKeyPacket*, EncryptedDataPacket(message).
signed :- OnePassSignaturePacket, message, SignaturePacket.
compressed :- CompressedDataPacket(message).
literal :- LiteralDataPacket.
```

(b) Grammar for OpenPGP message composition from RFC 4880 (simplified excerpt). This grammar does not include rules for compatibility with older PGP versions.

Figure 16: Valid OpenPGP message and its grammar specification.

Status lines are emitted by GnuPG as packets are processed recursively. However, the status lines do not represent packet boundaries nor the depth of nesting. As a consequence, the status interface is a flat projection of the nested structure, and some information is lost in the process.

Message Composition Attacks GnuPG outputs status lines as a side-effect of recursive processing of packets in OpenPGP messages. This has led to signature spoofing attacks in the past, where an attacker can prepend or append additional unsigned plaintext to a message [14]. We verified that current versions of GnuPG handle this correctly, and could not find any similar issues for signature verification.

Encryption Spoofing Some attacks to spoof signature verification can also be used to spoof decryption results, causing the email client to indicate an encrypted message where in fact the plaintext was transmitted in the clear. Although by itself this is not a security violation, it is concerning and might be a precursor or building stone for other attacks.

Besides the obvious adaptation of our UI redressing and status line injection attacks, we found a flaw in the message composition verification of GnuPG. Since 2006 [14], GnuPG only allows at most one plaintext (i.e., one Literal Data Packet) in a message. However, GnuPG does not verify that the plaintext of an encrypted (non-conforming) message is actually contained within the encrypted part of the message. By replacing the plaintext in an Encrypted Data Packet with a dummy packet ignored by GnuPG (the OpenPGP standard makes a provision for private/experimental packet types), and prepending or appending the (unencrypted) Literal Data Packet, we can cause GnuPG to output the same status lines as for a properly encrypted message, excluding the order. The following output shows the result for a properly encrypted message (differences in red and bold):

```
[GNUPG:] BEGIN_DECRYPTION
[GNUPG:] PLAINTEXT 62 0
[GNUPG:] DECRYPTION_OKAY
[GNUPG:] END_DECRYPTION
```

The next output shows the result for an empty Encrypted Data Packet, followed by a Literal Data Packet in the clear:

```
[GNUPG:] BEGIN_DECRYPTION
[GNUPG:] DECRYPTION_OKAY
[GNUPG:] END_DECRYPTION
[GNUPG:] PLAINTEXT 62 0
```

Both messages are displayed identically (resulting in a perfect forgery) in Thunderbird, Evolution, Mutt, and Outlook, revealing the flexibility of the PGP message format, GnuPG's parser, and the GnuPG status line parsers in email client applications.

7.4 Short Key PGP IDs

Short key IDs of 32 bit (the least significant 4 bytes of the fingerprint) were used in the PGP user interface, on business cards, by key servers, and other PGP-related utilities in the past until pre-image collisions were demonstrated to be efficient in practice [15]. Unfortunately, the Horde/IMP email client still uses short key IDs internally to identify public keys and automatically downloads them from key servers to cache them in internal data structures. Our attempts to exploit these collisions for ID attacks were inconsistent due to caching effects, which is why we did not include these attacks in the evaluation. Horde/IMP should mitigate these attacks by using full-length fingerprints to identify PGP keys.

7.5 GPG API Attacks Beyond Email

Based on source code searches on GitHub[16] and Debian,[17] we looked for software applications or libraries other than email clients which might be susceptible to API signature spoofing attacks. Candidates were programs that invoke GnuPG with `--status-fd 2`, thereby conflating the logging messages with the status line API, and programs that do not correctly anchor regular expressions for status lines (involving `[GNUPG:]`). We identified three broad classes of programs using the GnuPG status line API: (1) Wrapper libraries that provide an abstraction layer to GnuPG, usually for particular programming languages. (2) Applications using certificate pinning to verify the integrity of files when stored under external control (cloud storage), including version control systems like Git. (3) Package managers that use GnuPG for integrity protection of software packages.

We found vulnerable software in all three categories, but due to the large number of libraries and applications using GnuPG, we could not yet perform an extensive review. Also, we found that the available code search engines are not a good match for the task of identifying applications calling an external application through a shell interface. We therefore fear that there may still be a significant number of vulnerable applications using GnuPG out there.

Python-GnuPG Python-gnupg[18] is a library interface to GnuPG for the Python language. It uses the status line interface and conflates it with the logging messages, making 717 applications using it[19] potentially susceptible to the embedded filename injection attack described above, depending on how and in which context they use the Python library.

Bitcoin Source Code Repository Integrity The Bitcoin project uses the Git version control system, which supports signatures on individual software patches to verify the integrity of the whole repository as it changes over time. A shell script using GnuPG to verify the integrity of all commits is included in the distribution.[20] This script uses the status line API and does not anchor the regular expressions, making it susceptible to the malicious user ID injection attack described above. An attacker who can inject arbitrary keys into the keyring of the user can simply re-sign modified source code commits and thus bypass the verification script.

The Bitcoin source code is frequently used as the basis for other crypto-currencies,[21] and thus this error may propagate to many other similar projects such as Litecoin.

Signature Bypass in Simple Password Store Pass[22], a popular password manager for UNIX, uses the GnuPG status line API to encrypt password files and digitally sign configuration files and extension plugins. It does not anchor the regular expressions, making it susceptible to the malicious

[16] *GitHub Code Search*, `https://github.com/search`

[17] *Debian Code Search*, `https://codesearch.debian.net/`

[18] V. Sajip, *python-gnupg* – A Python wrapper for GnuPG, `https://pythonhosted.org/python-gnupg/`

[19] According to *libraries.io*, https://libraries.io/pypi/python-gnupg/usage

[20] *Bitcoin Source Code*, `https://github.com/bitcoin/bitcoin/blob/master/contrib/verify-commits/gpg.sh`

[21] According to *GitHub* the Bitcoin code has 21379 forks as of Nov. 2018

[22] J. A. Donenfeld, *pass*, `https://www.passwordstore.org/`

user ID injection attack described above. If `pass` is used to synchronize passwords over untrusted cloud storage, an attacker with control over that storage can add their public key to the configuration, which causes `pass` to transparently re-encrypt the passwords to the attackers key (in addition to the user's key) over time. Also, if extension plugins are enabled, the attacker can get remote code execution. This finding is documented as CVE-2018-12356.

Yarn Package Manager Yarn is a package manager by Facebook for the JavaScript runtime Node.js. The Yarn installer script[23] primarily relies on TLS to secure the integrity of the installation package itself. However, it also attempts to use GnuPG signature verification, presumably to secure the integrity of the installer from the build server to the download server, which can be different from the server that hosts the installer script (e.g., for nightly builds).

Unfortunately, Yarn fails to do any form of certificate pinning or trust management, and will accept any valid signature by any key in the local keyring, even by untrusted keys. If the attacker can inject a public key into the user's keyring, and perform a MiTM attack against one of Yarn's download servers, the attacker can replace the installation package with one containing a backdoor, thereby gaining remote code execution on the user's machine. This finding is documented as CVE-2018-12556.

7.6 Unsuccessful Cryptograpic Attacks

We analyzed 19 of 20 OpenPGP email clients from Table 2 (all but Airmail, which we could not test, see subsection 7.2) and all 22 email clients supporting S/MIME signatures from Table 3 if they are vulnerable to well-known attacks on the PKCS#1v1.5 signature scheme for RSA with exponent $e = 3$. Specifically, we checked for mistakes in the handling of the padding [16] and ASN.1 structures [17]. All tested clients resisted our attempts.

8 Related Work

The OpenPGP standard [6] only defines how to sign the email body. Critical headers such as `SUBJECT` and `FROM` are not signed if no further extensions, such as Memory Hole [12], Secure Header Fields [18] or XML-based techniques as described in [19], are applied. Levi et al. [20] developed a GUI for users to better understand the limitations of S/MIME digital signatures for emails, i.e., showing which parts of the email are actually signed by whom. Usability papers such as [21] discuss the difficulties that inexperienced users have to manually verify the validity of a signature and understand the different PGP trust levels.

[23] *Yarn installer script*, `https://yarnpkg.com/install.sh`

It is well known that messages signed by a certain entity can be reused in another context. For example, a malicious former employer in possession of a signed "I quit my job" message by Alice can simply re-send this message to Alice's new employer. Such attacks have been referred to as "surreptitious forwarding" in 2001 by Davis [22] who also showed how to strip signatures in various encryption schemes. Gillmor [23] touches upon the problems of partially-signed Inline PGP messages as well as non-signed attachments and the UI challenge such constructs present for MUAs. Furthermore, he demonstrates a message tampering attack through header substitution: by changing the encoding, a signed message with the content *€13/week* can be presented as *£13/week*, while the signature remains valid.

Recently, Poddebniak et al. [9] described two attacks to directly exfiltrate the plaintext of OpenPGP encrypted messages. One works by wrapping the ciphertexts into attacker-controlled MIME parts. This is related to our MIME wrapping attacks on signatures, as both techniques exploit missing isolation between multiple MIME parts. However, we present attacks which do not require mail clients to put trusted and untrusted input into the same DOM, using advanced approaches such as `cid:` URI scheme references.

GnuPG had signature spoofing bugs in the past. In 2006, it was shown that arbitrary unsigned data can be injected into signed messages [14]. Until 2007, GnuPG returned a valid signature status for a message with arbitrary text prepended or appended to an Inline PGP signature, allowing an attacker to forge the contents of a message without detection [24].

In 2014, Klafter et al. [15] showed practical collisions in 32-bit PGP key IDs, named the "Evil 32" attack, which can be dangerous in case a client requests a PGP key from a keyserver using the 32-bit ID. Collisions with 64-bit IDs are also feasible but require more computing power, therefore only full 160-bit fingerprints should be used.

"Mailsploit" [25] enables attackers to send spoofed emails, even if additional protection mechanism like DKIM are applied, by abusing techniques to encode non-ASCII chars inside email headers as described in RFC 1342. In 2017, Ribeiro [26] demonstrated that CSS rules included in HTML emails can be loaded from a remote source, leading to changes in the appearance of—potentially signed—emails after delivery.

Our partial (◐) and weak (○) forgery attacks can potentially be detected by carefully inspecting the GUI or manually clicking to receive more signature details. A user study analyzing user behavior would be necessary to reveal the real impact of such inconsistencies. Lausch et al. [27] reviewed existing cryptographic indicators used in email clients and performed a usability study. With their 164 "privacy-aware" participants they were able to select the most important indicators. They argue that further research with participants with general skills should be performed in the future work. It would be intersting to extend this user study with specific

corner cases from our paper. Similar studies were performed to analyze the usage of TLS indicators in web browsers. Schechter et al. performed laboratory experiments where they analyzed user behavior in respect to different browser warnings [28]. For example, they found out that all 63 tested users used their banking credentials on banking websites delivered over pure HTTP. Sunshine et al. [29] performed a study on SSL warning effectiveness. Their conclusion is that blocking unsafe connections and minimizing TLS warnings would improve the warning effectiveness and user security. Felt et al. studied TLS indicators in different browsers, including Chrome, Firefox, Edge, and Safari [30]. They performed a user study with 1329 participants to select the most appropriate indicators reporting positive as well as negative TLS status. The selected indicators have then been adopted by Google Chrome. Other researchers concentrated on special cases like the evaluation of Extended Validation certificates [31] or TLS indicators in mobile browsers [32].

9 Future Work

User Study On most clients the evaluation results were obvious. In the case of perfect or partial forgery, little to no discussion is needed because it should be clear that a user can not distinguish between a valid and a spoofed signature. However, some clients displayed conflicting information, for example, an erroneous seal *and* a success dialog. Other clients expect a more elaborate workflow from the user, such as clicking through the UI.

We currently classify a weak forgery as "not vulnerable" because the user has at least a chance to detect it and we did not measure if users actually detected it. A user study could clarify whether our weak forgery findings (e.g., conflicting security indicators) would convince email users, and answer the following research questions: Do users pay attention to email security indicators in general? Do users examine digital signature details (in particular for partial forgeries)? How do users react once they detect broken signatures?

We believe that such a study would help to understand the current email security ecosystem and our paper lays the foundations for such a study.

S/MIME Signatures in AS2 *Applicability Statement 2 (AS2)* as described in RFC 4130 is an industry standard for secure Electronic Data Interchange (EDI) over the Internet. It relies on HTTP(S) for data transport and S/MIME to guarantee the authenticity and integrity of exchanged messages. As critical sectors such as the energy industry heavily rely on AS2 for their business processes, it would be interesting to evaluate if our attacks (e.g., in the CMS class) can be applied to AS2. Unfortunately, we did not have the opportunity to test commercial AS2 gateways yet.

Email Security Fuzzing As described in subsection 7.1, our tests with different attack vectors led to unintentional crashes in several email applications. More rigorous testing and systematic fuzzing with MIME, S/MIME and PGP structures could uncover further vulnerabilities.

10 Conclusion

We demonstrated practical email signature spoofing attacks against many OpenPGP and S/MIME capable email clients and libraries. Our results show that email signature checking and correctly communicating the result to the user is surprisingly hard and currently most clients do not withstand a rigorous security analysis. While none of the attacks directly target the OpenPGP or S/MIME standards, or the underlying cryptographic primitives, they raise concerns about the practical security of email applications and show that when dealing with applied cryptography, even if the cryptosystem itself is considered secure, the overall system needs to be looked at and carefully analyzed for vulnerabilities. Implementing countermeasures against these vulnerabilities is very challenging and, thus, we recommend that OpenPGP, MIME, and S/MIME offer more concrete implementation advices and security best practices for developing secure applications in the future.

Acknowledgements

The authors would like to thank Kai Michaelis and Benny Kjær Nielsen for insightful discussions about GnuPG and its secure integration into the email ecosystem, and our anonymous reviewers for many insightful comments.

Juraj Somorovsky was supported through the Horizon 2020 program under project number 700542 (FutureTrust). Jens Müller was supported by the research training group 'Human Centered System Security' sponsored by the state of North-Rhine Westfalia.

References

[1] J. Postel, "Simple Mail Transfer Protocol," August 1982. RFC0821.

[2] D. Crocker, "Standard for the format of ARPA internet text messages," August 1982. RFC0822.

[3] H. Hu and G. Wang, "End-to-end measurements of email spoofing attacks," in *27th USENIX Security Symposium (USENIX Security 18)*, (Baltimore, MD), pp. 1095–1112, USENIX Association, 2018.

[4] J. Callas, L. Donnerhacke, H. Finney, and R. Thayer, "OpenPGP message format," November 1998. RFC2440.

[5] B. Ramsdell, "S/MIME version 3 message specification," June 1999. RFC2633.

[6] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer, "OpenPGP message format," November 2007. RFC4880.

[7] R. Housley, "Cryptographic Message Syntax (CMS)," September 2009. RFC5652.

[8] P. Resnick, "Internet message format," October 2008. RFC5322.

[9] D. Poddebniak, C. Dresen, J. Müller, F. Ising, S. Schinzel, S. Friedberger, J. Somorovsky, and J. Schwenk, "Efail: Breaking S/MIME and OpenPGP email encryption using exfiltration channels," in *27th USENIX Security Symposium (USENIX Security 18)*, (Baltimore, MD), pp. 549–566, USENIX Association, 2018.

[10] B. Ramsdell and S. Turner, "Secure/Multipurpose Internet Mail Extensions (S/MIME) version 3.2 message specification," January 2010. RFC5751.

[11] F. Strenzke, "Improved message takeover attacks against S/MIME," Feb. 2016. `https://cryptosource.de/posts/smime_mta_improved_en.html`.

[12] D. K. Gillmor, "Memory Hole spec and documentation." `https://github.com/autocrypt/memoryhole`, 2014.

[13] "CVE-2013-4402." Available from MITRE, 2013.

[14] "CVE-2006-0049." Available from MITRE, 2006.

[15] R. Klafter and E. Swanson, "Evil 32: Check your GPG fingerprints." `https://evil32.com/`, 2014.

[16] D. Bleichenbacher, "Forging some RSA signatures with pencil and paper." Presentation in the rump Session CRYPTO 2006, Aug. 2006.

[17] A. Furtak, Y. Bulygin, O. Bazhaniuk, J. Loucaides, A. Matrosov, and M. Gorobets, "BERserk: New RSA signature forgery attack." Presentation at Ekoparty 10, 2014.

[18] B. Ramsdell, "S/MIME version 3 certificate handling," June 1999. RFC2632.

[19] L. Liao, *Secure Email Communication with XML-based Technologies*. Europ. Univ.-Verlag, 2009.

[20] A. Levi and C. B. Güder, "Understanding the limitations of S/MIME digital signatures for e-mails: A GUI based approach," *computers & security*, vol. 28, no. 3-4, pp. 105–120, 2009.

[21] A. Whitten and J. D. Tygar, "Why Johnny can't encrypt: A usability evaluation of PGP 5.0," in *Proceedings of the 8th Conference on USENIX Security Symposium - Volume 8*, SSYM'99, (Berkeley, CA, USA), pp. 14–14, USENIX Association, 1999.

[22] D. Davis, "Defective sign & encrypt in S/MIME, PKCS#7, MOSS, PEM, PGP, and XML," in *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, (Berkeley, CA, USA), pp. 65–78, USENIX Association, 2001.

[23] D. K. Gillmor, "Inline PGP signatures considered harmful." `https://dkg.fifthhorseman.net/notes/inline-pgp-harmful/`, 2014.

[24] "CVE-2007-1263." Available from MITRE, 2007.

[25] S. Haddouche, "Mailsploit." `https://mailsploit.com/`, 2017.

[26] F. Ribeiro, "The ROPEMAKER email exploit," 2017.

[27] J. Lausch, O. Wiese, and V. Roth, "What is a secure email?," in *European Workshop on Usable Security (EuroUSEC)*, 2017.

[28] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer, "The emperor's new security indicators," in *2007 IEEE Symposium on Security and Privacy (SP '07)*, pp. 51–65, May 2007.

[29] J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor, "Crying wolf: An empirical study of SSL warning effectiveness," in *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, (Berkeley, CA, USA), pp. 399–416, USENIX Association, 2009.

[30] A. P. Felt, R. W. Reeder, A. Ainslie, H. Harris, M. Walker, C. Thompson, M. E. Acer, E. Morant, and S. Consolvo, "Rethinking connection security indicators," in *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*, (Denver, CO), pp. 1–14, USENIX Association, 2016.

[31] R. Biddle, P. C. van Oorschot, A. S. Patrick, J. Sobey, and T. Whalen, "Browser interfaces and extended validation SSL certificates: An empirical study," in *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, CCSW '09, (New York, NY, USA), pp. 19–30, ACM, 2009.

[32] C. Amrutkar, P. Traynor, and P. C. van Oorschot, "Measuring SSL indicators on mobile browsers: Extended life, or end of the road?," in *Information Security* (D. Gollmann and F. C. Freiling, eds.), (Berlin, Heidelberg), pp. 86–103, Springer Berlin Heidelberg, 2012.

Scalable Scanning and Automatic Classification of TLS Padding Oracle Vulnerabilities

Robert Merget[1], Juraj Somorovsky[1], Nimrod Aviram[2], Craig Young[3], Janis Fliegenschmidt[1], Jörg Schwenk[1], and Yuval Shavitt[2]

[1]Ruhr University Bochum
[2]Department of Electrical Engineering, Tel Aviv University
[3]Tripwire VERT

Abstract

The TLS protocol provides encryption, data integrity, and authentication on the modern Internet. Despite the protocol's importance, currently-deployed TLS versions use obsolete cryptographic algorithms which have been broken using various attacks. One prominent class of such attacks is CBC padding oracle attacks. These attacks allow an adversary to decrypt TLS traffic by observing different server behaviors which depend on the validity of CBC padding.

We present the first large-scale scan for CBC padding oracle vulnerabilities in TLS implementations on the modern Internet. Our scan revealed vulnerabilities in 1.83% of the Alexa Top Million websites, detecting nearly 100 different vulnerabilities. Our scanner observes subtle differences in server behavior, such as responding with different TLS alerts, or with different TCP header flags.

We used a novel scanning methodology consisting of three steps. First, we created a large set of probes that detect vulnerabilities at a considerable scanning cost. We then reduced the number of probes using a preliminary scan, such that a smaller set of probes has the same detection rate but is small enough to be used in large-scale scans. Finally, we used the reduced set to scan at scale, and clustered our findings with a novel approach using graph drawing algorithms.

Contrary to common wisdom, exploiting CBC padding oracles does not necessarily require performing precise timing measurements. We detected vulnerabilities that can be exploited simply by observing the content of different server responses. These vulnerabilities pose a significantly larger threat in practice than previously assumed.

1 Introduction

In 2002, Vaudenay presented an attack which targets messages encrypted with the Cipher Block Chaining (CBC) mode of operation [39]. The attack exploits the malleability of the CBC mode, which allows altering the ciphertext such that specific cleartext bits are flipped, without knowledge of the encryption key. The attack requires a server that decrypts a message and responds with 1 or 0 based on the message validity. This behavior essentially provides the attacker with a cryptographic oracle which can be used to mount an adaptive chosen-ciphertext attack. The attacker exploits this behavior to decrypt messages by executing adaptive queries.Vaudenay exploited a specific form of vulnerable behavior, where implementations validate the CBC padding structure and respond with 1 or 0 accordingly.

This class of attacks has been termed *padding oracle attacks*. Different forms of padding oracle attacks were demonstrated to break cryptographic hardware [6], XML Encryption [23], or web technologies like Java Server Faces [33] and ASP.NET web applications [15]. Rizzo and Duong used a padding oracle attack to steal secrets and forge authentication tokens, gaining access to sensitive data [15]. In all of these works, the attacker was able to use a direct side channel – different error messages – to instantiate a padding oracle and decrypt confidential data.

Transport Layer Security (TLS) employs CBC mode in a MAC-then-Pad-then-Encrypt scheme which makes it potentially vulnerable to these attacks. Indeed, different types of CBC padding oracles have been used to break confidentiality TLS connections [39, 4, 3, 20]. All these attacks require the attacker to perform precise timing measurements. This requirement stems from the properties of the TLS protocol; after establishing a TLS connection, all TLS error messages are sent encrypted and are of the same length. Therefore, even if an attacker is able to cause the server to send different error messages, the attacker is generally unable to distinguish between the different encrypted responses.

Since most previous analyses have only analyzed padding oracle attacks based on timing side channels, they required testing an implementation in a local environment. These evaluations uncovered many new vulnerabilities [4, 3, 20]. However, implementing a proper countermeasure to these vulnerabilities is very challenging and requires complex constant-time implementations. It is not surprising that the implementation of such countermeasures could introduce

new attacks. For example, in an attempt to fix the Lucky 13 padding oracle, the OpenSSL cryptographic library introduced a different vulnerability where OpenSSL responded with different TLS alert messages [37]. Analysis of implementations in lab settings therefore requires laborious testing for each new version of different implementations. This is obviously unrealistic, and therefore this type of analysis is performed sporadically.

Given the complexity of constant-time TLS padding verification, we expect that vulnerabilities similar to the one introduced by OpenSSL [37] could have been introduced in other implementations as well. Therefore, this work moves away from the above method of lab analyses and evaluates CBC padding oracles using large-scale Internet scans. We attempt to answer the two following questions: *How prevalent are padding oracle vulnerabilities? Are these attacks only exploitable by using timing side-channels?*

Contributions. In our work, we employ a novel scanning methodology that is capable of scanning for TLS CBC padding oracles at scale. We use this methodology to find new padding oracle vulnerabilities and perform responsible disclosures. We identify nearly 100 different padding oracles. We show that some of them can be exploitable without subtle timing side channels and thus pose a significantly larger threat in practice compared to most recently-discovered padding oracles.

New large-scale scanning methodology. Scanning at scale for padding oracles is challenging. Such scans detect vulnerabilities by sending different malformed inputs and observing server behavior. As shown by Böck et al. [9], in some cases these inputs only trigger vulnerabilities when using specific TLS versions or cipher suites. Scanning with all possible combinations of protocol versions, cipher suites and malformed inputs is not feasible since it would require an enormous number of connections to each scanned host.

We overcome this limitation by carefully selecting a set of probes, which allows for effective scans at scale. We systematically analyzed padding oracles previously described in the literature [39, 4, 3, 20, 37, 27, 25, 10, 29, 28]. We then carefully selected 25 inputs exhibiting padding oracle malformities, which we refer to as *malformed records*. These TLS records exhibit different combinations of valid and invalid padding and MAC, and are generated using the TLS-Attacker framework [37].

Even with only 25 malformed records, scanning with every combination of malformed record, TLS version and cipher suite would be impractical. We refer to these combinations as *test vectors*. We performed a preliminary scan on 50,000 random TLS hosts with all test vectors. We then reduced our test vector set, such that all vulnerabilities detected in the preliminary scan are still triggered by the reduced set. We were able to scan the Alexa Top 1 Million websites with this reduced test vector set within three days. Our scanner observes different server responses, not only in the TLS layer, but also in the TCP layer, similar to [9]. Our results indicate that about 1.83% of TLS servers are vulnerable to CBC padding oracle attacks.

Minimizing false positives. When a host first displays vulnerable behavior, we rescan it to make sure the behavior is not a scanning artifact. We only consider a host to be vulnerable if it responds identically in three separate scans to each of our test vectors. It is unlikely that hosts will be mislabeled as vulnerable under this criterion. We therefore believe our statistics for vulnerability are a conservative lower estimate.

Nearly 100 different padding oracle vulnerabilities. The detected vulnerabilities have to be clustered in order to notify different vendors. Until now, this was done manually [9]. To achieve this automatically, we re-scan vulnerable hosts against a larger set of test vectors. We refer to the set of the host responses to all test vectors as the host's *response map*. This response map is essentially a fingerprint of the host's vulnerability. We then cluster the scanned hosts according to their response maps. This process identified 93 different response maps, i.e., 93 different vulnerabilities. These vulnerabilities include different behaviors, ranging from typical padding oracles with different TLS alert messages [39], to TCP connection timeouts triggered by specific invalid MAC bytes, or closed connections observed when using invalid padding values.

We treat distinct response maps as distinct vulnerabilities. We argue that this is the natural way to count vulnerabilities since it captures the case of the same vulnerability occurring in similar, yet different implementations. Consider two hosts that respond identically to all test vectors. These hosts likely share an identical or very similar part of the implementation that causes the vulnerability to manifest with identical response maps. However, they do not necessarily share the exact same code. They may use different versions of the same TLS library, or two different libraries with a shared component.

Effective clustering of vulnerable hosts. Before we responsibly disclosed our findings to the affected parties, we grouped the vulnerable hosts by their response maps. To further refine our grouped servers, we used a novel approach based on a two-dimensional force-directed graph drawing ForceAtlas2 algorithm [21]. This algorithm allowed us to create a graph of vulnerable server hosts and thus, efficiently handle our responsible disclosure process.

New vulnerabilities that are realistic to exploit. For padding vulnerabilities to be exploitable, the attacker needs

to distinguish between different responses to correct and incorrect padding. This is usually not the case in TLS: Even if a server sends two different alert messages, the messages are encrypted, and the attacker cannot observe the difference. For this reason, most previous padding oracle attacks against TLS relied on timing measurements to distinguish between different error cases [4, 3, 20].

However, we show that many TLS implementations exhibit *observable* differences between correct and incorrect padding. For example, a server may gracefully close the TCP connection in one error case and ungracefully close it in a different case. Similarly, some servers send a different number of alert messages depending on specific padding errors. Both behaviors are easily observable.

Responsible disclosure and ethical considerations. In collaboration with affected website owners, we responsibly disclosed our findings to several vulnerable vendors. As a result of a successful attack, the attacker is able to decrypt secret values repeatedly transmitted in the TLS connection. By performing our scans, we were not able to reconstruct server private keys or other confidential data. We performed our scans with dummy data and never attempted to decrypt real user traffic.

We responsibly disclosed our findings among others to the following vendors and affected parties: IBM, Amazon, Slack, Cisco, Citrix, Oracle, Heroku, Netflix, Sonicwall, Venmo and Vine.

2 Background

The TLS protocol provides confidentiality, integrity, and authentication on the modern Internet. The latest version of the protocol is TLS 1.3 [31]. This version is gradually being deployed as of this writing. Until TLS 1.3 is fully deployed, the latest version in widespread use is TLS 1.2 [14]. Modern clients and servers typically also support two previous versions, TLS 1.0 and 1.1 [12, 13]. In the rest of the paper, we discuss only versions 1.0 to 1.2, which are commonly used today and share a similar structure.

The TLS protocol consists of two phases. In the first phase, called the *handshake*, the client and server choose the cryptographic algorithms that will be used for the session and establish session keys. In the second phase, the peers can securely send and receive application data, which is encrypted and authenticated using the keys and algorithms established in the previous phase.

The aforementioned choice of cryptographic algorithms is called a TLS *cipher suite* [14]. More precisely, a cipher suite is a concrete selection of algorithms for all of the required cryptographic tasks. Cipher suites are named by concatenating their choices for these algorithms. For example, the cipher suite `TLS_RSA_WITH_AES_128_CBC_SHA` uses RSA public-key encryption in order to establish a shared session key in the first phase, and also uses symmetric AES-CBC encryption with a 128-bit key and SHA-1-based HMACs in order to encrypt and authenticate data in the second phase.

2.1 The TLS Handshake

The client initiates the TLS handshake with a `ClientHello` message. This message advertises the TLS versions and cipher suites supported by the client. The server then responds with a `ServerHello` message specifying the selected cipher suite. It also sends its certificate in the `Certificate` message and indicates the end of transmission with the `ServerHelloDone` message. The client then generates a secret value called the `premaster secret`, encrypts it under the server's RSA key, and sends the encrypted ciphertext in a `ClientKeyExchange` message. Having shared knowledge of the premaster secret, both parties now derive symmetric encryption and MAC keys to be used in the session, based on the premaster secret. Finally, both parties send the `ChangeCipherSpec` and `Finished` messages. The `ChangeCipherSpec` message notifies the receiving peer that subsequent messages will be encrypted and authenticated under the session keys, and using the symmetric encryption and HMAC algorithms specified in the cipher suite. The `Finished` message contains an HMAC computed over all the previous handshake messages based on a key derived from the premaster secret. As this message is sent after the `ChangeCipherSpec` message, it is the first message in the session which is encrypted and authenticated using symmetric encryption and MAC. If the `Finished` message correctly decrypts and verifies on both sides, both parties can now securely exchange application data.

2.2 CBC Mode

There are many possible encryption algorithms in TLS, but we focus on the CBC encryption mode in this work. In CBC mode, each plaintext block is XOR'ed to the previous ciphertext block before being encrypted by the block cipher. Formally, if we denote plaintext blocks by $p_i, i = 0, \ldots$, ciphertext blocks by c_i and the encryption with a block cipher under key k as $Enc_k(\cdot)$, then $c_i = Enc_k(p_i \oplus c_{i-1}), i = 1, \ldots$. The above holds for all blocks except the first one, where there is no previous ciphertext block – instead, that block is XOR'ed with an initialization vector (IV) before encryption: $c_0 = Enc_k(p_0 \oplus IV)$.

CBC mode malleability. The CBC mode allows an attacker to perform meaningful plaintext modifications without knowing the symmetric key. Concretely, assume the attacker knows some block of the original plaintext p_i, and wants to alter the ciphertext such that block i instead decrypts to p'_i. The attacker can change the previous ciphertext block c_{i-1} to $c'_{i-1} = c_{i-1} \oplus p_i \oplus p'_i$. This comes at the cost of

corrupting the previous block, which now decrypts to some value that the attacker, in general, cannot predict.

Furthermore, the attacker can change the order of blocks while using this technique. If the attacker knows the plaintext block p_i and replaces ciphertext block c_j with c_i, then block j will now decrypt to $p'_j = p_i \oplus c_{i-1} \oplus c_{j-1}$.

This "malleability" property of CBC mode has been used in many cryptographic attacks, and is also a cornerstone of the attacks presented here.

2.3 TLS Record Layer

The TLS record layer encapsulates protocol messages. In essence, the record layer wraps the protocol message with a header containing the message length, message type, and protocol version. Once `ChangeCipherSpec` messages are exchanged, subsequent TLS records will encapsulate messages which are encrypted.

In our work, we focus on cipher suites using the CBC mode of operation. These cipher suites use a Message Authentication Code (MAC) to protect the authenticity of TLS records and encrypt application data using a block cipher in CBC mode (e.g., AES or 3DES). The TLS specification prescribes the *MAC-then-Pad-then-Encrypt* mechanism [14]. The encryptor first computes a MAC over the plaintext, concatenates the MAC to the plaintext, pads the message such that its length is a multiple of the block length, and finally encrypts the MAC'ed and padded plaintext using a block cipher in CBC mode.

TLS specifies the exact value of the padding bytes. The last byte of the padded plaintext specifies how many padding bytes are used, excluding that last byte. The value of the rest of the padding bytes is identical to the value of the last byte. For example, if 4 padding bytes are used including the last byte, then the value of all four bytes will be `0x03`.

To demonstrate the full process, if the encryptor encrypts five bytes of data with the `TLS_RSA_WITH_AES_128_CBC_SHA` cipher suite, he uses HMAC-SHA (whose output is 20 bytes long) and AES-CBC. After applying HMAC-SHA to the original plaintext, the concatenation is 25 bytes in length, which fits into two AES 16-byte blocks. The encryptor will typically select the minimum viable amount of padding, which would be 7 bytes in this case. The first block contains the data and the first 11 HMAC bytes. The second block contains the remaining 9 HMAC bytes and 7 bytes of padding `0x06`, see Figure 1. Note that the encryptor can also choose longer padding and append 23, 39, ...or 247 padding bytes (while setting the value of the padding bytes accordingly).

3 A Brief History of Padding Oracle Attacks

One of the main design failures in SSLv3 and TLS is the specification of the *MAC-then-Pad-then-Encrypt* scheme in CBC cipher suites. This scheme was responsible for a series of attacks on TLS implementations named padding oracle attacks. Even though the countermeasures are explicitly summarized in the TLS specification [14, Section 6.2.3.2], their correct implementation is challenging.[1]

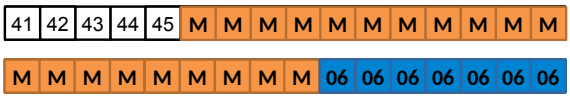


Figure 1: When processing five plaintext bytes with AES-CBC and HMAC-SHA, the encryptor needs to append 20 bytes of the HMAC-SHA output and seven bytes of padding.

3.1 Vaudenay's Padding Oracles

In 2002, Vaudenay showed that the MAC-then-Pad-then-Encrypt scheme introduces potential vulnerabilities in security protocols, in the form of so-called padding oracles [39]. The attacks leveraging these vulnerabilities are based on the malleability of the CBC mode of operation. We focus on the case of TLS.

Consider the TLS record layer when using CBC mode. After decryption, the decrypting party needs to verify the padding bytes and the MAC bytes. The natural way to implement these two checks is first to verify the padding bytes and, if they verify correctly, then verify the MAC bytes. If the padding bytes are invalid, it is natural for an implementation to emit an error message, without checking the MAC bytes. On the other hand, if the padding bytes are valid but the MAC is invalid, it is then natural to emit a (potentially different) error message.

Assume a decryptor that indeed emits two different error messages in these cases. The attacker can decrypt the last byte of any message block p_i as follows. He sets the last ciphertext block to c_i and replaces the last byte of the previous block c_{i-1} with a value between 0 and 255. If the last cleartext byte is `0x00`, then the padding will be valid (other forms of valid padding are much less likely). When the padding byte correctly verifies, the attacker detects this by observing that the decryptor emitted an "invalid MAC" error, rather than an "invalid padding" error. The attacker learns the value of the last byte of p_i after sending at most 255 ciphertexts to the decryptor.

Using his knowledge of the last plaintext byte, the attacker can proceed to decrypt the second-to-last byte of p_i. By doing so, he aims to create valid padding of length 2. More generally, using this technique, the attacker can iteratively decrypt every byte in p_i. We omit the formal description of the rest of the attack and refer the reader to [39].

[1] We note that the countermeasures summarized in [14] do not protect from timing-based attacks [4].

Note that the above attack relies on the ability to distinguish between ciphertexts decrypting to valid and invalid padding. It would therefore appear trivial for TLS implementations to prevent this attack by making sure they always emit the same error message. Indeed, Vaudenay was unaware of a way for an attacker to directly distinguish between these two cases in the context of TLS. The reason is that even if the TLS error messages differ, their distinction is impossible since they are encrypted with TLS session keys. This is one of the challenges we address in our work.

3.2 BEAST Attack Model

One question left open in Vaudenay's paper is how to exploit what he terms an "exploding oracle" – an oracle that is usable only until it first returns a negative answer. This models the problem where a TLS implementation will abort the session as soon as a message doesn't decrypt correctly. Hence, an attacker that relies on changing messages in a TLS session would not be able to continue the attack as soon as the first decryption error arises.

Canvel et al. used a model where the client repeatedly connects to the server [11], observing that this occurs due to polling behavior of email clients at the time, and exfiltrating an authentication password. The BEAST attack [34] essentially used the same model, but rather relied on the behavior of modern web browsers. In the simplest form of the BEAST model, a victim is tricked into visiting a malicious website controlled by the attacker. That website contains javascript which causes the victim browser to repeatedly connect to the victim website. Every website request then contains the user authentication cookie, which is automatically sent by the browser. This behavior allows the attacker to force the victim to repeatedly send encrypted values to the server.

Our attacks work in this model. We assume that the attacker can cause the victim client to repeatedly connect to a victim server while retransmitting the same sensitive information. We also assume the attacker is a man in the middle (MitM) and can change messages in transit. This model has now become standard in literature for modern attacks.

3.3 POODLE

The predecessor to TLS, SSLv3, uses a similar *MAC-then-Pad-then-Encrypt* scheme. However, unlike TLS, the value of the padding bytes in SSLv3 is under-specified. The last byte of the plaintext denotes how many padding bytes are present, but the rest of the padding bytes can take any value.

Consider a message with one full block of 16 padding bytes. The last block of plaintext will have a last byte of `0x0F`, and the first 15 bytes can take any value. Therefore, an attacker can use the techniques described in Section 3.1 to replace the last block with any block whose last byte decrypts to `0x0F`, and obtain a validly padded message. This property of SSLv3 led to a devastating attack called "POODLE". See [27] for a full description of the attack.

Although POODLE relies on the under-specification of the padding bytes in SSLv3, it surprisingly also affects TLS implementations. In essence, there is nothing forcing a careless TLS developer to verify the (specified) padding bytes after decryption; a TLS implementation will interoperate just fine even if it does not check the padding bytes at all. In fact, it is *easier* for the developer to reuse the same code that handles SSLv3 padding in a TLS implementation. This has led to a variant of the POODLE attack that affects TLS implementations [25]. Even after these two high-profile discoveries, variants of POODLE continued emerging [10, 29, 28]. These works detected different TLS record processing vulnerabilities; some TLS implementations only verified the first MAC byte, the others skipped validation of specific padding bytes.

3.4 Lucky 13 and Other Timing Attacks

In 2013, AlFardan and Paterson [4] used a similar technique to break TLS confidentiality and dubbed their attack "Lucky 13". The attack relies on an important observation: Common HMAC functions require different processing times when processing inputs of different lengths. By performing clever padding byte manipulations, the attacker can force the server to execute HMAC computations on plaintexts of different lengths. This is because the padding length determines the amount of data used as input into the HMAC function. The attacker can then measure the different processing times and learn information about the padding byte. We refer the reader to [4] for the full attack description.

The fix to Lucky 13 was to change the MAC verification code in TLS implementations to be constant-time, regardless of the number of processed cleartext blocks. This is possible, but writing and maintaining such code is hard, even for experts. In 2016, Somorovsky identified a bug in the patched code of OpenSSL [37]. The bug introduced a similar and even more severe vulnerability which allowed an attacker to distinguish between two alert messages. A different message could be triggered if the decrypted message only contained two or more valid padding blocks.

Amazon's s2n TLS library was released in 2015 [24], after the Lucky 13 attack was published. s2n's developers were aware of Lucky 13 and introduced specific countermeasures that seemed to render the code constant-time, thereby preventing the attack. They also introduced randomized timing delays to make the attack more difficult, in the unexpected case that the code turned out to be vulnerable. Despite all these efforts, s2n was still vulnerable to variants of Lucky 13 [3, 35]. All vulnerabilities were found despite the code having been formally verified.

3.5 Bleichenbacher's Attack and its Variants

Bleichenbacher's attack [8] is also a form of a padding oracle attack. Rather than targeting symmetric encryption, it targets a padding scheme used in RSA encryption, called PKCS#1 v1.5. It also similarly exploits a malleability property of RSA encryption and relies on a decryptor (i.e., a server) emitting error messages in case of invalidly-padded cleartexts. The standard countermeasure is similar to that of CBC padding oracles; the server must not behave differently when encountering error states in RSA decryption. This countermeasure has become part of the TLS standard.

However, implementing the countermeasure correctly is challenging. Böck et al. scanned for vulnerable TLS servers vulnerable to Bleichenbacher's attack [9]. They found vulnerabilities in servers used by high-profile websites such as Facebook and Paypal. Interestingly, their vulnerabilities could be triggered by using different TLS protocol flows or exploiting TCP connection states (TCP resets or timeouts). As with CBC padding oracles, Bleichenbacher's attack shows a similar sequence of an attack variant being discovered every few years in different contexts [26, 22, 6].

4 Scanning and Evaluation Methodology

The ultimate goal of our research is to estimate the number and the impact of padding oracle vulnerabilities and report our findings to the responsible vendors. To accomplish this, we proceed in three steps. We first define a list of test vectors potentially triggering observable differences which result in padding oracles. We then reduce this test vector list and perform a large-scale scan. Finally, we analyze the identified vulnerabilities and responsible vendors.

4.1 Test Vector Generation

In order to detect padding oracles in implementations, we connect and send various malformed records. These records contain different malformities in regards to the padding, MAC, and application data. We then observe if there are any differences in responses, in the TLS layer, or in lower layers. An implementation that responds differently to two malformed records may be vulnerable.

It is infeasible to test with all possible malformed records. For example, a vulnerable implementation could correctly check all padding bytes unless the padding bytes are exactly 16 bytes long, in which case the implementation does not check a specific bit in the padding.[2] Since there could be up to 256 padding bytes, testing the correct validation of each bit for all possible padding lengths would require testing with $\sum_{i=1}^{256} 8i = 263,168$ different records. These records need to be tested with different cipher suites or protocol versions which makes such a comprehensive test infeasible. We therefore carefully selected a set of malformed records which are motivated by previous research.

We concede this way of selecting the set of malformed records means we can only detect vulnerabilities that are similar to known ones. However, this approach is cost-effective and well-suited to large-scale scans. Since only a limited number of messages can be sent to individual servers during large-scale scans, automatic approaches for the test vector generation, like fuzzing, are usually infeasible.

4.1.1 Malformed Records

Our malformed records are all 80 bytes in length. Equal lengths ensure that differences in responses are likely caused by a padding oracle vulnerability and are not false positives triggered by different record lengths. Unusual record lengths may lead to errors that are unrelated to decryption; for example, recent OpenSSL versions respond with a different error message if the encrypted TLS record is shorter than the MAC length. We decided to use 80 bytes to have enough room for an HMAC output combined with two full padding blocks. This allows us to construct records protected by SHA-384, whose output is 48 bytes in length. We summarize our 25 malformed records in the following paragraphs. See also Table 1 for a summary of these malformed records for the case of `TLS_RSA_WITH_AES_128_CBC_SHA`.

Flipped MAC bits. We start with a valid record containing application data, a MAC, and four padding bytes. We then create three malformed records based on this record: One by flipping the most significant bit in the first MAC byte, one by flipping a middle bit in the middle of the MAC bytes, and one by flipping the least significant bit of the last MAC byte. We chose these malformed records to detect implementations where the MAC is not completely checked. The specific bit flipping positions are motivated by the recent OpenSSL vulnerability [1], where OpenSSL only checked the least significant bit of each byte on some platforms, and by further vulnerabilities caused by incomplete MAC validations [29, 28].

Missing One MAC byte. We start with a valid record containing empty application data, but with valid MAC and padding. We then modify it to create two malformed records: One where we delete the first MAC byte, and one where we delete the last MAC byte. We then add another padding byte in both messages. These malformed records could also trigger vulnerabilities caused by incomplete MAC validations and are indirectly motivated by [28].

Missing MAC. Motivated by [37], we created two malformed records which only contain padding and do not con-

[2] The above behavior may sound contrived, but similar behaviors have been found in the wild, see e.g. [29, 28, 37].

Nr.	MAC			Padding		
	Len	Pos	Modification	Len	Pos	Modification
1	20	20	⊕ 0x01	56	–	–
2	20	11	⊕ 0x08	56	–	–
3	20	1	⊕ 0x80	56	–	–
4	19	1	DEL	56	–	–
5	19	20	DEL	56	–	–
6	0	–	–	80	ALL	0x4F
7	0	–	–	80	ALL	0xFF
8	20	–	–	60	1	⊕ 0x80
9	20	–	–	60	31	⊕ 0x08
10	20	–	–	60	60	⊕ 0x01
11	20	1	⊕ 0x80	60	–	–
12	20	9	⊕ 0x08	60	–	–
13	20	16	⊕ 0x01	60	–	–
14	20	1	⊕ 0x01	60	1	⊕ 0x80
15	20	1	⊕ 0x01	60	31	⊕ 0x08
16	20	1	⊕ 0x01	60	60	⊕ 0x01
17	20	–	–	6	1	⊕ 0x80
18	20	–	–	6	3	⊕ 0x08
19	20	–	–	6	6	⊕ 0x01
20	20	1	⊕ 0x80	6	–	–
21	20	9	⊕ 0x08	6	–	–
22	20	16	⊕ 0x01	6	–	–
23	20	1	⊕ 0x01	6	1	⊕ 0x80
24	20	1	⊕ 0x01	6	3	⊕ 0x08
25	20	1	⊕ 0x01	6	6	⊕ 0x01

Table 1: A summary of our malformed records, as constructed for `TLS_RSA_WITH_AES_128_CBC_SHA`. The columns indicate length, position, and modification for MAC and padding bytes, respectively. ⊕ denotes XOR'ing the listed value in the listed position. DEL denotes deleting one byte in the listed position.

tain a MAC at all: One where we supply exactly 80 bytes of valid padding (`0x4F`), and one where we supply 80 bytes of incomplete padding of value `0xFF`. The latter is not only missing the MAC but also contains invalid padding since if the value of the last byte is `0xFF`, there should be 256 padding bytes.

Combining valid and invalid MAC and padding. The last group of malformed records contains messages with combinations of valid and invalid MAC and padding of three types: valid MAC and invalid padding, invalid MAC and invalid padding, and invalid MAC and valid padding. For each of these three types, we create three sub-types, depending on which bit positions we flip; we flip either the most significant, middle, or least significant bit in the first, middle, or 16th byte, respectively. For each of these nine sub-types, we create one version which contains application data, and one without. The length of the application data is chosen such that the padding bytes are contained within one plaintext block, while the malformed records without application data contain more than one block of padding. This aims to detect implementations which check only the last block of padding bytes.

4.1.2 Combining Malformed Records with Protocol Versions and Cipher Suites

We use each malformed record with several TLS protocol versions and cipher suites. As previously stated, we use the term *test vector* to refer to the combination of a malformed record, protocol version, and cipher suite. As we later show, testing each malformed record with different protocol versions and cipher suites is necessary; some vulnerabilities are only triggered with such specific combinations. At first glance this is surprising, but this actually follows the findings of [9]. We conjecture that implementations may use completely different code stacks depending on the negotiated version and cipher suite, and some vulnerabilities are only present in a subset of those code stacks.

4.2 Empirical Test Vector Reduction

Depending on the configuration of the server, the above set of test vectors is quite large. Assuming a server supporting TLS 1.0 and TLS 1.1 with 10 CBC cipher suites, there would be $10 \cdot 2 \cdot 25 = 500$ test vectors. Note that every test vector requires establishing a new TLS connection and performing an expensive handshake. This large number of test vectors would not allow us to perform large-scale scans. On the other hand, removing test vectors could lead to false negatives and missing vulnerabilities. To reduce the number of test vectors without lowering the detection rate, we propose an empirical test vector reduction approach. We sample 50,000 random hosts which respond on port 443. We then perform a full scan on these hosts with the aforementioned 25 malformed records and all supported cipher suites and TLS version combinations. We can then analyze our test vector combinations and create the smallest set of test vectors detecting all padding oracle vulnerabilities. These empirical steps ensure that 1) with high probability we do not miss vulnerabilities, and 2) we can use the reduced set for large-scale analyses.

4.3 Clustering Vulnerabilities

Once we reduce the number of test vectors we can perform our full scan. For this purpose, we use one of the *Internet top lists* which typically contain a good mixture of up-to-date server implementations. Among Internet top lists, the Alexa Top 1 Million dataset contains the most significant number of hosts responding to TLS connections (about 75%) and is recommended for TLS scans [36].

After performing the TLS scan with a reduced vector set, we create a list of vulnerable hosts. We re-scan these hosts with our full test vector list. For every host, we store its *response map*. The response map describes the complete host behavior when responding to our test vectors. The response map consists of *cipher suite fingerprints*. A cipher suite fingerprint describes the server response behavior for a specific cipher suite and TLS version.

One of our major goals is to notify vulnerable vendors. For this purpose, it is necessary to group vulnerable hosts using the resulting response maps and contact their administrators to find out the vulnerable implementation version. Böck et al. performed this step manually and were able to approach the most important vendors [9]. However, such an approach is laborious and error-prone. We aim to group vulnerable implementations automatically.

Although grouping vulnerable hosts appears to be easy given all response maps, response maps differ even if they use the same vulnerable implementation version. TLS servers running identical implementations can use different configurations, enabling different cipher suites and TLS versions. For example, server A may be vulnerable to a padding oracle attack and has only one TLS cipher suite enabled: `TLS_RSA_WITH_AES_128_CBC_SHA256`. Server B is vulnerable using the same cipher suite fingerprint. However, server B is configured to use additional cipher suites as well which are not vulnerable to the attack. Are these two servers using the same implementation or just a similar one? To estimate this, we devised a novel approach based on a two-dimensional force-directed graph drawing algorithm [21]. These algorithms embed a network of nodes on a plane that allows for spatially interpreting the network. They do so by creating a two-dimensional graph which contains as few crossing edges as possible. In our approach we use the ForceAtlas2 algorithm [21]: *ForceAtlas2 simulates a physical system in order to spatialize a network. Nodes repulse each other like charged particles, while edges attract their nodes, like springs. These forces create a movement that converges to a balanced state. This final configuration is expected to help the interpretation of the data [21].*

We represent the scanning results as a graph as follows: Each node in the graph represents a host. Each pair of hosts is connected by an edge if their response maps do not include different cipher suite fingerprints for the same cipher suite.

This approach works well on our dataset, and servers exhibiting similar vulnerabilities are grouped closely. We augment the graph by coloring nodes according to their degree (i.e., their number of edges). The resulting visualization indeed allows identifying similar implementations. We show the concrete results in Section 8.

5 Large Scale TLS Scanning

We developed our padding oracle test vectors with TLS-Attacker [37], a framework for systematic analyses of TLS implementations. TLS-Attacker supports creating malicious TLS workflows and message malformities. TLS-Attacker has already been used for detecting padding oracle attacks, but only against specific implementations in lab conditions, not at scale. Our approach of creating an optimized set of test vectors was not previously included in this framework.

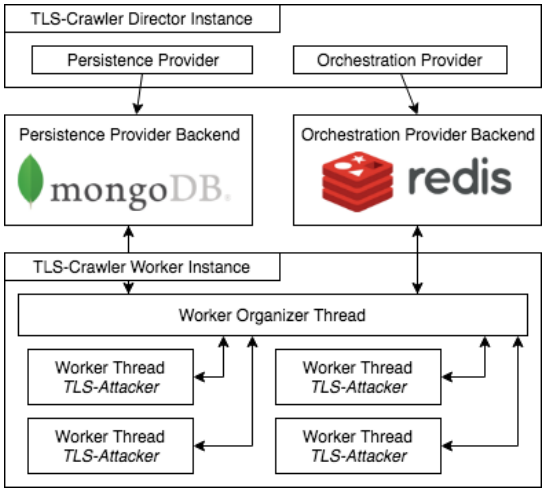


Figure 2: Our TLS scanning infrastructure is based on well-established components for data persistence and on TLS-Attacker for performing TLS evaluations.

5.1 TLS-Crawler

In order to scan a large number of hosts, we developed a framework which scans multiple servers in parallel and writes the results to a database. This allows us to parallelize the scan by using multiple machines. The database provides a querying interface for the scan data, which allows for easier analysis of the large result datasets. We call our framework TLS-Crawler.

TLS-Crawler is split into a director instance and potentially multiple worker instances. The worker instances perform the actual TLS host scans. Each worker instance implements a thread pool which distributes scanning work across available threads. The instance then bundles the results and coordinates parallelized database access. A director instance coordinates the worker instances. The director instance contains an orchestration provider responsible for the coordination and distribution of scanning tasks across workers. The results are persisted in a database using a persistence provider. We use MongoDB[3] as the persistence provider, and orchestrate instances via a Redis queue.[4] Figure 2 visualizes the TLS-Crawler architecture.

5.2 Performing the TLS Scans

Before scanning each host with test vectors, we perform a brief scan in order to learn the CBC cipher suites and TLS protocol versions supported by the host. We excluded export and anonymous cipher suites from these tests since they are already trivially broken by a MitM attacker. We then perform our scan using our set of test vectors for each CBC cipher

[3] `https://www.mongodb.com`
[4] `https://redis.io`

suite and its supported protocol version.

Previous large-scale TLS scans have mostly focused on vulnerabilities in the TLS handshake [9, 2], certificates [19], or vulnerabilities which could be triggered before the TLS handshake succeeds [17]. These previous scans only require performing a successful handshake once, usually with a commonly supported cipher suite. In contrast, in order to test for padding oracle vulnerabilities, it is necessary to perform a full TLS handshake for each tested cipher suite. This is complicated by TLS implementations exhibiting intolerances [7] which might prevent a server from completing the TLS handshake, or even responding to the initial `ClientHello` message. We tried to minimize the effect of these intolerances on our scans, but 20% of servers exhibited enough intolerances that we could not effectively scan them.

Even completing a TLS handshake does not guarantee we can effectively scan a host. For example, in some tests, the target hosts temporarily stopped responding for a few seconds. This is likely because the servers crashed or blocked our requests as part of a Denial-of-Service defense. In order to avoid false negatives from such scans, we scan multiple hosts in parallel (up to 2000) such that no host is overloaded by our requests. Additionally, we wait at least 10 seconds between scanning a host with two cipher suite/version pairs, further limiting the load on scanned hosts.

When performing these scans it is critical to select an appropriate timeout. If the timeout is too low, we might miss responses due to high server load. Conversely, a high timeout value would decrease the scanning performance. Setting a high timeout value also means we no longer distinguish between a server immediately closing the connection, and requiring a noticeable time to recover and close the connection. Additionally, the server's answers may span multiple TCP packets, so there is no simple way to ascertain the scanner has received the server's answer in full at any point in time. (Some responses do not include a TCP RST or FIN packet.) We empirically determined that a timeout of one second works well in practice, and mostly guarantees that the server did have enough time to process our record and respond. However, even when using this timeout value, we found servers that responded non-deterministically due to high load or various bugs.

To work around non-deterministic responses, we rescanned each suspected vulnerability in order to avoid false positives. We only consider a server as vulnerable if it responds identically in three separate scans to each of our test vectors.

6 Evaluation

For the scans, we used a machine with 2 Xeon E5-2683v5 CPUs (with a total of 64 cores) and 48 GB of RAM. The scan used an average of 5Mbit/s of upstream data and 15Mbit/s of downstream data.

6.1 Pre-Scanning with All Malformed Records

We performed a preliminary scan of 50,000 random TLS hosts, aimed at reducing the set of malformed records. The scan took place in October 2018 and required three days. The results confirmed that the choice of key exchange algorithm and protocol version indeed affects whether a given host exhibits CBC padding oracle vulnerabilities. We then reduced the set of malformed records. To do this, we first identified all vulnerable hosts, i.e. hosts that would be identified when scanning with the full set of malformed records. We then examined subsets of malformed records of increasing sizes, and for each subset, examined the number of hosts that would be identified when scanning only with this subset of malformed records. This process was stopped when a subset of four malformed records identified all vulnerable hosts. That is, all hosts that would be identified when scanning with the full set of malformed records, would also be identified when scanning with the reduced set of malformed records. This reduced set includes the following malformed records (all of these records are 80 bytes in length):

1. A record with missing MAC and correct padding (of value `0x4F`).

2. A record with missing MAC and incorrect padding (of value `0xFF`).

3. An empty record with no application data, with invalid padding and valid MAC. The highest bit in the first padding byte is flipped.

4. An empty record with no application data, with valid padding and invalid MAC. The lowest bit in the first MAC byte is flipped.

Please note that we still test every TLS host with all of its supported cipher suites and TLS protocol versions.

Is the malformed record set reduction lossy? The reduced malformed record set detects all vulnerabilities detected by the larger, original malformed record set, *on the sample data of the preliminary scan*. It is natural to ask whether there are hosts that are vulnerable to a malformed record from the original set, but not to a malformed record from the reduced set. There are obviously no such hosts in the sample data, but there could be such hosts outside of the sample. If there is a large number of such hosts on the Internet, then the malformed record reduction process would be lossy, i.e. by using fewer malformed records, we detect fewer vulnerabilities in the full scan. As we now explain, this source of scanning inaccuracy is likely small enough to not materially affect our results. Put another way, the reduced set of malformed records likely detects most vulnerabilities

triggered by the full set of malformed records, not just on the sample data.

Indeed, let p denote the percentage of hosts, out of all TLS-speaking hosts, that are vulnerable to one malformed record from the full set of malformed records, but not to any malformed records from the reduced set. I.e., p describes the percentage of hosts that the reduction misses; we will now show it is rather small. In the random sample of $N = 50000$ hosts used for the preliminary scan, we did not encounter any such hosts. In order to compute the 99% confidence interval, we require $(1-p)^N = 0.01$. Solving for p, we obtain $p = 0.0092\%$. We therefore determine with 99% confidence that there are at most 0.0092% additional vulnerable hosts that our scans miss due to the malformed record reduction.

We provide an intuitive explanation of the above, for the reader's convenience. As per the above calculation, we estimate the percentage of vulnerable hosts on the Internet that would be missed because we scan with the reduced set of malformed records is 0.0092%. Censys [16] estimates there are about 42.4 million hosts which serve TLS on port 443 as of February 2019. Therefore, our estimate is that the reduction misses at most $42400000 \cdot 0.0092\% = 3900$ hosts. Intuitively, the term "99% confidence interval" means there is roughly a 1% chance that this estimate is wrong, i.e. that there are more than 3900 such hosts on the Internet.

6.2 Alexa Top Million Scan

We used the reduced set of malformed records to scan the Alexa Top Million websites. Among the top lists, Alexa Top 1 Million provides the highest percentage of hosts supporting TLS [36] and is thus suitable for large-scale TLS scans. The list likely includes most high-profile TLS implementations.

The scan required approximately 72 hours. Of the initial one million hosts, 785,295 responded on port 443. We were able to perform TLS handshakes with CBC cipher suites with 627,493 hosts. We excluded all other hosts from the evaluation. We discovered a total of 18,257 Alexa Top Million hosts (1.83%) which are vulnerable to padding oracle attacks.

The data supports our conjecture that implementations may be vulnerable on a cipher suite with one protocol version, but not vulnerable on the same cipher suite with a different protocol version. A total of 649 servers were only vulnerable in either TLS 1.0 or TLS 1.1/1.2 although the vulnerable cipher suite was supported in the other version. Similarly, in some cases, the negotiated key exchange algorithm affects whether implementations exhibit a CBC vulnerability. 601 hosts were vulnerable on one cipher suite, but not on another cipher suite with a different key exchange algorithm but the same symmetric cipher and HMAC function. A total of 3,247 hosts were vulnerable on all CBC cipher suites they supported.

After identifying vulnerable hosts, we rescanned them with the full set of test vectors to get their full response maps. As noted above, to label a host as vulnerable we require the response maps to be consistent across three different scans.

6.3 Results of Our Clustering Approach

Analyzing each vulnerable host manually is infeasible. We therefore clustered the vulnerable hosts, such that hosts exhibiting the same cipher suite fingerprints are clustered together. This minimizes the manual work required to identify the vendor (or vendors) responsible for each vulnerable behavior. We reiterate that this clustering is not trivial, as explained in Section 4.3.

We identified 93 different cipher suite fingerprints. Table 2 summarizes the 40 most common cipher suite fingerprints. Using the first row as an example, 7297 hosts responded with `BAD_RECORD_MAC` and `CLOSE_NOTIFY` TLS alerts and timed out the connection for malformed records 11 and 12 (🕓). For all other malformed records these hosts closed the TCP connection (🚫) after sending the same TLS alerts.

We also identified four groups exhibiting behavior similar to the CVE-2016-2107 vulnerability in OpenSSL [37] (cipher suite fingerprints #41, #75, #14, and #54 in Table 2). They respond to malformed records 6 and 7 (see Table 1) with a `RECORD_OVERFLOW` TLS alert. To all other malformed records they respond with `BAD_RECORD_MAC`. These are likely unpatched OpenSSL implementations, or security appliances running older OpenSSL versions.

For vulnerable cipher suites on the same host, cipher suite fingerprints are largely consistent. Of hosts exhibiting at least one vulnerable cipher suite, 99.6% have an identical cipher suite fingerprint on all vulnerable cipher suites. We removed the remaining 0.4% of hosts to make clustering easier. However, hosts sharing the same cipher suite fingerprint on vulnerable cipher suites don't necessarily share the same implementation. As an example, consider two hosts, A and B, with two cipher suites supported by both hosts, 1 and 2. A is vulnerable on cipher suite 1 with cipher suite fingerprint X, but is not vulnerable on cipher suite 2. B is not vulnerable on cipher suite 1, but is vulnerable on cipher suite 2 with the same cipher suite fingerprint X. This difference indicates the hosts don't share the same implementation, as we would expect the shared implementation to have a consistent set of vulnerable cipher suites. (We concede that it is possible the hosts exhibit different behavior because of different configuration flags despite sharing the same implementation, but we consider this unlikely).

We denote the above situation (in its general form) as "contradictory response maps"; two hosts exhibiting the same cipher suite fingerprint on vulnerable cipher suites, but where there exists a cipher suite supported by both hosts such that one host is vulnerable on that cipher suite and the other host is not. We refer to the complement situation as "compatible response maps".

Nr.	Cipher suite fingerprint									Strength		Count
	1,2,3,20,21	4,5	6	7	8,9	10,16,19,22–25	11,12	13,14,15	17,18	R1	R2	
15	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⏲	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	👁	*S*	7297
41	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{22} 🔒 ⊘	F_{22} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	👁‍🗨	*W*	4387
84	ϟ	ϟ	ϟ	ϟ	⏲	ϟ	ϟ	ϟ	F_{20} 🔒 ϟ	👁	*P*	2313
75	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{22}W 🔒 ⊘	F_{22}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	👁‍🗨	*W*	940
21	F_{80} 🔒 ⊘	F_{80} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{80} 🔒 ⊘	F_{80} 🔒 ⊘	F_{80} 🔒 ⊘	F_{80} 🔒 ⊘	F_{80} 🔒 ⊘	👁‍🗨	*W*	687
23	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20} 🔒 ⏲	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	👁	*W*	458
68	⊘	⊘	⊘	⊘	⏲	⊘	⊘	⊘	⏲	👁	*P*	248
0	ϟ	ϟ	ϟ	ϟ	⏲	ϟ	ϟ	ϟ	A ⊘	👁	*P*	194
79	⊘	⊘	⊘	⊘	⊘	⊘	F_{20}W 🔒 ⏲	⊘	⊘	👁	*W*	151
10	F_{40} 🔒 ⊘	F_{40} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{40} 🔒 ⊘	F_{40} 🔒 ⊘	F_{40} 🔒 ⊘	F_{40} 🔒 ⊘	F_{40} 🔒 ⊘	👁‍🗨	*W*	98
85	⊘	⊘	⊘	⊘	⏲	⊘	⊘	⊘	A ⊘	👁	*P*	83
2	ϟ	ϟ	⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	👁	*S*	76
61	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	ϟ	ϟ	ϟ	F_{20} 🔒 ⊘	ϟ	ϟ	👁‍🗨	*S*	54
6	ϟ	ϟ	ϟ	ϟ	F_{40} 🔒 ⊘	ϟ	ϟ	ϟ	F_{40} 🔒 ⊘	👁	*P*	52
62	⏲	⏲	⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	👁	*S*	47
33	ϟ	ϟ	ϟ	ϟ	⏲	ϟ	ϟ	ϟ	⏲	👁	*P*	43
31	⊘	⊘	⊘	⊘	⏲	⏲	⊘	⏲	⏲	👁	*P*	36
76	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	⏲	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	⏲	👁	*P*	34
77	F_{20}W 🔒 ⊘	F_{50}W 🔒 ⊘	F_{50}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	👁‍🗨	*S*	28
14	F_{20} 🔒 ϟ	F_{20} 🔒 ϟ	F_{22} 🔒 ϟ	F_{22} 🔒 ϟ	F_{20} 🔒 ϟ	F_{20} 🔒 ϟ	F_{20} 🔒 ϟ	F_{20} 🔒 ϟ	F_{20} 🔒 ϟ	👁‍🗨	*W*	24
24	F_{20}W 🔒 ϟ	F_{20}W 🔒 ϟ	F_{20}W 🔒 ϟ	F_{20}W 🔒 ϟ	F_{20}W 🔒 ϟ	F_{20}W 🔒 ϟ	F_{20}W 🔒 ⏲	F_{20}W 🔒 ϟ	F_{20}W 🔒 ϟ	👁	*W*	21
38	F_{80} 🔒 ⊘	F_{80} 🔒 ⊘	F_{80} 🔒 ⊘	ϟ	ϟ	ϟ	F_{80} 🔒 ⊘	ϟ	ϟ	👁	*S*	19
4	ϟ	ϟ	ϟ	ϟ	⏲	ϟ	ϟ	ϟ	F_{20} 🔒 ⏲	👁	*P*	15
54	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	F_{22} 🔒 ⏲	F_{22} 🔒 ⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	👁‍🗨	*W*	12
74	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⏲	F_{20}W 🔒 ⏲	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⏲	F_{20}W 🔒 ⊘	F_{20}W 🔒 ⊘	👁	*W*	9
7	ϟ	ϟ	ϟ	ϟ	⊘	ϟ	ϟ	ϟ	⊘	👁	*P*	8
37	F_{20} 🔒 ⊘	F_{50} 🔒 ⊘	F_{50} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	👁‍🗨	*W*	7
51	⊘	⊘	⊘	⊘	⏲	⊘	⊘	⊘	⊘	👁	*W*	7
59	A ⊘	ϟ	ϟ	ϟ	ϟ	ϟ	⏲	ϟ	ϟ	👁	*S*	7
66	ϟ	ϟ	⏲	ϟ	⏲	ϟ	⏲	⏲	ϟ	👁	*W*	7
70	A ⊘	A ⊘	⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	👁	*S*	7
11	F_{20} 🔒 ⊘	⏲	⏲	⏲	⏲	⏲	F_{20} 🔒 ⊘	⏲	⏲	👁	*P*	5
42	F_{20} 🔒 ⊘	F_{21} 🔒 ⊘	F_{21} 🔒 ⊘	F_{21} 🔒 ⊘	F_{21} 🔒 ⊘	F_{21} 🔒 ⊘	F_{20} 🔒 ⊘	F_{21} 🔒 ⊘	F_{21} 🔒 ⊘	👁‍🗨	*S*	5
89	ϟ	ϟ	⏲	⊘	⊘	⊘	⏲	⊘	⊘	👁	*S*	5
3	⏲	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	⏲	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	👁	*S*	4
26	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{10} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	👁‍🗨	*W*	4
28	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	⏲	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	F_{20} 🔒 ⊘	AW 🔒 ⊘	👁	*P*	4
35	ϟ	ϟ	ϟ	ϟ	F_{20}W 🔒 ⊘	ϟ	ϟ	ϟ	F_{20}W 🔒 ⊘	👁	*P*	4
73	⊘	F_{80} 🔒 ⊘	F_{80} 🔒 ⊘	⊘	⊘	⊘	⊘	⊘	⊘	👁	*W*	4
9	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	F_{20} 🔒 ⊘	F_{20} 🔒 ⏲	F_{20} 🔒 ⏲	👁	*W*	3

Table 2: Analysis of the 40 most common cipher suite fingerprints, each consisting of responses to 25 malformed records. For ease of reading, we group together malformed records for which responses are identical within each cipher suite fingerprints. We use the following notation: `Application` message (A), Fatal Alert with error code k (F_k), Warning Alert (W), connection closed (⊘), TCP reset (ϟ), timeout (⏲). We use the following TLS Alert codes: `UNEXPECTED_MESSAGE` (10), `BAD_RECORD_MAC` (20), `DECRYPTION_FAILED_RESERVED` (21), `RECORD_OVERFLOW` (22), `DECOMPRESSION_FAILURE` (30), `HANDSHAKE_FAILURE` (40), `ILLEGAL_PARAMETER` (47), `DECODE_ERROR` (50), `DECRYPT_ERROR` (51), `INTERNAL_ERROR` (80). Alerts with code `CLOSE_NOTIFY` always used the `warning` level. 🔒 denotes an encrypted response. The oracle *strength* definition is provided in Section 7; observable differences are depicted with 👁, unobservable differences with 👁‍🗨. We use *W* and *S* for weak and strong padding oracles, respectively (a strong and observable oracle is exploitable). *P* represents behavior similar to POODLE (which is also exploitable if it is observable).

We then use a graph algorithm in order to further split host groups. For each group of hosts with an identical cipher suite fingerprint, we construct a graph where each node represents a host. We draw an edge between two hosts if and only if their response maps are compatible. We then embed the graph in a two-dimensional plane using the ForceAtlas2 algorithm, as implemented in the Gephi software.[5] The ForceAtlas2 algorithm clusters together nodes connected by an edge, so nodes with compatible response maps are clustered together. Identically configured servers which behave identically will be connected to the same nodes and will therefore have the same degree. Since these servers are connected to the same nodes, ForceAtlas2 will draw them close to one another. By coloring the nodes by their degree it becomes easy to manually spot similarly configured and identically behaving implementations in the graph. These subgroups can then be examined for candidates for manual analysis and responsible disclosure. [6]

Example for one vulnerability group. An example of this visualization is provided in Figure 3. The figure clearly shows two distinct sub-groups which do not share edges (meaning their response maps are contradictory and they likely do not share the same implementation). Hosts shown in green are vulnerable on `TLS_RSA_WITH_AES_128_CBC_SHA` and `TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA`, while servers shown in pink are only vulnerable to `TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA` and not on `TLS_RSA_WITH_AES_128_CBC_SHA`. Interestingly the hosts in the middle of the graph (mostly in teal) do not support `TLS_RSA_WITH_AES_128_CBC_SHA` (they are vulnerable on `TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA`). They may share their implementation with either the green or pink group and therefore share edges with the members of both groups. Hosts in red are very similar to the pink group but do not share edges with the teal group. This means that either a third group exists, or the teal group actually belongs to the green group and the red group belongs to the pink group. Individual nodes are likely rare configurations of one of the implementations of the bigger groups. We performed a DNS lookup and determined both groups are operated by a Czech hosting company.

This approach allowed us to also contact other prominent websites in each group and ask what TLS implementation they use.

[5] `https://github.com/gephi/gephi`

[6] We note that further grouping by the server agent string could provide more insights into the different groups. However, it is also very likely that it would also falsify our results. In many cases, TLS is terminated in reverse proxies or firewalls, and the server agent string is generated on a different machine handling HTTP traffic.

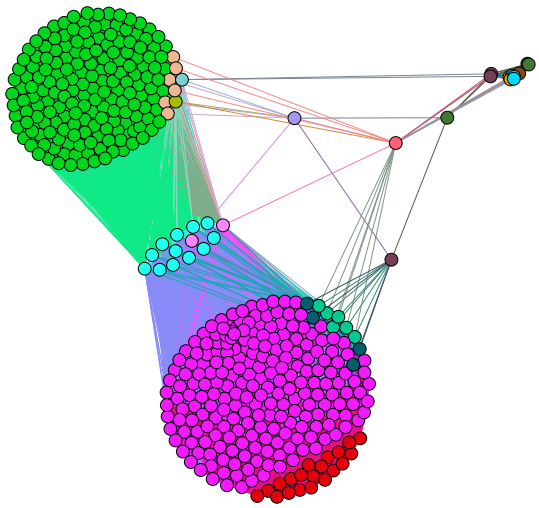


Figure 3: Visualisation of group #23 from Table 2.

Breakdown of response maps. Figure 4 visualizes the prevalence of the various cipher suite fingerprints. A few very common vulnerabilities account for the majority of vulnerable hosts. The newly-discovered vulnerabilities in Amazon/OpenSSL and Citrix account for slightly more than half of all vulnerable hosts. These are listed as #15 and #84 and described in more detail in Section 8.2. In addition, response maps #41 and #75, which likely stem from implementations based on unpatched OpenSSL versions, account for roughly a third of vulnerable hosts. Response map #23 is found in the above-mentioned Czech hosting company.

7 Realistically Exploitable Padding Oracles

Not all of the oracles we identified enable effective decryption attacks. The rest of this section explains exploitation in more detail.

The padding oracles we discovered are based on direct message side channels, i.e. on TLS implementations where two error states trigger different error responses from the TLS server. They may be exploitable in the BEAST attacker model, which relies on two assumptions: (a) the victim client visits a website under the attacker's control, which triggers HTTPS requests to the victim server, and (b) the attacker is a MitM and can observe the session and modify transmitted ciphertexts. In addition to those standard assumptions, an oracle is exploitable if it satisfies two additional requirements: *(R1) Observability* and *(R2) Perfect padding distinguishability*.

7.1 (R1) Observability

Unlike timing side channels, little attention has been paid to direct message side channels in the case of TLS, and common wisdom seems to assume they are *unobservable* to the

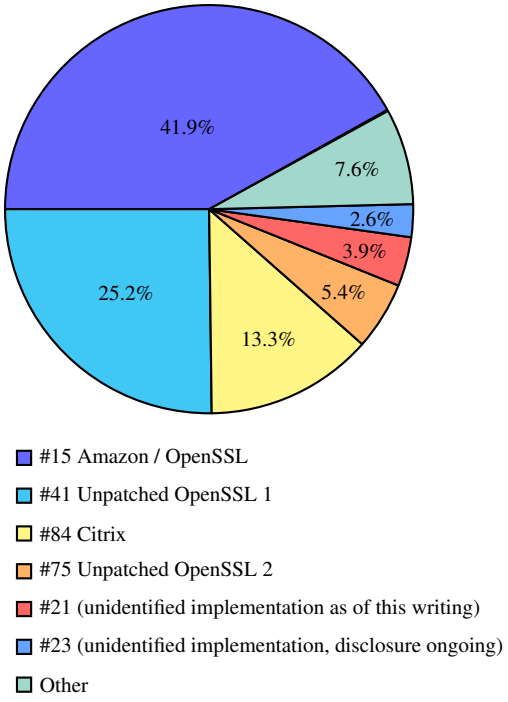


Figure 4: A visualization of the prevalence of cipher suite fingerprints. A few widely-prevalent vulnerabilities account for the majority of vulnerable hosts. Out of the above cipher suite fingerprints, #84 and #15 are exploitable. They are described in more detail in Section 8.2.

attacker. Indeed, this is true for implementations which send a single alert in all error cases and the behavior is identical except perhaps for the content of the alert message. Such behavior cannot be exploited by the attacker to create a side channel because the alert message is encrypted. However, we identified many cases where implementations do exhibit an observable difference in behavior. These observable differences can roughly be divided into two classes:

- **TCP layer.** We found implementations which leak information about the padding validity in the TCP layer. For example, in the case of Amazon, most test vectors with invalid padding caused the server to immediately close the TCP connection. However, specific, carefully crafted test vectors caused the server to abort the TLS session while keeping the TCP connection open.
- **Number of TLS records.** We observed TLS servers that responded with a different number of records based on the padding validity. While the attacker cannot decrypt these records, he is able to observe the total ciphertext length. For example, the servers from group 23 (see Table 2) responded with *one* TLS alert in the case of valid padding, while for invalid padding they responded with *two* TLS alerts.

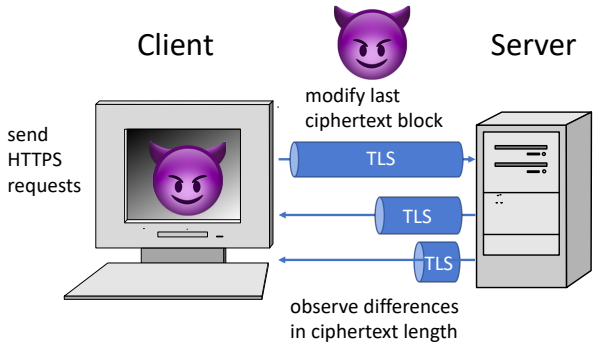


Figure 5: Exploiting observable error-based padding oracles in a BEAST scenario. Differences in total ciphertext length result from different numbers of TLS alerts being sent.

Consider an attacker $\mathscr{A}$ who can distinguish between the two cases of `valid_padding` and `invalid_padding` based on the validity of the last padding byte (see Figure 6). The attacker decrypts an HTTPS session cookie as follows:[7]

1. $\mathscr{A}$ lures the victim client to load a web page he controls. This web page contains JavaScript code which sends HTTPS requests to the victim server, with a URL of $\mathscr{A}$'s' choice.
2. $\mathscr{A}$ observes the first TLS handshake and determines if the negotiated cipher suite is vulnerable to padding oracle attacks. If not, he aborts.
3. If a vulnerable cipher suite is used, $\mathscr{A}$ instructs the client to send another HTTPS request, modifying the URL such that the first character of the session cookie is the last byte in cipher block c_i.
4. As a MitM, $\mathscr{A}$ intercepts the ciphertext $(c_1, ..., c_i, ..., c_n)$ and modifies it such that c_i becomes the last ciphertext block, for example by replacing c_n with c_i.
5. Decryption of this last block c_i is a pseudorandom transform, so the padding will likely be invalid, triggering an observable `invalid_padding` error event.
6. In about 1 out of 256 requests, the padding will randomly be valid. When the padding is valid, it is most likely to be one byte in length, as depicted in Figure 6. The preceding bytes will be parsed by the TLS server as MAC data, and will be invalid with overwhelming probability. In this case, $\mathscr{A}$ observes a `valid_padding` error event, and computes the first character of the HTTPS

[7] We present here a more general form of the attack, which is also applicable to POODLE-style oracles. This form requires 256 sessions on average in order to decrypt one plaintext byte [27]. For oracles which completely disregard the MAC, there is a faster form which requires 128 sessions on average to decrypt one plaintext byte.

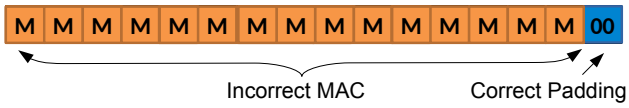


Figure 6: Our attacks rely on a vulnerable server that delivers different responses based on the validity of the last padding byte.

session cookie as $c_{n-1}[-1] \oplus c_{i-1}[-1]$, where the $[-1]$ operator denotes taking the last byte of a block.

7. $\mathscr{A}$ then prepares another HTTPS URL where the second character of the session cookie is shifted to the last byte of c_i, and starts again with step 3.

7.2 (R2) Perfect Padding Distinguishability

In the above example, we considered a simple oracle that allows for distinguishing between `valid_padding` and `invalid_padding` based on the validity of the last padding byte. However, even when providing different responses, implementations do not necessarily expose such simple oracles. For example, an older OpenSSL version responds with a different alert message only in the specific case of an empty record containing at least two full valid padding blocks [37]. We identified vulnerable implementations that only respond differently to ciphertexts containing several valid padding or MAC bytes. Such vulnerabilities are less likely to be exploitable since using the algorithm above, the attacker would need to perform far more than 256 oracle queries to decrypt each byte. The attacker may be able to overcome this limitation by inserting bytes of his choice directly after the cookie value. Due to the malleability property of CBC, it is only possible to insert one block of successive chosen data. Therefore, CBC allows for the creation of practical exploits if the number of chosen padding bytes is smaller than the block size.[8]

Therefore, in our impact estimation, we take a conservative approach. To consider a vulnerable implementation as exploitable, we require that it responds with `valid_padding` to ciphertexts with at most one block of valid padding. We call such oracles *strong* and refer to other oracles as *weak*. In addition to these two oracles, we consider oracles which do not correctly validate the complete CBC padding and only validate the MAC. We refer to such oracles as *POODLE oracles*. These oracles could also be exploited by applying attacks similar to POODLE.

Column *R2* in Table 2 identifies the oracle strength. For example, servers with the second most prevalent cipher suite fingerprint (#41) respond to malformed records #6 and #7 from Table 1 with a `RECORD_OVERFLOW`. In all other cases, the servers send the `BAD_RECORD_MAC` alerts. We consider this group to be weak since the attacker needs to send more than one block of valid padding to trigger the `RECORD_OVERFLOW` alert with a malformed record #6 or #7.

We consider servers with cipher suite fingerprint #2 to be strong oracles. The servers from this group respond with a TCP connection reset (ϟ) if they receive a malformed record with a valid padding (see malformed records #20 and #21). There are also several groups with behavior similar to POODLE. These groups ignore modifications in the MAC bytes and respond differently to malformed records #8, #9, #17, and #18.

7.3 Exploitability

We consider observable POODLE and observable strong oracles as exploitable. We consider all other oracles as non-exploitable. However, note that weak oracles may be exploitable using more advanced techniques. Our estimate of the number of exploitable hosts is, therefore, a conservative lower estimate.

Estimation of exploitable hosts. Our scan identified 18,257 hosts vulnerable to padding oracle attacks. Of those, 11,225 (61.4%) exhibit observable vulnerabilities that allow an attacker to distinguish between two malformed records. See also column *R1* in Table 2. At least 10,688 hosts provided strong or POODLE-styled oracles, which is 58% of vulnerable hosts. See also column *R2* in Table 2. In total, 10,501 hosts are practically exploitable, i.e. they meet both requirements.

Are CBC cipher suites negotiated? Most modern browsers support AEAD cipher suites. If a vulnerable server prefers AEAD cipher suites, they would likely be negotiated, and this precludes CBC attacks. 31,651 hosts or 4.03% only support RC4 or CBC cipher suites. Most modern browsers have disabled support for RC4 cipher suites due to [30], so modern browsers would likely negotiate CBC cipher suites with these hosts. Of those hosts, 1,400 were vulnerable to padding oracle attacks.

8 Findings

In this section we review our assumptions and present notable vulnerabilities we found in different implementations.

8.1 Do our Initial Assumptions Hold?

We performed our scans under the assumption that scanning with different cipher suites and protocol version is necessary

[8] Decrypting parts of the cookies with weak oracles or exploiting weak oracles could also be possible with extended techniques. We do not analyze the exploitability of these more complex oracles. Such an analysis would likely need to be done manually for each oracle and would need to consider specific browser behaviors.

in order to detect vulnerable hosts. As explained below, our findings confirm this assumption.

Is scanning with different protocol versions necessary? Böck et al. found that some servers exhibit RSA padding oracle vulnerabilities only on some of the protocol versions they support [9]. As noted in Section 3.5, we suspected the same holds for CBC padding vulnerabilities. Our findings confirm this assumption: We identified at least 744 hosts that support the same cipher suite in both TLS 1.0 and 1.2, but are vulnerable when using that cipher suite only in one of those versions. In some cases the vulnerable protocol version is the newer version, and in other cases, the older one. As an example of the former case, vine.co was vulnerable using TLS 1.2 with the `TLS_RSA_WITH_3DES_EDE_CBC_SHA` cipher suite, but was not vulnerable when using the same cipher suite in TLS 1.0.

Surprisingly, when only one protocol version is vulnerable with the same cipher suite, there are more cases where the newer version is vulnerable. Out of those 744 hosts, 120 hosts are vulnerable in TLS 1.0 but not in TLS 1.2, and 624 are vulnerable in TLS 1.2 but not in TLS 1.0.

Is scanning with different cipher suites necessary? Böck et al. also found that scanning with different cipher suites is necessary to detect as many vulnerabilities as possible [9]. In the above work, this finding held even when scanning with cipher suites using different symmetric ciphers, while the vulnerability was in the (theoretically unrelated) RSA implementation.

We find similar behavior in our results. We identified at least 601 hosts with two cipher suites, one vulnerable and one secure, where the only difference between the two cipher suites is the *key exchange algorithm*. This finding is unintuitive, as one would expect an implementation to be uniformly vulnerable or secure on all cipher suites with the same symmetric cipher. To give one example, one website is secure when using `TLS_RSA_WITH_AES_256_CBC_SHA256` with TLS 1.2, but is vulnerable when using `TLS_DHE_RSA_WITH_AES_256_CBC_SHA256`, also with TLS 1.2.

Rationale behind the server behaviors. Both behaviors may seem unintuitive but are actually expected. Many implementations take completely different code paths depending on the negotiated cipher suite or protocol version. These code paths may, for example, rely on hardware acceleration or use an optimized assembly implementation when possible. It is therefore likely (and, as we see, common) to find implementations that exhibit vulnerabilities only in some of the supported cipher suites and protocol versions, even when the same symmetric cipher is used.

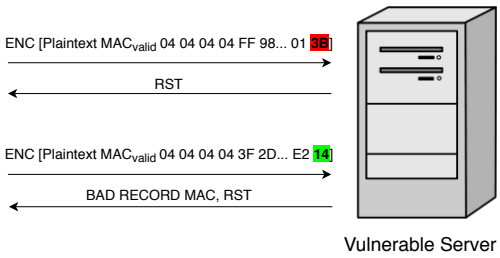


Figure 7: Behavior of Citrix implementations with cipher suite fingerprint #84.

8.2 Notable Vulnerabilities

In our scans we identified multiple devices from Cisco, two different IBM servers, and multiple devices from Sonicwall and Oracle. In the following, we describe specific vulnerabilities we identified and responsibly disclosed in Citrix, OpenSSL, and IBM servers.

Our disclosure is still an ongoing process. Our recent findings and the current state of countermeasures implemented by affected vendors are summarized on `https://github.com/RUB-NDS/TLS-Padding-Oracles`.

Amazon/OpenSSL. With the help of the Amazon security team, we identified a vulnerability (cipher suite fingerprint #15) which was mostly found on Amazon servers and Amazon Web Services (AWS). Hosts affected by this vulnerability immediately respond to most records with `BAD_RECORD_MAC` and `CLOSE_NOTIFY` alerts, and then close the connection. However, if the hosts encounter a zero-length record with valid padding and a MAC present, they do not immediately close the TCP connection, regardless of the validity of the MAC. Instead, they keep the connection alive for more than 4 seconds after sending the `CLOSE_NOTIFY` alert. This difference in behavior is easily observable over the network. Note that the MAC value does not need to be correct for triggering this timeout, it is sufficient to create valid padding which causes the decrypted data to be of zero length. Therefore, we classify this as a strong oracle which is also exploitable.

Further investigations revealed that the Amazon servers were running an implementation which uses the OpenSSL 1.0.2 API. In some cases, the function calls to the API return different error codes depending on whether a MAC or padding error occurred. The Amazon application then takes different code paths based on these error codes, and the different paths result in an observable difference in the TCP layer. The vulnerable behavior only occurs when AES-NI is not used.

We had in fact previously tested the vulnerable OpenSSL code manually, in lab settings, but had not identified this vulnerability. This is because the vulnerability only manifests

under a combination of specific conditions: subtle interactions between OpenSSL and external code, and only when AES-NI is not used, which is rare nowadays. We view this as an illustrative example of the usefulness of large-scale scans in detecting vulnerabilities that lab tests may sometimes miss.

We suspect this OpenSSL behavior underlies a number of similar vulnerabilities we identified, not only vulnerability #15. Therefore, we hope that once OpenSSL releases a patch, other vulnerabilities will be fixed as a result. The issue was assigned CVE-2019-1559.

The IBM vulnerabilities. We found multiple vulnerabilities in servers hosted by IBM. One of the vulnerabilities is described by cipher suite fingerprint #77 in Table 2. Affected servers respond with a `BAD_RECORD_MAC` alert if either the MAC or the padding is incorrect. If the padding is correct and the MAC is incomplete or not present, the server responds with a `DECODE_ERROR` alert. The latter behavior occurs even if the records are too short to contain a MAC, as long as the record contains at least two blocks of ciphertext, independently of the used MAC algorithm. An attacker can send only two blocks with an IV, which guarantees there is not enough room for a MAC. This provides the attacker with a classic CBC padding oracle. We therefore consider this a strong oracle. Since the alerts are encrypted, we classify this vulnerability as unobservable, and the oracle is therefore not exploitable.

The IBM security team decided to disable CBC cipher suites on the affected servers and to only support AES-GCM.

Citrix. The described vulnerability is identified by cipher suite fingerprint #84 in Table 2. The vulnerable implementation first checks the last padding byte and then verifies the MAC. If the MAC is invalid, the server closes the connection. This is done with either a connection timeout or an `RST`, depending on the validity of the remaining padding bytes. However, if the MAC is valid, the server checks if all other remaining padding bytes are correct. If they are not, the server responds with a `BAD_RECORD_MAC` and an `RST` (if they are valid, the record is well-formed and is accepted). We visualize this behavior in Figure 7. This behavior can be exploited with an attack similar to POODLE. Since the oracle is also observable, we consider this group as exploitable. We first detected this vulnerability in Amazon Web Services. In cooperation with the Amazon security team, we determined that Citrix Application Delivery Controller (ADC) and NetScaler Gateway are responsible for this behavior. The vulnerability was assigned CVE-2019-6485.

9 Related Work

We now highlight past work that focused on large-scale scans for vulnerabilities on the modern Internet. For a survey of related work on padding oracle attacks, we refer the reader to Section 3. ZMap [18] is a network scanner capable of reaching high scanning speeds. Durumeric et al. [17] used ZMap to scan the IPv4 address space to quantify the impact of the Heartbleed vulnerability [32]. Heninger et al. [19] scanned TLS and SSH for weak keys generated using insufficient entropy. Adrian et al. [2] introduced the Logjam vulnerability and used Internet-wide scanning to quantify its effects, depending on attacker computational resources. Aviram et al. [5] introduced the DROWN vulnerability and similarly used Internet-wide scanning to quantify its effects. Böck et al. [9] performed large-scale scans for Bleichenbacher's vulnerability, while also observing side channels such as changes in the TCP connection state, as we do here. Valenta et al. [38] scanned for known vulnerabilities in elliptic curve implementations, searching for a combination that could enable a powerful attack named CurveSwap.

10 Conclusions and Future Work

This work demonstrates that padding oracle vulnerabilities still exist on the modern Internet and will likely continue to threaten users' security. These vulnerabilities are often hard to detect: they may rely on subtle side channels or require specifically-crafted inputs in order to trigger.

In the past, major new TLS attacks had positive effects on the ecosystem. For example, the work by Adrian et al. [2] resulted in an "enforcement" effort, where major browsers changed their behavior and refused to connect to servers with weak DH parameters. It is an interesting open question how the security community can better help server operators detect and remediate more subtle kind of vulnerabilities (CBC oracles in particular, and other classes of vulnerabilities in general).

One solution in the context of CBC oracles would be to disallow CBC cipher suites altogether. Recently, major browser vendors have declared their intention to remove support for the old 1.0 and 1.1 TLS versions. This forces many server operators to upgrade their implementations or change configuration. Indeed, a case could be made that browser vendors can also remove support for CBC cipher suites, forcing again server operators to upgrade. These changes are not without their costs; they usually require notice of months in advance, may require coordination between browser vendors, and obviously, create additional work for server operators.

Our results again confirm that large-scale scans make it feasible to uncover a large variety of security vulnerabilities, previously not detected by lab testing. We believe that our approach is of general interest when performing large-scale scans, not only in the context of TLS. One open question is how to identify vulnerable implementation versions and their vendors. In the SSH and IPsec protocols, these data are typically transmitted as message fields in the protocol.

Transmitting such data in TLS would make disclosure easier, but on the other hand would lead to privacy issues and easier fingerprinting.

Acknowledgments

We would like to thank Dennis Felsch who assisted us with our hardware and network infrastructure, and our anonymous reviewers for many insightful comments. Additionally, we would like to thank the Amazon, Citrix and OpenSSL teams for their professional responses and help with disclosure.

Nimrod Aviram was supported by a scholarship from The Israeli Ministry of Science and Technology, a scholarship from The Check Point Institute for Information Security, and a scholarship from The Yitzhak and Chaya Weinstein Research Institute for Signal Processing. Juraj Somorovsky was supported by the European Commission through the FutureTrust project (grant 700542-Future-Trust-H2020-DS-2015-1). Robert Merget was supported by the German Federal Ministry for Economic Affairs and Energy with initiative "IT-Sicherheit in der Wirtschaft", through the SIWECOS project.

References

[1] Openssl security advisory. CVE-2018-0733.

[2] ADRIAN, D., BHARGAVAN, K., DURUMERIC, Z., GAUDRY, P., GREEN, M., HALDERMAN, J. A., HENINGER, N., SPRINGALL, D., THOMÉ, E., VALENTA, L., ET AL. Imperfect forward secrecy: How diffie-hellman fails in practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 5–17.

[3] ALBRECHT, M. R., AND PATERSON, K. G. Lucky microseconds: A timing attack on amazon's s2n implementation of TLS. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I* (2016), pp. 622–643.

[4] ALFARDAN, N. J., AND PATERSON, K. G. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. *2013 IEEE Symposium on Security and Privacy 0* (2013), 526–540. `http://www.isg.rhul.ac.uk/tls/TLStiming.pdf`.

[5] AVIRAM, N., SCHINZEL, S., SOMOROVSKY, J., HENINGER, N., DANKEL, M., STEUBE, J., VALENTA, L., ADRIAN, D., HALDERMAN, J. A., DUKHOVNI, V., KÄSPER, E., COHNEY, S., ENGELS, S., PAAR, C., AND SHAVITT, Y. DROWN: Breaking TLS Using SSLv2. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, Aug. 2016), pp. 689–706.

[6] BARDOU, R., FOCARDI, R., KAWAMOTO, Y., STEEL, G., AND TSAY, J.-K. Efficient Padding Oracle Attacks on Cryptographic Hardware. In *Advances in Cryptology – CRYPTO* (2012), Canetti and R. Safavi-Naini, Eds.

[7] BENJAMIN, D. Tls ecosystem woes, Jan. 2018. Real World Crypto Symposium.

[8] BLEICHENBACHER, D. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In *Advances in Cryptology — CRYPTO '98*, vol. 1462 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1998.

[9] BÖCK, H., SOMOROVSKY, J., AND YOUNG, C. Return of bleichenbacher's oracle threat (ROBOT). In *27th USENIX Security Symposium (USENIX Security 18)* (Baltimore, MD, 2018), USENIX Association, pp. 817–849.

[10] BÖCK, H. A little POODLE left in GnuTLS (old versions), Nov. 2015. `https://blog.hboeck.de/archives/877-A-little-POODLE-left-in-GnuTLS-old-versions.html`.

[11] CANVEL, B., HILTGEN, A., VAUDENAY, S., AND VUAGNOUX, M. Password Interception in a SSL/TLS Channel. In *Advances in Cryptology - CRYPTO 2003*, vol. 2729 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, Aug. 2003.

[12] DIERKS, T., AND ALLEN, C. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), Jan. 1999. Obsoleted by RFC 4346, updated by RFCs 3546, 5746, 6176, 7465, 7507.

[13] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346 (Proposed Standard), Apr. 2006. Obsoleted by RFC 5246, updated by RFCs 4366, 4680, 4681, 5746, 6176, 7465, 7507.

[14] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685.

[15] DUONG, T., AND RIZZO, J. Cryptography in the web: The case of cryptographic design flaws in ASP.NET. In *IEEE Symposium on Security and Privacy* (2011).

[16] DURUMERIC, Z., ADRIAN, D., MIRIAN, A., BAILEY, M., AND HALDERMAN, J. A. A search engine backed by Internet-wide scanning. In *22nd ACM Conference on Computer and Communications Security* (Oct. 2015).

[17] DURUMERIC, Z., LI, F., KASTEN, J., AMANN, J., BEEKMAN, J., PAYER, M., WEAVER, N., ADRIAN, D., PAXSON, V., BAILEY, M., ET AL. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference* (2014), ACM, pp. 475–488.

[18] DURUMERIC, Z., WUSTROW, E., AND HALDERMAN, J. A. Zmap: Fast internet-wide scanning and its security applications.

[19] HENINGER, N., DURUMERIC, Z., WUSTROW, E., AND HALDERMAN, J. A. Mining your ps and qs: Detection of widespread weak keys in network devices.

[20] IRAZOQUI, G., INCI, M. S., EISENBARTH, T., AND SUNAR, B. Lucky 13 strikes back. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2015), ASIA CCS '15, ACM, pp. 85–96.

[21] JACOMY, M., VENTURINI, T., HEYMANN, S., AND BASTIAN, M. Forceatlas2, a continuous graph layout algorithm for handy network visualization designed for the gephi software. *PLOS ONE 9*, 6 (06 2014), 1–12.

[22] JAGER, T., SCHINZEL, S., AND SOMOROVSKY, J. Bleichenbacher's attack strikes again: breaking PKCS#1 v1.5 in XML Encryption. In *Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-14, 2012. Proceedings* (2012), S. Foresti and M. Yung, Eds., LNCS, Springer.

[23] JAGER, T., AND SOMOROVSKY, J. How To Break XML Encryption. In *The 18th ACM Conference on Computer and Communications Security (CCS)* (Oct. 2011).

[24] LABS, A. W. S. s2n: An implementation of the tls/ssl protocols.

[25] LANGLEY, A. The POODLE bites again, Nov. 2014. `https://www.imperialviolet.org/2014/12/08/poodleagain.html`.

[26] MEYER, C., SOMOROVSKY, J., WEISS, E., SCHWENK, J., SCHINZEL, S., AND TEWS, E. Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks. In *23rd USENIX Security Symposium, San Diego, USA* (August 2014).

[27] MÖLLER, B., DUONG, T., AND KOTOWICZ, K. This POODLE bites: exploiting the SSL 3.0 fallback, 2014.

[28] PETTERSSEN, Y. The POODLE has friends.

[29] PETTERSSEN, Y. There are more POODLEs in the forest.

[30] POPOV, A. Prohibiting RC4 Cipher Suites. RFC 7465 (Proposed Standard), Feb. 2015.

[31] RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, 2018.

[32] RIKU, ANTTI, MATTI, AND MEHTA. Heartbleed, cve-2014-0160, 2015. `http://heartbleed.com/`.

[33] RIZZO, J., AND DUONG, T. Practical padding oracle attacks. In *Proceedings of the 4th USENIX conference on Offensive technologies* (Berkeley, CA, USA, 2010), WOOT'10, USENIX Association, pp. 1–8.

[34] RIZZO, J., AND DUONG, T. Here Come The XOR Ninjas, May 2011.

[35] RONEN, E., PATERSON, K. G., AND SHAMIR, A. Pseudo constant time implementations of tls are only pseudo secure. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), ACM, pp. 1397–1414.

[36] SCHEITLE, Q., HOHLFELD, O., GAMBA, J., JELTEN, J., ZIMMERMANN, T., STROWES, S. D., AND VALLINA-RODRIGUEZ, N. A Long Way to the Top: Significance, Structure, and Stability of Internet Top Lists. In *Internet Measurement Conference (IMC'18), IMC'18 Community Contribution Award* (Boston, USA, Nov. 2018), ACM, pp. 478–493.

[37] SOMOROVSKY, J. Systematic fuzzing and testing of tls libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 1492–1504.

[38] VALENTA, L., SULLIVAN, N., SANSO, A., AND HENINGER, N. In search of curveswap: Measuring elliptic curve implementations in the wild. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)* (2018), IEEE, pp. 384–398.

[39] VAUDENAY, S. Security Flaws Induced by CBC Padding — Applications to SSL, IPSEC, WTLS... In *Advances in Cryptology — EUROCRYPT 2002*, vol. 2332 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, Apr. 2002.

The KNOB is Broken: Exploiting Low Entropy in the Encryption Key Negotiation Of Bluetooth BR/EDR

Daniele Antonioli
Singapore University of Technology and Design
daniele_antonioli@mymail.sutd.edu.sg

Nils Ole Tippenhauer
CISPA Helmholtz Center for Information Security
tippenhauer@cispa.saarland

Kasper Rasmussen
Department of Computer Science University of Oxford
kasper.rasmussen@cs.ox.ac.uk

Abstract

We present an attack on the encryption key negotiation protocol of Bluetooth BR/EDR. The attack allows a third party, without knowledge of any secret material (such as link and encryption keys), to make two (or more) victims agree on an encryption key with only 1 byte (8 bits) of entropy. Such low entropy enables the attacker to easily brute force the negotiated encryption keys, decrypt the eavesdropped ciphertext, and inject valid encrypted messages (in real-time). The attack is stealthy because the encryption key negotiation is transparent to the Bluetooth users. The attack is standard-compliant because all Bluetooth BR/EDR versions require to support encryption keys with entropy between 1 and 16 bytes and do not secure the key negotiation protocol. As a result, *the attacker completely breaks Bluetooth BR/EDR security without being detected*. We call our attack **Key Negotiation Of Bluetooth (KNOB)** attack.

The attack targets the firmware of the Bluetooth chip because the firmware (Bluetooth controller) implements all the security features of Bluetooth BR/EDR. As a standard-compliant attack, it is expected to be effective on any firmware that follows the specification and on any device using a vulnerable firmware. We describe how to perform the KNOB attack, and we implement it. We evaluate our implementation on more than 14 Bluetooth chips from popular manufacturers such as Intel, Broadcom, Apple, and Qualcomm. Our results demonstrate that all tested devices are vulnerable to the KNOB attack. We discuss countermeasures to fix the Bluetooth specification and its implementation.

1 Introduction

Bluetooth BR/EDR (referred for the rest of this paper as Bluetooth), is a short-range wireless technology widely used by many products such as mobile devices, laptops, IoT and industrial devices. Bluetooth provides security mechanisms to achieve authentication, confidentiality and data integrity at the link layer [6, p. 1646].

The security and privacy of Bluetooth has been attacked and fixed several times, going all the way back to Bluetooth v1.0. [15, 32]. Several successful attacks on the (secure simple) pairing phase [28, 13, 4] have resulted in substantial revisions of the standard. Attacks on Android, iOS, Windows and Linux implementations of Bluetooth were also discussed in [2]. However, little attention has been given to the security of the *encryption key negotiation protocol*, e.g., the Bluetooth security overview in the latest Bluetooth core specification (v5.0) does not mention it [6, p. 240].

The encryption key negotiation protocol is used by two Bluetooth devices to agree on the entropy of the link layer encryption key. Entropy negotiation was introduced in the specification of Bluetooth to cope with international encryption regulations and to facilitate security upgrades [6, p. 1650]. To the best of our knowledge, all versions of the Bluetooth standard (including the latest v5.0 [6]) *require* to use entropy values between 1 and 16 bytes. The specification of Bluetooth states this requirement as follows: "For the encryption algorithm, the key size may vary between 1 and 16 octets (8 - 128 bits)" [6, p. 1650]. Our interpretation of this requirement is that any device to be standard-compliant has to support encryption keys with entropy varying from one to sixteen bytes. The attack that we present in this work confirms our interpretation.

The encryption key negotiation protocol is conducted between two parties as follows: the initiator proposes an entropy value N that is an integer between 1 and 16, the other party either accepts it or proposes a lower value or aborts the protocol. If the other party proposes a lower value, e.g., $N-1$, then the initiator either accepts it or proposes a lower value or it aborts the protocol. At the end of a successful negotiation the two parties have agreed on the entropy value of the Bluetooth encryption key. The entropy negotiation is performed over the Link Manager Protocol (LMP), it is not encrypted and not authenticated, and it is transparent to the Bluetooth users because LMP packets are managed by the firmware of the Bluetooth chips and they are not propagated to higher layers [6, p. 508].

In this paper we describe, implement and evaluate an attack capable of making two (or more) victims using a Bluetooth encryption key with 1 byte of entropy without noticing it. The attacker then can easily brute force the encryption key, eavesdrop and decrypt the ciphertext and inject valid ciphertext without affecting the status of the target Bluetooth piconet. In other words, *the attacker completely breaks Bluetooth BR/EDR security without being detected*. We call this attack the **Key Negotiation Of Bluetooth (KNOB)** attack.

The KNOB attack can be conducted remotely or by maliciously modifying few bytes in one of the victim's Bluetooth firmware. Being a standard-compliant attack it is expected to be effective on any firmware implementing the Bluetooth specification, regardless of the Bluetooth version. The attacker is not required to posses any (pre-shared) secret material and he does not have to observe the pairing process of the victims. The attack is effective even when the victims use the strongest security mode of Bluetooth (Secure Connections). The attack is stealthy because the application using Bluetooth and even the operating systems of the victims cannot access or control the encryption key negotiation protocol (see Section 3.2 for the details).

After explaining the attack in detail, we implement it leveraging our development of several Bluetooth security procedures to generate valid link and encryption keys, and the InternalBlue toolkit [21]. Our implementation allows a man-in-the-middle attacker to intercept, manipulate, and drop LMP packets in real-time and to brute force low-entropy encryption keys, without knowing any (pre-shared) secret. We have disclosed our findings about the KNOB attack with CERT and the Bluetooth SIG, and following that, we plan to release our tools as open-source at `https://github.com/francozappa/knob`. This will enable other Bluetooth researchers to take advantage of our work.

We summarize our main contributions as follows:

- We develop an attack on the encryption key negotiation protocol of Bluetooth BR/EDR that allows to let two unaware victims negotiate a link-layer encryption key with 1 byte of entropy. The attacker then is able to brute force the low entropy key, decrypt all traffic and inject arbitrary ciphertext. The attacker does not have to know any secret material and he can target multiple nodes and piconets at the same time.

- We demonstrate the practical feasibility of the attack by implementing it. Our implementation involves a man-in-the-middle attacker capable of manipulating the encryption key negotiation protocol, brute forcing the key and decrypting the traffic exchanged by two (or more) unaware victims.

- All standard-compliant devices should be vulnerable to our attack, including the ones using the strongest Bluetooth security mode. In order to demonstrate that this problem has not somehow been fixed in practice, we test more than 14 different Bluetooth chips and find all of them to be vulnerable.

- We discuss what changes should be made, both to the Bluetooth standard and its implementation, in order to counter this attack.

Our work is organized as follows: in Section 2 we introduce the Bluetooth BR/EDR stack. In Section 3 we present the Key Negotiation Of Bluetooth (KNOB) attack. An implementation of the attack is discussed in Section 4. We evaluate the impact of our attack in Section 5 and we discuss the attack and our proposed countermeasures in Section 6. We present the related work in Section 7. We conclude the paper in Section 8.

2 Background

2.1 Bluetooth Basic Rate/Extended Data Rate

Bluetooth Basic Rate/Extended Data Rate (BR/EDR), also known as Bluetooth Classic, is a widely used wireless technology for low-power short-range communications maintained by the Bluetooth Special Interest Group(SIG) [6]. Its physical layer uses the same 2.4 GHz frequency spectrum of WiFi and (adaptive) frequency hopping to mitigate RF interference. A Bluetooth network is called a piconet and it uses a master-slave medium access protocol. There is always one master device per piconet at a time. The devices are synchronized by maintaining a reference clock signal, defined as CLK. Each device has a Bluetooth address (BTADD) consisting of a sequence of six bytes. From left to right, the first two bytes are defined as non-significant address part (NAP), the third byte as upper address part (UAP) and the last three bytes as lower address part (LAP).

To establish a secure Bluetooth connection two devices first have to pair. This procedure results in the establishment of a long-term shared secret defined as *link key*, indicated with K_L. There are four types of link key: initialization, unit, combination and master. A initialization key is always generated for each new pairing procedure. A unit key is generated from a device and utilized to pair with every other device, and its usage is not recommended because it is insecure. A combination key is generated using Elliptic Curve Diffie Hellman (ECDH) on the P-256 elliptic curve. This procedure is defined as Secure Simple Pairing (SSP) and it provides optional authentication of the link key. Combination keys are the most secure and widely used. A master key is generated only for broadcast encryption and it has limited usage. The master key is temporary, while the others are semi-permanent. A semi-permanent key can persist until a new link key is requested (link key is bonded) or it can change within the same session (link key is not bonded). In this paper we deal with combination link keys generated using authenticated SSP.

The specification of Bluetooth defines custom security procedures to achieve confidentiality, integrity and authentication. In the specification their names are prefixed with the letter E. In particular, a combination link key K_L is mutually authenticated by the E_1 procedure. This procedure uses a public nonce (AU_RAND) and the slave's Bluetooth address (BTADD_S) to generate two values: the Signed Response (SRES) and the Authenticated Ciphering Offset (ACO). SRES is used over the air to verify that two devices actually own the same K_L.

The symmetric encryption key K_C is generated using the E_3 procedure. When the link key is a combination key E_3 uses ACO (computed by E_1) as its Ciphering Offset Number (COF) parameter, together with K_L and a public nonce (EN_RAND). E_1 and E_3 use a custom hash function defined in the specification of Bluetooth with H. The hash function is based on SAFER+, a block cipher that was submitted as an AES candidate in 1998 [22].

Once the encryption key K_C is generated there are two possible ways to encrypt the link-layer traffic. If both devices support Secure Connections, then encryption is performed using a modified version of AES CCM. AES CCM is an authenticate-then encrypt cipher that combines Counter mode with CBC-MAC and it is defined in the IETF RFC 3610 [14]. As a side note, the specification of Bluetooth defines a message authentication codes (MAC) with the term message integrity check (MIC). If Secure Connections is not supported then the devices use the E_0 stream cipher for encryption. The cipher is derived from the Massey-Rueppel algorithm and it is described in the specification of Bluetooth [6, p. 1662]. E_0 requires synchronization between the master and the slaves of the piconet, this is achieved using the Bluetooth's clock value (CLK).

Modern implementations of Bluetooth provides the Host Controller Interface (HCI). This interface allows to separate the Bluetooth stack into two components: the host and the controller. Each component has specific responsibilities, i.e., the controller manages low-level radio and baseband operations and the host manages high-level application layer profiles. Typically, the host is implemented in the operating system and the controller in the firmware of the Bluetooth chip. For example BlueZ and Bluedroid implement the HCI host on Linux and Android, and the firmware of a Qualcomm or Broadcom Bluetooth chip implements the HCI controller. The host and the controller communicate using the Host Controller Interface (HCI) protocol. This protocol is based on commands and events, i.e., the host sends (acknowledged) commands to the controller, and the controller uses events to notify the host.

The Link Manager Protocol (LMP) is used over the air by two controllers to perform link set-up and control for Bluetooth BR/EDR. LMP is neither encrypted nor authenticated. The LMP packets do not propagate to higher protocol layers, hence, the hosts (OSes) are not aware about the LMP packets exchanged between the Bluetooth controllers.

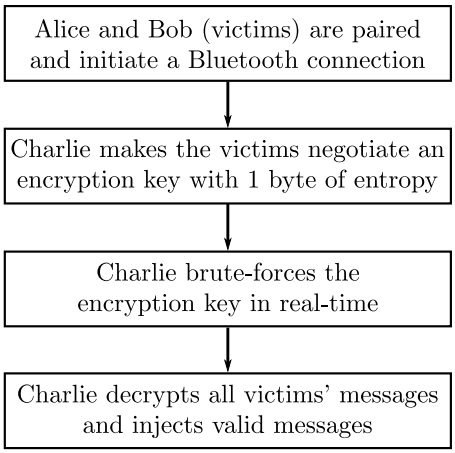


Figure 1: High level stages of a KNOB attack.

3 Exploiting Low Entropy in the Encryption Key Negotiation Of Bluetooth BR/EDR

In this section we describe the Key Negotiation Of Bluetooth (KNOB) attack. The attack allows Charlie (the attacker) to reduce the entropy of the encryption key of any Bluetooth BR/EDR (referred as Bluetooth) connection to 1 byte, without being detected by the victims (Alice and Bob). The attacker can brute force the encryption key without having to know any (pre-shared) secret material and without having to observe the Secure Simple Pairing protocol. As a result, the attacker can eavesdrop and decrypt all the traffic and inject arbitrary packets in the target Bluetooth network (piconet). The attack works regardless the usage of Secure Connections (the strongest security mode of Bluetooth). The KNOB attack high level stages are shown in Figure 1 and they are described in detail in the rest of this section.

3.1 System and Attacker Model

We assume a system composed of two or more legitimate devices that communicate using Bluetooth (as described in Section 2). One device is the master and the others are slaves. Without loss of generality, we focus on a piconet with one master and one slave (Alice and Bob). We indicate their Bluetooth addresses with BTADD_M and BTADD_S, and the Bluetooth clock with CLK. The clock is used for synchronization and it does not provide any security guarantee. The victims are capable of using Secure Simple Pairing and Secure Connections. This combination enables the highest security level of Bluetooth and should protect against eavesdropping and active man in the middle attacks. For example, if both devices have a display their users have to confirm that they see the same numeric sequence to mutually authenticate.

The attacker (Charlie) wants to decrypt all messages exchanged between Alice and Bob and inject valid encrypted messages, without being detected. The attacker has no access

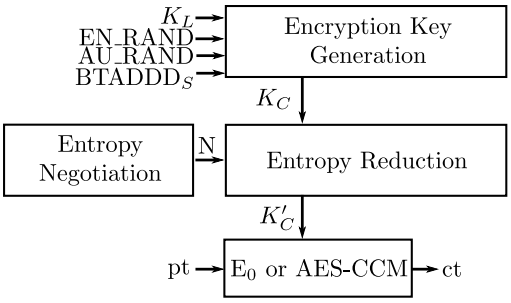


Figure 2: Generation and usage of the Bluetooth link layer encryption key (K'_C). Firstly, K_C is generated from K_L and other public parameters. K_C has 16 bytes of entropy, and it is not directly used as the encryption key. K'_C, the actual encryption key, is computed by reducing the entropy of K_C to N bytes. N is an integer between 1 and 16 and it is the result of the encryption key negotiation protocol. The N byte entropy K'_C is then used for link layer encryption by either the E_0 or the AES-CCM cipher.

to any (pre-shared) secret material. i.e., the link key K_L and the encryption key K_C. Charlie can observe the public nonces (EN_RAND and AU_RAND), the Bluetooth clock and the packets exchanged between Alice and Bob.

We define two attacker models: a *remote* attacker and a *firmware* attacker. A remote attacker controls a device that is in Bluetooth range with Alice and Bob. He is able to passively capture encrypted messages, actively manipulate unencrypted communication, and to drop packets using techniques such as network man-in-the-middle and manipulation of physical-layer signals [31, 26]. The firmware attacker is able to compromise the firmware of the Bluetooth chip of a single victim using techniques such as backdoors [7], supply-chain implants [12], and rogue chip manufacturers [27]. The firmware attacker requires no access to the Bluetooth host (OS) and applications used by the victims.

3.2 Negotiate a Low Entropy Encryption Key

Every time a Bluetooth connection requires link-layer encryption, Alice and Bob compute an encryption key K_C based on K_L, BTADD$_S$, AU_RAND, and EN_RAND (see top part of Figure 2). K_L is the link key established during secure simple pairing and the others parameters are public. Assuming ideal random number generation, the entropy of K_C is always 16 bytes.

K_C is not directly used as the encryption key for the current session. The actual encryption key, indicated with K'_C, is computed by reducing the entropy of K_C to N bytes. N is the outcome of the Bluetooth *encryption key negotiation protocol* (Entropy Negotiation in Figure 2). The protocol is part of the Bluetooth specification since version v1.0, and it was introduced to cope with international encryption regulations and

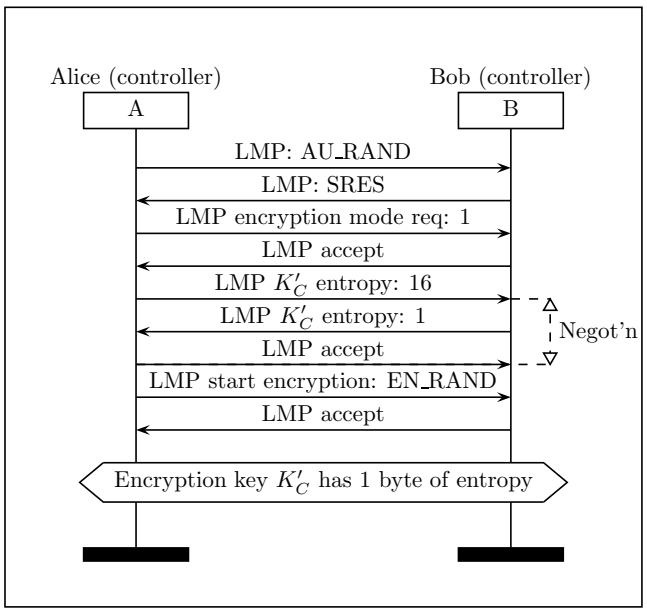


Figure 3: Alice and Bob negotiate 1 byte of entropy for the encryption key (K'_C). The protocol is run by Alice and Bob controllers (implemented in their Bluetooth chip) over the air using LMP.

to facilitate security upgrades [6, p. 1650]. The specification of the Bluetooth encryption key negotiation protocol contains three significant problems:

1. It allows to negotiate entropy values as low as 1 byte, regardless the Bluetooth security level.
2. It is neither encrypted nor authenticated.
3. It is implemented in the Bluetooth controller (firmware) and it is transparent to the Bluetooth host (OS) and to the user of a Bluetooth application.

Hence, an attacker (Charlie) can convince any two victims (Alice and Bob) to negotiate N equal to 1, the lowest possible, yet standard-compliant, entropy value. As a result the victims compute and use a Bluetooth encryption key (K'_C) with one byte of entropy. The victims (and their OSes) are not aware about the entropy reduction of K'_C because the negotiation happens between the victims' Bluetooth controller (firmware) and the packets do not propagate to the victims' Bluetooth host (OS).

To understand how an attacker can set N equal to 1 (or to any other standard-compliant value), we have to look at the details of the encryption key negotiation protocol. The protocol is run between the Bluetooth chip of Alice and Bob. In the following, we provide an example where Alice (the master) proposes 16 bytes of entropy, and Bob (the slave) is only able to support 1 byte of entropy (see Figure 3). The standard enables to set the minimum and maximum entropy

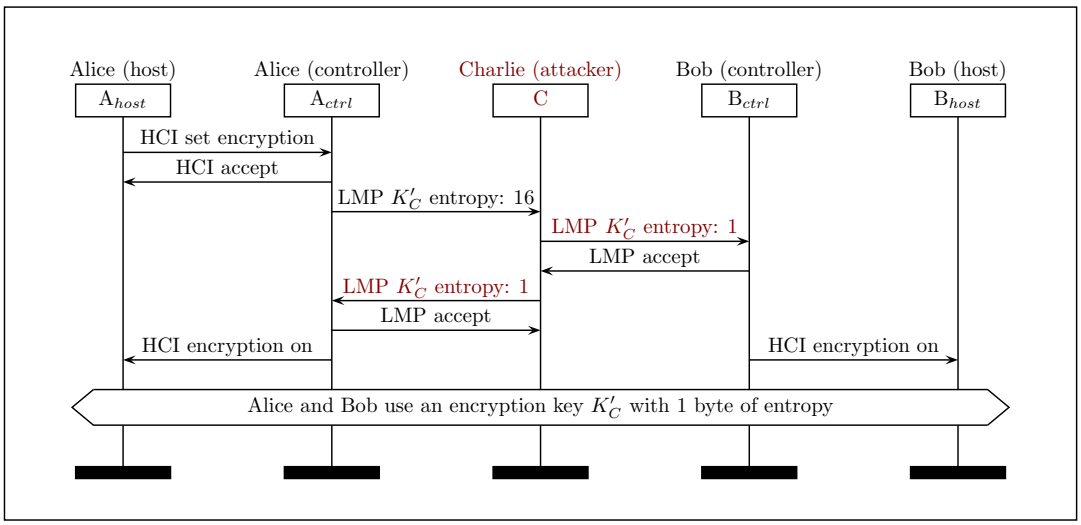


Figure 4: The KNOB attack sets the entropy of the encryption key (K'_C) to 1 byte. Alice requests Bob to activate encryption and starts the encryption key negotiation protocol. The attacker (Charlie) changes the entropy suggested by Alice from 16 to 1 byte. Bob accepts Alice's proposal and Charlie changes Bob's acceptance to a proposal of 1 byte. Alice, who originally proposed 16 bytes of entropy and she is asked to use 1 byte accepts the (standard-compliant) proposal. Charlie drops Alice's acceptance message because Bob already accepted Alice's proposal (modified by Charlie). Charlie does not know any pre-shared secret and does not observe SSP.

values by setting two parameters defined as L_{min} and L_{max}. These values can be set and read only by the Bluetooth chip (firmware). Indeed, our scenario describes a situation where Alice's Bluetooth firmware declares $L_{max} = 16$ and $L_{min} = 1$, and Bob's Bluetooth firmware declares $L_{max} = L_{min} = 1$.

The encryption key negotiation protocol is carried over the Link Manager Protocol (LMP). The first two messages in Figure 3 allow Alice to authenticate that Bob possesses the correct K_L. Then, with the next two messages, Alice requests to initiate Bluetooth link layer encryption and Bob accepts. Now, the negotiation of N takes place (Negot'n in Figure 3). Alice proposes 16 bytes of entropy. Bob can either propose a smaller value or accept the proposed one or abort the negotiation. In our example, Bob proposes 1 byte of entropy because it is the only value that he supports and Alice accepts it. Then, Alice requests to activate link-layer encryption and Bob accepts. Finally, Alice and Bob compute the same encryption key (K'_C) that has 1 byte of entropy. Note that, the Bluetooth hosts of Alice and Bob do not have access to K_C and K'_C, they are only informed about the outcome of the negotiation. The key negotiation procedure can also be initiated by the Bob (the slave), resulting in the same outcome.

Figure 4 describes how the attacker (Charlie) manages to let Alice and Bob agree on a K'_C with 1 byte of entropy when both Alice and Bob declare $L_{max} = 16$ and $L_{min} = 1$. In this Figure we also show the local interactions between hosts and controllers to emphasize that at the end of the negotiation the hosts are not informed about N and K'_C.

The attack is performed as follows: Alice's Bluetooth host requests to activate (set) encryption. Alice's Bluetooth controller accepts the local requests and starts the encryption key negotiation procedure with Bob's Bluetooth controller over the air. The attacker intercepts Alice's proposed key entropy and substitutes 16 with 1. This simple substitution works because LMP is neither encrypted nor integrity protected. Bob's controller accepts 1 byte. The attacker intercepts Bob's acceptance message and change it to an entropy proposal of 1 byte. Alice thinks that Bob does not support 16 bytes of entropy and accepts 1 byte. The attacker intercepts Alice' acceptance message and drops it. Finally, the controllers of Alice and Bob compute the same K'_C with one byte of entropy and notify their respective hosts that link-layer encryption is on.

It is reasonable to think that the victim could prevent or detect this attack using a proper value for L_{min}. However, the standard does not state how to explicitly take advantage of it, e.g., deprecate L_{min} values that are too low. The standard states the following: "The possibility of a failure in setting up a secure link is an unavoidable consequence of letting the application decide whether to accept or reject a suggested key size." [6, p. 1663]. This statement is ambiguous because it is not clear what the definition of "application" is in that sentence. As we show in Section 5, this ambiguity results in no-one being responsible for terminating connections with low entropy keys in practice. In particular, the entity who decides whether to accept or reject the entropy proposal is the firmware of the Bluetooth chip by setting L_{min} and L_{max} and participating in the entropy negotiation protocol. The

"application" (intended as the Bluetooth application running on the OS using the firmware as a service) cannot check and set L_{min} and L_{max}, and it is not directly involved in the entropy acceptance/rejection choice (that is performed by the firmware). The application can interact with the firmware using the HCI protocol. In particular, it can use the HCI Read Encryption Key Size request, to check the amount of negotiated entropy *after* the Bluetooth connection is established and theoretically abort the connection. This check is neither required nor recommended by the standard as part of the key negotiation protocol.

The low entropy negotiation presented in Figure 4 can be performed by both attacker models presented in Section 3.1. The remote attacker has the capabilities of dropping and injecting valid plaintext (the encryption key negotiation protocol is neither encrypted nor authenticated). The firmware attacker can modify few bytes in the Bluetooth firmware of a victim to always negotiate 1 byte of entropy. Furthermore, the negotiation is effective regardless of who initiates the protocol and the roles (master or slave) of the victims in the piconet.

3.3 Brute forcing the Encryption Key

Bluetooth has two link layer encryption schemes one is based on the E_0 cipher (legacy) and the other on the AES-CCM cipher (Secure Connections). Our KNOB attack works in both cases. If the negotiated entropy for the encryption key (K'_C) is 1 byte, then the attacker can trivially brute force it trying (in parallel) the 256 K'_C's candidates against one or more cipher texts. The attacker does not have to know what type of application layer traffic is exchanged, because a valid plaintext contains well known Bluetooth fields, such as L2CAP and RFCOMM headers, that the attacker can use as oracles.

We now describe how to compute all 1 byte entropy keys when E_0 and AES-CCM are in use. Each encryption mode involves a specific entropy reduction procedure that takes N and K_C as inputs and produces K'_C as output (Entropy Reduction in Figure 2). The specification of Bluetooth calls this procedure Encryption Key Size Reduction [6].

$$K'_C = g_2^{(N)} \otimes \left(K_C \bmod g_1^{(N)} \right) \qquad (E_s)$$

In case of E_0, K'_C is computed using Equation (E_s), where N is an integer between 1 and 16 resulted from the encryption key negotiation protocol (see Section 3.2). $g_1^{(N)}$is a polynomial of degree $8N$ used to reduce the entropy of K_C to N bytes. The result of the reduction is encoded with a block code $g_2^{(N)}$, a polynomial of degree less or equal to $128-8N$. The values of these polynomials depend on N and they are tabulated in [6, p. 1668]. If $N = 1$, then we can compute the 256 candidate K'_C by multiplying all the possible 1 byte reductions $K_C \bmod g_1^{(1)}$ (the set `0x00...0xff`) with $g_2^{(1)}$ (that equals to `0x00e275a0abd218d4cf928b9bbf6cb08f`).

In case of AES-CCM the entropy reduction procedure is simpler than the one of E_0. In particular, the $16-N$ least significant bytes of K_C are set to zero. For example, when $N = 1$ the 256 K'_C candidates for AES-CCM are the set `0x00...0xff`.

In the implementation of our KNOB attack brute force logic, we pre-compute the 512 keys with 1 byte of entropy and we store them in a look-up table to speed-up comparisons. Table 4 in Appendix A shows the first twenty K'_C with 1 byte of entropy for E_0 and AES-CCM. More details about the brute force implementation are discussed in Section 4.

3.4 KNOB Attack Implications

The Key Negotiation Of Bluetooth (KNOB) attack exploits a vulnerability at the architectural level of Bluetooth. The vulnerable encryption key negotiation protocol endangers potentially all standard compliant Bluetooth devices, regardless their Bluetooth version number and implementation details. We believe that the encryption key negotiation protocol has to be fixed as soon as possible.

In particular the KNOB attack has serious implications related to its *effectiveness*, *stealthiness*, and *cost*. The attack is effective because it exploits a weakness in the specification of Bluetooth. The Bluetooth security mode does not matter, i.e., the attack works even with Secure Connections. The implementation details do not matter, e.g., whether Bluetooth is implemented in hardware or in software. The time constraints imposed by the Bluetooth protocols do not matter because the attacker can eavesdrop the traffic and brute force the low-entropy key offline. The type of connection does not matter, e.g., the attack works with long-lived and short-lived connections. In a long-lived connection, e.g.,victims are a laptop and a Bluetooth keyboard, the attacker has to negotiate and brute force a single low-entropy K'_C. In a short-lived connection, e.g.,victims are two devices transferring files over Bluetooth, the attacker has to negotiate and brute force multiple low-entropy K'_C over time re-using the same attack technique without incurring in significant runtime and computational overheads.

The attack is stealthy because only the Bluetooth controllers (implemented in the victims' Bluetooth chip) are aware of N and K'_C. By design, the controllers are not notifying the Bluetooth hosts (implemented in the OSes) about N, but only about the outcome of the entropy negotiation. The users and the Bluetooth application developers are unaware of this problem because they use Bluetooth link-layer encryption as a trusted service.

The attack is cheap because it does not require a strong attacker model and expensive resources to be conducted. We expect that a remote attacker with commercial-off-the-shelf devices such as a software defined radio, GNU Radio and a laptop can conduct the attack.

3.5 KNOB Attack Root Causes

The root causes of the KNOB attack are shared between the specification and the implementation of Bluetooth BR/EDR confidentially mechanisms. On one side the specification is defining a vulnerable encryption key negotiation protocol that allows devices to negotiate low entropy values. On the implementation side (see Section 5), the Bluetooth applications that we tested are failing to check the negotiated entropy in practice. This is understandable because they are implementing a specification that is not mandating or explicitly recommending an entropy check.

We do not see any reason to include the encryption key negotiation protocol in the specification of Bluetooth. From our experiments (presented in Section 5) we observe that if two devices are not attacked they always use it in the same way (a device proposes 16 bytes of entropy and the other accepts). Furthermore, the entropy reduction does not improve runtime performances because the size of the encryption key is fixed to 16 bytes even when its entropy is reduced.

4 Implementation

We now discuss how we implemented the KNOB attack using a reference attack scenario. In particular, we explain how we manipulate the key negotiation protocol, brute force the encryption key (K'_C) using eavesdropped traffic, and validate K'_C by computing it from K_L as a legitimate device (as in Figure 2). In our attack scenario, the attacker is able to decrypt the content of a link-layer encrypted file sent from a Nexus 5 to a Motorola G3 using the Bluetooth OBject EXchange (OBEX) profile. A Bluetooth profile is the equivalent of an application layer protocol in the TCP/IP stack.

Our implementation required significant efforts mainly due to the lack of low-cost Bluetooth protocol analyzers and software libraries implementing the custom Bluetooth security primitives (such as the modified SAFER+ block cipher). Using our implementation we conducted successful KNOB attacks on more than 14 different Bluetooth chips, the attacks are evaluated in Section 5.

4.1 Attack Scenario

To describe our implementation we use an attack scenario with two victims a Nexus 5 and a Motorola G3, Table 1 lists their relevant specifications. The Nexus 5 is used also as a man-in-the-middle attacker by adding extra code to its Bluetooth firmware. This setup allows us to *simulate* a remote man-in-the-middle attacker (more details in Section 4.2). To perform eavesdropping, we use an Ubertooth One [24] with firmware version 2017-03-R2 (API:1.02). To the best of our knowledge, Ubertooth One does not capture all Bluetooth BR/EDR packets, but it is the only open-source, low-cost, and practical eavesdropping solution for Bluetooth that we know about. To brute force K'_C and decrypt the ciphertext we use a ThinkPad X1 laptop running a Linux based OS.

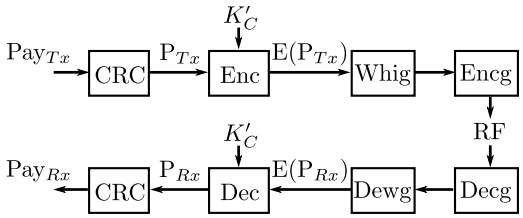


Figure 5: Transmission and reception of an E_0 encrypted payload. The concatenation of the payload and its CRC (P_{Tx}) is encrypted, whitened, encoded and then transmitted. On the receiver side the steps are applied in the opposite order. RF is the radio frequency wireless channel.

The victims use the following security procedures: Secure Simple Pairing to generate K_L (the link key) and authenticate the users, the entropy reduction function from Equation (E_s), and E_0 legacy encryption. The victims use legacy encryption because the Nexus 5 does not support Secure Connections. Nevertheless, the KNOB attack works also with Secure Connections.

Every E_0-encrypted packet that contains data is transmitted and received as in Figure 5. A cyclic redundancy checksum (CRC) is computed and appended to the payload (Pay_{Tx}). The resulting bytes (P_{Tx}) are encrypted with E_0 using K'_C. The ciphertext is whitened, encoded, and transmitted over the air. On the receiver side the following steps are applied in sequence: decoding, de-whitening, decryption, and CRC check. The encryption and decryption procedures are the same because E_0 is a stream cipher, i.e., the same keystream is XORed with the plaintext and the ciphertext. Whitening and encoding procedures do not add any security guarantee and the Ubertooth One is capable of performing both procedures.

4.2 Manipulation of the Entropy Negotiation

We implement the manipulation of the encryption key negotiation protocol (presented in Section 3.2) by extending the functionalities of InternalBlue [21] and using it to patch the Bluetooth chip firmware of the Nexus 5. Our InternalBlue modifications allow to manipulate all incoming LMP messages *before* they are processed by the entropy negotiation logic, and all outgoing LMP messages *after* they've been processed by the entropy negotiation logic. The entropy negotiation logic is the code in the Nexus 5 Bluetooth firmware that manages the encryption key negotiation protocol, and we do not modify it. As a result, we can use a Nexus 5 (or any other device supported by InternalBlue) as a victim and a remote KNOB attacker without having to deal with the practical issues related with wireless attacks over-the-air.

InternalBlue is an open-source toolkit capable of interfacing with the firmware of the BCM4339 Bluetooth chip in

		Bluetooth			
Phone	OS	Version	MAC	SC	Chip
Nexus 5	Android 6.0.1	4.1	48:59:29:01:AD:6F	No	Broadcom BCM4339
Motorola G3	Android 6.0.1	4.1	24:DA:9B:66:9F:83	Yes	Qualcomm Snapdragon 410

Table 1: Relevant technical specifications of Nexus 5 and Motorola G3 devices used to describe our attack implementation. The SC column indicates if a device supports Secure Connections.

Nexus 5 phones. To use it, one has to root the target Nexus 5 and compile and install the Android Bluetooth stack with debugging features enabled. InternalBlue allows to patch the firmware in real-time (e.g., start LMP monitoring) and read the ROM and the RAM of firmware at runtime. InternalBlue provides a way to hook and execute arbitrary code in the Bluetooth firmware. At the time of writing, InternalBlue is not capable of hooking directly the key negotiation logic. However, we managed to extend it to enable two victims (one is always the Nexus 5) to negotiate one (or more) byte of entropy.

Our manipulation of the entropy negotiation works regardless the role of the Nexus 5 in the piconet and it does not require to capture any information about the Secure Simple Pairing process. Assuming that the victims are already paired, we test if two victims are vulnerable to the KNOB attack as follows:

1. We connect over USB the Nexus 5 with the X1 laptop, we run our version of InternalBlue, and we activate LMP and HCI monitoring.

2. We connect and start the Ubertooth One capture over the air focusing only on the Nexus 5 piconet (using UAP and LAP flags).

3. We request a connection from the Nexus 5 to the victim (or vice versa) to trigger the encryption key negotiation protocol over LMP.

4. Our InternalBlue patch changes the LMP packets as Charlie does in Figure 4.

5. If the victims successfully complete the protocol, then they are vulnerable to the KNOB attack and we can decrypt the ciphertext captured with the Ubertooth One.

We now describe how we extended InternalBlue to perform the fourth step of the list. In this context, the most important file of InternalBlue is `internalblue/fw_5.py`. This file contains all the information about the BCM4339 firmware, and it provides two hooks into the firmware, defined by Mantz (the main author of InternalBlue) as `LMP_send_packet` and `LMP_dispatcher`. The former hook allows to execute code every time an LMP packet is about to be sent and the latter whenever an LMP packet is received. The hooks are intended for LMP monitoring, and we upgraded them to be used also for LMP manipulation.

Listing 1 shows three ARM assembly code blocks that we added to `fw_5.py` to let the Nexus 5 and the Motorola G3 negotiate 1 byte of entropy. In this case the Nexus 5 is the master and it initiates the encryption key negotiation protocol. The first block translates to: if the Nexus 5 is sending an LMP K'_C entropy proposal then change it to 1 byte. This block is executed when the Nexus 5 starts an encryption key negotiation protocol. The code allows to propose any entropy value by `moving` a different constant into `r2` in line 5.

The second block from Listing 1 translates to: if the Nexus 5 is receiving an LMP accept (entropy proposal), then change it to an LMP K'_C entropy proposal of 1 byte. This code is used to let the Nexus 5 firmware believe that the other victim proposed 1 byte, while she already accepted 1 byte (assuming that she is vulnerable). The third blocks translates to: if the Nexus 5 is sending an LMP accept (entropy proposal), then change it to an LMP preferred rate. This allows to obtain the same result of dropping an LMP accept packet because the LMP preferred rate packet does not affect the state of the encryption key negotiation protocols. We developed and used similar patches to cover the other attack cases: Nexus 5 is the master and does not initiate the connection, Nexus 5 is the slave and initiates the connection and Nexus 5 is the slave and does not initiate the connection.

4.3 Brute Forcing the Encryption Key

Once the attacker is able to reduce the entropy of the encryption key (K'_C) to 1 byte, he has to brute force the key value (key space is 256). In this section we explain how we brute forced and validated a E_0 encryption key with 1 byte of entropy. The key was used in one of our KNOB attacks to decrypt the content of a file transferred over a link layer encrypted Bluetooth connection.

The details about the E_0 encryption scheme are presented in Figure 6, we describe them backwards starting from the E_0 cipher. E_0 takes three inputs: BTADD_M, CLK26-1 and K'_C. CLK26-1 are the 26 bits of CLK in the interval CLK[25:1] (assuming that CLK stores its least significant bit at CLK[0]). The BTADD_M is the Bluetooth address of the master and it

Listing 1 We add three ARM assembly code blocks to `internalblue/fw_5.py` to negotiate K'_C with 1 byte of entropy. In this case the Nexus 5 is the master and it initiates the encryption key negotiation protocol.

```
# Send LMP Kc' entropy 1 rather than 16
ldrb  r2, [r1]
cmp   r2, #0x20
bne   skip_sent_ksr
mov   r2, #0x01
strb  r2, [r1, #1]
skip_sent_ksr:

# Recv LMP Kc' entropy 1 rather than LMP accept
ldrb   r2, [r1]
cmp    r2, #0x06
bne    skip_recv_aksr
ldrb   r2, [r1, #1]
cmp    r2, #0x10
bne    skip_recv_aksr
mov    r2, #0x20
strb   r2, [r1]
mov    r2, #0x01
strb   r2, [r1, #1]
skip_recv_aksr:

# Send LMP_preferred rate rather than LMP accept
# Simulate an attacker dropping LMP accept
ldrb  r2, [r1]
cmp   r2, #0x06
bne   skip_send_aksr
ldrb  r2, [r1, #1]
cmp   r2, #0x10
bne   skip_send_aksr
mov   r2, #0x48
strb  r2, [r1]
mov   r2, #0x70
strb  r2, [r1, #1]
skip_send_aksr:
```

is a public parameter. We did not have to implement the E_0 cipher because we found an open-source implementation [8] which we verified against the specification of Bluetooth. To provide valid K'_C candidates to E_0 we had to implement the E_s entropy reduction procedure. This procedure takes an input with 16 bytes of entropy (K_C) and computes an output with N bytes of entropy (K'_C). E_s involves modular arithmetic over polynomials in Galois fields and we use the BitVector [16] Python module to perform such computations.

Our Python brute force script takes a ciphertext (captured over the air using Ubertooth One) and tries to decrypt it by using the E_0 cipher with all possible values of K'_C. We validate our script by decrypting the content of a file sent from the Nexus 5 to the Motorola G3 using the OBEX Bluetooth profile after the negotiation of 1 byte of entropy. The content of the file (in ASCII) is `aaaabbbbccccdddd`. We discuss several brute forcing practical issues in Section 6.3.

Once we found the matching plaintext we wanted to verify that the brute forced key was effectively the one in use by the victims. To do that we had to implement E_1 and E_3, the former

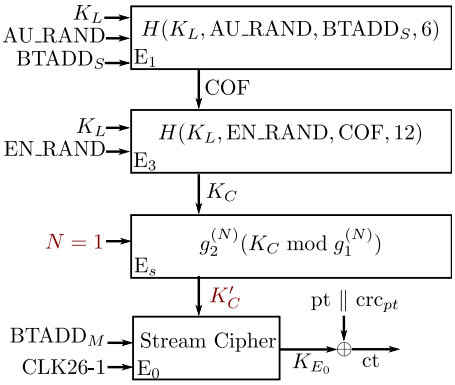


Figure 6: Implementation of the KNOB attack on the E_0 cipher. The attacker makes the victims agree on a K'_C with one byte of entropy ($N = 1$) and then brute force K'_C, without knowing K_L and K_C.

is used to compute the Ciphering Offset Number (COF), the latter to compute K_C (see Figure 6). Both procedures use a custom hash function defined in the specification of Bluetooth with H. We write E_1 and E_3 equations and label them with their respective names as follows:

$$SRES\|ACO = H(K_L, \text{AU_RAND}, \text{BTADD}_S, 6) \quad (E_1)$$
$$K_C = H(K_L, \text{EN_RAND}, \text{COF}, 12) \quad (E_3)$$

Figure 7 shows how E_3 uses the H hash function, H internally uses SAFER+, a block cipher that was submitted as an AES candidate in 1998 [22]. SAFER+ is used with 128 bit block size (8 rounds), in ECB mode, and only for encryption. SAFER+' (SAFER+ prime) is a modified version of SAFER+ such that the input of the first round is added to the input of the third round. This modification was introduced in the specification of Bluetooth to avoid SAFER+' being used for encryption [6, p. 1677].

We implemented in Python both SAFER+ and SAFER+' including the round computations and the key scheduling algorithm. We tested the two against the specification of Bluetooth (where they are indicated with A_r and A_r' [6, p. 1676]). We also implemented the E and O blocks from Figure 7. The E block is an extension block that transforms the 12 byte COF into a 16 byte sequence using modular arithmetic. The same block is applied to the 6 byte BTADD$_S$ in E_1. The O block is offsetting K_L using algebraic (modular) operations and the largest primes below 257 for which 10 is a primitive root. We implement the E and O blocks in Python and we tested them against the specification of Bluetooth. Then, we were able to implement H and to use it to implement and test E_3 and E_1.

We validate the brute forced K'_C by using the necessary parameters from Figure 6 to compute K'_C from K_L. We captured the parameters using the Bluetooth logging capabilities offered by Android. Table 2 shows an example of actual public and private values used during one of our KNOB attacks. We

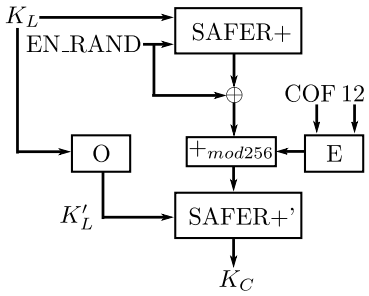


Figure 7: Bluetooth defines H a custom hash function based on SAFER+. H is used to compute K_C from K_L, EN_RAND, and COF (see Equation E_3).

Name	Value
Public	
$BTADD_M$	`0xccfa0070dcb6`
$BTADD_S$	`0x829f669bda24`
AU_RAND	`0x722e6ecd32ed43b7f3cdbdc2100ff6e0`
EN_RAND	`0xd72fb4217dcdc3145056ba488bea9076`
SRES	`0xb0a3f41f`
N	`0x1`
Secret	
K_L	`0xd5f20744c05d08601d28fa1dd79cdc27`
COF=ACO	`0x1ce4f9426dc2bc110472d68e`
K_C	`0xa3fccef22ad2232c7acb01e9b9ed6727`
K'_C	`0x7fffffffffffffffffffffffffffffff`

Table 2: Public and secret values (in hexadecimal representation) collected during a KNOB attack involving authenticated SSP and E_0 encryption. The encryption key (K'_C) has 1 byte of entropy.

plan to release our code implementing E_s, E_1 and E_3 as open-source to help researchers interested in Bluetooth's security, after we complete the responsible disclosure of our findings[1].

4.4 Implementation for Secure Connections

The specification of Bluetooth allows to perform the KNOB attack even when the victims are using Secure Connections. We already implemented the entropy reduction function of the brute force script over AES–CCM. However, at the time of writing, InternalBlue is not capable of patching the firmware of a Bluetooth chip that supports Secure Connections, indeed we are not able to implement the low entropy negotiation part of the attack using InternalBlue.

[1] See `https://github.com/francozappa/knob`

5 Evaluation

Our implementation of the KNOB attack (presented in Section 4) allows to test if any device accepts an encryption key with 1 byte of entropy ($N = L_{min} = 1$). We focus our discussion on the attack best case (1 byte of entropy) while arguably any entropy value lower than 14 bytes could be considered not secure for symmetric encryption [3].

After successfully conducting the KNOB attack on a Nexus 5 and a Motorola G3 we conducted other KNOB attacks on more than 14 unique Bluetooth chips (by attacking 21 different devices). Each attack is easy to reproduce and testing if a device is vulnerable is a matter of seconds.

Based on our experiments, we concluded that there are no differences between the specification and the implementation of both the Bluetooth controller (implemented in the firmware) and the Bluetooth host (implemented in the OS and usable as an interface by a Bluetooth application). In the former case the specification is not enforcing any minimum L_{min} and it is not protecting the entropy negotiation protocol. The firmware's implementers (to provide standard-compliant products) are allowing the negotiation of 1 byte of entropy with an insecure protocol. The only exception is the Apple W1 chip where an attacker can only reduce the entropy to 7 bytes. In the latter case, the Bluetooth specification is providing an HCI Read Encryption size API but it is not mandating or recommending its usage, e.g., a mandatory check at the end of the LMP entropy negotiation. The host's implementers are providing this API and the applications that we tested are not using it.

5.1 Evaluation Setup

To perform our evaluation we collected as many devices as possible containing different Bluetooth chips. At the time of writing, we were able to test chips from Broadcom, Qualcomm, Apple, Intel, and Chicony manufacturers. For each chip we conducted the KNOB attack following the same five steps presented in Section 4.2. As explained earlier, the Nexus 5 is used as a (remote) attacker and a victim. For each test we recorded the manipulated encryption key negotiation protocol over LMP in a pcapng file and we manually verified the protocol's outcome with Wireshark.

Our evaluation setup is not hard to reproduce and easy to extend because it does not require expensive hardware and uses open-source software. We would like to see other researchers evaluating more Bluetooth chips and devices that currently we do not posses, e.g., Apple Watches.

5.2 Evaluation Results

Table 3 shows our evaluation results. Overall, we tested more than 14 Bluetooth chips and 21 devices. The first column contains the Bluetooth chip names. We fill the entries of this

Bluetooth chip	Device(s)	Vuln?
Bluetooth Version 5.0		
Snapdragon 845	Galaxy S9	✓
Snapdragon 835	Pixel 2, OnePlus 5	✓
Apple/USI 339S00428	MacBookPro 2018	✓
Apple A1865	iPhone X	✓
Bluetooth Version 4.2		
Intel 8265	ThinkPad X1 6th	✓
Intel 7265	ThinkPad X1 3rd	✓
Unknown	Sennheiser PXC 550	✓
Apple/USI 339S00045	iPad Pro 2	✓
BCM43438	RPi 3B, RPi 3B+	✓
BCM43602	iMac MMQA2LL/A	✓
Bluetooth Version 4.1		
BCM4339 (CYW4339)	Nexus 5, iPhone 6	✓
Snapdragon 410	Motorola G3	✓
Bluetooth Version $\leq$ 4.0		
Snapdragon 800	LG G2	✓
Intel Centrino 6205	ThinkPad X230	✓
Chicony Unknown	ThinkPad KT-1255	✓
Broadcom Unknown	ThinkPad 41U5008	✓
Broadcom Unknown	Anker A7721	✓
Apple W1	AirPods	*

Table 3: List of Bluetooth chips and devices tested against the KNOB attack. ✓indicates that a chip accepts one byte of entropy. * indicates that a chip accepts at least seven bytes of entropy. We note that, all chips and devices implementing any specification of Bluetooth are expected to be vulnerable to the KNOB attack because the entropy reduction feature is standard-compliant.

column with Unknown when we are not able to find information about the chip manufacturer and/or model number. The second column lists the devices that we tested grouped by chip, e.g., the Snapdragon 835 is used both by the Pixel 2 and the OnePlus 5. The third column contains a ✓ if the Bluetooth chip accepts 1 byte of entropy and a * if it accepts at least 7 bytes. The table's rows are grouped by Bluetooth version in four blocks: version 5.0, version 4.2, version 4,1 and version lower or equal than 4.0.

From the third column of Table 3 we see that all the chips accept 1 byte of entropy (✓) except the Apple W1 chip (*) that requires at least 7 bytes of entropy. Apple W1 and its successors are used in devices such as AirPods, and Apple Watches. Seven bytes of entropy are better than one, but not enough to prevent brute force attacks. For example, the Data Encryption Standard (DES) uses the same amount of entropy and DES keys were brute forced multiple times with increasing efficacy [19].

Table 3 also demonstrates that the vulnerability spans across different Bluetooth versions including the latest ones such as 5.0 and 4.2. This fact confirms that the KNOB attack is a significant threat for all Bluetooth users and we believe that the specification of Bluetooth has to be fixed as soon as possible.

6 Discussion

6.1 Attacking Other Bluetooth Profiles

Cable replacement wireless technologies such as Bluetooth are widely used for all sorts of applications including desktop, mobile, IoT, industrial and medical devices. Bluetooth defines its set of application layer services as profiles. In Section 4 we describe an attack on the OBject EXchange (OBEX) Bluetooth profile, where the attacker breaks Bluetooth security by decrypting the content of an encrypted file without having access to any (pre-shared) secret. Here we describe three KNOB attacks targeting other popular Bluetooth profiles. As in the OBEX case, the attacks have serious implications in terms of security and privacy of the victims. To the best of our knowledge, all the profiles that we discuss in this section rely only on the link-layer for their security guarantees and they are widely used across different vendors. Our list of attacks is not exhaustive and an attacker might exploit the vulnerable encryption key negotiation protocol of Bluetooth in other creative ways.

HID profile The attacker could perform a remote keylogging attack on any device that uses the Human Interface Device (HID) profile. This profile is used by input-output devices such as keyboards, mice and joysticks. As a result, the attacker can sniff sensitive information including passwords, credit card numbers, and emails regardless if these information are then encrypted on the (wired or wireless) Ethernet link.

Bluetooth tethering The attacker could mount a remote man-in-the-middle attack when the victim uses Bluetooth for tethering. Tethering is used by a device, acting as an hotspot, to share Internet connectivity with other devices in range. Bluetooth transports Ethernet over the Bluetooth Network Encapsulation Protocol (BNEP) [5]. This protocol encapsulates Ethernet frames and transports them over (link-layer encrypted) L2CAP. As a result, the attacker can sniff all Internet traffic of the victims using a Bluetooth hotspot.

A2DP profile The attacker could record and inject audio signals when the victim uses the Advanced Audio Distribution Profile (A2DP) profile. As a result, the attacker is able to record phone and Voice over IP (VoIP) calls even if the call is encrypted (e.g., 4G and Skype). The attacker can also tamper with voice commands sent to a personal assistant, e.g.,

Siri and Google Assistant. Recent mobile devices, such as smartphone and tablets, are particularly vulnerable to this threat because Bluetooth is a convenient solution to the lack of an analog audio connector (audio jack).

6.2 Attacking Multiple Nodes and Piconets

In our paper we describe the implementation of KNOB attacks targeting two victims. If a Bluetooth piconet contains more than two devices, then (in the worst case for the attacker) each master-slave pair uses a dedicated set of keys. In this scenario the KNOB attack still works because it can be parallelized with minimal effort. For example, the attacker may run the same attack script on different computing units, such as processes or machines, and let each computing unit target a master-slave pair. Each parallel instance of the attack negotiates an encryption key with one byte of entropy, captures the exchanged ciphertext, and brute forces the encryption key. For example, an attacker is able to decrypt all the traffic from a victim using multiple Bluetooth I/O devices to interact with his device e.g., a laptop connected with a keyboard, a mouse and an headset.

The KNOB attack is effective even if the attacker wants to target multiple piconets (Bluetooth networks) at the same time. In this case the attacker has to follow and use a different Bluetooth clock (CLK) value for each piconet to compute the correct encryption key. This is not a problem because the attacker can use parallel KNOB attack instances, where each instance follows a pair of devices in a target piconet.

6.3 Practical Implementation Issues

We spent considerable time to fine tune our brute force script. One main reason is that Ubertooth One, used to sniff Bluetooth BR/EDR packets over the air, does not reliably capture all packets and clock values (CLK). This is true even if we limit our capture to a specific piconet by setting the UAP and LAP parameters. As a result, we had to include extra logic in our brute force script to iterate over different CLK values and E_0 keystream offsets. Our basic brute force logic only iterates over the encryption key space (256 iterations). The extra logic can be removed if we get access to a commercial-grade Bluetooth protocol analyzer such as Ellisys [10] or similar. Unfortunately, these devices are very expensive.

We implemented our attack by simulating a remote attacker using InternalBlue. Alternatively, we could have conducted the attacks over the air using signal manipulation [26] and (reactive) jamming [31]. However, the InternalBlue setup is simpler, more reliable, cheaper, and easier to reproduce than the over-the-air setup and it affects the victims in the same way as a remote attacker.

6.4 Countermeasures

In this section we propose several countermeasures to the KNOB attack. We divide them into two classes: legacy compliant and non legacy compliant. The former type of countermeasure does not require a change to the specification of Bluetooth while the latter does. We already proposed these countermeasures to the Bluetooth SIG and CERT during our responsible disclosure.

Legacy compliant. Our first proposed legacy compliant countermeasure is to require a minimum and maximum amount of negotiable entropy that cannot be easily brute forced, e.g., require 16 bytes of entropy. This means fixing L_{min} and L_{max} in the Bluetooth controller (firmware) and results in the negotiation of proper encryption keys. Another possible countermeasure is to automatically have the Bluetooth host (OS) check the amount of negotiated entropy each time link layer encryption is activated and abort the connection if the entropy does not meet a minimum requirement. The entropy value can be obtained by the host using the HCI Read Encryption Key Size Command. This solution requires to modify the Bluetooth host and it might be suboptimal because it acts on a connection that is already established (and possibly in use), not as part of the entropy negotiation protocol. A third solution is to distrust the link layer and provide the security guarantees at the application layer. Some vendors have done so by adding a custom application layer security mechanism on top of Bluetooth (which, in case of Google Nearby Connections, was also found to be vulnerable [1]).

Non legacy compliant. A non legacy compliant countermeasure is to modify the encryption key negotiation protocol by securing it using the link key. The link key is a shared (and possibly authenticated) secret that should be always available before starting the entropy negotiation protocol. The new protocol should provide message integrity and might also provide confidentiality. Preferably, the specification should get rid of the entropy negotiation protocol and always use encryption keys with a fixed amount of entropy, e.g., 16 bytes. The implementation of these solutions only requires the modification of the Bluetooth controller (firmware).

7 Related Work

The security and privacy guarantees of Bluetooth were studied since Bluetooth v1.0 [15, 32]. Particular attention was given to Secure Simple Pairing (SSP), a mechanisms that Bluetooth uses to generate and share a long term secret (defined as the link key). Several attacks on the SSP protocol were proposed [28, 13, 4]. The Key Negotiation Of Bluetooth (KNOB) attack works regardless of security guarantees provided by SSP (such as mutual user authentication).

The most up to date survey about Bluetooth security was provided by NIST in 2017 [25]. This survey recommends to use 128 bit keys (16 bytes of entropy). It also describes the key negotiation protocol, and considers it as a security issue when one of the connected devices is malicious (and not a third party). Prior surveys do not consider the problem of encryption key negotiation at all [9] or superficially discuss it [29].

The various implementation of Bluetooth were also analyzed and several attacks were presented on Android, iOS, Windows and Linux implementations [2]. Our attack works regardless of the implementation details of the target platform, because if any implementation is standard-compliant then it is vulnerable to the KNOB attack.

The security of the ciphers used by Bluetooth has been extensively discussed by cryptographers. The SAFER+ cipher used by Bluetooth for authentication purposes was analyzed [17]. The E_0 cipher used by Bluetooth for encryption was also analyzed [11]. Our attack works even with perfectly secure ciphers. For our implementation of the custom Bluetooth security procedures (presented in Section 4) we used as main references the specification of Bluetooth [6] and third-party hardware [18] and software [20] implementations.

Third-party manipulations of key negotiation protocols were also discussed in the context of WiFi, for example key reuse in [30]. Compared to those attacks, our attack exploits not only implementation issues, but a standard-compliant vulnerability of the specification of Bluetooth.

Protocol downgrade attacks were discussed in the context of TLS[23], where the two parties are negotiating the cipher suite to use. We note that in contrast to our scenario, for TLS the application developers have commonly direct control over the cipher suites that will be offered by their applications. Therefore, avoiding a fallback to legacy encryption standards can be prevented by the developers. To the best of our knowledge, this is not the case for Bluetooth, as the protocols does not enforce any mandatory checks on the encryption key's entropy.

8 Conclusion

In this paper we present the Key Negotiation Of Bluetooth (KNOB) attack. Our attack is capable of reducing the entropy of the encryption key of any Bluetooth BR/EDR connection to 1 byte (8 bits). The attack is standard-compliant because the specification of Bluetooth includes an insecure encryption key negotiation protocol that supports entropy values between 1 and 16 bytes. As a main consequence, an attacker can easily negotiate an encryption key with low entropy and then brute force it. The attacker is effectively breaking the security guarantees of Bluetooth without having to posses any (pre-shared) secret material. The attack is stealthy because the vulnerable entropy negotiation protocol is run by the victims' Bluetooth controller and this protocol is transparent to the Bluetooth host (OS) and the Bluetooth application used by the victims. We expect that the attack could be run in parallel to target multiple devices and piconets at the same time.

We demonstrate that the KNOB attack can be performed in practice by implementing it to attack a Nexus 5 and a Motorola G3. In our attack we decrypt a file transmitted over an authenticated and link-layer encrypted Bluetooth connection. Brute-forcing a key with 1 byte of entropy introduces a negligible overhead enabling an attacker to decrypt all the ciphertext and to introduce valid ciphertext even in real-time.

We evaluate the KNOB attack on more than 14 Bluetooth chips from different vendors such as Broadcom, Qualcomm and Intel. All the chips accept 1 byte of entropy except the Apple W1 chip that accepts (at least) 7 bytes of entropy. Frankly, we were expecting to find more non standard-compliant chips like the Apple W1. Before submitting the paper, we reported our findings to the Computer Emergency Response Team (CERT) and the Bluetooth Special Interest Group (SIG). Both organizations acknowledged the problem and we are collaborating with them to solve it. After our responsible disclosure, we plan to release the tools that we developed to implement the attacks as open-source.

The KNOB attack is a serious threat to the security and privacy of all Bluetooth users. We were surprised to discover such fundamental issues in a widely used and 20 years old standard. We attribute the identified issues in part to ambiguous phrasing in the standard, as it is not clear who is responsible for enforcing the entropy of the encryption keys, and as a result no-one seems to be responsible in practice. We urge the Bluetooth SIG to update the specification of Bluetooth according to our findings. Until the specification is not fixed, we do not recommend to trust any link-layer encrypted Bluetooth BR/EDR link. In Section 6.4 we propose legacy and non legacy compliant countermeasures that would make the KNOB attack impractical. We also recommend the Bluetooth SIG to create a dedicated procedure enabling researchers to securely submit new potential vulnerabilities, similarly to what other companies, such as Google, Microsoft and Facebook, are offering.

References

[1] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. Nearby Threats: Reversing, Analyzing, and Attacking Google's "Nearby Connections" on Android. In *Network and Distributed System Security Symposium (NDSS)*, February 2019.

[2] Armis Inc. The Attack Vector BlueBorne Exposes Almost Every Connected Device. `https://armis.com/blueborne/`, Accessed: 2018-01-26.

[3] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Recommendation for key man-

agement part 1: General (revision 3). *NIST special publication*, 800(57):1–147, 2012.

[4] Eli Biham and Lior Neumann. Breaking the bluetooth pairing–fixed coordinate invalid curve attack. http://www.cs.technion.ac.il/~biham/BT/bt-fixed-coordinate-invalid-curve-attack.pdf, Accessed: 2018-10-30.

[5] Bluetooth SIG. Bluetooth Network Encapsulation Protocol. http://grouper.ieee.org/groups/802/15/Bluetooth/BNEP.pdf, Accessed: 2018-10-28, 2001.

[6] Bluetooth SIG. Bluetooth Core Specification v5.0. https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=421043, Accessed: 2018-10-28, 2016.

[7] Bob Cromwell. The Problem With Government-Imposed Backdoors. https://cromwell-intl.com/cybersecurity/backdoors.html, Accessed: 2019-2-4.

[8] Arnaud Delmas. A C implementation of the Bluetooth stream cipher E0. https://github.com/adelmas/e0, Accessed: 2018-10-28.

[9] John Dunning. Taming the blue beast: A survey of bluetooth based threats. *IEEE Security & Privacy*, 8(2):20–27, 2010.

[10] Ellisys. Ellisys protocol test solutions. https://www.ellisys.com/, Accessed: 2018-10-28.

[11] Scott Fluhrer and Stefan Lucks. Analysis of the E0 encryption system. In *International Workshop on Selected Areas in Cryptography*, pages 38–48. Springer, 2001.

[12] Glenn Greenwald. *No place to hide: Edward Snowden, the NSA, and the US surveillance state*. Metropolitan Books, 2014.

[13] Keijo Haataja and Pekka Toivanen. Two practical man-in-the-middle attacks on bluetooth secure simple pairing and countermeasures. *IEEE Transactions on Wireless Communications*, 9(1), 2010.

[14] IETF. Counter with CBC-MAC (CCM). https://www.ietf.org/rfc/rfc3610.txt, Accessed: 2018-10-28.

[15] Markus Jakobsson and Susanne Wetzel. Security weaknesses in Bluetooth. In *Cryptographers' Track at the RSA Conference*, pages 176–191. Springer, 2001.

[16] Avinash Kak. BitVector.py. https://engineering.purdue.edu/kak/dist/BitVector-3.4.8.html, Accessed: 2018-10-28.

[17] John Kelsey, Bruce Schneier, and David Wagner. Key schedule weaknesses in SAFER+. In *The Second Advanced Encryption Standard Candidate Conference*, pages 155–167, 1999.

[18] Paraskevas Kitsos, Nicolas Sklavos, Kyriakos Papadomanolakis, and Odysseas Koufopavlou. Hardware implementation of Bluetooth security. *IEEE Pervasive Computing*, (1):21–29, 2003.

[19] Sandeep Kumar, Christof Paar, Jan Pelzl, Gerd Pfeiffer, and Manfred Schimmler. Breaking ciphers with copacobana–a cost-optimized parallel code breaker. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 101–118. Springer, 2006.

[20] Musaria K Mahmood, Lujain S Abdulla, Ahmed H Mohsin, and Hamza A Abdullah. MATLAB Implementation of 128-key length SAFER+ Cipher System.

[21] Dennis Mantz. Internalblue. https://github.com/seemoo-lab/internalblue, Accessed: 2018-10-30.

[22] James L Massey, Gurgen H Khachatrian, and Melsik K Kuregian. Nomination of SAFER+ as candidate algorithm for the Advanced Encryption Standard (AES). *NIST AES Proposal*, 1998.

[23] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE bites: exploiting the SSL 3.0 fallback. https://www.openssl.org/~bodo/ssl-poodle.pdf, Accessed: 2019-02-04, 2014.

[24] Michael Ossmann. Project Ubertooth. https://github.com/greatscottgadgets/ubertooth, Accessed: 2018-11-01.

[25] John Padgette. Guide to bluetooth security. *NIST Special Publication*, 800:121, 2017.

[26] Christina Pöpper, Nils Ole Tippenhauer, Boris Danev, and Srdjan Čapkun. Investigation of signal and message manipulations on the wireless channel. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, December 2011.

[27] Jordan Robertson and Michael Riley. The Big Hack: How China Used a Tiny Chip to Infiltrate U.S. Companies. https://www.bloomberg.com/news/features/2018-10-04/the-big-hack-how-china-used-a-tiny-chip-to-infiltrate-america-s-top-companies, Accessed: 2018-10-30.

[28] Yaniv Shaked and Avishai Wool. Cracking the Bluetooth PIN. In *Proceedings of the conference on Mobile systems, applications, and services (MobiSys)*, pages 39–50. ACM, 2005.

[29] Juha T Vainio. Bluetooth security. In *Proceedings of Helsinki University of Technology, Telecommunications Software and Multimedia Laboratory, Seminar on Internetworking: Ad Hoc Networking, Spring*, volume 5, 2000.

[30] Mathy Vanhoef and Frank Piessens. Key reinstallation attacks: Forcing nonce reuse in WPA2. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1313–1328. ACM, 2017.

[31] Matthias Wilhelm, Ivan Martinovic, Jens B Schmitt, and Vincent Lenders. Short paper: reactive jamming in wireless networks: how realistic is the threat? In *Proceedings of the fourth ACM conference on Wireless network security*, pages 47–52. ACM, 2011.

[32] Ford-Long Wong and Frank Stajano. Location privacy in Bluetooth. In *European Workshop on Security in Ad-hoc and Sensor Networks*, pages 176–188. Springer, 2005.

A Appendix

The Key Negotiation Of Bluetooth (KNOB) attack reduces the entropy of the encryption key (K'_C) to 1 byte (key space has 256 elements). Table 4 shows twenty encryption keys with one byte of entropy both for E_0 and AES-CCM.

E_0 K'_C in hex, MSB on the left	AES-CCM K'_C in hex, MSB on the left
0x00000000000000000000000000000000	0x00000000000000000000000000000000
0x00e275a0abd218d4cf928b9bbf6cb08f	0x01000000000000000000000000000000
0x01c4eb4157a431a99f2517377ed9611e	0x02000000000000000000000000000000
0x01269ee1fc76297d50b79cacc1b5d191	0x03000000000000000000000000000000
0x0389d682af4863533e4a2e6efdb2c23c	0x04000000000000000000000000000000
0x036ba322049a7b87f1d8a5f542de72b3	0x05000000000000000000000000000000
0x024d3dc3f8ec52faa16f3959836ba322	0x06000000000000000000000000000000
0x02af4863533e4a2e6efdb2c23c0713ad	0x07000000000000000000000000000000
0x0713ad055e90c6a67c945cddfb658478	0x08000000000000000000000000000000
0x07f1d8a5f542de72b306d746440934f7	0x09000000000000000000000000000000
0x06d746440934f70fe3b14bea85bce566	0x0a000000000000000000000000000000
0x063533e4a2e6efdb2c23c0713ad055e9	0x0b000000000000000000000000000000
0x049a7b87f1d8a5f542de72b306d74644	0x0c000000000000000000000000000000
0x04780e275a0abd218d4cf928b9bbf6cb	0x0d000000000000000000000000000000
0x055e90c6a67c945cddfb6584780e275a	0x0e000000000000000000000000000000
0x05bce5660dae8c881269ee1fc76297d5	0x0f000000000000000000000000000000
0x0e275a0abd218d4cf928b9bbf6cb08f0	0x10000000000000000000000000000000
0x0ec52faa16f3959836ba322049a7b87f	0x11000000000000000000000000000000
0x0fe3b14bea85bce5660dae8c881269ee	0x12000000000000000000000000000000
0x0f01c4eb4157a431a99f2517377ed961	0x13000000000000000000000000000000

Table 4: List of twenty K'_C used by E_0 (left column) and AES-CCM (right column) when $N = 1$ (key space is 256).

From IP ID to Device ID and KASLR Bypass*

Amit Klein
Bar-Ilan University

Benny Pinkas
Bar-Ilan University

Abstract

IP headers include a 16-bit ID field. Our work examines the generation of this field in Windows (versions 8 and higher), Linux and Android, and shows that the IP ID field enables remote servers to assign a unique ID to each device and thus be able to identify subsequent transmissions sent from that device. This identification works across all browsers and over network changes. In modern Linux and Android versions, this field leaks a kernel address, thus we also break KASLR.

Our work includes reverse-engineering of the Windows IP ID generation code, and a cryptanalysis of this code and of the Linux kernel IP ID generation code. It provides practical techniques to partially extract the key used by each of these algorithms, overcoming different implementation issues, and observing that this key can identify individual devices. We deployed a demo (for Windows) showing that key extraction and machine fingerprinting works in the wild, and tested it from networks around the world.

1 Introduction

Online browser-based user tracking is prevalent. Tracking is used to identify users and track them across many sessions and websites on the Internet. Tracking is often performed in order to personalize advertisements or for surveillance purposes. It can either be done by sites that are visited by users, or by third-party companies which track users across multiple web sites and applications. [2] specifically lists motivations for web-based fingerprinting as "fraud detection, protection against account hijacking, anti-bot and anti-scraping services, enterprise security management, protection against DDOS attacks, real-time targeted marketing, campaign measurement, reaching customers across devices, and limiting number of access to services".

Tracking methods Existing tracking mechanisms are usually based on either *tagging* or *fingerprinting*. With tagging, the tracking party stores at the user's device some information, such as a cookie, which can later be tracked. Modern web standards and norms, however, enable users to opt-out from tagging. Furthermore, tagging is often specific for one application or browser, and therefore a tag that was stored in one browser cannot be identified when the user is using a different browser on the same machine, or when the user uses the private browsing feature of the browser. Fingerprinting is implemented by having the tracking party measure features of the user's machine (for example the set of installed fonts). Corporates, however, often install a single "golden image" (standard set of software packages) on many *identical* (hardware-wise) machines, and therefore it is hard to obtain fingerprints that distinguish among such machines.

In this work we present a new tracking mechanism which is based on extracting data used by the IP ID generator (see Section 1.1). It is the first tracking technique that is able to simultaneously (a) cross the private browsing boundary (i.e. compute the same tracking ID for a private mode tab/window of a browser as for a regular tab/window of the browser); (b) work across different browsers; (c) address the "golden image" problem; and (d) work across multiple networks; all this while maintaining a very good coverage of the platforms involved. To our knowledge, no other tracking method (or a combination of several tracking techniques) achieves all these goals simultaneously. Moreover, the Windows variant of this technique also survives Windows shutdown+startup (but not restart).

Our techniques are realistic: for Windows we only need to have control over 8-30 IP addresses (in 3-13 class B networks), and for Linux/Android, we only need to control 300-400 IP addresses (can be in a single class B network). The Windows technique was successfully tested in the wild.

1.1 Introduction to IP ID

The IP ID field is a 16 bit IP header field, defined in RFC 791 [11]. It is used to facilitate de-fragmentation, by marking IP fragments that belong to the same IP datagram. The IP protocol assembles fragments into a datagram based on the fragment source IP, destination IP, protocol (e.g. TCP or UDP) and IP ID. Thus, it is desirable to ensure that given the same source address, destination address and protocol, the IP ID does not repeat itself in short time intervals. Simultaneously, the IP ID should not be predictable (across different destination IP addresses) since "[IP ID] predictability allows traffic analysis, idle scanning, and even packet injection in specific cases" [30].

Designing an IP ID generation algorithm that meets both requirements is not straightforward. Since IPv4 was standardized, several schemes have emerged:

- Global counter – This approach was used in the early IPv4 days due to its simplicity and its non-repetition

*An extended version of this paper can be found at http://www.securitygalore.com/site3/usenix2019.

period of 65536 global packets. However it is extremely predictable and thus insecure, hence abandoned.

- Counter/bucket based algorithms – This family of algorithms, suggested by RFC 7739 [7, Section 5.3], is the focus of our work. It uses a table of counters, and a hash function that maps a combination of a source IP address, destination IP address, key and sometimes other elements into an index of an entry in the table. IP ID is generated by choosing the counter pointed to by the hash function, possibly adding to it an offset (which may depend on the IP endpoints, key, etc.), and finally incrementing the counter. The non-repetition period in this family is 65536 global packets, and at the same time knowing IP ID values for one pair of source and destination IP addresses does not reveal anything about the IP IDs of pairs in other buckets.
- Searchable queue-based algorithm – This algorithm maintains a queue of the last several thousand IP IDs that were used. The algorithm draws random IDs until one is found that is not in the queue. Then this ID is used as the next IP ID, pushed to the queue, and the least-recently used value is popped from the queue. This algorithm ensures high unpredictability, and guarantees a non-repetition period as long as the queue.

Windows (version 8 and later) and Linux/Android implement variants of the counter-based algorithm. MacOS and iOS implement a searchable queue algorithm.

1.2 Introduction to KASLR

KASLR (Kernel Address Space Layout Randomization) is a security mechanism designed to defeat attack techniques such as ROP (Return-Oriented Programming [27]) that rely on the predictability of kernel code addresses. KASLR-enabled kernels randomize the kernel image load address during boot, so that kernel code addresses become unpredictable. While, e.g. in the Linux x64 kernel, the entropy of the load address is 9 bits, a brute force attack is deemed irrelevant since each failure usually ends in a system freeze ("kernel panic"). A typical KASLR bypass enables the attacker to obtain a kernel address (from which, addresses to useful kernel code gadgets can be calculated as offsets) without de-stabilizing the system.

1.3 Our Approach

The IP ID generation mechanisms in Windows and in Linux (UDP only) both compute the IP ID as a function of the source IP address, the destination IP address, and a key K which is generated when the source machine is restarted and is never changed afterwards. We run a cryptanalysis attack which analyzes the IP ID values that are sent by a device and extracts the key K. This key can then be used to identify the source device, because subsequent attacks will yield the same key value (until the device is restarted).

In more detail, IP ID generation in both systems maintains a table of counters and uses a hash function to choose which counter is used for each connection. It seems hard to deploy an attack based on the *value* of the counter, since each IP ID might depend on a different counter. Instead, our attack techniques rely on identifying and exploiting *collisions* which map two destination IP addresses to the same counter. This enables us to extract information about the key that caused the hash values to collide (Linux), or (in Windows) extract information about the offset of the IP ID from the counter. These values depend on K and therefore enable us to learn K and identify the machine.

Our approach does not rely on an a-priori knowledge of the counter values. Moreover, after we reconstruct K, we can reconstruct the current counter values (in full or in part) by sending traffic to specially chosen IP addresses, obtaining their IP ID values and with the knowledge of K, work back the counter values that were used to generate them.

Linux/Android KASLR bypass Support for network namespaces (part of container technology) was introduced in Linux kernel 4.1. With this change, the key K was extended to include 32 bits of a kernel address (the address of the `net` structure for the current namespace). Thus, reconstructing K also reveals 32 bits of a kernel address, which suffices to reconstruct the full address and be able to bypass KASLR.[1]

Conclusion In general, our work demonstrates that the usage of a non-cryptographic algorithm for the generation of attacker observable values such as IP ID, may be a security vulnerability even if the values themselves are not security-sensitive. This is due to an attacker's ability to extract the key used by the algorithm generating the values, and use this key to track or attack the system.

1.4 Advantages of our Technique

Tracking machines based on the key that is used for generating the IP ID has multiple advantages:

Browser Privacy Mode: Since our technique exploits the behavior of the IP packet generator, it is not affected if the browser runs in privacy mode.

Cross-Browser: Since our technique exploits the behavior of the IP packet generator, it yields the same device ID regardless of the browser used. It should be noted that browsers (like Tor browser) that relay transport protocols through other servers are not affected by our technique.

Network change: Tracking works across different networks since our technique uses bits of K as a device ID, and K does not depend on the device's IP address or network.

[1] Through our IP ID attack we were also able to achieve partial KASLR bypass, and a partial list of loaded drivers, with regards to Windows 10 RedStone 4. This attack was based on an additional initialization bug in Windows. However, that bug was repaired in the October 2018 security update and the corresponding KASLR bypass is not effective anymore.

The "Golden Image" Challenge: Since each device generates its own key K in a random fashion at O/S restart, even devices with identical software and hardware will most likely have different K values and thus different device IDs.

Not easily turned off: IP ID generation is built into the kernel, and cannot be modified or switched off by the user. Furthermore, the Windows attack can use simple HTTP traffic. The Linux/Android attack requires WebRTC which cannot be turned off for mobile Chrome and Firefox.

VPN resistant: The device ID remains the same when the device uses an IP-layer VPN.

Windows shutdown+startup vs. restart: The Fast Startup feature of Windows 8 and later,[2] which is enabled by default, saves the kernel to disk on shutdown, and reloads it from disk on system startup. Therefore, K is not re-initialized on startup, and keeps its pre-shutdown value. This means that the tracking technique for Windows survives system shutdown+startup. On restart, in contrast, the kernel is initialized from scratch, and a new value for K is generated, i.e. the old device ID is no longer in effect.

Scalability: Our technique can support billions of devices (Windows, Linux, newer Androids), as the device ID is random, and thus ID collisions are only expected due to the birthday paradox. Thus the probability of a single device not to have a unique ID is very low.

It should be noted that in the Linux/Android case, due to the use of 300-400 IP addresses, the need to "dwell" on the page for 8-9 seconds, and (in newer Android devices) the excessive attack time, there are use cases in which the technique may be considered invasive and/or inapplicable.

Additional Contributions: In addition to the cross-browser tracking technique for Windows and Linux, and the KASLR bypass with respect to Linux, we also provide the first full public documentation of the IP ID generation algorithm in Windows 8 and later versions, obtained via reverse-engineering of the relevant parts of Windows kernel `tcpip.sys` driver, and a cryptanalysis of said algorithm. We also show a demo implementation of the Windows tracking technique and provide results from an extensive in-the-wild experiment spanning 75 networks in 18 countries, demonstrating the applicability of the attack.

We disclosed the vulnerabilities to Microsoft and Linux. Microsoft fixed the issue in Windows April 2019 Security Update (CVE-2019-0688).[3] Linux fixed the kernel address disclosure (CVE-2019-10639) together with partially addressing the key-based tracking technique (by extending the key to 64 bits) in a patch[4] applied to Linux kernel versions 5.1-rc4, 5.0.8, 4.19.35, 4.14.112, 4.9.169 and 4.4.179. For 3.18.139 and 3.16.67, Linux applied a patch[5] we developed, that extends the key to 64 bits. The key-based tracking technique (CVE-2019-10638) is fully addressed in a patch,[6] part of kernel version 5.2-rc1, and will be back-ported to kernel versions 5.1.7, 5.0.21, 4.19.48 and 4.14.124.

Note: many non-essential details of the attack, as well as proofs for false positive bounds for Windows, are deferred to the extended version of the paper.

2 The Setting

We assume that device tracking is carried out over the web, using an HTML snippet (which can be embedded by a 3rd party site/page). The snippet forces the browser to send TCP or UDP traffic (one packet per destination IP suffices) to multiple IP addresses under the tracker's control (8-30 addresses for Windows, 300-400 for Linux/Android). Ideally, such transmission would be rapid. In our experiments, this can be done in few seconds or less.

For the Windows attack, the tracker needs to choose the IP addresses according to some trivial constraints (the Linux IP addresses are not subject to any constraints). A discussion of the exact constraints and their trade-offs can be found in the extended paper. At the server side, the tracker collects the IP ID values sent by the client to each of the IPs, and computes a device ID consisting of bits of the key in the device's kernel data that is used to calculate the IP ID.

Additional scenarios (KASLR bypass and internal IP disclosure) for Linux/Android attacks are described in the extended paper.

3 Related Work

Many tracking techniques were suggested in prior research. At large, proposals can be categorized by their passive/active nature. We use the terminology defined in [31]:

- A *fingerprinting* technique measures properties already existing in the browser or operating system, collecting a combination of data that ideally uniquely identifies the browser/device without altering its state.
- A *tagging* technique, in contrast, stores data in the browser/device, which uniquely identifies it. Further access to the browser can "read" the data and identify the device.

[2] https://blogs.msdn.microsoft.com/olivnie/2012/12/14/windows-8-fast-boot

[3] https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/CVE-2019-0688

[4] "netns: provide pure entropy for net_hash_mix()" (https://github.com/torvalds/linux/commit/355b98553789b646ed97ad801a619ff898471b92)

[5] "inet: update the IP ID generation algorithm to higher standards" (https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=55f0fc7a02de8f12757f4937143d8d5091b2e40b)

[6] "inet: switch IP ID generator to siphash" (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=df453700e8d81b1bdafdf684365ee2b9431fb702)

As described in Section 1, fingerprinting techniques typically cannot guarantee the uniqueness of the device ID, in particular with respect to corporate machines cloned from "golden images". Tagging techniques store data on the device, and as such they are more easily monitored and evaded. A comprehensive discussion of tracking methods can be found in Google Chromium's web page "Technical analysis of client identification mechanisms" [12].

3.1 IP ID Research

Device tracking via IP ID: Using IP ID is proposed in [5] (2002) to detect multiple devices behind a NAT, assuming an IP ID implementation using a *global* counter. But nowadays none of the modern operating systems implements IP ID as a global counter. A similar concept is presented by [25] for a single destination IP (the DNS resolver) which theoretically works for devices that have per-IP counter (Windows, to some extent). However, this technique does not scale beyond a few dozen devices, due to IP ID collisions (the IP ID field provides at most 2^{16} values), and requires ongoing access to the traffic arriving at the DNS resolver.

Predictable IP ID: The predictability of IP ID may theoretically be used in some conditions to track devices. [6] describes a technique to predict the IP ID of a target, but requires the adversary to have a fully controlled device alongside it behind the same NAT. Also this technique only handles sequential increments (e.g. not time-based). As such, it is inapplicable to the more general scenarios handled in this paper. This technique is then used in [9] to poison DNS records.

OS Fingerprinting: [32] suggests using $IPID = 0$ as a fingerprint for some operating systems.

Measuring traffic: [29] samples IP ID values from servers whose IP ID is a global counter, to estimate their outbound traffic.

IP ID Algorithm Categorization: [28] provides practical classification of IP ID generation algorithms and measurements in the wild.

Fragmentation attacks: While not directly related to the properties of the IP ID field, it should be noted that attack techniques abusing fragmentation are known. RFC 1858 [26] lists several such attacks, e.g. the "tiny frgament" attack and the "overlapping fragement" attack.

Windows IP ID research: In parallel to our research, Ran Menscher published on Twitter his research on Windows IP ID [23]. That research reverse-engineered part of the Windows IP ID generation algorithm (without revealing how the index to the counter array is calculated). The analysis of this algorithm is based on two assumptions: (1) that the technique is applied shortly after restart, when the relevant memory buffer contains zeroes in a large part of its cells; and (2) that the attacker controls or monitors traffic to pairs of IP addresses which differ in single, specific bit position (including positions in the left half of the address). Based on these extreme assumptions, the attacker can extract the key easily, and use it to expose kernel 31-bit data quantities (though without learning where in the array this data resides).

The uninitialized memory issue exploited by this attack was fixed in Microsoft's October 2018 Security Update [24], which invalidated assumption (1), rendering Menscher's attack completely ineffective. Our attack and our demo, on the other hand, still work against systems that were patched with this update. Our work has multiple contributions over Menscher's attack: (1) We provide the full details of the IP ID algorithm. (2) Our analysis does not rely on the array data, and is thus still in effect after applying the October 2018 Security Update which initializes the array with random data. (3) Our analysis does not require the extreme requirements on the relations between the addresses of the controlled/monitored IP addresses. (4) Our kernel data exposure provides positions of the data, not just data quantities (though our kernel exposure technique, too, was eliminated with the October 2018 Security Update). (5) It should also be noted that unlike our attack, Menscher's technique could not be used for tracking, since as the cell arrays become non-zero when they are incremented, the attack becomes ineffective.

3.2 PRNG seed/key extraction

Our approach involves breaking the random number generator algorithm used by operating systems to generate the IP ID value and obtain the seed/key used by the algorithm. Similar strategies were used to different ends. For example, [17] broke the PRNG of the Witty worm to obtain the seed, from which they learned the infection time of the Internet nodes. [14] broke the Javascript `Math.Random()` PRNG of several browsers, obtained the seed and used it as a browser instance tracking ID. [15] broke the `Math.Random()` PRNG of Adobe Flash, obtained the seed and used it to extract the machine clock speed.

4 Tracking Windows 8 (and Later) Devices

In this section we first present the algorithm that is used for generating the IP ID in Windows 8 (and later) devices. The input to this algorithm includes a key which is generated at system restart. We then describe how a remote server can identify 45 bits of this key. This data enables to remotely and uniquely identify machines.

4.1 IP ID Generation

IP ID prior to Windows 8 In versions of Microsoft Windows up to and including Windows 7, the IP ID was generated sequentially and globally. That is, for each outgoing IP packet, a global counter would be incremented by 1 and the

result (truncated to 16 bits) would be used [25]. These older Windows versions are out of scope for this paper.

The source code of the algorithm that is used for generating IP ID values in Windows is not public. However, we recovered the exact algorithm using reverse engineering, and verified its correctness by comparing its output to IP ID values generated by live Windows systems.

Technical details The algorithm was obtained by reverse-engineering parts of the `tcpip.sys` driver of 64-bit Windows 10 RedStone 4 (April 2018 Update, Build 1803). Apparently this algorithm is in use starting with Windows 8 and Windows Server 2012. Notice that the code is not specific to IPv4, and can be used with IPv6, which is why the key K is defined as 320 bits - more than required to support IPv4.[7] For IPv4 pre RedStone 5, only 106 key bits are used.

Toeplitz hash The IP ID generation is based on the Toeplitz hash function defined in [10]. Let us first define the *Toeplitz hash*, $T(K,I)$, which is a bilinear transformation from a binary vector K in $GF(2)^{320}$, and an input which is a binary string I (where $|I| \leq 289$) to the output space $GF(2)^{32}$. For a binary vector V, denote by V_i the i-th bit in the vector, with bit numbering starting from 0. The i-th bit of $T(K,I)$ ($0 \leq i \leq 31$) is defined as the inner product between I and a substring of K starting in location i. Namely

$$T(K,I)_i = \bigoplus_{j=0}^{|I|-1} I_j \cdot K_{i+j} \tag{1}$$

IP ID generation The IP ID generation algorithm itself uses keys K (`tcpip!TcpToeplitzHashKey`) which is a 320 bit vector, and $K1$ and $K2$ which are 32 bits each. All these keys are generated once during Windows kernel initialization (using `SystemPrng` and `BCryptGenRandom`).

In addition to these constant keys, the algorithm uses a dynamic array of M counters, denoted $\beta[0],\ldots,\beta[M-1]$, where M is a power of 2, and is specifically set to $M = 8192$.

Algorithm 1 describes how Windows 8 (and later) generates an IP ID for a packet delivered from IP_{SRC} to IP_{DST}, while updating a counter in β.

The algorithm uses the keys, and the source and destination IP addresses, to pick a random index i for a counter in β, and an offset. The algorithm outputs the sum of the counter $\beta[i]$ and the offset, and increments the counter.

Notation We use the notation $\mathrm{Num}(a_0,a_1,\ldots,a_{31})$ for the number represented in binary by the bits a_i, namely the number $\sum_{i=0}^{31} a_i \cdot 2^{31-i}$. (Network byte order is used throughout the paper for representing IP addresses as bit vectors, e.g. 127.0.0.1 is 01111111.00000000.00000000.00000001.)

Properties of the Toeplitz hash Our attack uses the following properties of T, which follow from the linearity of this transformation:

$$T(K,I||(0,0,\ldots,0)) = T(K,I) \tag{2}$$

[7] Our tracking technique can be probably adapted to IPv6, but since IPv6 is out of scope for this paper, we did not test this.

Therefore the trailing zeros in the input of T in the computation of v on line 3 of Algorithm 1, have no effect on the output. Also,

$$T(K,I_1||I_2) = T(K,I_1) \oplus T(K,0^{|I_1|}||I_2) \tag{3}$$

Therefore it is possible to decompose the second input of T to two parts, and rephrase the computation as the XOR of two separate expressions.

4.2 Reconstructing the Key K

To reconstruct the key, the device needs to be measured. The measurements only take a few seconds, and are thus assumed to take place from the same network. I.e., the *device's* source IP address, IP_{SRC}, is fixed (though possibly unknown). A first set of measurements directs the client device to J IP addresses from the same class B network. A second set of measurements directs the client device to G *pairs* of IP addresses, each pair in the same class B network, with G different class B network pairs in the set.

Once the device is measured, the attack proceeds in two phases. The first phase of the attack recovers 30 bits of the key using the first set of measurements. The second phase of the attack reveals additional 15 bits of the key using the second set of measurements. Overall, the measurements reveal 45 bits of the key, which suffices to uniquely identify machines from a large population, with high probability.

Section 4.5 describes how to optimally choose the parameters J and G given limits on the number of IP addresses that are available (L) and the processing time that is allowed ($\mathscr{T}$). For $L = 30$ IP addresses (typical low budget limit), and attack run time limit of $\mathscr{T} = 1$ seconds on a single Azure B1s machine ($\alpha = 0.001$ from Section 5.2), the optimal parameter values are $J = 6, G = 12$.

4.3 Extracting Bits of K - Phase 1

Denote by $IP^{g,j}$, $IPID^{g,j}$ and $\beta[i_g]^{g,j}$ the values of the destination IP address, the IP ID and $\beta[i]$ (prior to increment) respectively, with respect to the j-th packet in the g-th class B network that is used in the attack (j and g are counted 0-based). The first phase of the attack uses only a single class B network, and therefore g is set to 0 in this phase. We thus use the following shorthand notation: $IP^j = IP^{0,j}$, $IPID^j = IPID^{0,j}$ and $\beta_g = \beta[i_g]^{g,0}$.

A major observation is that only the first half of IP_{DST} is used to calculate i in Algorithm 1. Therefore packets that are sent to different IP addresses in the same class B network, have an identical index i into the counter table, and use the same counter $\beta[i]$. Denote the value of i for the g-th class B network as i_g.

If these packets are sent in rapid succession (i.e. when no other packet is sent in-between with $i = i_g$), then $\beta[i_g]^{g,j} = \beta_g + j \mod 2^{32}$, and therefore the output in line 5 of the

Algorithm 1 Windows 8 (and later) IP ID Generation

1: **procedure** GENERATE-IPID
2: $i \leftarrow \text{Num}(K2 \oplus T(K,(IP_{DST})_{0,\ldots,\frac{|IP_{DST}|}{2}-1}) \oplus T(K,IP_{SRC})) \mod M$
3: $v \leftarrow \beta[i] + \text{Num}(K1 \oplus T(K,IP_{DST}||IP_{SRC}||0^{32})) \mod 2^{32}$
4: $\beta[i] \leftarrow (\beta[i]+1) \mod 2^{32}$
5: return $v \mod 2^{15}$ ▷ $v \mod 2^{16}$ for Windows 10 RedStone 5

algorithm is calculated with $\beta[i_g]^{g,j} = \beta_g + j \mod 2^{15}$ (for simplicity, in Windows 10 RedStone 5, we discard the most significant bit of the IP ID).

We focus in this phase on the first class B network, b_0, with J destination IP addresses in it. Note that the offset that is calculated in line 3 is the difference between the IPID and the counter $\beta[i_0]$ prior to its increment.

The attack enumerates over the values of the $\beta_0 \mod 2^{15}$ counter. For each possible value it calculates the differences between the observed J IPIDs and the corresponding values of the counter, arriving at the offsets calculated in line 3. By observing pairs of IPIDs, it is possible to identify the correct value of $\beta_0 \mod 2^{15}$ as well as 30 bits of the key.

In more detail, for each possible value of $\beta_0 \mod 2^{15}$ the attack calculates the difference

$$IPID^j - (\beta_0 + j \mod 2^{15}) \mod 2^{15}$$

which, for the right value of the counter should be equal to the offset that is calculated in line 3. Namely to

$$\text{Num}(K1 \oplus T(K,IP^j||IP_{SRC}||0^{32})) \mod 2^{15}$$

This value can be expressed as $(K1 \oplus T(K,IP^j||IP_{SRC}||0^{32}))_{17,\ldots,31}$. Applying eq. (2) and eq. (3), this expression is simplified into:

$$(K1 \oplus T(K,IP^j) \oplus T(K,0^{32}||IP_{SRC}))_{17,\ldots,31}$$

.

The attack takes two different j values and computes the XOR of the two corresponding such quantities. This results in the following expression (where we denote by Vec a representation of a number in $[0,2^{32})$ as a vector in $GF(2)^{32}$):

$$(\text{Vec}(IPID^j - (\beta_0 + j) \mod 2^{15}) \oplus$$
$$\text{Vec}(IPID^{j'} - (\beta_0 + j') \mod 2^{15}))_{17,\ldots,31} =$$
$$T(K,IP^j \oplus IP^{j'})_{17,\ldots,31}$$

This yields 15 linear equations ($i = 17,\ldots,31$) on K since (from eq. (1)):

$$T(K,IP^j \oplus IP^{j'})_i = \bigoplus_{m=0}^{31} (IP^j \oplus IP^{j'})_m \cdot K_{i+m}$$

Since all IP^j belong to the same class B network, $IP^j \oplus IP^{j'}$ always has 0 for its first 16 bits, and therefore m can start at 16. Due to obvious linear dependencies, only $J-1$ sets of such equations are useful (e.g. all pairs with $j' = 0$), with a total of $15(J-1)$ linear equations for bits $K_{33},\ldots,K_{62}$. That is, for $j = 1,\ldots,J-1$ and $i = 17,\ldots,31$, the equations are:

$$\bigoplus_{m=16}^{31} (IP^j \oplus IP^0)_m \cdot K_{i+m} =$$
$$(\text{Vec}(IPID^j - (\beta_0 + j) \mod 2^{15}) \oplus$$
$$\text{Vec}(IPID^0 - (\beta_0) \mod 2^{15}))_i \quad (4)$$

Speeding up the computation using preprocessing The coefficients of K in eq. (4) are controlled by the server and are known at setup time. Therefore it is possible to preprocess the computation of Gaussian elimination. Namely, compute a matrix Z that, when multiplied by the observed values, reveals bits of the key. This preprocessing is only important for efficiency, therefore we defer the details to the extended paper.

Attack summary

1. The tracker needs to control J IP addresses in the same class B network.
2. During setup time, the tracker calculates, using Gaussian elimination, a matrix $Z \in GF(2)^{15(J-1)\times 15(J-1)}$, based on the values of these IP addresses.
3. In real time, the tracker gets IP ID values from the device, from packets sent to the J destination IP addresses under the tracker's control.
4. The tracker then guesses 14 bits ($\beta_0 \mod 2^{14}$ - the most significant bit of $\beta_0 \mod 2^{15}$ cancels itself in eq. (4)) of the counter that is used for these IP addresses, calculates vectors D^j ($j = 1,\ldots,J-1$), where $D^j = (\text{Vec}(IPID^j - (\beta_0 + j) \mod 2^{15}) \oplus \text{Vec}(IPID^0 - (\beta_0) \mod 2^{15}))_{17,\ldots,31}$, and performs a matrix-by-vector multiplication of Z and the vector $(D^0,\ldots,D^{J-1})$.

   For the correct value of $\beta_0 \mod 2^{14}$ this computation results in a vector of $15(J-1)$ bits, whose first 30 bits are $K_{33},\ldots,K_{62}$ and the remaining bits are zero.
5. The attacker identifies the right value of the counter by comparing to zero the $15(J-1)-30$ bits starting at position 31: if $15(J-1)-30 \gg 14$, this verification statistically guarantees the correctness of the solution (up to a flipped most significant bit in $\beta_0 \mod 2^{14}$, see the extended paper.)

Overall this process reveals 30 bits of the key as well as the value ($\beta_0 \mod 2^{14}$).

The attack takes $2^{14} \cdot (15(J-1))^2$ bit operations (for enumeration over the possible key values and for the matrix-by-vector and $(15(J-1))^2$ memory bits (for Z). As explained in Section 4.5, we set $J = 6$ and therefore this overhead is very small.

The tracker obtains the (correct) value $\beta_0 \mod 2^{14}$, which will be used in the next phase. While it is guaranteed that the correct K and $\beta_0 \mod 2^{14}$ will be found, the algorithm may emit additional candidates (with incorrect $\beta_0 \mod 2^{14}$). The false positive probability of both phases of the attack is analyzed in the extended paper.[8]

4.4 Extracting Bits of K - Phase 2

Given 30 bits of K ($K_{33},\ldots,K_{62}$) and the value ($\beta_0 \mod 2^{14}$), recovered in Phase 1, the attack can be extended to learn a total of up to 45 key bits ($K_{18},\ldots,K_{62}$). This is done in the following way. The offset for $IPID^0$ computed in line 3 of Algorithm 1 is:

$$\mathrm{Num}(K1 \oplus T(K,IP^0) \oplus T(K,0^{32}||IP_{SRC})) \mod 2^{15} = (IPID^0 - \beta_0) \mod 2^{15}$$

The following equation follows from the previous one:

$$(K1 \oplus T(K,0^{32}||IP_{SRC}))_{17,\ldots,31} = T(K,IP^0)_{17,\ldots,31} \oplus \mathrm{Vec}(IPID^0 - \beta_0 \mod 2^{15})_{17,\ldots,31}$$

The tracker looks at pairs of IP addresses in the remaining B classes ($b_1,\ldots,b_G$), each pair in a different class B network. Denote each such pair as ($IP^{g,0}, IP^{g,1}$), with the order inside the pair conforming to the order of packet transmission, and the packets being transmitted in rapid succession. Substituting the above into the definition of $IPID$ yields:

$$\begin{aligned} IPID^{g,j} &= \beta_g + j + \mathrm{Num}(\, T(K,IP^0)_{17,\ldots,31} \\ &\oplus \mathrm{Vec}(IPID^0 - \beta_0 \mod 2^{15})_{17,\ldots,31} \\ &\oplus T(K,IP^{g,j})_{17,\ldots,31}\,) \mod 2^{15} \end{aligned}$$

Using the linearity of T, this is simplified into:

$$\begin{aligned} IPID^{g,j} = \beta_g + j + \mathrm{Num}\,(\, & T(K,IP^0 \oplus IP^{g,j})_{17,\ldots,31} \oplus \\ & \mathrm{Vec}(IPID^0 - \beta_0 \mod 2^{15})_{17,\ldots,31}\,) \mod 2^{15} \end{aligned}$$

Let us use the notation

$$\begin{aligned} S^{g,j} = \mathrm{Num}\,(\, & T(K,IP^0 \oplus IP^{g,j})_{17,\ldots,31} \oplus \\ & \mathrm{Vec}(IPID^0 - \beta_0 \mod 2^{15})_{17,\ldots,31}\,) \mod 2^{15} \end{aligned}$$

Then this equation becomes

$$IPID^{g,j} = \beta_g + j + S^{g,j} \mod 2^{15}$$

Subtracting the IPIDs of the two consecutive packets in the same B class (with $j = 0$ and $j = 1$) cancels the value of the counter β_g, and yields:

$$(IPID^{g,1} - IPID^{g,0}) \mod 2^{15} = 1 + S^{g,1} - S^{g,0} \mod 2^{15} \quad (5)$$

The left side of the equation is observed by the tracker. The right side can be computed based on $\beta_0 \mod 2^{15}$ and $K_{17},\ldots,K_{62}$. The tracker already knows these values except for $K_{18},\ldots,K_{33}$, and therefore only needs to enumerate over the 2^{15} possible values of $K_{18},\ldots,K_{32}$ and eliminate all values which do not agree with the equation. We discuss this procedure in depth in the extended paper.

Attack summary:

1. The tracker needs to control additional G pairs of IPs (each pair in its own class B network).
2. Given IP IDs for these pairs, the tracker enumerates over additional 15 key bits, and then, for each pair of IP addresses, calculates both sides of eq. (5) and compares them. For this calculation the tracker can choose K_{17} and the leftmost bit of $\beta_0 \mod 2^{15}$ arbitrarily, as they will both cancel themselves.
3. In theory, each IP pair should yield a 2^{15} elimination power for identifying the right key, but see the extended paper for a more accurate analysis.
4. In the calculation, the leading term (in terms of run time) is computing $T(K,I)_{17,\ldots,31}$ (where $|I| = 32$), which takes $14|I|$ bit operations, and is used twice. Thus, the run-time is roughly $2^{15} \cdot 2 \cdot 14 \cdot 32$ bit operations (there is no multiplication by G since the first pair is likely to eliminate almost all false guesses).

At the end of Phase 2, the tracker obtains:

- A partial key vector (or some candidates) $K_{18},\ldots,K_{62}$ (45 bits), which is specific to the device since it was set during kernel initialization, and does not depend on IP_{SRC}. These bits serve as a device ID.
- The value

  $$\begin{aligned} (K1 \oplus T(K,0^{32}||IP_{SRC}))_{18,\ldots,31} = {} & T(K,IP^0)_{18,\ldots,31} \\ & \oplus \mathrm{Vec}(IPID^0 - \beta_0 \mod 2^{14})_{18,\ldots,31} \end{aligned}$$

  This value allows the tracker to calculate (assuming $K_{18},\ldots K_{62}$ are known) the value of the counter $\beta[i] \mod 2^{14}$ for any destination IP address whose IP ID is known (provided the source IP is IP_{SRC}).[9]

[8] Note: Throughout the paper, we assume that $\mathrm{rank}(C) = 30$. This results in a single key vector per guessed $\beta_0 \mod 2^{14}$. We discuss the conditions on $IP^0,\ldots,IP^{J-1}$ to meet this assumption in the extended paper. If $\mathrm{rank}(\ker(C)) > 0$, then each guess of $\beta_0 \mod 2^{14}$ yields $2^{\mathrm{rank}(\ker(C))}$ possible keys. Thus small values of $\mathrm{rank}(\ker(C))$ are acceptable.

[9] This is useful for reconstructing the table β of counters – this table is not correctly initialized (pre October 2018 Security Update), and therefore is populated with kernel data that happens to be (in build 1803) data structures containing kernel address pointers.

4.5 Choosing Optimal G and J

For Windows, we assume budget-oriented constraints, namely L available IP addresses and $\mathscr{T}$ CPU time per measurement. We need to set the number J of IP addresses from the same class B network to which the client is directed in the first set of measurements, and the number G of pairs of IP addresses, each pair in the same class B network, used in the second set of measurements.

Our goal is to optimize for minimum false positives. The first constraint can be expressed as $J+2G \leq L$. As for the second constraint, the leading term of the time of the attack run is $\alpha \cdot (J!)$ (Appendix A.1.2), where α expresses the computing platform's strength. Therefore, we can approximate the second constraint as $\alpha \cdot (J!) \leq \mathscr{T}$. Additionally, there are inherent constraints: $J-1 \geq 3$ to let Phase 1 suggest a single key candidate to Phase 2 (most of the time), and $G \geq 2$ to let Phase 2 provide a single final key (most of the time).

Given these constraints, we want to minimize the leading term in false positives, $2 \cdot 2^{-\frac{G+J-1}{2}}$ (Appendix A.2), i.e. we need to maximize $G+J$. Since we "pay" two IP addresses for each increment of G and only one IP address for each increment of J, we should make J as large as possible (as long as G is valid), so the solution is:

$$J = \min(\max(\{J \mid \alpha J! \leq \mathscr{T}\}), L-4)$$

(As stated in Section 4.2, for $L=30$, $\mathscr{T}=1$ sec., and $\alpha = 0.001$, the optimal combination is $J=6, G=12$.)

4.6 Practical Considerations

We discuss in Appendix A.1 different issues that appear when deploying the attack. These issues include ways to emit the needed traffic from the browser, handling packet loss and out-of-order packet transmission, handling interfering packets, and limiting the false-positive and false-negative error probabilities.

The run time of the key extraction attack is less than a second even on a very modest machine. The *dwell time* (time duration in which the page needs to be loaded in the browser) is 1-2 seconds for a WebSocket implementation. It is possible to minimize the dwell time by moving to WebRTC (STUN).

Longevity: the device ID is valid until the machine restarts (mere shutdown+start does not invalidate the device ID due to Windows' Fast Start feature). A typical user needs to restart his/her Windows machine only for some Windows updates, i.e. with a frequency of less than once per month.

The attack is scalable: with 41 bits, the probability of a device to have a unique ID is very high, even for a billion device population; false positives are also rare (2.1×10^{-6} – Table 3), and false negatives can be made negligible (Appendix A.1.4). From resource perspective, the attack uses a fixed number of servers, RAM/disk and ($L=30$) IPs. The required CPU power is linear in the number of devices measured per time unit, and in the Windows case is negligible. Network consumption per test is also negligible (assuming WebRTC/STUN implementation – 1.5KB at the IP layer.)

4.7 Attack Improvements and Variants

A **fast-track identification of already-seen keys** can be obtained in the following way: Once bits of a key K are extracted, they will be stored for comparison against future connections. When a device is to be measured, the tracker first goes through all stored K bit strings, and tests the measured data for compatibility with each one of them. This amounts to guessing the bits of β_0 one by one, starting from the least significant, and eliminating via eq. (5), using mod 2^n where n is the number of β_0 bits guessed so far. The CPU work per key is thus almost negligible.

The original attack can also be sped up using **incremental evaluation**. The details are in the extended paper.

4.8 Environment Factors

We demonstrate here that the tracking attack can be deployed in almost every setting that can be reasonably expected.

HTTPS: In essence, there should be no problem in having the snippet use WebSocket over HTTPS (`wss://` URL scheme) for TCP packets.

NAT: Typically NAT (Network Address Translator) devices do not alter IP IDs, and thus do not affect the attack.

Transparent HTTP Proxy / Web Gateway: Such devices may terminate the TCP connection and establish their own connections (with IP ID from their own network stack) and thus render our technique completely ineffective. However, typically these devices do not interfere with HTTPS (TCP port 443) traffic, and UDP traffic, so these alternatives can be used by the tracker.

Forward HTTP proxy: When a browser is configured to use a forward proxy server, even HTTPS traffic is routed to it by the browser. However, it may still be the case that UDP traffic (which is not handled by HTTP forward proxies) can be used by the technique.

Tor-based browsers and similar browsers: Browsers that forward TCP traffic to proxy servers (and disallow or forward UDP requests) are incompatible with the tracking technique as they do not expose IP header data generated on the device. Since "Tor transports TCP streams, not IP packets",[10] this applies to all Tor-based products, such as the Tor browser and Brave's "Private Tabs with Tor" and therefore they are not covered by our technique.

Windows Defender Application Guard (WDAG): This new technology in Windows 10 enables the user to launch the Edge browser in a virtual environment. While the device ID

[10] https://www.torproject.org/docs/faq.html.en#RemotePhysicalDeviceFingerprinting

in this virtual environment is independent of the device ID of the main operating system, it is consistent among **all** WDAG Edge instances. Furthermore, unlike the "main" Windows device ID, the WDAG device ID does not change with operating system restart, hence the WDAG device ID lives longer than the main Windows device ID. It should be noted that WDAG is only available for Edge browser in Windows 10 Enterprise/Pro edition, and requires high-end hardware.

IP-Level VPN: We experimented with F-Secure FreeDome (www.f-secure.com/en/web/home_global/freedome) and PureVPN (www.purevpn.com/). Both VPNs supported our technique.

IPv6 and IPsec: We do not know whether IPv6 or IPsec packets use the same IP ID generation mechanism. This requires further research.

Javascript disabled: Tracking can also work when Javascript (or any client side scripting) is not available, e.g. with the NoScript browser extension [20]. We discuss this in the extended paper.

4.9 Possible Countermeasures

We list here some obvious ways of modifying Algorithm 1 and their impact:

- Increasing M (the size of the table of counters) – surprisingly, this has very little effect on the basic tracking technique, since no assumptions were made on M in the first place. It does affect the β reconstruction technique.
- Changing T into a cryptographically strong keyed-hash function – while this change eliminates the original attack, it is still possible to mount a weaker attack that only tracks a device while its IP_{SRC} does not change. In fact, this applies to the entire abstract scheme proposed in [7, Section 5.3]. See the extended paper for details.
- Changing the algorithm altogether (this is our recommendation). A robust algorithm relies on industrial-strength cryptography, large enough key space, and strong entropy source for the key, and uses them to generate IP IDs which (a) have guaranteed non-repetition period; (b) are difficult to predict; and (c) do not leak useful data. The algorithm used in macOS/iOS [30] is a good example. This eliminates the attack altogether.

5 Field Experiment – Attacking Windows Machines in the Wild

We set up a fully operational system to test the IP ID behavior in the wild, as well as to verify that the technique for extracting device IDs for Windows machine works as expected.

5.1 Setup

As explained in Appendix A.1.3, in order to avoid false positives (which almost always happen due to false keys that differ from the true key in a few most significant bits), we need to trim the most significant bits from the key – i.e. use the key's tail. For the full production setup (30 IP addresses), we calculated that a tail of 41 bits will suffice. Due to logistic and budgetary constraints, in our experiment we used only 15 IP addresses (rather than 30) for the key extraction (and 2 more IPs for verification), with $J = 5, G = 5, Q = 1$. Thus we lowered the tail length to 40, and used the 40 bits $K_{23}, \ldots, K_{62}$ as a device ID. That is, for this experiment, we traded the device ID space size for a smaller probability of false positives.

We then used WebSocket traffic to the additional pair of IP addresses (from a class B network that is different than those in the initial set of 15 IPs) to verify the correctness of the key bits extracted. In this experiment, since we do not extract $K_{17,\ldots,22}$ we can only compute the least significant 9 bits of the IPID, adapting eq. (5) into:

$$IPID^{g,1} \mod 2^9 = IPID^{g,0} \pm 1 + S^{g,1} - S^{g,0} \mod 2^9$$

(We need to use ± 1 since we cannot know the order of packet generation. Thus given knowledge of $IPID^{g,0}$ we have two candidates for $IPID^{g,1}$, out of a space of $2^9 = 512$ values.) A random choice of two values yields a success rate of $1/256$. We deem our algorithm to be valid if it consistently yields the correct value (in one of the candidates) in all tests.

We asked "Friends and Family" to browse to the demo site using Windows 8 or later, from various networks.

5.2 Results

Network distribution The experiment was conducted from July 22nd, 2018 to October 20th, 2018. We collected data on 75 different class B networks. The networks are well dispersed across 18 countries and 4 continents. The networks are also usage-diverse (home networks, SMB networks, corporate networks, university networks, public hotspots and cellular networks). We asked the users who connected to our demo site to use multiple regular browsers and networks, and connect at different times, and verified that the device ID remained the same in all these connections.

Failures to extract a key – IP ID modification In only 6 networks out of 75 (8%) we could not extract the key and therefore concluded that the IP ID was not preserved by the network. These six networks did not include any major ISP and seem to be used by relatively few users: they included an airport WiFi network, a government office, and a Windows machine connecting through one cellular hotspot (hotspots that we tested in other cellular networks did not change the IP ID). Of those six networks, in 3 networks we had clear indication that a transparent proxy or a web security gateway was in path. In such cases, moving to WebSocket over HTTPS, or to UDP would probably have addressed the issue. Another case was a forward proxy (moving to UDP would have possibly addressed it). In the two final cases, the exact

nature of interference was not identified. We can say then that optimistically, only 2 networks out of 75 (2.7%) are incompatible with the tracking technique, maybe even less (as it is still quite possible these two TCP gateways are actually transparent proxies).

Positive results In the remaining 69 networks, for 4 networks we did not keep traffic for the additional two IPs, thus we could not verify the key extraction. For the rest 65 networks, our algorithm extracted a single 40-bit key, and correctly predicted the least significant 9 bits of the IPID of the second IP in the last pair (i.e. the correct value was one of the two candidates computed by the algorithm). This verifies the correctness of the algorithm and the key bits it extracts.

Lab verification We tested a machine in the lab with the above test setup to obtain 40 bits of K. Then, using WinDbg in local kernel mode, we obtained `tcpip!TcpToeplitzHashKey`, extracted the 40 bits from it and compared to the 40 bits calculated by the snippet – as expected, they came out identical.

Actual run time We estimate the overall runtime for $J = 6, G = 12$ on a single Azure B1s machine to be 0.73 seconds.

Packet loss and false negatives We analyzed 79 valid tests and found only 3 cases wherein the analysis logic failed to provide a device ID (additional test from the same devices succeeded in extracting a key). In all such cases a manual analysis indicates that this is due to packet loss. Appendix A.1.4 describes additional logic that can be used to reduce false negatives to a negligible level.

6 Linux and Android

The scope of our research is Linux kernel 3.0 and above. Also, we only investigated the x64 (typical desktop Linux) and ARM64 (Android) CPU architectures, although almost all of the analysis is not architecture-specific.

6.1 Attack Outline

In order to track a Linux/Android device, the tracker needs to control several hundred IP addresses. The tracking snippet forces the browser to rapidly emit UDP packets to each such IP (using WebRTC and specifically the STUN protocol, which enables sending bursts of packets closely spaced in time to controlled destination addresses). It also collects the device's source IP address (using WebRTC as well or a different approach described in the extended paper.)

The tracker collects IP IDs from all IP addresses, and identifies bucket collisions by looking for IP pairs whose IP IDs are in close proximity. Recall that the choice of the bucket is a function of the source and destination IP addresses, and a device key. The tracker enumerates over the key space to find the (correct) key which generates collisions for the same pairs for which collisions were observed. The key that is found is the device ID.

6.2 IP ID Generation in Linux

The Linux kernel implementation of IP ID differs between TCP and UDP [16]. The TCP implementation always used a counter per TCP connection (initialized with a hash of the connection endpoints and a secret key, combined with a high resolution timer) and as such, is not interesting to us (collisions are meaningless). The implementation of IP ID for stateless over-IP protocols (e.g. UDP) has gone through an interesting evolution process. We focus on short datagrams, i.e. datagrams shorter than MTU (maximum transmission unit), that do not undergo fragmentation. We designate the IP ID generation algorithms as A_0, A_1, A_2 and A_3, in their order of evolution.

A_0: In early Linux kernels, the IP ID for short datagrams was simply set to 0.

A_1 and A_2: In Linux kernel 3.16.0 (released August 2014), IP ID for short datagrams became dynamic (just like it has always been for long UDP datagrams).[11] This was back-ported to various active Linux 3.x branches (see Table 2). The generation algorithm in general has an array of $M = 2048$ buckets, each containing a value $0 \leq \beta < 2^{16}$ and a time-stamp τ of the last time this bucket was used. The bucket array is initialized at boot time with random data (using a PRNG). The algorithm also uses the following parameters

- *key* – a 32-bit key (`ip_idents_hashrnd`) which is initialized upon first IP transmission with random data.
- h – a hash function. Older and newer versions of Linux used different hash functions (A_1 and A_2, resp.) The details of the hash functions are described in the extended paper since they not important for understanding the attack.
- *protocol* – the IP "next level" protocol number (for UDP, this value is 17). Nominally 8-bit field, extended to 32-bit by zero-filling the most significant bits.
- $\text{RANDOM}(x), x > 0$ – a PRNG (a 96/128 bit Tausworthe Generator) which receives x as a parameter and provides a random integer in the range $[0, x)$. (We define $\text{RANDOM}(0) = 0$). Note that $\text{RANDOM}(1) = 0$.

The IP ID generation algorithm is defined in Algorithm 2. The procedure picks an index to a counter as a function of the source and destination IP address, the protocol and the key. It picks a random value which is smaller than or equal to the time that passed (measured in ticks, with tick frequency of f per second) since the last usage of this counter, increments the counter by this value, and outputs the result.

A_3: Starting with Linux 4.1, the net namespace of the kernel context, *net* (a 64-bit *pointer* in kernel space) is included in the hash calculation, conditional on a compilation flag `CONFIG_NET_NS` (which is on by default for Linux 4.1

[11] See function `__ip_select_ident` in https://elixir.bootlin.com/linux/v3.16/source/net/ipv4/route.c.

Algorithm 2 Linux IP ID Generation (A_1/A_2)

1: **procedure** GENERATE-IPID
2: $i \leftarrow h(IP_{DST}, IP_{SRC}, protocol, key) \mod M$
3: $hop \leftarrow 1 + \text{RANDOM}(t_{now} - \tau[i])$
4: $\beta[i] \leftarrow (\beta[i] + hop) \mod 2^{16}$
5: $\tau[i] \leftarrow t_{now}$
6: return $\beta[i]$

and later, and for Android kernel 4.4 and later). The modification is for step 2, which now reads:

$$i \leftarrow h(IP_{DST}, IP_{SRC}, protocol \oplus g(net), key) \mod M$$

where $g(x)$ is a right-shift (by ρ bits) and a truncation function that returns 32 bits from x. We designate this algorithm as A_3.

To summarize, there are four flavors of IP ID generation (for short stateless protocol datagrams) in Linux:

1. A_0 - IP ID is always 0 (in ancient kernel versions)
2. A_1 / A_2 - Both versions use Algorithm 2, with the different implementations of h.
3. A_3 - Algorithm 2, adding net namespace to the calculation.

Of interest to us are algorithms A_1 to A_3. We focus mostly on UDP, as this is a stateless protocol which can be emitted by browsers.

The resolution f of the timer t in the algorithm is determined by the kernel compile-time constant `CONFIG_HZ`. A common value for older Android Linux kernels is 100(Hz). Newer Android Linux kernels (4.4 and above) use 300 or 100 (or rarely, 250). The default for Linux is $f = 250$.[12] In general, for tracking purposes, a lower value of f is better.

Note that *key* and *net* are generated during the operating system initialization, which, unlike Windows, happens during restart *and* during (shutdown+)start.

6.3 Setting the Stage

Our technique for tracking Android (and Linux) devices uses HTML5's WebRTC[1] both to discover the internal IP address of the device and to send multiple UDP packets. It works best when the WebRTC STUN [21] traffic is bursty. In order to analyze the effectiveness of the technique we investigated the following features, focusing on Android devices.

Android Versions and Linux Kernel Versions The Android operating system is based on the Linux kernel. However, Android versions do not map 1:1 to Linux kernel versions. The same Android version may be built with different Linux kernel versions by different vendors, and sometimes by the same vendor. Moreover, when an Android device updates its Android operating system, typically its Linux kernel remains on the same branch (e.g. 3.18.x). Android vendors also typically use somewhat old Linux kernels. Therefore, many Android devices in the wild still have Linux 3.x kernels, i.e. use algorithm A_1 or A_2.

Sending Short UDP Datagrams to Arbitrary Destinations, or "Set Your Browsers to STUN" The technique requires sending UDP datagrams from the browser to multiple destinations. The content of the datagrams is immaterial, as the tracker is interested only in the IP ID field. We use WebRTC (specifically – STUN) to send short UDP datagrams (with no control over their content) to arbitrary hosts. The `RTCPeerConnection` interface can be used to instruct the browser's WebRTC engine to use a list of presumably STUN servers, and even allows setting the UDP destination port per each host. The browser then sends STUN "Binding Request" (UDP short datagram) to the destination host and port.

To send STUN requests to multiple servers (in Javascript), create an array `A` of strings in the form `stun:`*host*`:`*port*, then invoke the constructor `RTCPeerConnection({iceServers: A}, ...)` in a regular WebRTC flow e.g. [13] (applying the fix from [8]).

Another option (specific to Google Chrome) is to send requests over gQUIC (Google QUIC) protocol, which uses UDP as its transport. This is less ideal since the traffic is less bursty, its transmission order isn't deterministic, and there is an overhead in HTTPS requests and in gQUIC packets.

Browser Distribution in Android We want to estimate the browser market share of "supportive" browsers (Chrome-like and Firefox) in the Android OS. Based on April 2018 figures for operating systems,[13] combined with mobile browsers distribution in April 2018,[14] we conclude that the Chrome-like browsers (Google Chrome, Opera Mini, Baidu, Opera) comprise 90% of the browser usage in Android. Adding Firefox (even though its STUN traffic is less bursty, Firefox can still be tracked at least for $f = 100$) gets this figure up to 92%.

Chrome's STUN Traffic Shape Chrome sends the STUN requests to the list of supposedly STUN servers, in bursts. A single burst may contain the full list of the requested STUN servers (in ascending order of destination IP address), or a subset of the ordered list (typically with a missing range of destination hosts). We measured 1014 bursts (to $L = 400$ destination IP addresses) emitted by a Google Pixel 2 mobile phone (Android 8.1.0, kernel 4.4.88), running Google

[12] https://elixir.bootlin.com/linux/v4.19/source/kernel/Kconfig.hz

[13] https://netmarketshare.com/operating-system-market-share.aspx

[14] https://netmarketshare.com/browser-market-share.aspx

Chrome 67 browser. The vast majority of bursts last between 0.1 seconds to 0.2 seconds, and the maximal burst duration was 0.548 seconds. Thus we use an upper bound of $\delta_L = 0.6$ seconds for a single burst duration.

Chrome emits up to 9 bursts with increasing time delays, at the following times (in seconds, where $t = 0$ is the first burst): 0, 0.25, 0.75, 1.75, 3.75, 7.75, 15.75, 23.75, 31.75.[15] We label these bursts $B_0, \ldots, B_8$ respectively, and we will be interested in B_4 and B_5, as they're sufficiently far from their neighbors. Thus, we are only interested in the first 8-9 seconds of the STUN traffic.

UDP Latency Distribution While WebRTC traffic is emitted by the browser in well defined, ordered bursts, one cannot assume the traffic will retain this "shape" when arriving to the destination servers. Indeed, even order among packets within a burst is not guaranteed at the destination. Understanding the latency distribution in UDP short datagrams is therefore needed in order to simulate the in-the-wild behavior, and consequently the efficacy of various tracking techniques. The latency of UDP datagrams is gamma-distributed according to [18] and [19]. However, for simplicity, we use normal distribution to approximate the in-the-wild latency distribution. On May 1st-6th 2018, we measured the latency of connections to a server in Microsoft Azure "East-US" location (in Virgina, USA) from 8 different networks located in Israel, almost 10,000km away. The maximum standard deviation was 0.081 seconds. Hereinafter, we will use a standard deviation value $\sigma = 0.1$ seconds as a worst case scenario for UDP jitter.

Packet Loss We identified two different packet loss scenarios:

- Packet loss during generation: the WebRTC packet stream (in Chrome-like browsers) is bursty in nature. In some bursts, we noticed large chunks of missing packets. These are quite rare (in the STUN traffic measurement experiment we got 29 such cases out of 1014 – 2.9%, though they are more common in Androids whose kernel is 4.x and have $f = 100$) and easily identified. We can safely ignore them because the tracker can detect a burst with a lot of missing packets, reject the sample and run the sampling logic again, or use a more sophisticated logic incorporating information from more than two bursts. Additionally, with $f = 100$ there are far less false pairs, which helps the analysis.
- Network packet loss: the UDP protocol does not guarantee delivery, and indeed packets get lost over the Internet. The loss rate is not high, however, and we estimate it to be $\leq 1\%$. This is also backed by research.[16]

[15] See https://chromium.googlesource.com/external/webrtc/+/master/p2p/base/stunrequest.cc).

[16] See http://www.verizonenterprise.com/about/network/latency/, and [4].

6.4 The Tracking Technique

The technique that we use is different than prior art techniques in focusing on bucket *collisions*. That is, in cases wherein UDP datagrams for two different destination IP addresses end up with IPID generated using the same counter.

The tracker needs to control L Internet IPv4 addresses, such that the IP-level traffic to these addresses (and particularly, the IP ID field) is available to the tracker. Ideally the IPs are all in the same network, so that they are all subject to the same jitter distribution. The tracker should be able to monitor the traffic to these IP addresses with time synchronization resolution of about 10 milliseconds (or less) - e.g. by having all the IPs bound to a single host.

With L different destination IP addresses and M buckets ($M = 2048$ in Algorithm 2), there are $\binom{L}{2}/M$ expected collisions, assuming no packet loss. In reality, the tracker can only obtain an *approximation* of this set. The goal is to reduce those false negatives and false positives to levels which allow assigning meaningful tracking IDs.

The basic property that enables the attacker to construct the approximate list is that in an IP ID generation the counter is updated by a random number which is smaller than 1 plus the multiplication of the timer frequency f and the time that passed since the last usage of that counter. Therefore for a true pair (IP^i, IP^j) where the IP ID generation for IP^i and IP^j used the same bucket (counter), the following inequality almost always holds:

$$0 < (IPID^j - IPID^i) \mod 2^{16} < f\Delta t + 10$$

(We use $f\Delta t + 10$ instead of $f\Delta t + 1$ to support up to 10 IPs colliding into the same bucket, as each collision may increment the counter by $\leq 1 + f\Delta t$ where Δt is from the *previous* collision. So the counter can end up incrementing no more than $f\Delta t + 10$ where Δt is the sum of the time difference between collisions, i.e. the time duration between the first collision and the last collision in the burst.)

Since we are looking at datagrams from the same burst we have an upper bound δ_L such that $\Delta t < \delta_L$, and therefore:

$$0 < (IPID^j - IPID^i) \mod 2^{16} < f\delta_L + 10$$

For two IP addresses which are *not* mapped to the same counter, the likelihood of this inequality to hold is only $\frac{(f\delta_L+10)-1}{2^{16}}$ which is $\ll 1$ when $f\delta_L \ll 2^{16}$. The key extraction algorithm (Section 6.6) will examine IP ID values in two different communication bursts, and this will further reduce the likelihood of a false positive. Note that the probability of a false positive pair in a given burst to survive into the next burst is roughly $\frac{f\delta_L+10}{f\Delta t} \approx \frac{\delta_L}{\Delta t}$ where Δt is the time between the consecutive bursts, whereas a true pair will occur in all bursts. Thus for the intersection of 2 consecutive bursts $\Delta t = 4$ seconds apart, the amount of false positives (in both bursts) will be ≈ 0.15 of their amount in a single burst.

6.5 Attack Phase 1 – Collecting Collisions

The tracking snippet needs to be rendered for at least 8.5 seconds, enough time for the browser to send the first 6 STUN bursts $(B_0,\ldots,B_5)$ – see Section 6.3. The tracking server splits the STUN traffic to bursts, based on the datagrams' time of arrival, and on the expected burst time offsets (see Section 6.3). For simplicity and ease of analysis, we henceforth only use traffic from bursts B_4 and B_5, which can be easily and unambiguously determined (since they are well separated in time from other bursts). We note that in some cases, requests in B_4 or in B_5 may be unsent, and in such cases we may need to resort to using e.g. B_3 and B_5 or similar combinations, but as long as these are "late" bursts (i.e. separated from their neighboring bursts by a enough σ units, where σ is the UDP jitter, see above), they can be separated without errors (or almost without errors) and the following analysis remains valid. If there are too many missing requests in a burst, the Tracking Server communicates with the Tracking Snippet, instructing it to retest the device.

Assuming no (or few) missing requests in B_4 and B_5, the Tracking Server starts analyzing the data per burst (in B_4 and B_5). For each burst the Tracking Server calculates a set of pair candidates by collecting pairs of IP addresses (IP^i, IP^j) for which $IP^i < IP^j$ and $0 < (IPID^j - IPID^i) \mod 2^{16} < \lambda_L$ where $\lambda_L = f\delta_L + 10$. It then identifies pairs which appear in the candidate sets of *both* bursts, and adds them to a set U of full candidates. This set forms a single *measurement* of a device. The tracker calculates the tracking ID based on U in Phase 2.

6.6 Attack Phase 2 – Exhaustive Key Search

In the second phase the tracking server runs an exhaustive search on the key space W where the key is 32 bits long for algorithms A_1 and A_2, 41 bits long for algorithm A_3 (Linux) and 48 bits for A_3 (Android). For each candidate key, the algorithm counts how many IP pairs in U are predicted by the candidate key. It is expected that only in one (the correct) key, this number will exceed a threshold ν, and in such case, this will be returned as the correct key (and the device ID). See Algorithm 3 for details (the algorithm uses the notation $h'(\ldots,k) = h(\ldots, protocol \oplus g(net), key)$ where k is split into $g(net), key$).

We assume here knowledge of the version of the algorithm (A) used – A_1, A_2 or A_3. For A_1 and A_2, the key space size is $|W| = 2^{32}$, and for A_3, it is 2^{41} for the x64 architecture and 2^{48} for the ARM64 architecture (see Section 6.7.)

As explained in Section 6.11, false positives ($|X| > 1$) are very rare – they can be handled but as this complicates the analysis logic, it is left out of the paper.

Attack run time Where $|U| = P$ pairs, the run time of Algorithm 3 is proportional to $|W|P$. P's distribution depends on f; Table 1 summarizes the expectancy and standard deviation for common f values. These were approximated by a computer simulation (100 million iterations.)

Algorithm 3 Exhaustive key search

```
1:  procedure GENERATE-ID(U, IP_src)    ▷ U is defined in Section 6.5
2:      if |U| < ν then
3:          return ERROR
4:      X ← ∅
5:      for all 0 ≤ k < W do
6:          Y ← {(IP^i, IP^j) ∈ U | h'(IP^i, IP_SRC, k) = h'(IP^j, IP_SRC, k)}
7:          if |Y| ≥ ν then
8:              X ← X ∪ {k}
9:      if |X| > 0 then
10:         return X    ▷ Needs special treatment if |X| > 1
11:     else
12:         return ERROR
```

Table 1: Approximated P distribution

f [Hz]	$E(P)$	$\sigma(P)$	$\sigma(P)/E(P)$
100	50.59	7.39	0.146
250	65.47	8.60	0.131
300	70.45	8.79	0.125

Time/memory optimization When the number of devices to measure is much smaller than $|W|$ it is possible to optimize the technique for repeat visits. The optimization simply amounts to keeping a list Λ of already encountered *key* values (or $(g(net), key)$ values), and trying them first. If a match is found (i.e., this is a repeat visit), there is clearly no need to continue searching the rest of the key space. Otherwise, the algorithm needs to go through the remaining key space.

Targeted tracking Even if the key space W is too large to make it economically efficient to run large scale device tracking, it is still possible to use it for targeted tracking. The use case is the following: The tracking snippet is invoked for a specific target (device), e.g. when a suspect browses to a honeypot website. At this point, the tracker (e.g. law enforcement body) extracts the key, possibly using a very expensive array of processors, and not necessarily in real time. Once the tracker has the target's key, it is easy to test any invocation of the tracking snippet against this particular key and determine whether the connecting device is the targeted device. Moreover, if the attacker targets a single device (or very few devices), it is possible to reduce the number of IP addresses used for re-identifying the device, by using only IP addresses which are part of pairs that collide (into the same counter bucket) under the known device key. Thus we can use a single burst with as few as 5 IP pairs per device to re-identify the device. The dwell time in this case drops to

near-zero.

6.7 The Effective Key Space in Attacking Algorithm A_3

In Algorithm A_3, 32 bits of the net namespace are extracted by a function we denote as $g()$, and are added to the calculation of the hash value. The attack depends on the effective keyspace size $|W| = |\{key\}| \times |\{g(net)\}| = 2^{32} \cdot |\{g(net)\}|$.

We analyzed the source code of Linux kernel versions 4.8 and above on x64, and 4.6 and above on ARM64, and found that if KASLR is turned off then the effective key space size is 32 bits in both x64 and ARM64. If KASLR is turned on, then the effective key space size is 41 bits in x64 and 48 bits in ARM64.

6.8 KASLR Bypass for Algorithm A_3

By obtaining $g(net)$ as part of Attack Phase 2 (Section 6.6), the attacker gains 32 bits of the address of the `net` structure. In single-container systems such as desktops and mobile devices, this `net` structure resides in the `.data` segment of the kernel image, and thus has a fixed offset from the kernel image load address. In default x64 and ARM64 configurations, the 32 bits of $g(net)$ completely reveal the random KASLR displacement of `net`. This suffices to reconstruct the kernel image load address and thus fully bypass KASLR.

6.9 Optimal Selection of L

Since IP addresses are at premium, we choose a minimal integer number L of IP addresses such that at the point ν where $Prob(FN) + Prob(FP)$ is minimal, $Prob(FP) + Prob(FN) \leq 10^{-6}$. We assume $f = 300$ (worst case scenario). For simplicity, at this stage we neglect packet loss, and assume that $\delta_L = \frac{L}{400}\delta_{400}$ (we assume $\delta_L \propto L$, and we measured δ_{400}). For false negatives, we use the Poisson approximation of birthday collisions [3] with $\lambda = \binom{L}{2}/M$. Therefore:

$$Prob(FN) \approx \sum_{i=0}^{\nu-1} \frac{\lambda^i e^{-\lambda}}{i!}$$

For false positives, we also assume that a burst contains the average number of false pairs and true pairs $A = \lfloor \frac{f\delta_L+10}{f\Delta t}\binom{L}{2}\frac{f\delta_L+10}{2^{16}} + \frac{\binom{L}{2}}{M} \rfloor$. We note that the probability for a single false key to match exactly k pairs is $\binom{A}{k}(\frac{1}{M})^k(1-\frac{1}{M})^{A-k}$. The probability of $|W|-1$ false keys to generate at least one false positive key is therefore:

$$Prob(FP) \approx 1 - \Big(\sum_{i=0}^{\nu-1}\binom{A}{i}(\frac{1}{M})^i(1-\frac{1}{M})^{A-i}\Big)^{|W|-1}$$

Assuming $|W| = 2^{48}$ (worst case – Android), we enumerated over all ν values for each L in $\{200, 250, \ldots, 500\}$ to find the optimal ν (per L). We found that $L = 400$ (with $\nu = 11$) is the minimal "round" L satisfying $Prob(FP)+Prob(FN) \leq 10^{-6}$ at its optimal ν.

6.10 A More Accurate Treatment for $L = 400$

Using a computer simulation, we approximated the distributions of all collisions $p_A(n)$ (using 10^8 simulation runs), and of true collisions $p_T(n)$ (using 10^9 simulation runs). The simulations took into account 1% packet loss. With these, we can calculate more accurate approximations:

$$Prob(FN) \approx \sum_{i=0}^{\nu-1} p_T(i)$$

$$Prob(FP) \approx 1 - \sum_n p_A(n)\Big(\sum_{i=0}^{\nu-1}\binom{n}{i}(\frac{1}{M})^i(1-\frac{1}{M})^{n-i}\Big)^{|W|-1}$$

(We use the convention $\binom{n}{k} = 0$ where $k > n$.) We enumerated over values $1 \leq \nu \leq 20$ for $L = 400$ and $|W| = 2^{48}$ (worst case – Android.) The minimal $Prob(FP) + Prob(FN)$ is at $\nu = 11$, where $Prob(FP) = 6.2 \times 10^{-10}$ and $Prob(FN) = 4.2 \times 10^{-8}$. We get the same optimal ν value for $L = 400$ as we got in Section 6.9, which means that the approximation steps we took there are reasonable.

6.11 Practical Considerations

Controlling packets from the browser As explained in Section 6.3, it is possible to emit UDP traffic to arbitrary hosts and ports using WebRTC. The packet payload is not controlled. The tracker can use the UDP destination port in order to associate STUN traffic to the same measurement.

Synchronization and packet transmission/arrival order Unlike the Windows technique, in the Linux/Android tracking technique there is no need to know the exact transmission order of the packets within a single burst.

False positives and false negatives Using a computer simulation with $L = 400$ destination IP addresses, a burst length of $\delta_L = 0.6$ seconds, and packet loss rate of 0.01, we calculated an approximation of for the false negative rate of 4.2×10^{-8} for $\nu = 11$, and an approximation for the false positive rate of 6.2×10^{-10}. These approximations were computed assuming $|W| = 2^{48}$ (worst case – Android). See Section 6.10 for more details.

Device ID collisions The expected number of pairs of devices with colliding IDs, due to the birthday paradox, and given R devices and a key space of size $|W|$, is $\binom{R}{2}/|W|$. For Algorithms A_1 and A_2 the key space size is $|W| = 2^{32}$, and will cause device ID collisions once there are several tens of thousands of devices. For $R = 10^6$ this will affect 0.00023 of the population (2 out of every 10,000 devices). For Alg. A_3, the key space size (with KASLR) is $\geq 2^{41}$, so collisions start showing up with R in the millions. Even for $R = 128 \cdot 10^6$, collisions affect only 0.00006 of the population.

Dwell time In order to record B_5, the snippet page needs to be loaded in the browser for 8-9 seconds. Navigating away from the page will immediately terminate the STUN traffic.
Environment factors All the UDP-related topics in Section 4.8 are applicable as environment factors on the Linux/Android tracking technique.
Longevity The device ID remains valid as long as the device is not shutdown or restarted. Mobile devices are rarely shut down, and are typically restarted only on updates, which happen once every several months, or even less frequently.
Scalability The attack is scalable. Device ID collisions are rare even with many millions of devices (see above). False positives and false negatives are also rare (less than 4.3×10^{-8} combined). From a resource perspective, the attack uses a fixed number of IPs and servers, and a fixed-size RAM/disk. The required CPU power is proportional to the number of devices measured per time unit. Network consumption per test is negligible – approx. 13.5KB/s (at the IP level) during measurement.

6.12 Possible Countermeasures

Increasing M Changing the algorithm to use a larger number M of counters, will reduce the likelihood of pairs of IP addresses using the same counter. In response to such a change the tracker can increase the number L of IP addresses that is uses. The expected number of collisions is $\binom{L}{2}/M$, and therefore increasing M by a factor of c requires the attacker to increase L by only a factor of $\sqrt{c}$.

On the other hand, δ_L also grows (probably linearly in L), and when $f\delta_L \geq 2^{16}$ no information is practically revealed to the tracker. It is probably safe to assume that the tracker can handle an increase of L by a factor of $\times 10$, which means that in order to stop the attacker the IP ID generation algorithm must increase M by more than $\times 100$, making it too memory expensive to be practical.
Increasing the key size (W) This can be an effective counter-measure for the exhaustive search phase, though the pair collection phase is unaffected by it. Yet some choices of the hash function h might still allow fast cryptanalysis.
Strengthening h Our analysis does not rely on any property of the hash function h, except that it is more-or-less uniform. Thus, changing h will not affect our results.
Replacing the algorithm See the last item in Section 4.9.

7 Experiment – Attacking Linux and Android Devices in the Lab

In order to verify that we can extract the key used by Linux and Android devices, we need to control hundreds of IP addresses. Controlling such a magnitude of Internet-routable IP addresses was logistically out of scope for this research. Therefore we had to settle for an in-the-lab setup, which naturally limited the number of devices we could test.

7.1 Setup

We connected the tested devices to our own WiFi access point, which advertised our laptop as a network gateway. Then we launched a Chrome-like browser inside the Linux/Android device, and navigated to a page containing a tracking snippet. The tracking snippet used WebRTC to force UDP traffic to a list of $L = 400$ hosts, and this traffic passed through our laptop (as a gateway) and was recorded.

We then ran the collision collection logic (Phase 1), and fed its output (IP pairs whose IP IDs collide) to the exhaustive key search logic (Phase 2). For KASLR-enabled devices, we also provided the algorithm with the offset (relative to the kernel image) of `init_net`, which we extracted from the kernel image file given the build ID (can be inferred e.g. from the `User-Agent` HTTP request header). We expected that the algorithm will output a single key, which will match a large part of the collisions.

7.2 Results

We tested 2 Linux laptops and 6 Android devices, together covering the vast majority of operating system and hardware parameters that regulate the IP ID generation. The results from all tests were positive - our technique extracted a single key and a kernel address of `init_net` where applicable (which was identical to the address in `/proc/kallsyms`). Note that due to hardware availability constraints, for the Pixel 2XL case ($|W| = 2^{48}$), we provided the algorithm with the correct 16 bit kernel displacement to reduce the key search to 2^{32}. Table 2 provides information about the common kernel versions, their parameter combinations and the tested devices.

The Attack time column is the extrapolated attack time in seconds with 10,000 Azure B1s machines, based on $E(P_f)$ from Table 1, i.e. the average attack time is $r \cdot |W| \cdot E(P_f)$ where r is the time it takes a single B1s machine to test a single key with a single pair, divided by 10,000. The standard deviation of the attack time for a given f is $r \cdot |W| \cdot \sigma(P_f)$, which is $\sigma(P_f)/E(P_f)$ in Table 1 times the average attack run time in Table 2. From a calibration run (single B1s machine, 10 pairs, 2^{32} keys, 294.83 seconds run time) we calculated $r = 6.8645 \times 10^{-13}$, and populated the Attack Time column in Table 1 with $r \cdot |W| \cdot E(P_f)$.

Applicability in-the-wild While our tests were carried out in the lab, we argue that the results are representative of an in-the-wild experiment with the same devices. We list the following potential differences between in-the-lab and in-the-wild experiment, and for each difference, we note why our experiment can be projected to an in-the-wild scenario.

- Packet loss: our technique is not sensitive to packet loss. We ran false positive/negative computer simulations (assuming 1% packet loss) supporting this fact.

Table 2: Common Linux/Android Kernels and Their Parameter Combinations

O/S	Kernel Version	Alg.	f [Hz]	KASLR	NET_NS	ρ	$\log_2 \lvert W \rvert$	Tested System	Attack Time [s]
Linux (x64)	4.19+	A_3	250	Yes	Yes	12	41	Dell Latitude E7450 laptop	99
Linux (x64)	4.8-4.18.x	A_3	250	Yes	Yes	6	41	Dell Latitude E7450 laptop	99
Android (ARM64)	4.4.56+, 4.9, 4.14	A_3	300/ 100	Yes	Yes	6/7	48	Pixel 2XL ($\rho = 6$)	13,612/ 9,775
Android (ARM64)	3.18.17+ 3.4.109+	A_2	100	No	No	Don't care	32	Redmi Note 4 Xiaomi Mi4	0.15
Android (ARM64)	3.18.0-3.18.6 3.10.53+ 3.4.103-3.4.108	A_1	100	No	No	Don't care	32	Samsung J7 prime Samsung S7 Meizu M2 Note	0.15

- Network latency: our technique is not sensitive to network latency (which is just a constant time-shift, from our perspective).
- UDP jitter: this only affects correctly splitting the traffic into bursts. Our technique uses the "late" bursts, thus assuring that the bursts are well separated time-wise and that a jitter of $\sigma = 0.1$s does not affect tracking.
- Network interference (IPID modification): this issue was already evaluated in-the-wild in the Windows experiment, and the Windows results can be applied to the Linux/Android use case.
- Packet reordering (within a burst): Our technique does not rely on packet order within a burst.

Thus we conclude that our results (and henceforth, the practicality of our technique) are applicable in-the-wild.

8 Conclusions

Our work demonstrates that using non-cryptographic random number generation of attacker-observable values (even if the values themselves are not security sensitive), may be a security vulnerability in itself, due to an attacker's ability to extract the key/seed used by the algorithm, and use it as a fingerprint of the system.

We stress that any replacement cryptographic algorithm must not be hampered by using a key that is too short, in order to avoid a key enumeration attack. Also, as a security measure, we strongly recommend generating unique keys for such cryptographic usage, without resorting to using secret data that is used for other purposes (which – in case of a cryptographic weakness in the algorithm – can leak out).

9 Acknowledgements

This work was supported by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Directorate in the Prime Minister's Office.

We would like to thank the anonymous reviewers for their feedback, Assi Barak for his help to the project, as well as Avi Rosen, Sharon Oz, Oshri Asher and the Kaymera Team for their help with obtaining a rooted Android device.

References

[1] B. Aboba, D. Burnett, T. Brandstetter, C. Jennings, A. Narayanan, J.-I. Bruaroey, and A. Bergkvist. WebRTC 1.0: Real-time communication between browsers. Candidate recommendation, W3C, June 2018. https://www.w3.org/TR/2018/CR-webrtc-20180621/.

[2] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel. FPDetective: dusting the web for fingerprinters. In *ACM CCS '13*, pages 1129–1140, 2013.

[3] R. Arratia, L. Goldstein, and L. Gordon. Two moments suffice for poisson approximations: The chen-stein method. *Ann. Probab.*, 17(1):9–25, 01 1989.

[4] D. Baltrunas, A. Elmokashfi, A. Kvalbein, and Ö. Alay. Investigating packet loss in mobile broadband networks under mobility. In *2016 IFIP Networking Conference and Workshops*, pages 225–233, 2016.

[5] S. M. Bellovin. A technique for counting Natted hosts. In *2nd SIGCOMM Workshop on Internet Measurement*, pages 267–272, 2002.

[6] Y. Gilad and A. Herzberg. Fragmentation considered vulnerable. *ACM Trans. Inf. Syst. Secur.*, 15(4):16:1–16:31, Apr. 2013.

[7] F. Gont. Security Implications of Predictable Fragment Identification Values. RFC 7739, Feb. 2016.

[8] Google. Issue 928273: unified plan breaks rtp datachannels with empty datachannel label, Feb. 2019.

[9] A. Herzberg and H. Shulman. Fragmentation considered poisonous. *CoRR*, abs/1205.4011, 2012.

[10] T. Hudek and D. MacMichael. RSS hashing functions.

[11] Information Sciences Institute (University of Southern California). Internet Protocol. RFC 791, Sept. 1981.

[12] A. Janc and M. Zalewski. Technical analysis of client identification mechanisms. https://www.chromium.org/Home/chromium-security/client-identification-mechanisms.

[13] M. Khan. RTCDataChannel for beginners. https://www.webrtc-experiment.com/docs/rtc-datachannel-for-beginners.html, 2013.

[14] A. Klein. Predictable javascript math.random and http multipart boundary string. http://www.securitygalore.com/site3/math_random_and_multipart_boundary.

[15] A. Klein. Detecting operation of a virtual machine (US patent 9384034), July 2016.

[16] J. Knockel and J. R. Crandall. Counting packets sent between arbitrary internet hosts. In *4th USENIX Workshop on Free and Open Communications on the Internet (FOCI 14)*, 2014.

[17] A. Kumar, V. Paxson, and N. Weaver. Exploiting underlying structure for detailed reconstruction of an internet-scale event. In *5th ACM SIGCOMM Conf. on Internet Measurement*, IMC '05, pages 33–33, 2005.

[18] H. Li and L. Mason. Estimation and simulation of network delay traces for voip in service overlay network. *2007 International Symposium on Signals, Systems and Electronics*, pages 423–425, 2007.

[19] S. Maheshwari, K. Vasu, S. Mahapatra, and C. S. Kumar. Measurement and analysis of UDP traffic over wi-fi and GPRS. *CoRR*, abs/1707.08539, 2017.

[20] G. Maone. NoScript. https://noscript.net/.

[21] P. Matthews, J. Rosenberg, D. Wing, and R. Mahy. Session Traversal Utilities for NAT (STUN). RFC 5389, Oct. 2008.

[22] A. Melnikov and I. Fette. The WebSocket Protocol. RFC 6455, Dec. 2011.

[23] R. Menscher. Exploiting Windows' IP ID randomization bug to leak kernel data and more (CVE-2018-8493). https://menschers.com/2018/10/30/what-is-cve-2018-8493/, November 2018.

[24] Microsoft. CVE-2018-8493 | windows TCP/IP information disclosure vulnerability. https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/CVE-2018-8493.

[25] L. Orevi, A. Herzberg, and H. Zlatokrilov. DNS-DNS: DNS-based de-nat scheme. In *Cryptology and Network Security CANS*, pages 69–88, 2018.

[26] D. Reed, P. S. Traina, and P. Ziemba. Security Considerations for IP Fragment Filtering. RFC 1858, Oct. 1995.

[27] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, Mar. 2012.

[28] F. Salutari, D. Cicalese, and D. Rossi. A closer look at ip-id behavior in the wild. In *International Conference on Passive and Active Network Measurement (PAM)*, Berlin, Germany, Mar. 2018.

[29] H. Shulman. Pretty bad privacy: Pitfalls of DNS encryption. In *13th Workshop on Privacy in the Electronic Society*, WPES '14, pages 191–200, 2014.

[30] M. J. Silbersack. darwin-xnu/bsd/netinet/ip_id.c. https://opensource.apple.com/source/xnu/xnu-4570.41.2/bsd/netinet/ip_id.c.

[31] H. Wramner. Tracking users on the world wide web. http://www.nada.kth.se/utbildning/grukth/exjobb/rapportlistor/2011/rapporter11/wramner_henrik_11041.pdf, 2011.

[32] M. Zalewski. *Silence on the Wire*. No Starch Press, 2005.

A Details of the Attack on Windows

A.1 Practical Considerations

A.1.1 Controlling Packets from the Browser

UDP: As explained in Section 6.3, it is possible to emit UDP traffic to arbitrary hosts and ports using WebRTC. The packet payload is not controlled. The tracker can use the UDP destination port in order to associate STUN traffic to the same measurement.

TCP: WebSocket [22] emits TCP traffic in a controlled fashion once a circuit is established, thus can be used by the snippet to fully control packet transmission. The downside of using TCP-based protocols is the TCP-level retransmission, which can introduce loss of synchronization between the device and the server side, regarding how many packets were sent. The tracker can use the packet payload to mark packets that belong to the same measurement.

Table 3: Common tail length probability - measured with 1000 randomly chosen sets of 30 IPs ($J=6, G=12, Q=3$), 10,000 tests each (10^7 tests altogether).

Common tail [bits]	Prob.
45	0.9937579
44	0.0058328
43	0.0003783

Common tail [bits]	Prob.
42	2.6×10^{-5}
41	2.9×10^{-6}
≤ 40	2.1×10^{-6}

A.1.2 Packet Transmission Order

We encountered cases in the wild where the packet payload generation order is not identical to packet transmission order. Specifically, Microsoft IE and Edge are prone to this behavior. This is only relevant in the same class B network (since there the original extraction algorithm makes an assumption on the order of the packets). Therefore, the tracker should try all possible permutations of packet order (per class B network IPs). In Phase 1, this means enumerating over all $\pi \in S_J$ ($J!$ permutations). For each permutation, use the following definition of D^j (instead of the original one):

$$D^j = (\mathrm{Vec}(IPID^j - (\beta_0 + \pi(j)) \mod 2^{15}) \oplus$$
$$\mathrm{Vec}(IPID^0 - (\beta_0 + \pi(0)) \mod 2^{15}))_{17,\ldots,31}$$

It follows that enumerating over all possible orders will increase the run time of Phase 1 by a factor of $J!$. In Phase 2, for each pair of IP addresses, there are only 2! permutations, and since the elimination is so powerful, this will only affect the run time due to the first pair, i.e. will double the run time.

A.1.3 Handling False Positives

The issue of false positive keys is covered in the extended paper. As mentioned there, the vast majority of false positive keys only differ from the correct key by a few leftmost bits. Table 3 demonstrates that with an optimal choice of 30 IP addresses, if the tracker keeps only 41 bits of the key tail, he/she will get multiple keys with probability 2.1×10^{-6}, which is sufficiently small even for a large scale deployment.

In the case where multiple keys are emitted by the algorithm (even after truncation, e.g. to 41 bits), two strategies can be applied: either (a) determining that this particular device cannot be assigned an ID (at the price of losing 2.1×10^{-6} of the devices); or (b) assigning multiple IDs to the device (which makes tracking the device more complicated and more prone to ID collisions).

A.1.4 Handling False Negatives and Interference

It is important to note that while there may be false positives, there are no algorithmic false negatives, i.e. the algorithm always emits the correct key (possibly along with incorrect keys), given the correct data. However, it is possible for the algorithm to receive incorrect data, in the sense that IP IDs are provided which are not (even after re-ordering) derived from an incrementing counter – i.e. there are "gaps" in the counter values associated with the IP IDs. This can happen either due to packet loss or due to interference.

Packet loss: In TCP, when a packet from the client to the server is lost, the client will note a missing ACK and will eventually retransmit the original data (with incremented IP ID). This will cause a gap in the counter values, which can be enumerated by the analysis logic (our analysis logic does not currently implement this). Another packet loss scenario is wherein the ACK packet from the server to the client is lost, and the client retransmits the original data. The server however receives two such packets, and can simply discard the one with incremented IP ID. Thus the "problematic" scenario is the one wherein the original data packet is lost.

Interference: Theoretically, an unrelated packet sent by the Windows device in between measurement packets may interfere with the measurement. However, this is very unlikely. First, the interfering packet must fall into the same counter bucket as that of the measurement packets – this happens with probability of $1/8192$ for a given bucket. Second, the timing is delicate – the interfering packet should be sent between the first and the last measurement packet. This time window is below 1 second, so overall the likelihood for interference is very low. When such an interference happens, it creates a gap in the IP ID values, which can be addressed as explained below.

Addressing "gaps": The analysis logic can compensate for up to l lost packets in the first class B network ($g=0$) by enumerating over all possible $\sum_{d=0}^{l} \binom{J-2+d}{J-2}$ gap configurations (each addendum counts all weak compositions of d into $J-1$ parts). In our experiment, we measured $l=4$ for some "difficult" networks, so for $J=6$ there are 126 gap configurations, thus a $\times 126$ factor in the runtime. Note that a gap in the transmissions for $g>0$ is much easier to handle, as $(IPID^{g,1} - IPID^{g,0})$ in eq (5) may now take values in $\{1,\ldots,l+1\}$, so there is no runtime factor for this case. When the total gap space is larger than l, the algorithm will yield no key. In such a case, the server can instruct the snippet to run another test. Therefore the actual false negative probability can be reduced to as small as necessary.

A.2 Optimizing the IP Set for Minimum False Positives

Since (from Table 3) keys with flipped K_{18} are the source of most false positives, the tracker should choose a set of IP addresses that minimizes (over Q) this false positive probability, $2^{-(J-1)-Q} + 2^{-(G-Q)}$ (this calculation can be found in the extended paper.) The said minimum is at $Q = {}^{G-(J-1)}/_{2}$, and yields false positive leading term of $2 \cdot 2^{-\frac{G+J-1}{2}}$. For $G=12$, $J=6$, the optimum is at $Q=3$ or $Q=4$.

When the Signal is in the Noise: Exploiting Diffix's Sticky Noise

Andrea Gadotti[*a], Florimond Houssiau[*a], Luc Rocher[*a,b], Benjamin Livshits[a], and Yves-Alexandre de Montjoye[†a]

[a]*Department of Computing and Data Science Institute, Imperial College London*
[b]*ICTEAM, Université catholique de Louvain*

Abstract

Anonymized data is highly valuable to both businesses and researchers. A large body of research has however shown the strong limits of the de-identification release-and-forget model, where data is anonymized and shared. This has led to the development of privacy-preserving query-based systems. Based on the idea of "sticky noise", Diffix has been recently proposed as a novel query-based mechanism satisfying alone the EU Article 29 Working Party's definition of anonymization. According to its authors, Diffix adds less noise to answers than solutions based on differential privacy while allowing for an unlimited number of queries.

This paper presents a new class of noise-exploitation attacks, exploiting the noise added by the system to infer private information about individuals in the dataset. Our first differential attack uses samples extracted from Diffix in a likelihood ratio test to discriminate between two probability distributions. We show that using this attack against a synthetic best-case dataset allows us to infer private information with 89.4% accuracy using only 5 attributes. Our second cloning attack uses dummy conditions that conditionally strongly affect the output of the query depending on the value of the private attribute. Using this attack on four real-world datasets, we show that we can infer private attributes of at least 93% of the users in the dataset with accuracy between 93.3% and 97.1%, issuing a median of 304 queries per user. We show how to optimize this attack, targeting 55.4% of the users and achieving 91.7% accuracy, using a maximum of only 32 queries per user.

Our attacks demonstrate that adding data-dependent noise, as done by Diffix, is not sufficient to prevent inference of private attributes. We furthermore argue that Diffix alone fails to satisfy Art. 29 WP's definition of anonymization. We conclude by discussing how non-provable privacy-preserving systems can be combined with fundamental security principles such as defense-in-depth and auditability to build practically useful anonymization systems without relying on differential privacy.

[†]Email: deMontjoye@imperial.ac.uk; Corresponding author.

We acknowledge the support from Agence Française de Développement. The opinions expressed in this article are the authors' own and do not reflect the view of the Agence Française de Développement.
Luc Rocher is the recipient of a doctoral fellowship from the Belgian National Fund for Scientific Research (F.R.S.-FNRS).

1 Introduction

Personal data holds a significant potential for researchers and organizations alike, yet its large-scale collection and use raise serious privacy concerns. While scientists have compared the impact of modern large-scale datasets of human behaviors to the invention of the microscope [1], numerous scandals, such as the recent Cambridge Analytica debacle [2] highlight the importance of privacy and data protection for the general public and modern societies.

Historically, the balance between using personal data and preserving people's privacy has relied, both practically and legally, on the concept of data anonymization. Anonymization, also called de-identification, is the process of transforming personal data to mask the identity of participants, e.g. by removing identifiers, coarsening data, or adding noise. The recent European General Data Protection Regulation (GDPR) defines anonymous information as "information which does not relate to an identified or identifiable natural person or to personal data rendered anonymous in such a manner that the data subject is not or no longer identifiable" [3]. Similar definitions exist in other protection laws around the world, such as the HIPAA privacy rule for medical data in the US and the ePrivacy regulation. These all state that anonymized data does not require consent from participants to be shared widely, as it cannot be traced back and potentially used against them.

While de-identification algorithms are widely used in industry and academia to transform and release anonymous datasets, a large body of research has shown that these practices are not resistant to a wide range of re-identification attacks [4–9]. Exposure of the limits of de-identification have led to less than happy conclusions by policy makers: for instance, the [US] President's Council of Advisors on Sci-

ence and Technology concluded that data anonymization "is not robust against near-term future re-identification methods. PCAST do not see it as being a useful basis for policy" [10].

Query-based systems. In response to the limits of de-identification, privacy researchers and companies have proposed query-based systems as an alternative. Such systems typically offer data analysts a remote interface to ask questions that return data aggregated from several, potentially many, records. Granting access to the data only through queries, without releasing the underlying raw data, mitigates the risk of typical re-identification attacks. Yet a malicious analyst can often submit a series of seemingly innocuous queries whose outputs, when combined, will allow them to infer private information about participants in the dataset.

Differential privacy. Privacy research has been increasingly focused on providing provable privacy guarantees to defend query-based systems against such attacks. Differential privacy [11] is the main privacy guarantee considered by researchers. Intuitively, a randomized algorithm is differentially private if the output does not depend on any single individual's record in the dataset. It has been shown that algorithms that satisfy differential privacy are robust to a very large class of attacks [12]. Yet, efficient differential privacy mechanisms are generally very use case-specific and, even if a large range of differentially private mechanisms have been proposed, there is still no practical widely deployed differential privacy solution for general-purpose data analytics [13]. To achieve strong privacy guarantees, practitioners must often sharply limit the data utility by adding large amounts of noise and restrict the total number of requests that the system is allowed to answer [14]. Moreover, while differential privacy is a strong guarantee, the risk of data breaches because of implementation issues remains, exposing systems to attacks that differential privacy should in theory rule out [15, 16]. Overall, with some rare exceptions, the complexity of correctly implementing differential privacy and choosing the right privacy budget often prevents practitioners from using it.

Alternatives to differential privacy. Diffix, a patented commercial solution that acts as an SQL proxy between an analyst and a protected database [17, 18], has recently been proposed by researchers affiliated with Aircloak and the Max Planck Institute for Software Systems as a practical alternative to differential privacy, based on the [EU] Article 29 Working Party (Art. 29 WP)'s definition of anonymous data. It defines a dataset as anonymous if the anonymization mechanism used protects against *singling out* (identify one user), *linkability* (match users across datasets), and *inference* (learn participants' records) attacks [18, 19]. Diffix relies on a novel noise addition framework called "sticky noise", which aims to give analysts a rich query syntax, minimal noise addition, and an infinite number of queries, all while satisfying the WP29 definition of anonymous.

The authors claim that data produced by Diffix (*i*) falls outside of the scope of the new European GDPR regulation; (*ii*) has been determined by the French National Commission on Informatics and Liberty (CNIL) to offer "GDPR-level anonymization" for all cases; and (*iii*) has been certified by TÜViT as fulfilling "all requirements for data collection and anonymized reporting" [20, 21]. Diffix is used in production and Aircloak reports working with partners such as Telefonica, DZ Bank and Cisco.

Exploiting Diffix's noise. In this paper we present a new class of attacks, called noise-exploitation attacks. The attacks work in three parts: (*i*) canceling out part of the sticky noise using multiple queries, (*ii*) exploiting the noise Diffix adds to one query in order to learn information about the query set associated to this query, and (*iii*) using logical equivalence between queries to obtain independent noise samples for the same query. We develop two noise-exploitation attacks that take advantage of the structure of Diffix's sticky noise to infer private (also called secret) attributes of individuals in the dataset, violating the inference requirement from the Article 29 WP definition of anonymization [19]. Our first attack, the differential attack, uses samples obtained by the difference between two queries' outputs, to discriminate between two distributions, depending on the value of the private attribute. We show that, under specific conditions, this attack potentially allows an attacker to infer private information of unique users with up to 96.8% accuracy knowing only 5 pieces of auxiliary information we call attributes.

Our second noise-exploitation attack, the cloning attack, uses dummy conditions that affect the output of queries conditionally to the value of the private attribute. This attack relies on weaker assumptions and automatically validates them with high accuracy, without the need for an oracle. It proceeds in two steps: (*i*) a *validation* step, searching for subsets of known attributes to use for the attack, that will satisfy the assumptions required for its success, and (*ii*) an *inference* step that uses the attributes found to predict users' private attribute's value. We perform the attack against four real-world datasets, and show that it can infer the private attributes of between 87.0% and 97.0% of all records across datasets. We then present an optimized cloning attack that targets 55.4% of the users and achieves the same accuracy using as little as 32 queries. This proves that introducing limits for the number of allowed queries would not protect against our attacks.

Contributions. This paper makes the following contributions:

- By developing and implementing two attacks, we demonstrate that Diffix alone does not currently satisfy the *inference* requirements of the Art. 29 WP. We make the datasets and code for the attacks and experiments available to other researchers.
- We show, using a collection of four previously published datasets that the assumptions made by our attacks are realistic. We establish that, across all datasets, between

93% and 100% of all users are value-unique (i.e. all records sharing the same set of attributes have the same private attribute).

- We make a range of defense-in-depth proposals, which can be used to improve the practical privacy guarantees of both Diffix and other non-differentially private data anonymization tools. While these measures will not result in differential privacy guarantees, they might provide adequate practical solution in many settings.
- We show, using the Diffix mechanism as our primary example, that anonymization mechanisms that do not rely on differential privacy might not be GDPR-compliant alone, and that naive data-dependent noise can lead to powerful attacks.

Paper organization. The rest of the paper is organized as follows. Section 2 describes the Diffix mechanism. Section 3 presents two attacks exploiting the noise added by the system to infer private attributes of individuals in the dataset. Section 4 shows, on real-world datasets, how an attacker can accurately attack the Diffix mechanism. Section 5 discusses the assumptions of the attacks and potential solutions for Diffix to thwart noise-exploitation attacks moving forward. Sections 6 and 7 summarize related work and provide our conclusions to build practically useful anonymization systems. Appendix A provides some details for the statistical tests used by the attack. Appendix B describes how to optimize the cloning attack to reduce the number of queries.

2 Summary of the Diffix framework

Here we summarize the Diffix framework as described in [18] and introduce notations for our attack. Diffix acts as an SQL proxy between an analyst and a *protected database D* where each row is an individual record and each column one attribute. The analyst can send SQL queries to Diffix, which will process the queries and then output a noisy answer.

We denote with $\mathcal{A}_D$ the set of attributes in the database D. For instance, $\mathcal{A}_D$ could contain 4 attributes $\mathcal{A}_D = \{gender, age, zip, HIV\}$ with HIV a binary attribute (0 or 1). A record x is a row of D with values for the attributes in $\mathcal{A}_D$. For example, with $\mathcal{A}_D$ as above, we could have $x = (M, 27, 55416, 1)$. We assume, for simplicity, that there is one and only one record for every user in D.

While Diffix can process a large part of the SQL syntax, we here focus on simple count queries:

```
SELECT count(*)
FROM table
WHERE condition_1 AND condition_2 [AND ...]
```

where every condition is an expression of the form "$attribute \square value$" with $\square$ being $=, \neq, \leq, <, \geq$, or $>$.

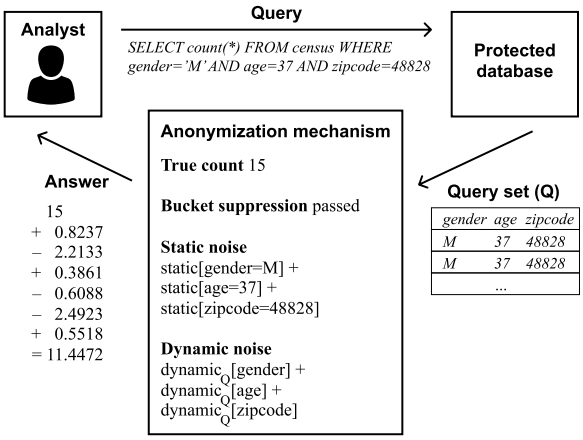


Figure 1: Diffix privacy-preserving architecture

For simplicity, we use a shorter notation for queries using "$\wedge$" for the logical AND:

$$Q \equiv \mathsf{count}(condition_1 \wedge condition_2 \wedge \ldots).$$

The following query would, for example, count the number of individuals in the database who are male, 37 years old, and live in the area with ZIP code 48828:

$$Q \equiv \mathsf{count}(gender = M \wedge age = 37 \wedge zip = 48828)$$

Diffix's privacy-preserving mechanism. Diffix protects privacy through *sticky noise* addition (static and dynamic noise) and bucket suppression (see Fig. 1). Let $Q \equiv \mathsf{count}(C_1 \wedge \ldots \wedge C_h)$, and denote by $Q(D)$ the true result of Q evaluated on D (without noise). Diffix's output for Q on D (without bucket suppression, see below) is:

$$\widetilde{Q}(D) = Q(D) + \sum_{i=1}^{h} \mathsf{static}[C_i] + \sum_{i=1}^{h} \mathsf{dynamic}_Q[C_i] \quad (1)$$

with $\mathsf{static}[C_i]$ the *static noise* for condition C_i, $\mathsf{dynamic}_Q[C_i]$ the *dynamic noise* for condition C_i in Q.

Static noise. Let C be a condition, for example $age = 34$. The *static noise* $\mathsf{static}[C]$ associated to C is a random number drawn from a normal distribution $\mathcal{N}(0,1)$. The value is generated by a pseudo-random number generator (PRNG), whose seed is a salted hash of the string literal C:

$$static_seed_C = \mathsf{XOR}(\mathsf{hash}(C), salt)$$

This ensures that the static noise associated with C is always the same independently of the query where C appears. The noise is "sticky" thereby preventing an attacker to send the same query multiple times, average out the results, and obtain a precise estimate of the private value (*averaging attack*) [18].

Dynamic noise. In the Diffix framework, every record in D is associated with a user ID, a unique string for that user.

These pseudonyms are used to compute the dynamic noise. Let $Q \equiv \mathsf{count}(C_1 \wedge \ldots \wedge C_h)$ be a query and C any condition C_i. The dynamic noise depends not only on C, but also on the *query set* of Q in the dataset D, i.e. the set of users which satisfies *all* conditions $C_1, \ldots, C_h$. More precisely, if the query set for Q on D is $S = \{uid_1, uid_2, \ldots, uid_m\}$, the dynamic noise for C ($\mathsf{dynamic}_Q[C]$) is generated from a normal distribution $\mathcal{N}(0,1)$ by the PRNG seeded with:

$$dynamic_seed = \mathsf{XOR}(static_seed_C, \mathsf{hash}(uid_1), \ldots, \mathsf{hash}(uid_m))$$

Note that we don't include D in the notation $\mathsf{dynamic}_Q[C]$, as the dataset is usually fixed and clear from the context.

The output $\widetilde{Q}(D)$ is therefore the realization of a random variable distributed as a normal distribution $\mathcal{N}(\mu, \sigma^2)$, with mean $\mu = Q(D)$ and variance $\sigma^2 = 2h$.

Example. Consider again the query

$$Q \equiv \mathsf{count}(gender = M \wedge age = 37 \wedge zip = 48828).$$

Diffix's output for Q on the database D is

$$\begin{aligned}\widetilde{Q}(D) = \quad & Q(D) \\ & + \mathsf{static}[gender = M] + \mathsf{dynamic}_Q[gender = M] \\ & + \mathsf{static}[age = 37] + \mathsf{dynamic}_Q[age = 37] \\ & + \mathsf{static}[zip = 48828] + \mathsf{dynamic}_Q[zip = 48828]\end{aligned}$$

where $\widetilde{Q}(D)$ is a random value drawn from a normal distribution $\mathcal{N}(Q(D), 6)$.

Static and dynamic noise layers are both needed to prevent *intersection attacks* [17, 18], a class of attacks that combine multiple queries to infer private attributes of records.

Bucket suppression. In addition to static and dynamic noise, Diffix implements another security measure called *bucket suppression*, similar to the classic query set size restriction. If the size of the query set is smaller than a certain threshold, the bucket suppression rejects the query. Previous research has shown that a fixed threshold constitutes a risk for privacy [22]. Diffix addresses this issue by using a *noisy* (and sticky to the query set) *threshold*. Specifically, suppose Diffix processes a query $Q \equiv \mathsf{count}(C_1 \wedge \ldots \wedge C_h)$. If $Q(D) \leq 2$, then the query gets suppressed. If $Q(D) > 1$, then Diffix computes a noisy threshold $T \sim \mathcal{N}(4, 1/2)$, using the seed:

$$threshold_seed = \mathsf{XOR}(salt, \mathsf{hash}(uid_1), \ldots, \mathsf{hash}(uid_m))$$

If $Q(D) < T$, the query is suppressed; otherwise, the noisy output $\widetilde{Q}(D)$ is computed and sent to the analyst. In the original Diffix mechanism [17], the queries are said to be "silently suppressed" when censored by bucket suppression. This could mean that (1) a value of 0 is returned as result, (2) a random value is returned or (3) Diffix displays an error message. In this paper, we assume that a bucket-suppressed query will return a value of zero. This gives less information to a potential attacker than an error message, as a value of zero can be the result of either noise addition or bucket suppression. We consider that returning a random result affects utility too significantly to be applied in practice.

3 Noise-exploitation attacks

Our noise-exploitation attacks, which we call *differential* and *cloning*, are both based on three observations. First, since the noise is sticky, it is possible to *cancel out* part of it using multiple queries. Second, since the noise depends on the query set, the noise itself leaks information about the query set. Third, exploiting logical equivalence between some queries, it is possible to circumvent the "stickiness" of the noise by repeating (almost) the same query and consequently obtain independent noise samples. Our differential attack uses samples in a likelihood ratio test to discriminate between two probability distributions depending on the value of the private (also called secret) attribute. Our cloning attack relies on dummy conditions that conditionally strongly affect the output of the query depending on the value of the secret attribute.

3.1 Differential noise-exploitation attack

We first define further notations: with $A \subseteq \mathcal{A}_D$ a set of attributes, $x^{(A)}$ is the *restriction* of the record x to A, i.e. the vector one obtains after removing from x every entry for attributes that are not in A. For example, if $\mathcal{A}_D = \{gender, age, zip, HIV\}$, $x = (M, 27, 55416, 1)$ and $A = \{gender, age, HIV\}$, then $x^{(A)} = (M, 27, 1)$. If A contains a single attribute a, we simply write $x^{(a)}$. So, for example, $x^{(gender)} = M$.

For this attack, we make the following assumptions:

- H1 The attacker wants to find out some information about Bob, the victim. The attacker knows that Bob's record is in the dataset. We denote Bob's record by x. The attacker has access to the protected dataset only through Diffix.
- H2 The attacker knows all of Bob's attributes for some set of attributes A. Our *background knowledge* (also called auxiliary information in the literature) is the restricted record $x^{(A)}$. This is a standard assumption in the literature on re-identification attacks [23, 24].
- H3 The attacker wants to infer a secret attribute s about Bob, the victim. That is, she wants to infer the value of $x^{(s)}$, where $s \notin A$. For simplicity of notation, s is a binary attribute, i.e. $x^{(s)} \in \{0, 1\}$. This means that the attack can be seen as a classifier, with the output of the attack being negative if the algorithm returns $x^{(s)} = 0$ and positive if it returns $x^{(s)} = 1$.
  While we here focus on the binary case, our results fairly easily extend to non-binary cases.
- H4 There exists an oracle Unique that takes as input any restricted record $z^{(R)}$ and outputs whether $z^{(R)}$ is *unique*.

$\mathsf{Unique}(z^{(R)})$ = True if and only if there is no other record y in D such that $z^{(R)} = y^{(R)}$.

For the attack to succeed, we first need to ensure that the background knowledge $x^{(A)}$ uniquely identifies Bob. This is given to us by the oracle $\mathsf{Unique}(x^{(A)})$. In this attack, we only attempt to infer secret attributes of records that are unique in the dataset. The cloning attack, presented later, extends this by requiring a weaker notion of uniqueness, which we call value-uniqueness.

While the existence of such an oracle is a strong assumption, it is weaker than Diffix's claims to protect against an "analyst [that] has full access to the inference dataset" [18], where the *inference dataset* is the original dataset with only the secret attribute $x^{(s)}$ removed. If the attacker has access to every record, she can easily verify that no other record shares the same restricted record $x^{(A)}$.

Firstly, the attack needs to bypass bucket suppression. For example, an attacker could ask how many records have both the background knowledge $x^{(A)}$ and the private attribute $s = 0$:

$$Q \equiv \mathsf{count}(a_1 = x_1 \wedge \ldots \wedge a_k = x_k \wedge s = 0). \tag{2}$$

with $A = \{a_1, \ldots, a_k\}$ and $x^{(A)} = (x_1, \ldots, x_k)$.

While an accurate answer to Q would immediately disclose the value of $x^{(s)}$, since $Q(D)$ can be either 0 (if $x^{(s)} = 1$) or 1 (if $x^{(s)} = 0$), this query will always be blocked by bucket suppression since the query set is either empty or {Bob}.

Intersection attacks have been proposed in the literature to circumvent similar kinds of restrictions [25]. Picking an attribute, e.g. a_1, we can define the following queries:

$$Q_1 \equiv \mathsf{count}(a_2 = x_2 \wedge \ldots \wedge a_k = x_k \wedge s = 0) \tag{3}$$

$$Q_1' \equiv \mathsf{count}(a_1 \neq x_1 \wedge a_2 = x_2 \wedge \ldots \wedge \ldots \wedge a_k = x_k \wedge s = 0) \tag{4}$$

As the record x is unique, by assumption, it is the only record that can differ between Q_1 and Q_1'. This allows us to directly compute $Q(D)$:

$$Q(D) = Q_1(D) - Q_1'(D) = \begin{cases} 0 & \text{if } x^{(s)} = 1 \\ 1 & \text{if } x^{(s)} = 0 \end{cases} \tag{5}$$

To prevent this vulnerability[1], Diffix adds static and dynamic noises:

$$\begin{aligned} \widetilde{Q}_1(D) = \quad & Q_1(D) \\ & + \sum_{i=2}^{k} \mathsf{static}[a_i = x_i] + \mathsf{static}[s = 0] \\ & + \sum_{i=2}^{k} \mathsf{dynamic}_{Q_1}[a_i = x_i] + \mathsf{dynamic}_{Q_1}[s = 0] \end{aligned}$$

[1] For the intersection attack to be successful, both Q_1 and Q_1' need to be large enough to not trigger the bucket suppression. We discuss this assumption later on.

and

$$\begin{aligned} \widetilde{Q}_1'(D) = \quad & Q_1'(D) \\ & + \mathsf{static}[a_1 \neq x_1] + \mathsf{dynamic}_{Q_1'}[a_1 \neq x_1] \\ & + \sum_{i=2}^{k} \mathsf{static}[a_i = x_i] + \mathsf{static}[s = 0] \\ & + \sum_{i=2}^{k} \mathsf{dynamic}_{Q_1'}[a_i = x_i] + \mathsf{dynamic}_{Q_1'}[s = 0]. \end{aligned}$$

The first part of our attack relies on noticing that $k-1$ static noise layers cancel out. Let

$$q_1 = \widetilde{Q}_1(D) - \widetilde{Q}_1'(D).$$

Then:

$$\begin{aligned} q_1 = \quad & Q_1(D) - Q_1'(D) \\ & - \mathsf{static}[a_1 \neq x_1] - \mathsf{dynamic}_{Q_1'}[a_1 \neq x_1] \qquad (\textit{fixed}) \\ & + \sum_{i=2}^{k} \mathsf{dynamic}_{Q_1}[a_i = x_i] + \mathsf{dynamic}_{Q_1}[s = 0] \\ & \qquad (\textit{dynamic}_{Q_1}) \\ & - \sum_{i=2}^{k} \mathsf{dynamic}_{Q_1'}[a_i = x_i] - \mathsf{dynamic}_{Q_1'}[s = 0] \\ & \qquad (\textit{dynamic}_{Q_1'}) \end{aligned}$$

leaving us with $2k+2$ noise layers: $\textit{fixed} \sim \mathcal{N}(0,2)$, $\textit{dynamic}_{Q_1} \sim \mathcal{N}(0,k)$, and $\textit{dynamic}_{Q_1'} \sim \mathcal{N}(0,k)$.

The second part of the attack relies on the fact that both $\mathsf{dynamic}_{Q_1}[a_i = x_i]$ and $\mathsf{dynamic}_{Q_1'}[a_i = x_i]$ (resp. $\mathsf{dynamic}_{Q_1}[s = 0]$ and $\mathsf{dynamic}_{Q_1'}[s = 0]$) relate to the same condition "$a_i = x_i$" (resp. "$s = 0$"). This means that the noise values for Q_1 and Q_1' will cancel out if Q_1 and Q_1' have the same query set. Therefore either $Q_1(D) - Q_1'(D) = 0$, and the $2k$ dynamic noise layers cancel out, or $Q_1(D) - Q_1'(D) = 1$, and no dynamic noise layer is canceled out:

$$q_1 \sim \begin{cases} \mathcal{N}(0,2) & \text{if } x^{(s)} = 1 \\ \mathcal{N}(1,2k+2) & \text{if } x^{(s)} = 0 \end{cases} \tag{6}$$

Using this result, an attacker can run a likelihood ratio test (see Appendix A) to estimate whether q_1 is distributed as $\mathcal{N}(0,2)$ or $\mathcal{N}(1,2k+2)$ and predict the value of $x^{(s)}$. The larger k is, the easier it becomes to discriminate between the two distributions. This alone already allows the attacker to infer Bob's secret, $x^{(s)}$, with better than random accuracy.

The third part of the attack allows us to strongly improve the accuracy of our inference. While the stickiness of the noise prevents us from running the same query again to collect more sample, we circumvent it by using different pairs of queries for which equation (6) is still true. Specifically, instead of removing (resp. negating) the condition $a_1 = x_1$, we remove

(resp. negate) other conditions $a_j = x_j$ for $j \leq k$, obtaining queries Q_j (resp. Q'_j) for $j \leq k$. In our notation:

$$Q_j \equiv \mathsf{count}\left(\bigwedge_{\substack{i=1\\ i\neq j}}^{k} a_i = x_i \wedge s = 0\right) \tag{7}$$

$$Q'_j \equiv \mathsf{count}\left(\bigwedge_{\substack{i=1\\ i\neq j}}^{k} a_i = x_i \wedge a_j \neq x_j \wedge s = 0\right) \tag{8}$$

Running all queries $\{(Q_j, Q'_j)\}_{j\leq k}$, the attacker collects a vector of realizations $\{q_j\}_{j\leq k}$ where $q_j = \widetilde{Q}_j(D) - \widetilde{Q}'_j(D)$. All q_j values are computed from different queries, which generate different noises. Hence, the noises all have different values (with probability 1). Nevertheless, the equation (6) is still true for each q_j, so we can combine them as k different samples from the same distribution, and estimate the value of $Q(D)$.

Finally, replacing the "$s = 0$" condition with "$s = 1$" in every pair (Q_j, Q'_j) defines k new pairs of queries $\{(R_j, R'_j)\}_{j\leq k}$. This allows us to obtain k more samples $\{r_j\}_{j\leq k}$ (with different noises and inverted results in equation (6)). This gives us a total of $2k$ samples before bucket suppression (see Appendix A), generated by issuing $4k$ queries.

On a technical note, observe that in principle we cannot be certain that the q_j's (resp. r_j's) are all independent samples, because two different queries Q_j and Q_l (resp. R_j and R_l) with $j \neq l$ might have the same query set. In that case, the dynamic noise layers associated to the same conditions would have the same values, and hence the total dynamic noises of the two queries would be heavily correlated. While this affects the accuracy of the likelihood ratio test, the impact is negligible in practice. Hereafter, we always assume that the samples are independent.

Full differential attack. For larger values of k, the queries used by the differential attack contain many conditions, and hence potentially select a low number of records. Dependending on the dataset, this might result in a large fraction of queries being bucket suppressed, leaving the attacker with few or no samples for the likelihood ratio test. To counteract this effect, we integrate the attack with a subset exploration step to obtain a *full differential attack*. Assume that the attacker knows a set A^* of the correct attributes for the victim with $|A^*| = k^*$, i.e. the background knowledge is $x^{(A^*)}$. The full attack proceeds as follows. The algorithm selects random subsets of A^* until it finds a subset $A \subseteq A^*$ such that $\mathsf{Unique}(x^{(A)})$ returns True. It then performs the differential attack using $x^{(A)}$ as background knowledge. For the likelihood ratio test, the attack considers only query pairs $(\widetilde{Q}_j(D), \widetilde{Q}'_j(D))$ or $(\widetilde{R}_j(D), \widetilde{R}'_j(D))$ which have outputs larger than zero in both entries. If no such pair exists, the algorithm samples a new subset A and iterates the procedure. If no feasible subset is found, the algorithm outputs NonAttackable. The subsets of A^* are sampled by decreasing size, as bucket suppression is less likely for lower values of k (but on the other hand the likelihood ratio test is less accurate).

The procedure FullDifferentialAttack presents in detail the algorithm outlined above.

Procedure DifferentialAttack($A, x^{(A)}, s$)

Input: known attributes (names A and values $x^{(A)}$), secret s

Output: True if $x^{(s)} = 1$, False if $x^{(s)} = 0$, NoSamples if $x^{(A)}$ does not yield any positive sample

$k \leftarrow |A|$, $\mathcal{Q} \leftarrow \emptyset$, $\mathcal{R} \leftarrow \emptyset$
for $j \leftarrow 1$ **to** k **do**
  $I \leftarrow \{i \in [1,k] \mid i \neq j\}$
  $\widetilde{Q} \leftarrow \mathsf{count}\left(\bigwedge_{i\in I} a_i = x_i \wedge s = 0\right)$
  $\widetilde{Q}' \leftarrow \mathsf{count}\left(\bigwedge_{i\in I} a_i = x_i \wedge a_j \neq x_j \wedge s = 0\right)$
  if $\widetilde{Q} > 0$ and $\widetilde{Q}' > 0$ **then**
    $q_j \leftarrow \widetilde{Q} - \widetilde{Q}'$
    $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{q_j\}$
  end if
end for
for $j \leftarrow 1$ **to** k **do**
  $I \leftarrow \{i \in [1,k] \mid i \neq j\}$
  $\widetilde{R} \leftarrow \mathsf{count}\left(\bigwedge_{i\in I} a_i = x_i \wedge s = 1\right)$
  $\widetilde{R}' \leftarrow \mathsf{count}\left(\bigwedge_{i\in I} a_i = x_i \wedge a_j \neq x_j \wedge s = 1\right)$
  if $\widetilde{R} > 0$ and $\widetilde{R}' > 0$ **then**
    $r_j \leftarrow \widetilde{R} - \widetilde{R}'$
    $\mathcal{R} \leftarrow \mathcal{R} \cup \{r_j\}$
  end if
end for
if $\mathcal{Q} = \emptyset$ and $\mathcal{R} = \emptyset$ **then**
  return NoSamples
end if
$f \leftarrow$ PDF of $\mathcal{N}(0,2)$, $\quad g \leftarrow$ PDF of $\mathcal{N}(1,2k+2)$
$L \leftarrow \prod_{q\in\mathcal{Q}} \frac{f(q)}{g(q)} \prod_{r\in\mathcal{R}} \frac{g(r)}{f(r)}$
return $L \geq 1$

3.2 Cloning noise-exploitation attack

In this section, we present an extension of the differential noise-exploitation attack, which we call *cloning attack*. This attack adds dummy conditions, that don't affect the query set, to queries, in such a way that several queries with different dummy conditions will have either identical or very different results conditional to the secret attribute's value.

We first introduce some new notations and definitions. Denoting by x the victim's entire record, the attacker's background information is now $x^{(A)} = (x^{(A')}, x^{(u)})$ with $A = A' \cup \{u\}$ and $|A| = k$. We use the shorthand (A', u) for $A' \cup \{u\}$. We also define a restricted record $z^{(A)}$ to be *value-unique* for the attribute s in D if $y^{(A)} = z^{(A)}$ implies $y^{(s)} = z^{(s)}$. That is,

Procedure FullDifferentialAttack($A^*, x^{(A^*)}, s$)

Input: known attributes (names A^* and values $x^{(A^*)}$), secret s
Output: True if $x^{(s)} = 1$, False if $x^{(s)} = 0$, NonAttackable if $x^{(A^*)}$ is not attackable

```
for k ← |A*| to 1 do
    for iter ← 1 to 100 do
        A ← RandomSubsetOfSize(A*, k)
        if Unique(x^(A)) and
          DifferentialAttack(A, x^(A), s) ≠ NoSamples
          then
            return DifferentialAttack(A, x^(A), s)
        end if
    end for
end for
return NonAttackable
```

every record that shares the same attributes of the restricted record $z^{(A)}$ holds the same value for the secret attribute s. To simplify the notation, we write $A' = x^{(A')}$ to indicate the condition that all attributes in A' must match the ones in $x^{(A')}$, i.e. $\wedge_{a \in A'} a = x^{(a)}$

The attack addresses several limitations of the differential attack, making it much stronger in practice.

First, the cloning attack does not require an oracle to confirm that the background information uniquely identifies a user. Instead, it replaces the oracle with a heuristic to automatically validate the assumptions.

Second, the differential attack has to ignore any pair $(\widetilde{Q}_j(D), \widetilde{Q}'_j(D))$ where at least one of the entries is not positive, as it cannot tell whether the null output comes from noise addition or from bucket suppression. This significantly reduces the total number of samples used. Our cloning attack instead uses "dummy conditions" that do not impact the user set. As queries now only differ in the dummy conditions, the corresponding query sets will always be identical. This allows us to rule out bucket suppression in case at least one output is greater than zero.

Third, while the differential attack requires records to be unique, the cloning attack only requires records to be value-unique. This is a weaker condition and makes the cloning attacks effective on a larger set of users.

Fourth, the cloning attack only requires that the set of users who share all attributes in $x^{(A')}$ is "large enough" to not be bucket suppressed. This is a weaker assumption than for the differential attack, where this needed to hold for a large number of subsets of A with one attribute removed. Furthermore, the cloning attack validates this automatically (and thus prevents bucket suppression) with high confidence.

While much stronger, the attack relies on the attacker being able to produce a set of distinct *dummy conditions* $\Delta = \{\Delta_j\}_{1 \le j \le |\Delta|}$, where each Δ_j is an SQL statement such that the set of users matching $A' = x^{(A')}$ is the same as the set of users matching $A' = x^{(A')} \wedge \Delta_j$. In section 5, we discuss how dummy conditions are easy to obtain, slow to detect, and how automatically filtering them might introduce new vulnerabilities.

Description of the attack. For each dummy condition Δ_j, we define the two queries:

$$Q_j \equiv \mathsf{count}\left(A' = x^{(A')} \wedge \Delta_j \wedge s = 0\right) \tag{9}$$

$$Q'_j \equiv \mathsf{count}\left(A' = x^{(A')} \wedge \Delta_j \wedge u \neq x^{(u)} \wedge s = 0\right) \tag{10}$$

With $q_j = \widetilde{Q}_j(D) - \widetilde{Q}'_j(D)$, we have:

$$\begin{aligned} q_j = \quad & Q_j(D) - Q'_j(D) \\ & - \mathsf{static}[u \neq x^{(u)}] \; - \mathsf{dynamic}_{Q_j}[u \neq x^{(u)}] \\ & + \sum_{i \in A'} \mathsf{dynamic}_{Q_j}[a^{(i)} = x^{(i)}] + \mathsf{dynamic}_{Q_j}[s = 0] \\ & - \sum_{i \in A'} \mathsf{dynamic}_{Q'_j}[a^{(i)} = x^{(i)}] - \mathsf{dynamic}_{Q'_j}[s = 0] \\ & + \left(\mathsf{dynamic}_{Q_j}[\Delta_j] - \mathsf{dynamic}_{Q'_j}[\Delta_j]\right) \end{aligned}$$

By the same argument we presented for the differential attack, if $x^{(s)} = 1$ then $Q_j(D) = Q'_j(D)$ and most dynamic and static noises cancel out, giving:

$$q_j = -\mathsf{static}[u \neq x^{(u)}] - \mathsf{dynamic}_{Q_j}[u \neq x^{(u)}] \tag{11}$$

As this value does not depend on the dummy condition used, we have that $q_1 = q_2 = \cdots = q_{|\Delta|}$.

On the contrary, if $x^{(s)} = 0$, then the noise layers do not cancel out with probability 1. As the noise values given by $\mathsf{dynamic}_{Q_j}[\Delta_j]$ and $\mathsf{dynamic}_{Q'_j}[\Delta_j]$ depend on Δ_j, the probability that all (or any) q_j are equal is zero.

We can therefore complete the attack by inferring that $x^{(s)} = 1$ if $q_1 = \cdots = q_{|\Delta|}$, and $x^{(s)} = 0$ otherwise. Under the current assumptions, the attack always infers the correct value with 100% confidence (up to pseudo-random collisions in Diffix's noise addition mechanism).

Robustness against rounding. In the previous section, we follow the Diffix papers [17, 18] and assume that $\widetilde{Q}(D)$ is returned directly without any rounding, admitting also negative values. We now consider the case where results are rounded to the nearest nonnegative integer, and propose a simple modification of our attack that accounts for this.

When the results of the queries Q_j and Q'_j are rounded, the corresponding q_j might not be identical if $x^{(s)} = 0$. However, the q_js will vary less if $x^{(s)} = 1$ than if $x^{(s)} = 0$. Hence, instead of checking if $q_1 = \ldots = q_{|\Delta|}$, we check if the q_j values are "similar" to one another. While for high values of k this is easy to detect, the total variance of the noise for low values

of k is small, making it harder to distinguish between the two hypotheses (i.e. whether the q_j values are "similar" or not). To overcome this issue, we "amplify" the noise for each query: instead of adding a single dummy condition Δ_j to the queries for Q_j and Q'_j, we add the conjunction $\wedge_{l\neq j}\Delta_l$:

$$Q_j \equiv \mathsf{count}\left(A' = x^{(A')} \wedge \bigwedge_{l\neq j} \Delta_l \wedge s = 0\right) \tag{12}$$

$$Q'_j \equiv \mathsf{count}\left(A' = x^{(A')} \wedge \bigwedge_{l\neq j} \Delta_l \wedge u \neq x^{(u)} \wedge s = 0\right) \tag{13}$$

This increases the total variance of the noise in q_j in the $x^{(s)} = 0$ case, making it easy to distinguish between the two hypotheses: all the q_j values will be very similar if $x^{(s)} = 1$ and fluctuate heavily if $x^{(s)} = 0$. Measuring the sample variance S^2 of $\{q_j\}_{1\leq j\leq|\Delta|}$, we infer that $x^{(s)} = 1$ if $S^2 \leq \sigma^*$, and $x^{(s)} = 0$ otherwise with a cutoff threshold σ^* chosen by the attacker. We include an empirical analysis of σ^* in the full version of this manuscript. The cloning attack is described in detail in the procedure CloningAttack.

Procedure CloningAttack($A', u, x^{(A',u)}, \Delta, s, v$)

Input: known attributes (names A', u and values $x^{(A',u)}$), dummy conditions Δ, secret s and target value v
Output: True if $x^{(s)} = v$, False if $x^{(s)} \neq v$

for $j \leftarrow 1$ **to** $|\Delta|$ **do**
  $\varphi \leftarrow A' = x^{(A')} \wedge \bigwedge_{l\neq j}\Delta_l$
  $\widetilde{Q} \leftarrow \mathsf{count}(\varphi \wedge s \neq v)$
  $\widetilde{Q}' \leftarrow \mathsf{count}\left(\varphi \wedge u \neq x^{(u)} \wedge s \neq v\right)$
  $q_j \leftarrow \widetilde{Q} - \widetilde{Q}'$
end for
$\overline{r} \leftarrow \frac{1}{|\Delta|}\sum_{j=1}^{|\Delta|} q_j, \quad S^2 \leftarrow \frac{1}{|\Delta|-1}\sum_{j=1}^{|\Delta|}(q_j - \overline{r})^2$
return $S^2 \leq \sigma^*$

Automated validation of the assumption. The cloning attack relies on two assumptions on the attacker's background knowledge $x^{(A',u)}$:

1. The queries $\{Q_j\}_{1\leq j\leq|\Delta|}$ and $\{Q'_j\}_{1\leq j\leq|\Delta|}$ in equations (12) and (13) are not bucket suppressed.
2. The user is value-unique in the dataset according to (A', u) for the secret attribute s.

We here propose procedures for an attacker to determine whether (A', u) satisfies the two assumptions with high probability.

Validating the first assumption can be done easily by submitting queries $\{Q_j\}_{1\leq j\leq|\Delta|}$ and $\{Q'_j\}_{1\leq j\leq|\Delta|}$ to Diffix. Recall that the threshold for bucket suppression for a query depends only on the corresponding query set. All the queries in $\{Q_j\}_{1\leq j\leq|\Delta|}$ have the same query set, and the same applies for $\{Q'_j\}_{1\leq j\leq|\Delta|}$. Hence, if *any* query Q_j is bucket suppressed (i.e. has output zero), then *all* queries in $\{Q_j\}_{1\leq j\leq|\Delta|}$ must have output zero, and similarly for $\{Q'_j\}_{1\leq j\leq|\Delta|}$. Thus, if *any* query Q_j and *any* query Q'_j have output higher than zero, we are sure that no query was bucket suppressed, and hence all q_j's are valid samples. The test is considered passed in this case, and failed otherwise. See the algorithm NoBucketSuppression for an implementation example.

Validating the second assumption relies on a heuristic. We run the query:

$$\mathsf{count}(A' = x^{(A')} \wedge u = x^{(u)})$$

and consider the assumption validated if the output is zero, and not otherwise. The idea is that if the output is larger than zero, then the query was not bucket suppressed, and many users are likely share the same attributes $x^{(A',u)}$, meaning that $x^{(A',u)}$ is unlikely to be value-unique. Experiments in section 4 show that this heuristic works very well on real-world datasets.

Procedure NoBucketSuppression($A', u, x^{(A',u)}, \Delta, s, v$)

Input: known attributes (names A', u and values $x^{(A',u)}$), dummy conditions Δ, secret s and target value v
Output: True if (A', u) passes the tests and is deemed to satisfy assumption 1, False otherwise

$ok_Q \leftarrow 0, ok_{Q'} \leftarrow 0$
for $j \leftarrow 1$ **to** $|\Delta|$ **do**
  $\varphi \leftarrow A' = x^{(A')} \wedge \bigwedge_{l\neq j}\Delta_l$
  $\widetilde{Q} \leftarrow \mathsf{count}(\varphi \wedge s \neq v)$
  $\widetilde{Q}' \leftarrow \mathsf{count}\left(\varphi \wedge u \neq x^{(u)} \wedge s \neq v\right)$
  if $\widetilde{Q} > 0$ **then**
    $ok_Q \leftarrow 1$
  end if
  if $\widetilde{Q}' > 0$ **then**
    $ok_{Q'} \leftarrow 1$
  end if
end for
return $ok_Q = 1$ & $ok_{Q'} = 1$

Procedure ValueUnique($A', u, x^{(A',u)}$)

Input: known attributes (names A', u and values $x^{(A',u)}$)
Output: True if (A', u) passes the tests and is deemed to satisfy assumption 2, False otherwise

$\widetilde{Q} \leftarrow \mathsf{count}(A' = x^{(A')} \wedge u = x^{(u)})$
return $\widetilde{Q} = 0$

The procedures CloningAttack and NoBucketSuppression both issue (the same) $2|\Delta|$ queries, while ValueUnique uses

only one query. Validating the assumptions and performing the attack thus requires only $2|\Delta|+1$ queries, for a given set of attributes (A', u). We empirically obtain accuracy above 93.3% with $|\Delta|$ as low as 10 (see section 4 and Appendix B).

Full cloning attack. Combining procedures CloningAttack, NoBucketSuppression and ValueUnique, we design a fully fledged procedure FullCloningAttack that performs the entire attack under the following assumptions:

- H1 The attacker knows that the victim's record x is in the dataset.
- H2 The attacker knows a set A^* of the correct attributes for the victim with $|A^*| = k^*$, i.e. the background knowledge is $x^{(A^*)}$.
- H3 The secret attribute $x^{(s)}$ is a binary attribute.

The full cloning attack includes a subset exploration step similar to the one used in the full differential attack. The algorithm selects random subsets A' of A^* (and an element u from $A^* \setminus A'$ at random) by decreasing size until it finds a subset that passes both tests, upon which it then performs the attack using $x^{(A',u)}$ as background knowledge. If no feasible subset is found, the algorithm outputs NonAttackable.

Procedure FullCloningAttack$(A^*, x^{(A^*)}, \Delta, s, v)$

Input: known attributes (names A^* and values $x^{(A^*)}$), dummy conditions Δ, secret s and target value v
Output: True if $x^{(s)} = v$, False if $x^{(s)} \neq v$

```
for k ← |A*| to 1 do
    for iter ← 1 to 100 do
        A' ← RandomSubsetOfSize(A*, k − 1)
        u ← RandomElement(A* \ A')
        if NoBucketSuppression(A', u, x^(A), Δ, s, v) and
          ValueUnique(A', u, x^(A*)) then
            return CloningAttack(A', u, x^(A), Δ, s, v)
        end if
    end for
end for
return NonAttackable
```

Reducing the number of queries. While Diffix allows each analyst to send arbitrarily many queries, we study how many queries are required to perform the cloning attack in practice. In Appendix B we present a heuristic that reduces the median number of queries by a factor of 100. Using this heuristic, the attack targets 55.4% of the users in the dataset, achieving 91.7% accuracy with a maximum of 32 queries per user in our experiments.

4 Experiments

In order to assess the effectiveness of our attacks, we implemented Diffix's mechanism for counting queries as described in the original paper [18]. The implementation outputs zero when queries are bucket-suppressed and results are rounded to the nearest nonnegative integer. We apply our attacks to four datasets and an additional synthetic dataset on which the assumptions of the differential attack are always validated.

4.1 Description of the datasets

In our experiments, we use the following datasets:

1. ADULT: U.S. Census dataset with 30,162 records and 11 attributes, incl. salary class as secret attribute [26].
2. CREDIT: credit card application dataset with 690 records and 16 attributes, incl. accepted credit as secret attribute [27].
3. CENSUS: U.S. Census dataset with 199,523 records and 42 attributes, incl. total personal income (digitized, null income as negative condition) as secret attribute [28].
4. CDR: synthetic collection of phone metadata with 2,000,000 records generated using real-world data for human behaviour and the geography of the UK for the location of antennas. Every user is a record of 11,674,870 binary attributes (an attribute being whether a user was geographically present at a certain place and time, and placed a call or received a text message). As the vast majority of the attributes in a record are null, the distribution of values for a random attribute is heavily skewed towards zero. To obtain a balanced experiment, for 50% of runs we select as the secret attribute a pair $(location, time)$ where the user was present, and for the other 50% we select a pair where the user was absent.

4.2 Differential noise-exploitation attack

Evaluation of the attack alone. We first test the differential attack on a synthetic dataset where all users satisfy the uniqueness assumption.

In the $\mathsf{Complete}_k$ dataset, every user is unique according to k attributes (excluding the secret attribute), whereas $k-1$ attributes always identify a *larger* set of users. This ensures that (*i*) every user is vulnerable to the attack and (*ii*) bucket suppression is unlikely to be triggered by the attack queries. To create the dataset, we fix an integer B and generate every possible $k-$tuple whose values are in $\{1,\ldots,B\}$. We then append to each tuple a random value of either 0 or 1 for the secret attribute. $\mathsf{Complete}_k$ contains B^k records, one for each combination of k attributes. For our experiments, we set $B = 12$, to ensure that close to no bucket suppression occurs. For computational reasons, as the size of the dataset in memory grows as B^k, the maximum k we can use is limited to 6.

Fig. 2 compares the accuracy $\mathrm{acc}(k)$ of the attack, knowing k attributes, on $\mathsf{Complete}_k$ with the theoretical accuracy. The procedure we use here is DifferentialAttack, which does *not* include subset exploration and has no access to the oracle.

Hence, this experiment simulates a realistic attacker. We report the empirical fraction of users whose secret attribute is correctly predicted, estimated by performing the differential attack on a sample of 1000 users. We also report the theoretical distribution of accuracy, (*i*) without rounding (closed-form expression, see Appendix A) and (*ii*) with rounding (numerical simulation, see Appendix A). For the Complete$_k$ dataset, the accuracy reaches 92.6% for 5 points. Even knowing only $k = 2$ attributes, the accuracy is above 66% both theoretically and empirically. While rounding has close to no effect on the theoretical accuracy of the attack, comparing the Complete$_k$ with the Theoretical (rounding) curves shows that bucket suppression and potential correlations between the samples in empirical experiments noticeably decrease the accuracy.

Evaluation on real-world datasets. We now evaluate the accuracy of the differential attack on 1,000 users selected at random in each of the four datasets. Contrary to the synthetic experiment, bucket suppression is more prevalent on real-world datasets. Therefore, we run the FullDifferentialAttack algorithm, knowing k^* attributes A^* that are selected at random for each record. If the attack outputs NonAttackable, the secret attribute is predicted at random (uniformly).

Fig. 3 shows, for each dataset and knowing k^* attributes, the percentage of unique individuals and the percentage of individuals, in each dataset, for which the secret is correctly inferred. The latter divided by the the former gives us the accuracy of our attack. Our attack realizes an accuracy of 68.4% for ADULT with $k^* = 10$, 64.0% for CREDIT with $k^* = 15$, 68.8% for CENSUS with $k^* = 40$, and 68.8% for CDR with $k^* = 6$.

Observe that the fraction of correctly inferred attributes

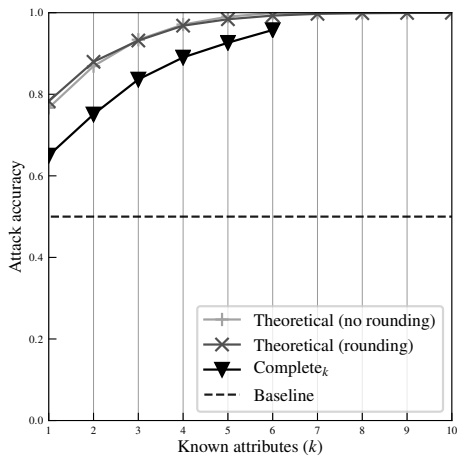


Figure 2: Accuracy of the differential attack when the uniqueness assumption is always validated and with balanced truth values. The baseline accuracy is 0.5 and represents the expected success rate when randomly predicting the secret attribute using a uniform prior.

Dataset	Attributes (k^*)	Value-unique	Predicted attackable	accuracy$_{pa}$	accuracy$_{all}$
ADULT	10	93.0%	96.8%	93.3%	87.0%
CREDIT	15	100.0%	100.0%	97.0%	97.0%
CENSUS	40	99.7%	94.6%	97.1%	91.6%
CDR	6	100.0%	100.0%	91.3%	91.3%

Table 1: Empirical results of the cloning attack on four real-world datasets.

plateaus with larger k^* for the CREDIT, CENSUS and CDR datasets. The reason is that, on these datasets, most users are unique for larger values of k^*. As explained in section 3, this makes bucket suppression more prevalent and reduces the total number of samples for the likelihood ratio test, which is a limitation of the differential attack.

4.3 Cloning noise-exploitation attack

We implement the attack as described by algorithm FullCloningAttack. As before, for each value of k^* we select 1,000 users at random, and for each user a random subset A^* of their attributes of size k^*. A^* represents the total number of attributes known to the attacker about the victim.

We set a threshold $\sigma^* = 0.7$ for the variance cutoff (see full version). We generate $|\Delta| = 10$ dummy conditions for $x_1 = a_1$ of the form $x_1 \neq b_j$ for $j \leq |\Delta|$, with b_j being some plausible values for x_1 different from a_1. We present the results when the attacker knows enough attributes (k^*) to identify every user, or up to all available attributes in the dataset.

Table 1 shows the proportion of records that are value-unique (third column) and fraction of the value-unique records that are predicted as attackable by procedures NoBucketSuppression and ValueUnique (fourth column). We then perform the cloning attack on all records that are predicted as attackable and report accuracy$_{pa}$, the fraction of predicted attackable records whose secret attribute was successfully inferred (fifth column). For completeness, we also report accuracy$_{all}$, the fraction of all records in the datasets (including the ones deemed NonAttackable) whose secret attribute was successfully inferred (last column).

Table 1 shows that the cloning attack—including the assumption validation step—performs really well on all datasets considered, between 87.0 and 97.0% of secret attributes and 97.0% on the CREDIT dataset when knowing 15 attributes.

Fig. 4 shows, knowing k^* attributes, the fraction of all records that are value-unique, predicted attackable, and correctly inferred. The curves for value-unique users and for predicted attackable users are always very close, suggesting

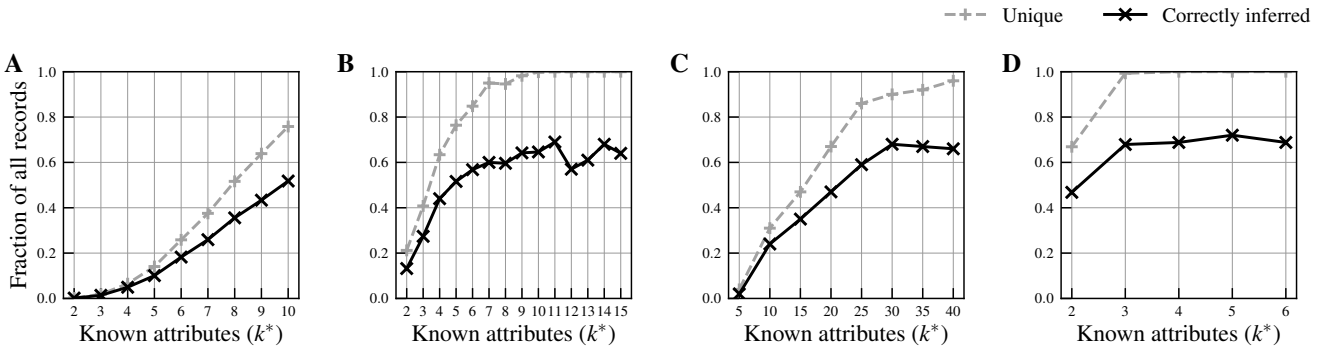


Figure 3: Results of the differential attack for the (A) ADULT, (B) CREDIT, (C) CENSUS, and (D) CDR datasets.

that the assumption validation step is effective. Out of all records predicted as attackable, most of them are correctly inferred, demonstrating that the attack works on targeted records across all k. For the CREDIT dataset, with only six attributes, the attack reaches the inference step for 95% of the users in the dataset, and correctly infers the secret attribute for 93% of the total records.

5 Discussion

5.1 Value-uniqueness and attribute predictability

Value-uniqueness plays an important role in the cloning attack. As Fig. 4 shows, it is a valid assumption for real-world datasets.

Value-uniqueness means that a group of people who share the same attributes also share the same secret attribute. If this group were to be large enough, the noise added by Diffix might not be enough to hide the secret attribute, which could then be revealed by using a simple count query. While this might be true for some datasets, it is not the case for any of the datasets we considered. For instance, the average size of the value-unique class (i.e. the set of value-unique users sharing the same restricted record) in the ADULT dataset is 1.44, with no class containing more than 4 users and similar numbers for the other datasets. This means that, most of the times, value-unique users are simply *unique*.

If secret attributes are predictable from the other attributes, a trained machine learning classifier could predict them with potentially high accuracy[2]. Despite our datasets coming from the machine learning literature, our attack does not rely at all on the predictability of secret attributes and performs equally well if no correlation at all exists between attributes and the secret attribute. We run our attack on a modified version of the ADULT dataset where sensitive attributes have been randomly sampled, thereby theoretically destroying any correlation. Our attacks perform as well on this modified dataset as on the original dataset. These results are included in the full version of this manuscript.

[2]Whether this would constitute a privacy attack is debated [29].

5.2 Producing and detecting dummy conditions

The cloning attack requires the attacker to provide a set of dummy conditions that affect the noise addition without affecting the query set. These conditions can be syntactic (e.g., $age \geq 15$ for the query $age = 23$), semantic (e.g., $status \neq retired$ for $age = 23$), or pragmatic (e.g., $age \neq 15$ against a database containing only adult individuals).

When the language is rich enough, detecting redundant clauses is not a trivial task. At the same time, the richer the syntax is, the more utility an analyst gets out of the system. Diffix offers a fairly rich syntax including boolean expression, GROUP BY, JOIN, seven aggregation functions, set membership, fourteen string functions, and ten maths functions. In this context, automatically detecting dummy conditions would likely require iterating recursively through every condition and evaluating the query with and without them, a costly operation. Moreover, if dummy conditions are detected by evaluating them on the dataset, filtering them might not be safe. Removing semantic and pragmatic dummy conditions from a query would indeed reduce the total variance of the added noise and leak information about the dataset itself (e.g., only adult individuals are present if the condition $age \neq 15$ is always removed).

5.3 Improving the attacks

The cloning attack can be modified to run a double NoBucketSuppression test: once with $v = 0$ and once with $v = 1$. If both pass and the ValueUnique test is passed as well, the attack proceeds with the actual inference procedure CloningAttack for both $v = 0$ and $v = 1$. The attack then makes a guess only if both inferences return the same value, and continues with the subset exploration otherwise (deeming the user NonAttackable if no working set of attributes is found).

We found that this modified attack improves the accuracy$_{pa}$ figure on all datasets (e.g. from 93.3% to 97.3% for ADULT,

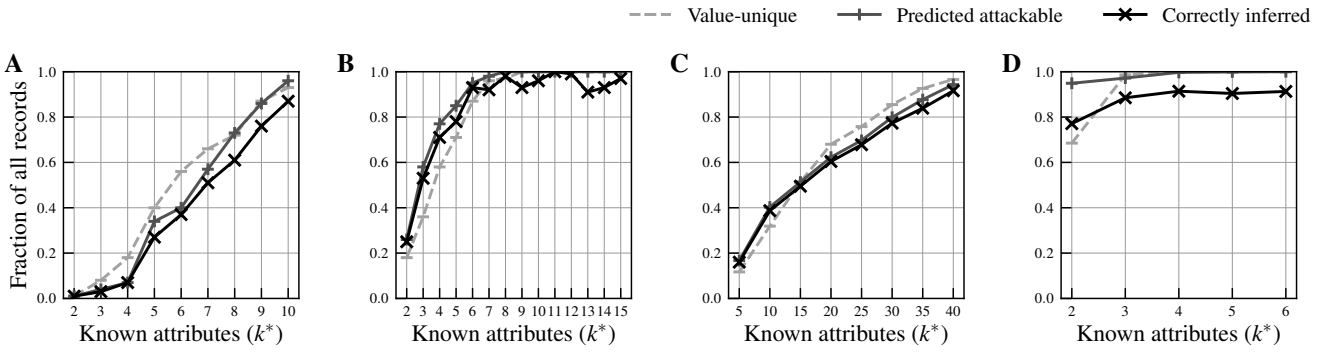


Figure 4: Results of the cloning attack for the (A) ADULT, (B) CREDIT, (C) CENSUS, and (D) CDR datasets.

all results available in the full version). However, double tests are more likely to fail, meaning that less users are predicted as attackable (from 96.8% to 87% in ADULT). Because of bucket suppression, this effect is particularly strong for datasets where the overall distribution of the secret attribute is very skewed, such as the CDR dataset where the number of predicted attackable users goes from 100% to 8.5%. Depending on the aims of the attacker (precision versus coverage), she might prefer the original or the double version of the cloning attack.

Other improvements. To properly quantify the strength of our attacks, none of them use prior knowledge on the distribution of the secret attribute. In practice, an attacker might want to use this information, e.g. obtaining it by querying Diffix. We discuss this in the full version of the manuscript. We also discuss how to generate more samples for the differential attack and outline how to generalize both attacks to infer non-binary attributes.

5.4 Defenses

In this section, we briefly outline some of the approaches that may be used to mitigate the effects of our noise-exploitations attacks – and other attacks – against Diffix and other privacy-preserving query-based systems. Overall, it is our belief that practical secure design principles apply here just as they do in many other contents. Specifically, privacy-preserving query-based system such as Diffix (regardless of whether they have provable guarantees or not) would benefit from a defense-in-depth approach, by monitoring the query stream for queries that are likely to lead to exploitation.

Intrusion detection. The set of queries generated by our attacks follow a specific template. Learning this pattern may help prevent noise-exploitation attacks, as well as potentially related attacks. A more sophisticated attacker might however vary the shape of the queries and interleave them with other more natural-looking queries, including over long periods of time.

Auditability. If the user of such a system is authenticated, then a suspicious-looking query stream can lead to temporary account suspension and further investigations of their activity, including after the fact, as new attacks are being uncovered.

Increased friction. Another strategy involves imposing time delays or financial charges on queries, for instance by charging by the number of queries, instead of using a subscription-based model. This strategy can be refined to, for instance, charge more or create longer delays for suspicious queries. This would make it more difficult to automate the inference process at scale.

Limited expressiveness. Instead of a rich syntax, the mechanism could allow only for a small set of conditions that are easier to validate. This could also include a limit to the number of conditions per query, or to the total number of conditions that may be used by the authenticated system user during a specific interval of time. This involves a compromise between rough data summarization and fine-grained queries, and limits the utility of the system in practice.

5.5 Disclosure

After we discovered and prototyped our differential attack, we reached out to the authors of Diffix and shared with them our manuscript, which subsequently appeared on ArXiv.org. A week later, the authors of Diffix published a blog post on their website [30] discussing our results. While they acknowledge our attack, they claim that it is not practical as the necessary assumptions are rarely met in the datasets they analyzed.

We disagree with this claim. First, the existence of the attack, *independently* of the dataset, contradicts both the spirit and the letter of GDPR's Art. 29 WP. Second, we showed that, albeit correct, the authors' analysis was insufficient. In this paper, we give an example of a dataset on which the differential attack is very effective, even without an oracle. Moreover, we demonstrate that there exist real-world datasets on which the necessary assumptions are met for a significant fraction of users. Third, the cloning attack, which we developed after-

wards, is able to validate its assumptions automatically and performs very well on a large range of real-world datasets.

The code of our differential and cloning attacks, as well as the experiments performed in this paper, are available at https://cpg.doc.ic.ac.uk/signal-in-the-noise.

6 Related work

Attacks on query-based systems. Diffix is an example of query-based system: the individual-level (often pseudonymous) data is stored on the data curator's server. Users access the data exclusively by sending queries that only return information aggregated from several records. While this setup prevents traditional re-identification attacks [4–9], a large range of attacks on query-based systems have been developed since the late 70's [25, 31]. Most of these attacks show how to circumvent privacy safeguards (for instance, query set size restriction and noise addition) in specific setups. In 2003, Dinur et al. [32] proposed the first example of an attack that works on a large class of query-based systems. In what they called a reconstruction attack, they showed that if the noise added to every query is at most $o(\sqrt{n})$, where n is the size of the dataset, then an attacker can reconstruct almost the entire dataset using only polynomially many queries. Sankararaman et al. [33] realized the first formal study of tracing attacks, introducing a theoretical attack model based on hypothesis testing. While reconstruction attacks aim at inferring one or more attributes of some record in the dataset (violating the inference requirement of the Art. 29 WP), the goal of tracing attacks is only to determine whether the data about a certain individual (more precisely, their record) is present in the dataset. Numerous reconstruction and tracing attacks have been proposed in the literature. These attacks address different limitations of previous ones, particularly the computational time required to perform them. A recent survey from Dwork et al. [34] gives a detailed overview of attacks on query-based systems.

Attacks on differential privacy. Differential privacy is a privacy guarantee that can be enforced by query-based systems. Differential privacy has been mathematically proven to be robust against a very large class of attacks [12] when used with an appropriate privacy budget ε. However, research has shown that attacks on *implementations* of differentially private systems exist. We give an overview in the full version.

Differential privacy for general-purpose analytics. Diffix was specifically created as an alternative to differential privacy to provide a better privacy/utility tradeoff for general-purpose analytics [35, 36]. Specifically, Diffix allows for infinitely many queries with little noise added to outputs.

General-purpose analytics usually refers to systems that allow analysts to send many queries of different type, and ideally permit to join different datasets. Some solutions based on differential privacy have been proposed, the main ones being PINQ [37], wPINQ [38], Airavat [39], and GUPT [40]. All of these systems however present limitations, e.g. simplicity of use and support for various operators that join different datasets [13]. In 2017, Johnson et al. [13] proposed a new framework for general-purpose analytics, called FLEX, developed in collaboration with Uber. FLEX enforces differential privacy for SQL queries without requiring any knowledge about differential privacy from the analyst. However, the actual utility achieved – level of noise added – by the current implementation of FLEX has been questioned [41].

Attacks on data-dependent noise. Values of Diffix's dynamic noise for a query depend on the query set (i.e. the set of users selected by the query), and hence on the data. This is what allows for our noise-exploitation attack to work. Data-dependent noise, also called instance-based noise, has been shown to provide significantly better accuracy than data-independent noise [42]. However, naive implementations of data-dependent noise can leak information about the data, a result Nissim et al. theorized as a potential way to attack the system [42]. To the best of our knowledge, our noise-exploitation attack is the first instance of an attack exploiting specifically data-dependent noise on deployed systems.

6.1 Other attacks on Diffix

We published the first version of our paper on ArXiv.org in April 2018, describing the differential attack. We updated it with a cloning attack in July 2018. Two months later, in October 2018, two other attacks on Diffix were disclosed. A membership attack by Pyrgelis et al. [43], based on a previous paper [44], and a reconstruction attack by Cohen and Nissim [45], based on previous work by Dinur et al. [32] and Dwork et al. [46]. These attacks are very different from ours and require a large number of queries in a typical setting, while our cloning attack can work with only 32 queries (see Appendix B). These are, to the best of our knowledge, the only three attacks specifically targeting Diffix.

Membership attack on location data. The attack by Pyrgelis et al. [43] is as follows: the attacker trains a machine learning algorithm on a *linkability* dataset (the attacker's background knowledge) to infer the presence of a user in a *protected* dataset (accessible only through queries on Diffix). Both datasets contain the full trajectories of users and half of the users are present in both datasets. The classifier is trained on the linkability dataset and queries on Diffix that count the number of people transiting in a certain area at a given time. The experimental results focus on the top 100 users with the highest number of reported locations in the linkability dataset. Out of 62 users present in both datasets, the classifier correctly infers the presence for 50 of them.

This attack presents three limitations. First, it is a membership attack and only allows an attacker to infer whether a person is in the protected dataset or not. Second, it assumes a strong adversary who has access to the full trajectory of a user exactly as it exists in the protected dataset, for a large number

of users. Third, the attack requires about 32,000 queries to assess the presence of a user. Membership attacks are however very useful when combined with inference attacks like ours, allowing an adversary to effectively verify our assumption that the victim is in the dataset.

Linear program reconstruction attack. The attack by Cohen et al. [45] focuses on reconstructing the dataset. In its simplest form, the attack assumes that the dataset contains n records, and each user $i \in [n]$ has a binary attribute s_i. The attack then selects random subsets of users and, for each subset $I \subseteq [n]$, queries Diffix for the result of $\sum_{i \in I} s_i$. This allows the attacker to produce a noisy linear system that can be solved using linear programming techniques to reconstruct the entire set of secret attributes $\{s_1, \ldots, s_n\}$ with perfect accuracy in polynomial time.

While this attack can successfully reconstruct the entire dataset, it presents two limitations compared to our attack.

First, it requires that the system allows queries of the type $\sum_{i \in I} s_i$, i.e. queries that select any analyst-defined set of users $I \subseteq [n]$, the "row-naming problem". The authors here exploit SQL functions supported by Diffix to define hash functions which they then use to select "random enough" sets of users. Following the disclosure, Aircloak restricted the available SQL functions to prevent the attack [47].

Second, to target a specific user, the attack would require a number of queries proportional to the number of records. Since the attacker does not know *which* name $i \in [n]$ corresponds to the victim's record, it is necessary to fully reconstruct at least a few columns entirely. The attacker would then perform a uniqueness attack on the reconstructed dataset to infer the secret attribute of the victim.

On the contrary, the number of queries used by our attacks is independent of the number of records in the dataset.

7 Conclusion

The Diffix mechanism has recently been proposed as an alternative to data anonymization methods and differential privacy, and is currently used in production. The mechanism is claimed to allow an analyst to submit an unbounded number of queries, while thwarting inference attacks, as defined by EU's Art. 29 WP. In this paper, we show that Diffix's anonymization mechanism is vulnerable to a new class of attacks, which we call noise-exploitation attacks. Our attacks leverage design flaws in Diffix's data-dependent noise to infer private attributes of an individual in the dataset, solely from prior knowledge about other attributes of this individual. In our opinion, Diffix alone and in its present state likely fails to satisfy the EU's Art. 29 WP requirements for data anonymization. Furthermore, our results show that naive data-dependent noise leads to highly vulnerable systems.

Our differential noise-exploitation attack, given little auxiliary information about the victim, combines specific queries and estimates how the noise is distributed to infer the value of the private attribute. In a synthetic best-case dataset, the attacker can predict with 92.6% accuracy private attributes, using only 5 attributes.

Our cloning noise-exploitation attack extends the first one by adding "dummy" conditions that do not change the selected query set. It relies on weaker assumptions, that are automatically validated with high accuracy by our algorithm. We evaluate its performances on four real-world datasets and find that it infers private attributes of between 87.0% and 97.0% of all records across datasets.

We finally recommend four defense-in-depth principles to defeat the de-anonymization attacks we describe.

References

[1] David Lazer, Alex Sandy Pentland, Lada Adamic, Sinan Aral, Albert Laszlo Barabasi, Devon Brewer, Nicholas Christakis, Noshir Contractor, James Fowler, Myron Gutmann, and Others. Life in the network: the coming age of computational social science. *Science*, 323(5915):721, 2009.

[2] Carole Cadwalladr and Emma Graham-Harrison. Revealed: 50 million facebook profiles harvested for cambridge analytica in major data breach. *The Guardian*, 17, 2018.

[3] Council of European Union. Regulation (EU) 2016/679. *OJ*, L 119:1–88, May 2016.

[4] L Sweeney. Weaving technology and policy together to maintain confidentiality. *J. Law Med. Ethics*, 25(2-3):98–110, 82, 1997.

[5] A Narayanan and V Shmatikov. Robust de-anonymization of large sparse datasets. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 111–125. ieeexplore.ieee.org, May 2008.

[6] Yves-Alexandre de Montjoye, César A Hidalgo, Michel Verleysen, and Vincent D Blondel. Unique in the crowd: The privacy bounds of human mobility. *Sci. Rep.*, 3:1376, 2013.

[7] Yves-Alexandre de Montjoye, Laura Radaelli, Vivek Kumar Singh, and Alex "sandy" Pentland. Unique in the shopping mall: On the reidentifiability of credit card metadata. *Science*, 347(6221):536–539, January 2015.

[8] Chris Culnane, Benjamin I P Rubinstein, and Vanessa Teague. Health data in an open world. *ArXiv e-prints*, December 2017.

[9] P Ohm. Broken promises of privacy: Responding to the surprising failure of anonymization. *UCLA Law Rev.*, 2010.

[10] President's Council of Advisors on Science and Technology. Big data and privacy: a technological perspective. Technical report, White House, January 2014.

[11] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. *Calibrating Noise to Sensitivity in Private Data Analysis*, page 265–284. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Mar 2006.

[12] Shiva P Kasiviswanathan and Adam Smith. On the 'semantics' of differential privacy: A bayesian formulation. *1*, 6(1), June 2014.

[13] Noah Johnson, Joseph P. Near, and Dawn Song. Towards practical differential privacy for sql queries. *ArXiv e-prints*, Jun 2017. arXiv: 1706.09479.

[14] Jun Tang, Aleksandra Korolova, Xiaolong Bai, Xueqiang Wang, and Xiaofeng Wang. Privacy loss in apple's implementation of differential privacy on macos 10.12. *ArXiv e-prints*, September 2017.

[15] Ilya Mironov. On significance of the least significant bits for differential privacy. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 650–661, New York, NY, USA, 2012. ACM.

[16] Andreas Haeberlen, Benjamin C Pierce, and Arjun Narayan. Differential privacy under fire. In *USENIX Security Symposium*, 2011.

[17] Paul Francis, Sebastian Probst Eide, and Reinhard Munz. Diffix: High-Utility database anonymization. In *Privacy Technologies and Policy*, pages 141–158. Springer International Publishing, 2017.

[18] P. Francis, S. Probst-Eide, P. Obrok, C. Berneanu, S. Juric, and R. Munz. Extended Diffix. *ArXiv e-prints*, June 2018.

[19] Article 29 Data Protection Working Party. Opinion 05/2014 on anonymisation techniques. April 2014.

[20] Aircloak. Announcing our seed investment. https://web.archive.org/web/20180426131132/https://blog.aircloak.com/announcing-our-seed-investment-26f392f1068a, October 2017.

[21] Aircloak. Building trust. https://web.archive.org/web/20180426104935/https://blog.aircloak.com/building-trust-35c74424efc6, July 2017.

[22] William Stallings, Lawrie Brown, Michael D Bauer, and Arup Kumar Bhattacharjee. *Computer Security: Principles and Practice*. Pearson Education, 2012.

[23] Nabil R. Adam and John C. Worthmann. Security-control Methods for Statistical Databases: A Comparative Study. *ACM Comput. Surv.*, 21(4):515–556, December 1989.

[24] Pierangela Samarati and Latanya Sweeney. Protecting privacy when disclosing information: K-anonymity and its enforcement through generalization and suppression. Technical report, technical report, SRI International, 1998.

[25] Leland L. Beck. A security machanism for statistical database. *ACM Trans. Database Syst.*, 5(3):316–3338, Sep 1980.

[26] Center for Machine Learning and Intelligent Systems. Adult dataset. https://archive.ics.uci.edu/ml/datasets/adult.

[27] Center for Machine Learning and Intelligent Systems. Credit dataset. https://archive.ics.uci.edu/ml/datasets/Credit+Approval.

[28] Center for Machine Learning and Intelligent Systems. Census dataset. https://archive.ics.uci.edu/ml/datasets/Census-Income+(KDD).

[29] Frank McSherry. Statistical inference considered harmful. https://github.com/frankmcsherry/blog/blob/master/posts/2016-06-14.md, 2016.

[30] Aircloak. Report on the diffix vulnerability announced by imperial college london and cu louvain. https://aircloak.com/report-on-the-diffix-vulnerability-announced-by-imperial-college-london-and-cu-louvain, April 2018.

[31] Dorothy E Denning. Are statistical data bases secure. In *Proc. AFIPS*, volume 2978, pages 525–530, 1978.

[32] Irit Dinur and Kobbi Nissim. Revealing information while preserving privacy. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, page 202–210. ACM, 2003.

[33] Sriram Sankararaman, Guillaume Obozinski, Michael I Jordan, and Eran Halperin. Genomic privacy and limits of individual detection in a pool. *Nature Genetics*, 41(9):965–967, Sep 2009.

[34] Cynthia Dwork, Adam Smith, Thomas Steinke, and Jonathan Ullman. Exposed! a survey of attacks on private data. *Annual Review of Statistics and Its Application*, 4(1):61–84, Mar 2017.

[35] Paul Francis. Mydata 2017 workshop abstract: Technical issues and approaches in personal data management.

https://aircloak.com/mydata-2017-workshop-abstract-technical-issues-and-approaches-in-personal-data-management, 2017.

[36] Paul Francis. Diffix: Enabling (aggregate) data markets with anonymization. https://aircloak.com/wp-content/uploads/mydata-market-aug17.pdf, 2017.

[37] Frank D McSherry. Privacy integrated queries: An extensible platform for privacy-preserving data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 19–30, New York, NY, USA, 2009. ACM.

[38] Davide Proserpio, Sharon Goldberg, and Frank McSherry. Calibrating data to sensitivity in private data analysis: A platform for differentially-private analysis of weighted datasets. *Proceedings VLDB Endowment*, 7(8):637–648, April 2014.

[39] Indrajit Roy, Srinath T V Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: Security and privacy for MapReduce. In *NSDI*, volume 10, pages 297–312, 2010.

[40] Prashanth Mohan, Abhradeep Thakurta, Elaine Shi, Dawn Song, and David Culler. GUPT: Privacy preserving data analysis made easy. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 349–360, New York, NY, USA, 2012. ACM.

[41] Frank McSherry. Uber's differential privacy .. probably isn't. https://github.com/frankmcsherry/blog/blob/master/posts/2018-02-25.md, 2018.

[42] Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. Smooth sensitivity and sampling in private data analysis. In *Proceedings of the Thirty-ninth Annual ACM Symposium on Theory of Computing*, STOC '07, pages 75–84, New York, NY, USA, 2007. ACM.

[43] Apostolos Pyrgelis, Carmela Troncoso, and Emiliano De Cristofaro. On location, time, and membership: Studying how aggregate location data can harm users' privacy. https://www.benthamsgaze.org/2018/10/02/on-location-time-and-membership-studying-how-aggregate-location-data-can-harm-users-privacy/, October 2018.

[44] Apostolos Pyrgelis, Carmela Troncoso, and Emiliano De Cristofaro. Knock Knock, Who's There? Membership Inference on Aggregate Location Data. In *Proceedings 2018 Network and Distributed System Security Symposium*, San Diego, CA, 2018. Internet Society.

[45] Aloni Cohen and Kobbi Nissim. Linear Program Reconstruction in Practice. *TPDP 2018*, October 2018.

[46] Cynthia Dwork, Frank McSherry, and Kunal Talwar. The Price of Privacy and the Limits of LP Decoding. In *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '07, pages 85–94, New York, NY, USA, 2007. ACM.

[47] Aircloak. Fix for the mit/georgetown univ attack on diffix. https://aircloak.com/fix-for-the-mit-georgetown-univ-attack-on-diffix, October 2018.

[48] G. A Young and Richard L Smith. *Essentials of statistical inference: G.A. Young, R.L. Smith.* Cambridge University Press, 2005.

[49] Munuswamy Sankaran. Approximations to the non-central chi-square distribution. *Biometrika*, 50(1–2):199–204, Jun 1963.

A Likelihood ratio test

Let X and Y be independent random variables. Suppose that we have two hypotheses about the distributions of X and:

$$\begin{aligned} H_0: &\quad X \sim \mathcal{N}(\mu_0, \sigma_0^2) \quad \text{and} \quad Y \sim \mathcal{N}(\mu_1, \sigma_1^2) \\ H_1: &\quad X \sim \mathcal{N}(\mu_1, \sigma_1^2) \quad \text{and} \quad Y \sim \mathcal{N}(\mu_0, \sigma_0^2) \end{aligned}$$

where $\mu_0, \mu_1, \sigma_0, \sigma_1$ are known and fixed values such that $\mu_0 < \mu_1$ and $\sigma_0^2 < \sigma_1^2$. $\mathcal{N}(\mu, \sigma^2)$ denotes the the normal distribution with mean μ and variance σ^2.

Suppose we have a vector of n realizations $\mathbf{x} = (x_1, \ldots, x_n)$ of X and a vector of n realizations $\mathbf{y} = (y_1, \ldots, y_n)$ of Y. We assume that all the $2n$ realizations are mutually independent. The standard frequentist way to accept the preferred hypothesis H_0 or refute it (in favor of H_1) would use a likelihood ratio test with a pre-defined confidence level from which to derive critical regions [48]. In our case we do not have a preferred hypothesis, and hence we define a slightly different test.

Let f and g denote the probability density functions of $\mathcal{N}(\mu_0, \sigma_0^2)$ and $\mathcal{N}(\mu_1, \sigma_1^2)$ respectively. We define the likelihood ratio function Λ as follows:

$$\Lambda(\mathbf{x}, \mathbf{y}) = \prod_{j=1}^{n} \frac{f(x_j)}{g(x_j)} \prod_{j=1}^{n} \frac{g(y_j)}{f(y_j)}.$$

We accept H_0 if $\Lambda(\mathbf{x}, \mathbf{y}) \geq 1$, and we accept H_1 if $\Lambda(\mathbf{x}, \mathbf{y}) < 1$.

Theoretical accuracy of the test. The test will sometimes yield the wrong result. It is possible to determine what is the probability that this happens. Such probability depends on mean and variance of the two specific normal distributions.

Fact. *Let* $p_{err_0} = \Pr[\Lambda(\mathbf{x}, \mathbf{y}) < 1 \mid H_0]$ *and* $p_{err_1} = \Pr[\Lambda(\mathbf{x}, \mathbf{y}) \geq 1 \mid H_1]$. *Then*

$$p_{err_0} = p_{err_1} = \Pr[\alpha_0 Z_0 - \alpha_1 Z_1 < 0]$$

where, for $i = 0, 1$, Z_i is a noncentral chi-squared distribution with n degrees of freedom and noncentrality parameter

$$\lambda_i = n\left(\frac{\mu_i}{\sigma_i} + \frac{\mu_0\sigma_1^2 - \mu_1\sigma_0^2}{\sigma_i(\sigma_0^2 - \sigma_1^2)}\right)^2$$

and

$$\alpha_i = \frac{\sigma_0^2 - \sigma_1^2}{2\sigma_{1-i}^2}.$$

To prove the fact, one considers the log-likelihood ratio function $\log\Lambda(\mathbf{x},\mathbf{y})$ and applies elementary algebra to the obtained expression to derive a linear combination of noncentral chi-squared distributions. We omit the details.

Since $p_{\text{err}_0} = p_{\text{err}_1}$, we refer to this quantity simply as p_{err}. The accuracy of the test is $\text{acc} = 1 - p_{\text{err}}$.

We now show how to apply the fact to our differential noise-exploitation attack. For simplicity, we suppose that Diffix's outputs are not rounded to the nearest nonnegative integer and bucket suppression is never triggered for the queries in the attack, so that every pair of queries $(\widetilde{Q}_j, \widetilde{Q}'_j)$ and $(\widetilde{R}_j, \widetilde{R}'_j)$ yields a valid sample. Thus, for k known attributes, we have two vectors of samples $\mathbf{q} = (q_1, \ldots, q_k)$ and $\mathbf{r} = (r_1, \ldots, r_k)$ and for every $j \leq k$:

$$q_j \sim \begin{cases} \mathcal{N}(0,2) & \text{if } x^{(s)} = 1 \\ \mathcal{N}(1,2k+2) & \text{if } x^{(s)} = 0 \end{cases}$$

$$r_j \sim \begin{cases} \mathcal{N}(1,2k+2) & \text{if } x^{(s)} = 1 \\ \mathcal{N}(0,2) & \text{if } x^{(s)} = 0 \end{cases}$$

We assume that the $2k$ samples in $\mathbf{q}$ and $\mathbf{r}$ are mutually independent. As discussed in section 3, this is not always guaranteed to be true, but it has close to no effect on the actual accuracy of the test. Let

$$\begin{aligned} H_0 &: \quad x^{(s)} = 1 \\ H_1 &: \quad x^{(s)} = 0. \end{aligned}$$

Let f and g denote the probability density functions of $\mathcal{N}(0,2)$ and $\mathcal{N}(1,2k+2)$ respectively. Observe that H_0 holds if and only if every $q_j \sim f$ and every $r_j \sim g$. Similarly, H_1 holds if and only if every $q_j \sim g$ and every $r_j \sim f$. Then we can apply the test defined above. Define

$$\Lambda(\mathbf{q},\mathbf{r}) = \prod_{j=1}^{k} \frac{f(q_j)}{g(q_j)} \prod_{j=1}^{k} \frac{g(r_j)}{f(r_j)}.$$

Our test concludes that $x^{(s)} = 1$ if $\Lambda(\mathbf{q},\mathbf{r}) \geq 1$, and $x^{(s)} = 0$ if $\Lambda(\mathbf{q},\mathbf{r}) < 1$.

To measure the theoretical accuracy of the attack for k known attributes, we can apply the fact to $\Lambda(\mathbf{q},\mathbf{r})$ with $\mu_0 = 0, \sigma_0^2 = 2, \mu_1 = 1, \sigma_1^2 = 2k+2$ and $n = k$, and finally find $\text{acc}(k) = 1 - p_{\text{err}}(k)$.

Fig. 2 shows the values of $\text{acc}(k)$ for increasing values of k. Computing the value of p_{err} requires an approximation of the cumulative distribution function of a linear combination of noncentral chi-squared distributions, for which an exact closed-form expression is not known [49]. We compute these values using the R package `sadists`[3] version 0.2.3.

Numerical simulation with rounding. If we suppose that Diffix's outputs are rounded to the nearest nonnegative integer, no simple expression can be determined for the error rate. To estimate the accuracy in this case, we numerically simulate the values of $\widetilde{Q}_j(D)$ and $\widetilde{Q}'_j(D)$ that would result from querying Diffix (without bucket suppression), for different values of the secret attribute $x^{(s)}$. We then obtain each sample as the difference of the rounded results:

$$q_j = \text{round}(\widetilde{Q}_j(D)) - \text{round}(\widetilde{Q}'_j(D)).$$

Finally, we perform the likelihood ratio test as for the continuous case (considering also null outputs) and check whether the result is correct. We use balanced truth values for $x^{(s)}$, and perform 1000 experiments (on different queries) for each value of $x^{(s)}$. The results are shown in Fig. 2.

B Reducing the number of queries

One of the main features of Diffix is that it allows analysts to send an unlimited amount of queries. Many privacy attacks work by issuing a relatively large number of queries (see also 6.1). Limiting the number of queries allowed by Diffix would thwart or significantly affect these attacks. While our actual attack procedures require a small number of queries ($2|\Delta| + 1$ for the cloning attack), the subset exploration step can sometimes explore many sets of attributes before finding an exploitable one. To minimize the number of queries, we replace the iterative exploration with a greedy heuristic that selects only one subset which is likely to work. We focus only on the cloning attack, as it does not require an oracle and achieves much better accuracy.

The cloning attack requires a set of attributes (A', u), where the restricted record $x^{(A',u)}$ uniquely identifies the victim, but the vector $x^{(A')}$ is shared across a larger population (to avoid bucket suppression). The FullCloningAttack starts with a larger set of attributes A^* and iteratively explores subsets of A^* to find a candidate (A', u). We replace this iterative process with a single deterministic step.

Intuitively, we want u to be as discriminative as possible, while for the attributes in A' to select as many users as possible. This is what the procedure GreedySelectSubset does. First, it computes the (approximate) fraction of users that share the same value $x^{(a)}$, for each attribute $a \in A^*$. Then it selects as u the attribute associated with the lowest fraction. Now suppose that N is the estimated total number of users in the dataset. The set A' is selected as the smallest set of attributes

[3] https://github.com/shabbychef/sadists

associated with the highest fraction, additionally requiring that the product of all the fractions for (A', u) is smaller than $1/N$. This ensures that, with high probability, the victim is uniquely identified by $x^{(A',u)}$.

Procedure GreedySelectSubset($A^*, x^{(A^*)}, s, v$)

Input: known attributes (names A^* and values $x^{(A^*)}$), secret s and target value v
Output: a set of attributes $(A', u) \subseteq A^*$

$N \leftarrow \mathsf{count}()$ // approx. tot. number of users
foreach $a \in A^*$ **do**
  $C_a \leftarrow \mathsf{count}(a = x^{(a)} \wedge s \neq v)$
  $\rho_a \leftarrow \frac{C_a}{N}$
end foreach
$\{\rho_1, \ldots, \rho_{|A^*|}\} \leftarrow \text{SortDescendingOrder}(\{\rho_a\}_{a \in A^*})$
$u \leftarrow a_{|A^*|}$
$i \leftarrow 1, \quad A' \leftarrow \emptyset$
while $\rho_u \prod_{a_i \in A'} \rho_{a_i} > \frac{1}{N}$ **do**
  $A' \leftarrow A' \cup \{a_i\}$
  $i \leftarrow i + 1$
end while
return (A', u)

We can modify the FullDifferentialAttack replacing the subset exploration with this heuristic. The modified full attack is described in the GreedyFullCloningAttack procedure.

Procedure GreedyFullCloningAttack($A^*, x^{(A^*)}, \Delta, s, v$)

Input: known attributes (names A^* and values $x^{(A^*)}$), dummy conditions Δ, secret s and target value v
Output: True if $x^{(s)} = v$, False if $x^{(s)} \neq v$

$(A', u) \leftarrow$ GreedySelectSubset($A^*, x^{(A^*)}, s, v$)
if NoBucketSuppression($A', u, x^{(A)}, \Delta, s, v$) and ValueUnique($A', u, x^{(A^*)}$) **then**
  return CloningAttack($A', u, x^{(A)}, \Delta, s, v$)
end if
return NonAttackable

Observe that the GreedySelectSubset procedure issues exactly $|A^*| + 1$ queries. The differential attack with the assumption validation step issues at most $2|\Delta| + 1$ queries. So, the GreedyFullCloningAttack algorithm requires at most $|A^*| + 2|\Delta| + 2$ queries.

Subset selection	Median n. queries	Max n. queries	Predicted attackable	accuracy$_{pa}$
Iterative	304	5310	96.8%	93.3%
Heuristic	32	32	55.4%	91.7%

Table 2: Empirical results of the cloning attack with iterative subset exploration and with the heuristic subset selection.

We compared the performances of the FullCloningAttack and GreedyFullCloningAttack on the ADULT dataset, with the salary class as secret attribute and the other 10 attributes as A^*. As in section 4, we used $|\Delta| = 10$ dummy conditions and ran the attack on 1000 random users. The results are summarized in Table 2.

The maximum (and median) number of queries used by GreedyFullCloningAttack for a single user is $10 + 2 \times 10 + 2 = 32$. The median number of queries used by GreedyFullCloningAttack is about 10 times higher, and the maximum is 100 times higher.

GreedyFullCloningAttack effectively attacks more than half of the users, as opposed to 96.8% of the users for the FullCloningAttack. This is due to the fact that the first attack tries a single subset of attributes per user. However, this figure is still remarkably high, given the huge reduction of required queries. Finally, the accuracy of the inference for the attacked users is almost the same.

We believe that these results give additional evidence of the power, extendability and practicability of our noise-exploitation attacks. Introducing additional optimizations, the accuracy could be improved and the number of queries could be further reduced (see full version).

FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation

Yaowen Zheng[1,2,3]*, *Ali Davanian*[2], *Heng Yin*[2], *Chengyu Song*[2], *Hongsong Zhu*[1,3], and *Limin Sun*[1,3]†
[1] Beijing Key Laboratory of IoT Information Security Technology,
Institute of Information Engineering, CAS, China
[2] University of California, Riverside, USA
[3] School of Cyber Security, University of Chinese Academy of Sciences, China
{zhengyaowen,zhuhongsong,sunlimin}@iie.ac.cn, adava003@ucr.edu, {heng,csong}@cs.ucr.edu

Abstract

Cyber attacks against IoT devices are a severe threat. These attacks exploit software vulnerabilities in IoT firmware. Fuzzing is an effective software testing technique for vulnerability discovery. In this work, we present FIRM-AFL, the first high-throughput greybox fuzzer for IoT firmware. FIRM-AFL addresses two fundamental problems in IoT fuzzing. First, it addresses compatibility issues by enabling fuzzing for POSIX-compatible firmware that can be emulated in a system emulator. Second, it addresses the performance bottleneck caused by system-mode emulation with a novel technique called augmented process emulation. By combining system-mode emulation and user-mode emulation in a novel way, augmented process emulation provides high compatibility as system-mode emulation and high throughput as user-mode emulation. Our evaluation results show that (1) FIRM-AFL is fully functional and capable of finding real-world vulnerabilities in IoT programs; (2) the throughput of FIRM-AFL is on average 8.2 times higher than system-mode emulation based fuzzing; and (3) FIRM-AFL is able to find 1-day vulnerabilities much faster than system-mode emulation based fuzzing, and is able to find 0-day vulnerabilities.

1 Introduction

The security impact of IoT devices on our life is tremendous. By 2020, the number of connected IoT devices will exceed the number of people [10]. This creates an unprecedented attack surface leaving almost everybody at danger. Even currently, the hackers leverage the lack of security in IoT devices to create large botnets (e.g., Mirai, VPNFilter and Prowli). These malware attacks exploit the vulnerabilities in IoT firmware to penetrate into the IoT devices. As a result, it is crucial for defenders to discover vulnerabilities in IoT firmware and fix them before attackers.

Fuzzing, a software testing technique that feeds a program with random inputs, has approved to be very effective in finding vulnerabilities in real-world programs. In particular, AFL [34], a coverage-guided greybox fuzzing tool, has been used widely in both industry and academia. For instance, most of the finalists in DARPA Cyber Grand Challenge used AFL as the primary vulnerability discovery component [2].

Challenges in IoT firmware fuzzing. Despite the effectiveness of fuzzing for programs on general-purpose platforms, it is generally not feasible to directly apply fuzzing on IoT firmware, due to its strong dependency on the actual hardware configuration. For instance, simply extracting a user-level program from a Linux-based firmware and fuzzing this program with AFL would not work in most cases.

To this end, recent researches propose a series of solutions, ranging from directly fuzzing the IoT devices (e.g., IoTFuzzer [14]), a hybrid solution that combines hardware and software emulation (e.g., AVATAR [33]), to a full system emulation (e.g., Firmadyne [13]). As a recent study by Muench et al. [28] points out, full system emulation yields the highest throughput, because IoT devices are much slower than a desktop workstation or a server.

Throughput is a key factor for the effectiveness of fuzzing. However, even for full system emulation, its performance is far from being ideal. According to our evaluation (§5), full system emulation is about 10 times slower than user-mode emulation (which is used by AFL). 10 times slowdown means approximately 10 times more computing resources are needed to find a vulnerability in an IoT program than its desktop counterpart. According to our analysis (§2.4), part of the enormous runtime overhead of full-system emulation comes from software implementation of memory management unit (i.e., SoftMMU) that is used to translate a guest virtual address into a host virtual address for every single memory access happening in the virtual machine. The other part of the overhead comes from the system calls emulation overhead.

*The work was done while visiting University of California, Riverside
†Corresponding author

Our solution: greybox fuzzing via augmented process emulation. In this work, we present, to the best of our knowledge, the first greybox fuzzer for IoT firmware, that achieves two design goals simultaneously: (1) *transparency* that is no modification should be needed for the program in firmware to be fuzzed, and (2) *efficiency* that is the fuzzing throughput of the overall system should come close to that of the user-mode emulation. Our key insight is to find a novel combination of full-system emulation and user-mode emulation to achieve the best of two worlds: generality from full-system emulation and efficiency from user-mode emulation.

More specifically, we propose a new technique called "augmented process emulation". As the name suggests, its main idea is to augment process (or user-mode) emulation with full system emulation. The program to be fuzzed is mainly run in user-mode emulation to achieve high efficiency, and switches to full system emulation only when necessary to ensure correct program execution, thus achieving generality.

To evaluate the feasibility of this technique, we implement a prototype system called FIRM-AFL, on top of AFL [34] and Firmadyne [13]. From a user's perspective, using FIRM-AFL, we can conduct coverage-guided greybox fuzzing on a user-specified program from an IoT firmware, the same as fuzzing a normal user-level program using AFL. Under the hood, FIRM-AFL occasionally switches to the full system emulation mode in Firmadyne, to ensure the given program can be correctly emulated.

We have evaluated FIRM-AFL with standard benchmarks and a set of real-world IoT firmware images. The evaluation results showed that (1) FIRM-AFL can faithfully emulate the target programs as if they were running in full-system emulation; (2) compared to a full-system emulation based fuzzer (TriforceAFL [29] with lightweight snapshot enabled), the throughput of FIRM-AFL is 8.2 times higher on average and (3) FIRM-AFL can find 1-day vulnerabilities 3 to 13 times faster than full-system emulation based fuzzer, and was able to find two 0-day vulnerabilities within 8 hours on a single machine.

Contributions. In summary, we make the following contributions in this paper:

- We point out that full system emulation exerts significant runtime overhead, and is far from ideal to serve as the base for IoT firmware fuzzing. We further investigate the root cause of this runtime overhead.
- We propose a novel technique called "augmented process emulation", to reconcile the contradictory characteristics of full-system emulation (high generality and low efficiency) and user-mode emulation (low generality and high efficiency).
- We design and implement the first coverage-guided greybox fuzzing platform for IoT firmware, FIRM-AFL.
- We extensively evaluate our system and show the overhead for each part of our system. Our improvements lead to 8.2 times speedup on average. As a result, FIRM-AFL could find 1-day vulnerabilities 3 to 13 times faster than full-system emulation, and was able to find two new vulnerabilities within 8 hours on a single machine.
- The current implementation of FIRM-AFL supports three CPU architectures, including mipsel, mipseb and armel, which cover 90.2% firmware images in the Firmadyne datasheet [4]. The source code of FIRM-AFL can be found at `https://github.com/zyw-200/FirmAFL`.

2 Background and Motivation

2.1 Fuzzing

Fuzzing is a software testing technique that aims to find bugs by executing the target program with random inputs and looking for interesting program behaviors such as the crashes. Based on how much information is collected and used from the execution, fuzzers can be categorized into blackbox, whitebox and greybox. A blackbox fuzzer treats the target program as a blackbox and does not utilize any feedback from the execution to guide the generation of random inputs. This approach was originally used to test Linux utilities [26]. On the other hand, a whitebox fuzzer selects the inputs based on a deep insight into the target program. This is usually achieved through expensive program analysis techniques like dynamic taint analysis and symbolic execution [22]. Finally, a greybox fuzzer improves the testing by utilizing limited information collected with lightweight monitoring techniques (e.g., code coverage).

The most popular greybox fuzzers are coverage-guided fuzzers. These fuzzers instrument the target program to collect code coverage information. The collected information is then used to guide the input generation—inputs that explore new execution paths will be used as *seeds* to generate new inputs while inputs that did not yield new coverage will be discarded. This simple strategy is extremely effective in practice. In fact, greybox fuzzers can even outperform whitebox fuzzers when targeting real-world applications. Their secret is *speed*, the lightweight instrumentation allows greybox fuzzers to execute hundreds or thousands times more inputs than whitebox fuzzers [32]. In other words, throughput is paramount for greybox fuzzers.

AFL [34] is a well-known greybox fuzzer. It can instrument the program either statically or dynamically. Static instrumentation is preferred when the source code is available. When the source code is not available, e.g., when fuzzing commercial off-the-shelf (COTS) programs, AFL utilizes a binary translator (i.e., user-mode emulation provided by QEMU [12]) to perform the instrumentation. For most IoT devices, because source code and design documents are often proprietary and

only firmware image might be available, dynamic instrumentation is the only viable option. As a matter of fact, even extracting the binary from the firmware is not always straightforward [14].

2.2 QEMU

QEMU [12] is a fast processor emulator based on dynamic binary translation. Unlike traditional emulators that interpret the target program instruction-by-instruction, QEMU translates several basic blocks at a time. More importantly, it caches translated blocks and uses block chaining to link them together. This allows the execution to remain inside the code cache (i.e., the logic of the target program) for the most of the time thus minimizes the overhead of the translation. Dynamic instrumentation can be performed during the translation to introduce new functionalities, such as branch monitoring [34] and taint propagation [19, 23].

Besides the translation of instructions, the next most important task is address space translation. The translation is done very differently based on the execution mode. In system mode, QEMU implements a software Memory Management Unit (MMU) to handle memory accesses. The software MMU maps Guest Virtual Addresses (GVAs) to the Host Virtual Addresses (HVA). This mapping process is transparent to the guest operating system (OS) meaning that QEMU still allows the guest OS to set up the GVA to Guest Physical Address (GPA) mapping through the interface of page tables and to handle page faults. Under the hood, QEMU inserts a GVA to GPA translation logic for every memory access. To speed up the translation, QEMU uses a software Translation Lookaside Buffer (TLB) to cache the translation results. Moreover, to avoid invalidating the code cache and block chaining whenever the address translation changes, all translated blocks are indexed using GPA and the block chaining is only performed when the two basic blocks are within the same physical page. GPA to HVA mapping is done using a linear mapping (i.e., HVA = GPA + OFFSET).

In contrast to system-mode emulation, in user-mode emulation, the Host Virtual Address (HVA) is calculated as the Guest Virtual Address (GVA) plus a constant offset. So this translation is much faster than the one in system-mode emulation.

2.3 Testing IoT Firmware

As IoT devices become a popular attack target, testing IoT programs to find vulnerabilities also becomes important. There are two main challenges in testing IoT programs. The first challenge is compatibility: many IoT programs depend on special hardware components of the device thus cannot be tested without proper support. The second challenge is code coverage: blackbox fuzzers are known to have low code coverage while whitebox fuzzers cannot scale to slightly larger code base [20]. Table 1 compares some representative efforts on IoT firmware testing using these two metrics.

Avatar [33] aims to enable dynamic program analysis for embedded firmware by providing better hardware component support. It achieves this goal through constructing a hybrid execution environment consists of both a processor emulator (QEMU) and real hardware where Avatar acts as a software proxy between the emulator and the real hardware. This allows Avatar to utilize the emulator to execute and analysis the instructions while channeling the I/O operations to the physical hardware. As a demonstration, the authors have applied S2E [15], a whitebox fuzzing tool to find vulnerabilities in the Redwire Econotag Zigbee sensor. Due to the involvement of whitebox fuzzing and slow hardware, the throughput of Avatar is expected to be very low.

IoTFuzzer [14] performs blackbox fuzzing directly on the real device. Its main advantage over previous blackbox fuzzing based approaches is that it performs the fuzzing through the companion mobile app of the target device. By automatically analyzing the data flow in the companion app to better understand the communication protocol, IoTFuzzer can generate better test cases that are more likely to trigger a bug. That said, based on its evaluation, IoTFuzzer never exceeds a throughput of 1 test case per second, which is slow (based on Table III in [14]).

Although it does not perform fuzzing, Firmadyne [13] adds hardware support for IoT firmware to the system mode QEMU. It provides support for both ARM and MIPS architectures that are popular among the IoT manufacturers. For hardware support, Firmadyne fully emulates the system by modifying the kernel and drivers to handle the IoT exceptions due to the lack of actual hardware. Compared to the former two solutions, this solution is easier to adapt to new IoT firmware and programs. The throughput of full-system emulation is usually better than the native execution [28].

Muench et al. [28] compare the throughput of a blackbox fuzzer [24] under different configurations, including native execution (directly sending inputs to the hardware), partial emulation (redirecting only hardware requests to the hardware), and full emulation. Their emulation is based on image replaying capability provided by PANDA [19]. They concluded that full emulation (FE) has the highest throughput mainly because the IoT processors are much slower than desktop processors. However, even in the best case, the throughput did not exceed 15 test cases per second [1].

AFL [34] is a well-known greybox fuzzer that can support binary-only fuzzing through user-mode QEMU. Unfortunately, lacking special hardware support, user-mode QEMU can not successfully emulate most IoT programs. For example, AFL with user-mode QEMU failed on all the programs used in our evaluation (Table 3). Moreover, simply adopting a full system emulator (e.g., Firmadyne) does not fully solve

[1] They reported 53390 cases/hour which is equal to 15 cases/second

	Avatar [33]	IoTFuzzer [14]	Firmadyne [13]	Muench et al. [28]	AFL [34]
Technique	Whitebox fuzzing	Blackbox fuzzing	PoC	Blackbox fuzzing	Greybox fuzzing
Compatibility	High	High	High	High	Low
Hardware Support	Hybrid	Real	Emulation	Mixed	None
Code Coverage	Medium	Low	N/A	Low	High
Throughput	Very Low	Low	Medium	Low to Medium	High
Zero-day Detection	Yes	Yes	No	Yes	Yes

Table 1: Comparison of IoT firmware testing tools.

the problem because the throughput is low.

In summary, existing IoT firmware testing tools do not provide satisfying code coverage yet sate-of-the-art fuzzers (e.g., AFL) cannot be easily applied to test IoT programs. So far, there is no greybox IoT fuzzer, not to mention a greybox IoT fuzzer with good throughput.

2.4 Motivations

Given the unsatisfying status-quo of IoT firmware testing tools, we aim to enable high-throughput greybox fuzzing for IoT programs. To this end, we decide to build the fuzzer based on emulation. This choice is based on two reasons. First, greybox fuzzing requires collecting execution information (e.g., branch coverage) to guide test case generation. As mentioned in §2.1, this is usually done through lightweight instrumentation. Since most IoT programs are only distributed in binary format, emulator-based instrumentation is the best available option. The second reason is performance. Although it might be possible to run instrumented binaries directly on the device, Muench et al. [28] have shown that full-emulation-based approach is actually faster than the real device, because the desktop processors are much faster.

Unfortunately, simply adopting a full system emulator (e.g., Firmadyne [13]) does not fully solve the problem because the throughput is not enough. For example, even with the full-emulation configuration, the fuzzer used in [28] never exceeded 15 test cases per second. To understand the bottleneck, we profiled the execution time of two networking tools (`basename` and `uptime`) under full-system emulation (with lightweight snapshot) and user-mode emulation. The results are shown in Table 2. Based on this measurement, we can see that the throughput of fuzzing can be significantly boosted if we can apply user-mode emulation to the target program. There are several bottlenecks that contribute to the execution time difference.

- *B1. Memory address translation*. In full-system emulation, QEMU uses a software MMU to perform address translation for *every* memory access. In contrast, in user-mode emulation, the address translation is much simpler. So even if we just consider time spent in user-mode execution, user-mode emulation uses much less time.
- *B2. Dynamic code translation*. The code translation process in user-mode emulation is faster than the full-system mode. In full-system mode, block chaining is limited to basic blocks in the same physical page, which means the translator is invoked more often than in user-mode emulation.
- *B3. Syscall emulation*. In user-mode emulation, system calls are handled directly by the host OS and hardware. Therefore, it is significantly faster than full-system emulation where the OS also runs in the emulator and the hardware devices are also emulated. Although hardware emulation is necessary to allow the target program to run correctly, not all system calls would rely on the special hardware. In other words, not all system calls require emulation.

In this work, we address all three bottlenecks to improve the throughput of IoT program fuzzing.

3 Augmented Process Emulation

3.1 Overview

The goal of this work is to enable high-throughput greybox fuzzing for IoT programs. As discussed in §2, to achieve this goal, we need to overcome two challenges: compatibility and performance. The first challenge can be solved through full-system emulation but this would result in poor performance. The second challenge can be solved through user-mode emulation but would result in poor compatibility. In this section, we present *augmented process emulation*, a new approach that brings the best of both full-system emulation and user-mode emulation.

Problem statement. Generally speaking, the goal of augmented process emulation is to correctly execute a program of an IoT firmware in a user-mode emulator, given the following requirements are satisfied:

(1) The firmware can be correctly emulated in a system emulator (e.g., system-mode QEMU). Fortunately, with the help of Firmadyne [13], a large portion of IoT firmware images are able to meet this requirement.

	system mode (ms)					user mode (ms)			
program	overall	sys exec	sys code trans	user exec	user code trans	overall	sys exec	user exec	user code trans
`basename`	4.08	1.79	0.53	1.41	0.35	0.34	0.02	0.11	0.22
`uptime`	7.48	2.39	0.76	2.79	1.55	0.89	0.04	0.31	0.54

Table 2: Runtime performance of system mode and user mode emulation

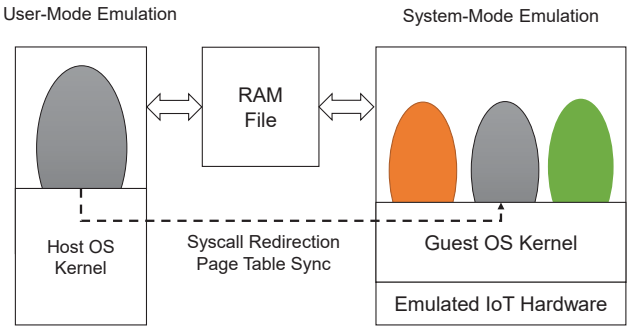


Figure 1: Overview of Augmented Process Emulation

(2) The firmware runs a POSIX-compatible operating system (OS). Fortunately, many IoT firmware images use Linux as the OS hence satisfy this requirement.

With augmented process emulation, we aim to achieve the following design goals:

- *Transparency.* The user-level program running in the augmented process emulation should behave as if it were run in the system-mode emulation.
- *High efficiency.* Since throughput is a dominating factor for fuzzing, the augmented process emulation needs to be as efficient as possible. Ideally, it should approximate the performance of pure user-mode emulation.

Solution overview. To achieve the design goals mentioned above, we resort to combine user-mode emulation with system-mode emulation in a novel manner. Figure 1 illustrates the overview of our solution.

At first, the IoT firmware boots up in the system-mode emulator and the user-level programs (including the one to be fuzzed) are launched properly inside the emulator. After the program to be fuzzed has reached at a predetermined point (e.g., the entry point of main function, or after receiving the first network packet), the process execution is then migrated to the user-mode emulation in order to gain high execution speed. Only at rare occasions, the execution is migrated back to the system-mode execution to ensure the correctness of execution.

To minimize the migration cost, the memory state is shared between these two emulation modes. More concretely, the physical memory of the virtual machine for the system-mode emulation is allocated as a memory-mapped file, called RAM file. This RAM file is also mapped into the address space of the user-mode emulation. Note that system-mode emulation and user-mode emulation access this RAM file in different ways. System-mode emulation treats the RAM file as physical memory, and thus accesses it by physical address, while user-mode emulation accesses the shared memory by virtual address. Therefore, the physical pages in the RAM file need to be mapped into the address space of user-mode emulation by their virtual addresses at a page granularity. As a result, when a page mapping is not established in the user-mode emulation, the process execution needs to be migrated to the system-mode emulation to establish this mapping. We will discuss more details about the memory mapping in §3.2.

With a proper memory mapping, the process should be able to execute correctly in the user-mode emulation, until it reaches a system call. Directly executing the system call locally on the host OS would not work in general, because the host OS and the OS in IoT firmware are different and the underneath hardware layers are also different. To ensure transparency, we need to migrate the execution to the system-mode emulation to process this system call. When the system call returns, we migrate the execution back to the user-mode emulation. More details will be discussed in §3.3.

3.2 Memory Mapping

Bootstrapping. When fuzzing a program with AFL, the program executes to a predetermined point, and then the fork server of AFL will repeatedly fork a new program instance on this point (which is referred to as fork point) and feed random inputs. Similarly, in this setting, we will boot up the IoT firmware in system-mode emulation and further launch the specified IoT program. Using Virtual Machine Introspection (VMI) provided by DECAF [23] (a system emulation based dynamic analysis platform), we are able to monitor the execution of the specified IoT program and get notified when the execution reaches to the predetermined fork point.

At this moment, we will walk the page table of the specified process and collect the virtual to physical page mapping information and send it over to the user-mode emulation side. Then for each mapping of virtual address (`va`) to physical address (`pa`), the user-mode emulation side establishes a mapping by calling `mmap` as below:

```
mmap(va, 4096, prot, MAP_FILE, ram_fd, pa);
```

The code above is self-explanatory. Essentially, we map a page of the RAM file with the physical address as offset into a specified virtual address. The argument `prot` is determined by the protection bits from the corresponding page table entry.

From this point onward, the execution in system-mode emulation is paused, the CPU state is sent over to user-mode emulation, and the execution resumes there.

Page fault handing. During the process execution in user-mode emulation, if the accessed memory addresses have already been mapped in this address space, the execution should proceed successfully. Otherwise, the host processor will raise a page fault. We register a signal handler for page fault in user-mode emulation, so the host OS will pass along the page fault event to the user-mode emulation. On receiving this signal, the user-mode emulation records the CPU state at the faulting instruction, pauses the execution, and passes the CPU state to the system-mode emulation side, expecting that the page fault can be handled in the system-mode emulation and a new mapping for the faulting virtual address can be established.

When the system-mode emulation receives the CPU state and resumes execution, the emulated processor will raise a page fault, since the page is not present. The page fault handler in the OS of the IoT firmware will respond to this page fault and attempt to establish the mapping. Most likely, this mapping will be established by the OS sooner or later (depending on the scheduling of numerous kernel threads and interrupt handlers) and the instruction that causes the page fault will be re-executed. In very rare cases, if the OS cannot establish a mapping for various reasons, it will kill the process.

A key question here is to determine when the page mapping has been established or an error occurs, so we can switch back to the user-mode emulation to maximize execution speed. The answer to this question is in fact non-trivial, because the OS is handling multiple tasks simultaneously and enormous amount of context switches may happen in the meantime.

To capture the right moment when a mapping is established, we instrument the end of each basic block. If the execution is currently within the specified process (or thread), it means the execution has returned from the kernel to the user space to resume the faulting instruction. The mapping must be present in the software TLB. So we can just directly find the mapping there. At this moment, we pass the mapping information and the CPU state back to the user-mode emulation, which will create this new mapping by calling `mmap` and resume the execution.

If for some reasons, an error occurs and the process gets killed, we can rely on the VMI (Virtual Machine Introspection) capability provided by DECAF [23] to get notified, and then the whole execution on both sides get terminated.

Preload page mapping. Modern operating systems load memory pages in a lazy manner. Although when a new process starts, all code pages are assigned into its address space, a mapping from each virtual page to its physical page is not really established until a page fault caused by the first memory access to it.

This lazy design has adverse effect on fuzzing performance. As we will discuss in §4.1, a child process is repeatedly forked from the parent process for each fuzzing iteration, and thus there are always a series of page faults caused by un-mapped code pages. This is especially harmful for our system, because the overhead of page fault handling is much more expensive than handling it locally on the host OS.

To solve this problem, we decide to preload the code pages of the given process in the physical memory and perform the mapping between the two modes. This helps us avoid repeatedly loading the code pages at every fuzzing iteration, and hence speed up the fuzzing throughput. To do that, we simulate the access to each program code page in the system-mode emulation during the bootstrap, to force the OS to map each page into the process' address space. As a result, we can reduce the number of page faults caused by these pre-loaded pages.

3.3 System Call Redirection

System calls and their implementation in IoT programs are different because of the underlying IoT hardware, firmware and requirements. Consequently, user-mode emulation of an IoT program will likely fail if the exceptions caused by the system calls are not properly handled (see §2). For example, most IoT devices have network interfaces that are not available on a local emulator. When an IoT program in the user-mode emulation executes a system call that needs to interact with a specific network interface in the IoT system, there will be a fault that needs to be handled. Another example is a system call that accesses NVRAM that is undefined for a desktop computer.

Therefore, to ensure execution correctness, we must redirect the system calls from the user-mode emulation to the system-mode emulation. More specifically, when the user-mode emulation encounters a system call, it pauses the execution, saves the current CPU state, and sends it over to the system-mode emulation. The system-mode emulation receives the CPU state and resumes execution. This will cause a mode switch into the kernel mode in the guest system to process the corresponding system call. Again, since the guest OS kernel is multi-tasking, there might be many context switches happening before the system call returns. So similar to how we handle page faults, we will instrument the end of each basic block. If the current basic block is in the kernel space, but next program counter is in the user level, and the current execution context is for the thread that makes the system call, we detect the moment when the system call returns. Then at this moment, we pause the execution in the system-mode emulation, save the CPU state, and pass it back to the user-mode

emulation, which will then resume the execution.

Optimizing filesystem-related system calls. While examining the system calls made by a set of IoT programs, we realize that many system calls are related to the file system. The IoT programs either attempt to access files or directories that already exist in the firmware or are newly created for only temporary uses. We propose an optimization for this set of system calls. We map the file system from the firmware image, and mount it as a directory in the host OS, such that the user-mode emulation can directly access it. In this way, the user-mode emulation can directly pass through the file-system related system calls to the host OS, instead of redirecting them to the system-mode emulation.

As shown in §5.3, filesystem-related system calls take a significant portion among all system calls, and thus this optimization makes a significant contribution for the final performance.

4 Firm-AFL Design and Implementation

Leveraging the technique described in §3, we design and implement FIRM-AFL, an enhancement of AFL [34] for fuzzing IoT firmware. In §4.1, we first describe the workflow of AFL, and then in §4.2, we present how we integrate augmented process emulation into the workflow of AFL.

4.1 Workflow of AFL

AFL is a coverage-guided greybox fuzzer. It maintains a seed queue that stores all the seeds, including the initial seeds chosen by the user as well as the ones that are mutated from the existing seeds and cause the program to reach unique code coverage.

The main program that drives the fuzzing process is `afl-fuzz`. It picks a seed from the seed queue, performs a random mutation, generates an input, and feeds this input to the target program (assuming it is a binary executable).

In order to collect the code coverage information from the execution of the target program, AFL starts the program using the user-mode QEMU, and instruments the branch transitions of the target program, and the code coverage information is encoded and stored in a bitmap.

Since during fuzzing we need to execute the target program repeatedly, AFL utilizes "fork" as a mechanism to speed up this process. It first runs the target program up to a certain point (e.g., the entry point of the main function) such that the program's code and data have been properly initialized, and then repeatedly forks a child process from it. In this way, the initial setup of a new process is skipped. For this reason, the parent process is called `fork-server`. Then the input is fed into the forked child process, and the coverage information is collected and stored in the bitmap, which is shared among

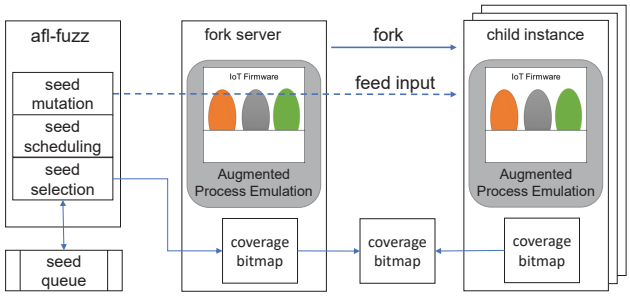


Figure 2: Overview of FIRM-AFL

all three processes (`afl-fuzz`, `fork-server`, and the child instance). `afl-fuzz` will compare the bitmap from the child instance and the accumulative bitmap from all past executions to determine if this mutated input should be kept as a new seed and stored in the seed queue.

4.2 AFL with Augmented Process Emulation

We would like to keep the workflow of AFL intact, but allow AFL to fuzz a target program in an IoT firmware image. To do so, we replace the user-mode QEMU with augmented process emulation, and the rest of the components remain unchanged. The new workflow is illustrated in Figure 2.

Bootstrapping. To fuzz a program in the IoT firmware image, we need to boot up the firmware image and launch the program after the system boots up. This is done in the system-mode emulation within `fork-server`.

We leverage Firmadyne [13] to correctly emulate a firmware image. We further integrate DECAF [23] with Firmadyne to make use of its VMI (Virtual Machine Introspection) capability. In this way, we are able to capture the precise moment when the target program is started or terminated. We can also know when the execution of the target program has reached the pre-determined fork point.

Forking. The default fork point chosen by AFL is the entry point of the main function. In our case, we are interested in finding vulnerabilities in the IoT programs that are triggered through the network interface. Therefore, we hook the network-related system calls. And the first invocation of any of these system calls becomes the fork point.

In the standard workflow of AFL, we can simply leverage the `fork` system call to fork a child process and start the next fuzzing instance. In our case, we not only need to fork a child process for the user-mode emulation, but also "fork" a new virtual machine instance for the system-mode emulation, because two modes must synchronize with each other.

Actually forking a new virtual machine would be too expensive. Instead, we can make a snapshot of the virtual machine

at the fork point, and when one fuzzing execution is finished, we can restore the snapshot. System-mode QEMU offers `save_snapshot` function that saves all the CPU registers and the memory space to a specific file. However, file write/read operations would still be very slow.

In our system, we implement a lightweight snapshot mechanism based on the Copy-on-Write principle. More concretely, we first mark the RAM file mapped into the system-mode QEMU as read-only. Then a memory write will cause a page fault. We make a copy of the page, and then mark this page as write-able. As such, we record all memory pages that have been modified during one fuzzing execution. When restoring the snapshot, we only need to write these recorded pages back.

Feeding input. The inputs are fed through instrumenting system calls. For the IoT programs that are receiving input from network interface, we instrument the network-related system calls in the user-mode emulation directly, so we don't need to redirect these system calls to the system-mode emulation.

Collecting coverage information. Since most of execution happens in the user-mode emulation and system-mode emulation is only needed for handling page faults and some system calls, we can simply instrument the branch transitions in user-mode QEMU to compute the coverage bitmap, just like how the original AFL does it in user-mode QEMU.

5 Evaluation

In this section, we evaluate the prototype implementation of our fuzzer FIRM-AFL. The purpose of this section is to test whether our approach has resolved the performance bottlenecks and achieved the two design goals. In short, we would like to answer following questions:

- *Transparency.* Can FIRM-AFL fuzz programs extracted from IoT firmware as if they are running inside a full-system emulator?
- *High efficiency.* How close is FIRM-AFL's throughput (executions/sec) to the throughput of a pure user-mode emulation based fuzzer?
- *Effectiveness of optimization.* Do our optimization techniques successfully resolved the performance bottlenecks we identified?
- *Effectiveness in vulnerability discovery.* How effective is FIRM-AFL in finding real vulnerabilities in IoT firmware?

Experiments setup. We used three sets of programs in our evaluation. The first set of programs are two standard benchmarks: `nbench` [9] and `lmbench` [7]. They are used to access the correctness of the emulation and the overhead of the emulation. The second set of programs consist of seven IoT programs from four different vendors (Table 3). We selected these program since they are the key service programs that handle network requests thus are good targets for remote attacks. They are used to access the performance of greybox fuzzing. The third dataset is the Firmadyne dataset which includes firmwares whose HTTP and uPnP services are related to 15 1-day exploits (Table 6). We collected them to evaluate the transparency and effectiveness of FIRM-AFL in vulnerability discovery.

Experiments (except the ones in §5.4) are conducted on a 8-core Intel(R) Core(TM) i7-3940XM 3.00GHz CPU machine with 23.5GB of RAM 1TB hard disk . The operating system is Ubuntu 16.04.5 LTS. The version of QEMU and AFL is 2.10.1 and 2.06b. We obtain each measurement value after every ten iterations. Our final reported numbers are the average value of 20 measurements. By default, we set fork point at the position after the network data received, and feed the random input provided by AFL engine.

5.1 Transparency

To evaluate the transparency of our augmented process emulation, we first evaluated our emulator with the `nbench` test suite. After generating the output, the benchmark will compare the outputs with expected outputs. If the generated outputs are wrong, then it implies the emulation is not correct. The results showed that our system can finish all the benchmarks without errors.

We also empirically evaluated the transparency of FIRM-AFL using the Firmadyne dataset [4]. We collected 120 firmware images with HTTP services and unique device models. We first tried to run HTTP service programs in them directly using user-mode QEMU. We extracted the file systems from the firmware images and used `chroot` to mount the file systems. However, all these programs crashed at the very beginning due to the lack of expected system environment. Then we tried to run them with normal inputs (the initial seeds) under full-system emulation, as well as under augmented process emulation. We observed that in both settings, all the programs could run properly. For each program, we further compared the system call sequences generated under full-system emulation as well as augmented process emulation, and confirmed that the system call sequences were identical.

Finally, we evaluated a set of exploits targeting known vulnerabilities listed in Table 6. For each vulnerability, we fed a proof-of-concept (PoC) exploit in both full-system emulation and augmented process emulation and compared the execution traces. We confirmed that the collected two traces are

Program	Size (KB)	Description	Vendor	Devices	Model	Version	CPU Arch
cgibin	129.4	CGI binary program	DLINK	Router	DIR-815	1.01	MIPSEL
httpd	90.2	Embedded HTTP server					
dnsmasq	162.3	Embedded DNS server					
dropbear	307.3	Embedded SSH server	TPLINK	Router	TL-WR940N	V4_160617	MIPSEB
httpd	1692	Embedded HTTP server					
jjhttpd	103.3	Embedded HTTP server	Trendnet	Router	TEW-813DRU	v1(1.00B23)	MIPSEB
lighttpd	327.3	Embedded HTTP server	Netgear	Router	WNAP320	3.0.5.0	MIPSEB

Table 3: IoT programs used for evaluation

identical.

In summary, this evaluation showed that FIRM-AFL can provide transparent emulation as if the program is executing in full-system emulation.

5.2 Efficiency

Benchmark	Augmented mode	User mode	Slowdown
Numeric sort	679.12	686.56	1.08%
String sort	78.36	79.54	1.48%
Bitfield	3.47E+08	3.45E+08	0.00%
FP emulation	163.85	161.72	0.00%
Fourier	1383.6	1,384.00	0.00%
Assignment	20.45	20.75	1.40%
IDEA	4,864.10	4,854.10	0.00%
Huffman	1,749.00	1,743.10	0.00%
Neural Net	1.93	1.95	0.60%
LU Decomp	61.26	61.92	1.00%

Table 4: `nbench` results, the unit is iterations/second. The last column shows the slowdown of augmented mode.

Syscall	Augmented mode	User mode	Overhead
null	0.48	0.48	0.00%
read	0.62	0.60	3.33%
write	0.57	0.52	9.62%
stat	1.31	1.24	5.64%
fstat	0.63	0.61	3.28%
open	2.61	2.50	4.40%
select file	3.52	3.48	1.15%
select tcp	32.74	12.64	159%
pipe(latency)	6.73	6.57	2.44%

Table 5: `lmbench` syscall testing results, the unit is microsecond. The last column shows the overhead of augmented mode.

Standard benchmarks. We evaluated the efficiency of our approach from two angles. First, we evaluated the performance overhead of augmented process emulation using standard performance benchmarks. The result of `nbench` is shown in Table 4. `nbench` is a CPU-bound benchmark suite. On this benchmark, the augmented mode did not impose much overhead, largely due to the fact that these benchmarks are relatively simple, so they do not require many memory synchronization operations and syscall redirection. To evaluate the overhead of syscall redirection, we used the `lmbench`. The result is shown in Table 5. As we can see, for syscalls that are executed locally (e.g., file related syscalls), the overhead is almost negligible. For syscalls that still require redirection (e.g., TCP related), the overhead is much higher.

Fuzzing throughput. In the second performance evaluation, we measured the throughput of FIRM-AFL, under different optimization levels:

(a) *Baseline*: we used TriforceAFL [29] as the baseline. TriforceAFL uses full-system emulation to support fuzzing IoT programs. To avoid rebooting the virtual machine, in this configuration, we added support for QEMU's stock snapshot mechanism (`qemu_savevm` and `qemu_loadvm`) to TriforceAFL. We also use VMI provided by DECAF [23] to capture the precise moment when program is started and terminated.

(b) *Lightweight snapshot*: in this configuration, we changed the snapshot mechanism to our lightweight snapshot (§4).

(c) *Augmented process emulation*: in this configuration, we switched the emulation mode from full-system mode to our augmented process emulation mode (§3).

(d) *Full*: in this configuration, we applied all optimization techniques, including selective syscall redirection.

Figure 3 shows the throughput improvement. Overall, lightweight snapshot boosted the throughput for about 9.3 times (b vs. a). Augmented process emulation boosted the throughput for about 3 times on average (c vs. b). With selective syscall redirection, the throughput had another boost for about 2.9 times on average (d vs. c). So compared with the best result on full-system emulation based fuzzing (b), FIRM-AFL (d) provided an average improvement of 8.2 times.

5.3 Effectiveness of Optimization

In §2.4, we identified three major bottlenecks of full-system emulation: memory address translation, dynamic code trans-

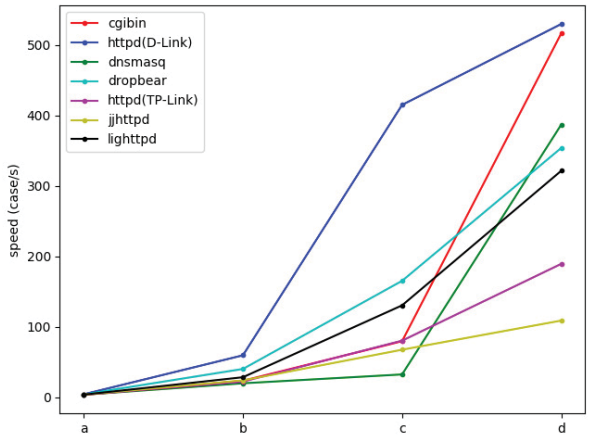


Figure 3: Fuzzing throughput of FIRM-AFL under different optimization level. The x-axis is the optimization level: (a) baseline, (b) w/ lightweight snapshot, (c) w/ augmented process emulation, and (d) w/ selective syscall redirection. Fuzzing throughput for each program is shown in a different color.

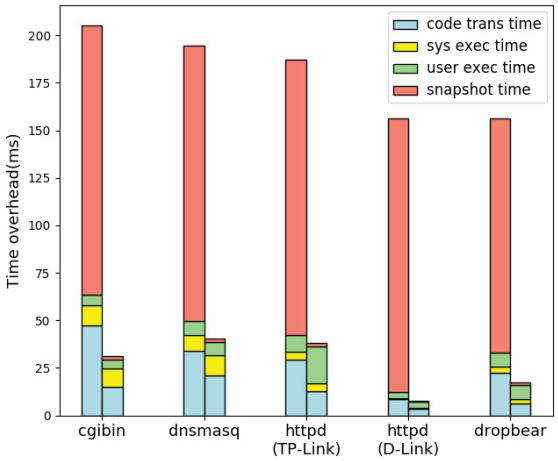


Figure 4: Execution time breakdown: system-mode emulation w/o and w/ lightweight snapshot.

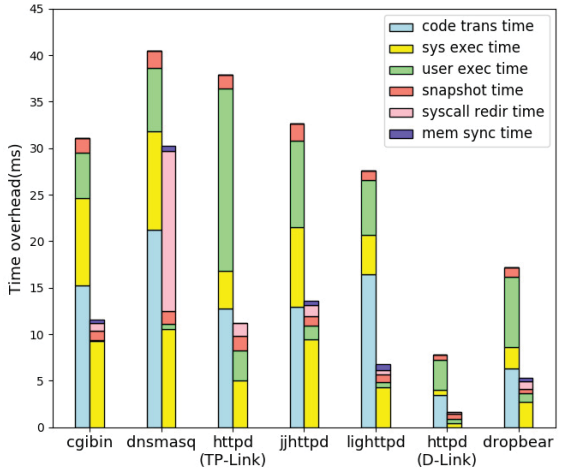


Figure 5: Execution time breakdown: augmented process emulation vs. full-system emulation.

lation, and syscall. In this section, we evaluated whether our optimization techniques successfully addressed these bottlenecks. For this purpose, we break down the total execution time into five parts:

- *User execution time*: the total time spent in executing the logic of the target program, this includes the time spent on software address translation.
- *Memory synchronization time*: in augmented emulation mode, time spent on setup the memory mapping between the user-mode emulator and the full-system emulator.
- *Code translation time*: total time spent on translating the target program.
- *Syscall execution time*: total time spent on system calls in an iteration of execution.
- *Syscall redirection time*: in augmented emulation mode, time spent on redirecting the system call to the full-system emulator.
- *Snapshot time*: the total time spent on storing and restoring memory and CPU states in an iteration of fuzzing. Note that different snapshot mechanisms have different time overhead values. We record the starting and ending time for each page store and restore operations.

Lightweight snapshot. Snapshot overhead only exists for the system-mode emulator. In augmented process emulation, a synchronization mechanism is required to ensure the consistency of snapshot between system and user mode. For these experiments, we measure the snapshot synchronization cost and add it to the snapshot overhead. When comparing the snapshot overhead in Figure 4 and Figure 5, we can see that the lightweight snapshot mechanism leads to more than 100x reduction in the snapshot overhead.

Augmented process emulation. Figure 5 shows the execution time breakdown of full-system emulation and augmented process emulation for the seven IoT programs. The total execution time on average reduces more than 50% except for `dnsmasq`. When analyzing breakdown of execution time, we can see huge reduction on user execution time and code translation time. On average, the user execution time (green bar) was reduced by about 9 times. This is mostly due to the elimination of software address translation. Even if we combine

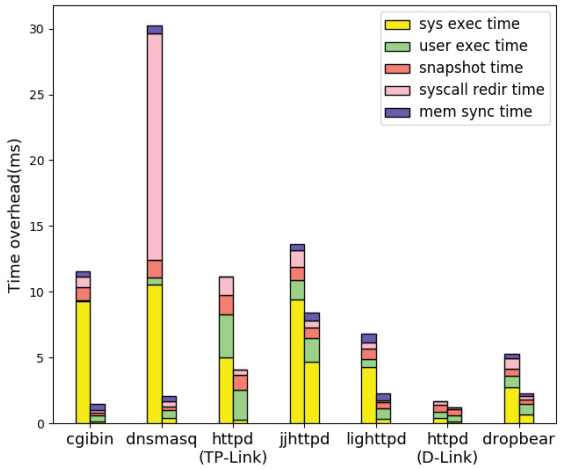


Figure 6: Execution time breakdown: augmented process emulation w/o and w/ selective syscall redirection.

the memory synchronization time (purple bar), the execution time was still reduced by about 5 times.

Another huge reduction is the code translation time. As briefly mentioned in §2, this is due to two optimization techniques. First, when running in full-system mode, QEMU only performs block chaining for basic blocks within the same physical page. This means the emulator has to be invoked to resolve control transfer between pages. In augmented process emulation, QEMU can link any basic blocks as long as they are translated. Second, when using full-system mode for fuzzing, the fuzzer (Triforce) will reset the virtual machine after processing each input. Although we have optimized this step with lightweight snapshot, the code cache will be reset during the restore. This means the same basic block could be translated repeatedly for every fuzzing iteration. In augmented process emulation, we can utilize the code cache pooling technique from AFL to avoid this re-translation. As a result, the amortized code translation time became very small.

Unfortunately, the reduction on user execution time and code translation time is at the cost of increase in overall syscall time, i.e., the combination of syscall execution time and syscall redirection time. In general, the more syscalls the target program issues, the higher the redirection overhead. This is why `dnsmasq` spent significantly more time on syscall redirection than the other programs: it issued more than one thousand system calls which caused more than two thousand state transitions between system mode and user mode. This highlights the necessity of selective syscall redirection.

Selective syscall redirection. Figure 6 shows the execution time breakdown with and without selective syscall redirection. Recall that the goal of redirecting system calls to the full-system emulator is to ensure correct emulation. However, not all system calls require special kernel or hardware support. Therefore, by locally executing system calls that can be fully supported by the host system (e.g., file system related syscalls), we reduce most of the syscall time without jeopardizing correctness. As shown in the figure, after applying this optimization, we observed a huge reduction in system call execution time, because many system calls are now executed by the host OS without address/code translation and device emulation. At the same time, we also observed reduction in syscall redirection time, which has a great impact on programs that issue many syscalls, like `dnsmasq`. A majority of syscalls issued by `dnsmasq` were file operations which can be handled locally by mounting the IoT firmware file system in the host OS. By doing so, the total execution time of `dnsmasq` can be reduced by another 14 times.

To summarize, this evaluation showed that our solutions (augmented process emulation and selective syscall redirection) have successfully addressed the three bottlenecks we identified in §2.4.

5.4 Vulnerability Discovery

In this section, we aim to evaluate how effective FIRM-AFL is in finding vulnerabilities in real-world IoT firmware images.

Data collection. We started with the Firmadyne dataset [4]. We collected these firmware images and tested the emulation condition and network reachability, and then checked the liveness of HTTP and uPnP services by probing their ports. Eventually, we obtained 288 firmware images with active HTTP and uPnP services. We then used getsploit [1] to collect exploits targeting HTTP and UPnP services from online resources, such as exploit-db [3], metasploit [8], and Packet Storm [6]. Then we fed these exploits into the 288 images, and eventually identified 15 exploits that can be launched successfully against 51 firmware images. Table 6 lists these 15 exploits.

We further ran the programs related to these 15 exploits in user-mode QEMU, and observed that only one program `tcapi` that is related to the last five exploits can continue to work in user-mode QEMU. This result once again confirms the necessity of augmented process emulation.

Experiment setup. As our focus in this case study is on fuzzing HTTP and uPnP services, which have well-structured protocol formats. To expedite fuzzing, we made use of the dictionary option "-x" in AFL. We collected keywords for HTTP (from honggfuzz [5]), uPnP and HTTP CGI services (extracted directly from binary programs) respectively. For each service, we then provided a normal service request as the initial seed .

Moreover, to avoid underestimating the performance of full-system emulation with its default snapshot implementation, we enabled lightweight snapshot in it.

Exploit ID	Vendor	Model	Version	Device	Program	Full-System Time to crash	FIRM-AFL Time to crash
CVE-2018-19242	Trendnet	TEW-632BRP	1.010B32	Router	httpd	21h43min	6h2min
CVE-2013-0230	Trendnet	TEW-632BRP	1.010B32	Router	miniupnpd	>24h	9h16min
CVE-2018-19241	Trendnet	TV-IP110WN	V.1.2.2	Camera	video.cgi	19h13min	4h55min
CVE-2018-19240	Trendnet	TV-IP110WN	V.1.2.2	Camera	network.cgi	12h0min	2h21min
CVE-2017-3193	DLink	DIR-850L	1.03	Router	hnap	21h3min	2h54min
CVE-2017-13772	TPLink	WR940N	V4	Router	httpd	>24h	>24h
EDB-ID-24926	DLink	DIR-815	1.01	Router	hedwig.cgi	16h38min	1h22min
EDB-ID-38720	DLink	DIR-817LW	1.00B05	Router	hnap	4h26min	1h29min
EDB-ID-38718	DLink	DIR-825	2.02	Router	httpd	>24h	22h3min
CVE-2016-1558	DLink	DAP-2695	1.11.RC044	Router	httpd	16h24min	2h32min
CVE-2018-10749	DLink	DSL-3782	1.01	Router	tcapi	247s	20s
CVE-2018-10748	DLink	DSL-3782	1.01	Router	tcapi	252s	22s
CVE-2018-10747	DLink	DSL-3782	1.01	Router	tcapi	249s	20s
CVE-2018-10745	DLink	DSL-3782	1.01	Router	tcapi	236s	25s
CVE-2018-8941	DLink	DSL-3782	1.01	Router	tcapi	281s	24s

Table 6: 1-day exploits

The experiments were conducted on a server with 40-core Intel Xeon(R) E5-2687W(v3) 3.10GHz CPU and 125GB of RAM.

Finally, to ensure our evaluation results on fuzzing performance are statistically significant, as suggested by Klees et al. [25], we ran each fuzzing experiment ten instances in parallel for 24 hours. In addition to FIRM-AFL, we also evaluated full system emulation with lightweight snapshot support. We report cumulative number of unique crashes found over time, using `plot_data` in AFL output files.

Evaluation results. We calculate the median time to first crash in full-system emulation and augmented process emulation respectively and record them in the last two columns of Table 6. We can see that FIRM-AFL can find a crash at least 3.6 times faster than full-system emulation, and in many cases more than 10 times faster.

We also plot cumulative number of unique crashes found over time by FIRM-AFL (blue), and fuzzing with full emulation (red) in Figure 7. In each plot, the solid line represents the median result from 10 rounds while the dashed lines represent the lower and upper bounds of 95% confidence intervals for a median. Since last five cases in Table 6 are related to the same program and the results are similar, we just plot the case for CVE-2018-10749 as the representative.

From the result, we can see that in spite of large variations across fuzzing runs, FIRM-AFL was able to find significantly more unique crashes and find them multiple times faster than full emulation. We further investigated these crashes and confirmed that most of these crashes were caused by the same known vulnerabilities. We indeed found two new vulnerabilities, which we will describe next.

0-day vulnerabilities. We discovered two 0-day vulnerabilities using FIRM-AFL, after 7.5 hours and 6 hours respectively. We also tried fuzzing these two programs with full-system emulation using the same initial seeds, and no crash was found within 24 hours. we reported them to IoT manufacturers and MITRE corporation. The details about these two vulnerabilities are described as below.

- CVE-2019-11417: Buffer overflow in Trendnet TV-IP110WN (firmware version: v.1.2.2 build 68). Attackers can exploit the device by using 'languse' parameter in `system.cgi`.
- CVE-2019-11418: Buffer overflow in Trendnet TEW-632BRP (firmware version: v.1.010B32). Attackers can exploit the device by crafting the `soapaction` HNAP interface.

6 Discussion

In this section, we discuss the limitations in our system and shed some light for future work.

Limitation on supported CPU architectures. The current implementation of FIRM-AFL supports the following CPU architectures: mipsel, mipseb and armel, which already account for 90.2% images in the Firmadyne dataset. We expect that supporting more CPU architectures is relatively easy, because the majority of the emulation logic in QEMU is implemented in an architecture-independent manner.

Limitation on supported IoT firmware. Even after more CPU architectures are supported, FIRM-AFL can only fuzz a

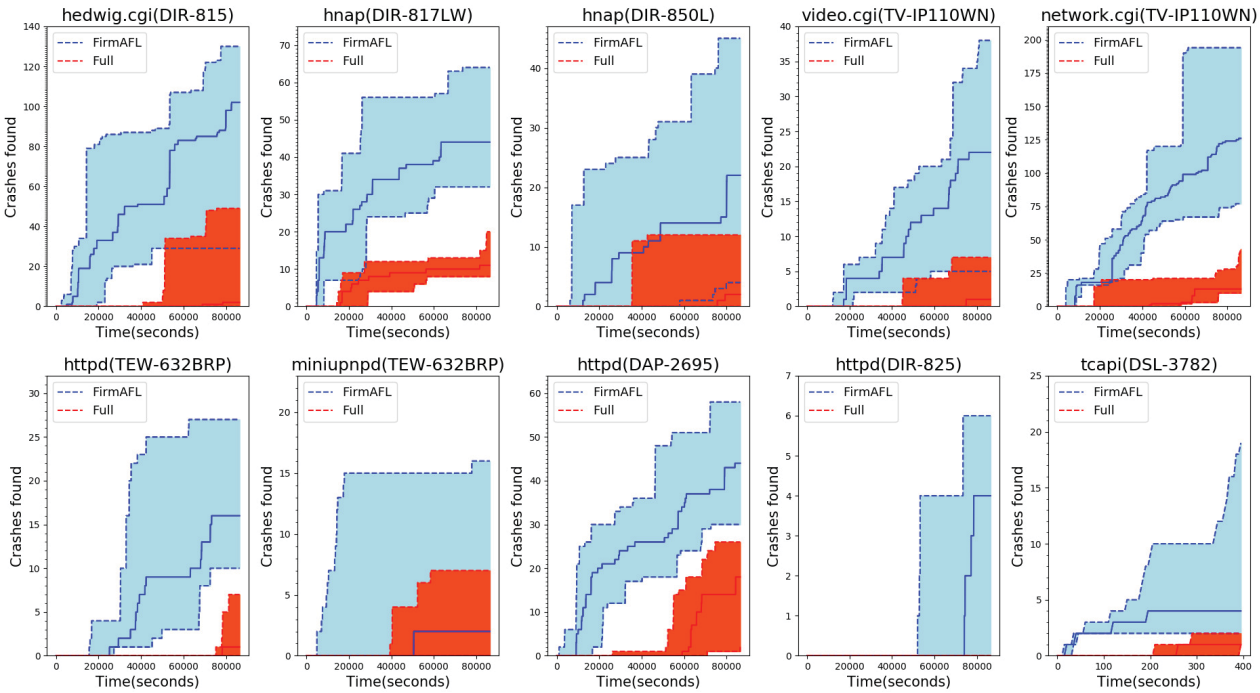


Figure 7: Crashes found over time

program in a firmware image that can be properly emulated by Firmadyne and runs a POSIX-compatible OS (e.g., Linux). This limitation stems deeply from the design of FIRM-AFL, and thus there is no simple solution. An improvement on IoT firmware emulation is orthogonal to this paper. We will leave it for future work. Supporting a non-POSIX program would require a virtualization layer, such that they can run properly within a POSIX process. We are not aware of an existing solution for this. Thus, it can be an interesting future work.

7 Related Work

With the increasing number of IoT devices and their security issues, several techniques are proposed to find the IoT devices vulnerabilities in an automatic manner. These techniques can be categorized into static or dynamic analysis. Lacking the source code of the IoT firmware, static analysis often relies on the binary image and reverse engineering techniques.

Static analysis. Costin et al. presented a large scale analysis of IoT firmware by coarse-grained comparison of files and modules [17]. Their approach is able to find a lot of known bugs within the common third-party projects used by different vendors. Cojocar et al. proposed another approach to heuristically identify parsers and complex processing logics from IoT firmware, and they find several vulnerabilities [16]. That said, these approaches suffer from high false positives and cannot find completely new vulnerabilities. Feng et al. presented a cross-platform bug search technique for firmware images [21]. The technique is based on high-level numeric features comparison, and only takes 0.1 second on average to finish all 154 vulnerabilities searching. Xu et al. further proposed a novel neural network-based approach to detect cross-platform binary code similarity [31]. It can significantly reduce training time and feature vector generation time, as well as improve search accuracy.

Firmalice is another IoT binary analysis framework that employs static analysis techniques [30]. Firmalice utilizes symbolic execution on the firmware binary and uses backward slicing to make the vulnerability analysis tractable. Firmalice focuses only on one slice of the program based on an analyst's specification. The specification provides a clue about the privileged program code. Isolating the potential vulnerable code, Firmalice makes the analysis scalable while also capable of finding new vulnerabilities. That said, Firmalice can only find the authentication vulnerabilities and relies on manual analysis for the slice specification.

Dynamic analysis. On the other hand, dynamic analysis techniques for IoT firmware require either the real devices or an emulation of some sort. Black-box fuzzing is a common approach to discover vulnerabilities by directly interacting with devices. Recently, several works have developed dynamic emulators for IoT devices. For example, Zaddach et al. developed a dynamic analysis framework for IoT devices by redirecting hardware requests from the emulator to

the actual hardware [33]. Based on it, Marius et al. developed a dynamic multi-target orchestration framework that can enable interoperability between different dynamic binary analysis framework, debuggers, emulators and real physical devices [27]. However, the large number of hardware limits its scalability, and also imposes a large overhead.

Chen et al. proposed a robust software-based full system emulation. Their emulation is based on kernel instrumentation [13]. Their goal is to perform automatic vulnerability verification that has no ability to find unknown vulnerabilities. Both Avatar and Firmadyne do not use techniques such as fuzzing that are capable of finding completely new vulnerabilities in real applications. Anderi et al. conducted dynamic analysis to achieve automated vulnerability discovery within embedded firmware images [18]. The tool aims at discovering web-interface related vulnerabilities by using web pentesting tools. However, it cannot find vulnerabilities of other modules in IoT firmware.

IoT fuzzing. For IoT fuzzing, and closest to our work, Muench et al. developed six live analysis heuristics including call stack tracing and call frame tacking [28]. Muench et al. built their system on top of Avatar [33] and PANDA [19], and their system can effectively detect memory corruption for IoT devices. However, this system takes target systems as black-box and feeds input from outside which imposes overhead on the devices startup and rebooting for each fuzzing session. Further, unlike greybox fuzzing, the input space exploration is very blind, and hence the chance of finding a bug is very low. In our work, we utilize greybox fuzzing, and aim to minimize each fuzzing iteration overhead so that the fuzzer can test more test cases in the same unit of time. In addition, Alimi et al. proposed to use fuzzing techniques and specific simulators (JCOP) to discover vulnerabilities in programs hosted into smart cards [11]. The methodology does not scale due to emulation problems of various kinds of IoT firmware.

8 Conclusion

Coverage-based greybox fuzzing has proven to be an effective way to find vulnerabilities in real-world programs. Yet, applying greybox fuzzing to IoT firmware has not been realized due to two main challenges. Firstly, state-of-the-art greybox fuzzers like AFL fail to run many IoT programs due to specific hardware dependencies. Secondly, solutions that can tackle the first challenge (e.g., by employing full-system emulation) yield very low throughput. We proposed a novel technique, augmented process emulation to address both challenges at the same time. With augmented process emulation, we achieve high throughput fuzzing by running the target program in a user-mode emulator and switch to a full-system emulator when the target program invokes a system call that has specific hardware dependencies.

We evaluated the transparency and the efficiency of FIRM-AFL, our prototype implementation of greybox IoT fuzzing based on the augmented process emulation. The results showed that our system is transparent and its throughput outperforms all the state-of-the-art IoT firmware fuzzers by one order of magnitude. Our case study further showed that FIRM-AFL could indeed find both 1-day vulnerabilities much faster than full-system emulation and was able to find two new vulnerabilities within only two hours on a single machine.

Acknowledgement

We thank our shepherd Dr. Yongdae Kim and the anonymous reviewers for their insightful comments on our work. This work is partly supported by Key Program of National Natural Science Foundation of China under Grant No. U1766215, National key R&D Program of China under Grant No. 2016YFB0800202, Strategic Priority Research Program of Chinese Academy of Sciences under Grant No. XDC02020500, International Cooperation Program of Institute of Information Engineering, CAS under Grant No. Y7Z0451104, National Science Foundation under Grant No. 1664315, Office of Naval Research under Award No. N00014-17-1-2893, Guangdong Province Key Area R&D Program of China under Grant No. 2019B010137004. We also thank the support provided by China Scholarship Council (CSC) for Yaowen Zheng's visiting to UCR. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] Command line utility for searching and downloading exploits. https://github.com/vulnersCom/getsploit.

[2] The cyber grand challenge. http://blogs.grammatech.com/the-cyber-grand-challenge.

[3] Exploit database - exploits for penetration testers, researchers, and ethical hackers. https://www.exploit-db.com/.

[4] Firmadyne datasheet. https://cmu.app.boxcn.net/s/hnpvf1n72uccnhyfe307rc2nb9rfxmjp/folder/6601681737.

[5] honggfuzz. a security oriented, feedback-driven, evolutionary, easy-to-use fuzzer with interesting analysis options. http://honggfuzz.com/.

[6] Information security services, news, files, tools, exploits, advisories and whitepapers. https://packetstormsecurity.com.

[7] LMbench - tools for performance analysis. http://www.bitmover.com/lmbench/.

[8] Metasploit | penetration testing software, pen testing security. https://www.metasploit.com.

[9] nbench. https://www.math.utah.edu/~mayer/linux/bmark.html.

[10] Gartner says 8.4 billion connected "things" will be in use in 2017, up 31 percent from 2016, gartner. http://www.gartner.com/en/newsroom/press-releases/2017-02-07-gartner-says-8-billion-connected-things-will-be-in-use-in-2017-up-31-percent-from-2016, February 2017.

[11] V. Alimi, S. Vernois, and C. Rosenberger. Analysis of embedded applications by evolutionary fuzzing. In *2014 International Conference on High Performance Computing Simulation (HPCS)*, pages 551–557, July 2014.

[12] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[13] Daming D. Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for Linux-based embedded firmware. In *Network and Distributed System Security Symposium, NDSS*, February 2016.

[14] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. IoTFuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *Networked and Distributed System Security Symposium (NDSS'18)*, February 2018.

[15] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[16] Lucian Cojocar, Jonas Zaddach, Roel Verdult, Herbert Bos, Aurélien Francillon, and Davide Balzarotti. PIE: Parser identification in embedded systems. In *Annual Computer Security Applications Conference (ACSAC'15)*, December 2015.

[17] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In *USENIX Security Symposium*, August 2014.

[18] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. Automated dynamic firmware analysis at scale: A case study on embedded web interfaces. In *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, May 2016.

[19] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, PPREW-5, 2015.

[20] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *IEEE Symposium on Security and Privacy*, May 2016.

[21] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 480–491, 2016.

[22] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium (NDSS'08)*, February 2008.

[23] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform. In *International Symposium on Software Testing and Analysis (ISSTA'14)*, July 2014.

[24] Pereyda J. boofuzz. https://github.com/jtpereyda/boofuzz, 2016.

[25] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*, October 2018.

[26] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, December 1990.

[27] Marius Muench, Aurélien Francillon, and Davide Balzarotti. Avatar^2: A multi-target orchestration platform. In *Workshop on Binary Analysis Research (BAR'18)*, February 2018.

[28] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *Network and Distributed System Security Symposium (NDSS'18)*, February 2018.

[29] NCC-Group. TriforceAFL. https://github.com/nccgroup/TriforceAFL, 2017.

[30] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In *Network and Distributed System Security Symposium (NDSS'15)*, February 2015.

[31] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS'17)*, October 2017.

[32] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *USENIX Security Symposium*, August 2018.

[33] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares. In *Network and Distributed System Security Symposium (NDSS'14)*, February 2014.

[34] M. Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl/.

Not Everything is Dark and Gloomy: Power Grid Protections Against IoT Demand Attacks

Bing Huang
The University of Texas at Austin
binghuang@utexas.edu

Alvaro A. Cardenas
University of California, Santa Cruz
alvaro.cardenas@ucsc.edu

Ross Baldick
The University of Texas at Austin
baldick@ece.utexas.edu

Abstract

Devices with high energy consumption such as air conditioners, water heaters, and electric vehicles are increasingly becoming Internet-connected. This new connectivity exposes the control of new electric loads to attackers in what is known as Manipulation of demand via IoT (MadIoT) attacks. In this paper we investigate the impact of MadIoT attacks on power transmission grids. Our analysis leverages a novel cascading outage analysis tool that focuses on how the protection equipment in the power grid as well as how protection algorithms react to cascading events that can lead to a power blackout. In particular, we apply our tool to a large North American regional transmission interconnection system consisting of more than 5,000 buses, and study how MadIoT attacks can affect this power system. To help assess the effects of such cyber attacks, we develop numerical experiments and define new and stronger types of IoT demand attacks to study cascading failures on transmission lines and their effects on the system frequency. Our results show that MadIoT attacks can cause a partition of the bulk power system, and can also result in controlled load shedding, but the protections embedded in the operation of the transmission grid can allow the system to withstand a large variety of MadIoT attacks and can avoid a system blackout.

1 Introduction

The vulnerability of Internet of Things (IoT) devices is a well-known problem [11, 25, 46]. Previous work has demonstrated that devices from cameras to door locks can be compromised directly or through their designated smart phone applications [29, 43]. A large-scale compromise of these devices can enable attackers to affect network infrastructures, as exemplified by the Distributed Denial of Service (DDoS) attacks by the Mirai botnet—which consisted of more than six hundred thousand IoT devices [13].

The collective effect of compromised IoT devices can go beyond traditional computer network infrastructures. Recent work proposed a novel form of attack called Manipulation of demand via IoT (MadIoT) [47], and showed that if an attacker compromised hundreds of thousands of high-energy IoT devices (such as water heaters and air conditioners), the attacker could cause various problems to the power grid, including (i) frequency instabilities, (ii) line failures, and (iii) increased operating costs. These attacks paint a dire picture of the security of the power grid as they show that a 30% increase in demand can trip all the generators in the US Western interconnection causing a complete system blackout, and a 1% increase of demand in the Polish grid results in a cascade of 263 transmission line failures, affecting 86% of the load in the system.

In this paper we re-evaluate the potential impact of MadIoT attacks by modeling in detail the protection equipment and the operational responses to sudden load changes in the power grid. Our analysis leverages a novel cascading outage analysis tool that focuses on how the protection equipment already embedded the power grid reacts during cascading events, where multiple protection equipment is activated one after the other.

Our analysis shows that while MadIoT attacks can create negative consequences on the power grid, the negative impact on the grid will not be as dire as originally thought. In particular, while the most powerful MadIoT attacks (assuming the attacker compromises more than 8 million air conditioners) might cause the power system to partition and operate as separate islands, or can also cause some controlled load shedding, our results show that creating a system blackout—which would require a black start period of several days to restart the grid— or even a blackout of a large percentage of the bulk power grid will be very difficult.

This paper is organized as follows. Section 2 introduces the background necessary to understand power systems and how our tool compares to state-of-the-art practices for cascading analysis. Section 3 presents the details of our simulations and models. Section 4 illustrates why our cascade analysis tool has advantages over competing alternatives in a simplified model used in previous work. Our main results focusing on the analysis of a large-scale North American interconnec-

tion undergoing MadIoT attacks are presented in Section 5. Section 7 summarizes related work and Section 8 provides conclusions, limitations, and future work.

2 Power Systems Background

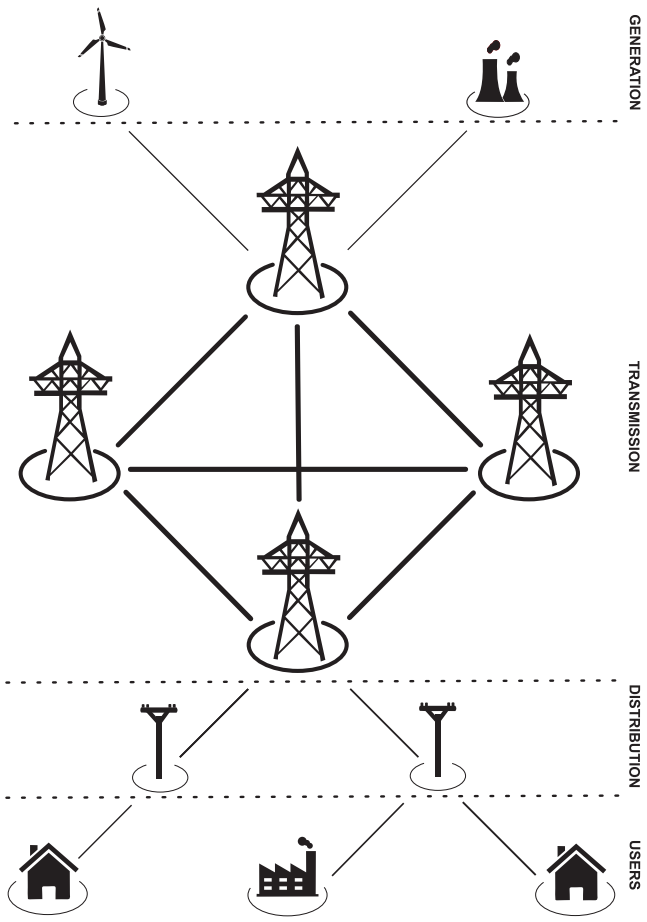


Figure 1: Generation and Transmission form the Bulk of the Power Grid. Transmission systems are redundant and have to satisfy the N-1 operation criterion, while Distribution systems are radial systems (non redundant) and affect a very small percentage of the system.

The objective of engineers and researchers in the power system industry is to deliver increasing amounts of electrical energy in a safe, clean, and economical manner [31]. The power grid has three major parts: (1) generation, (2) transmission, and (3) distribution. Electric power is generated wherever it is convenient and economical, and then it is transmitted at high voltages (100kV-500kV) in order to minimize energy losses—electrical power is equal to voltage times electrical current ($P = VI$), and given a constant power, high voltage lines have less electrical current, and therefore there is less energy lost as heat as the current moves through the transmission lines. Geographically, a distribution system is located in a smaller region thereby energy losses are less of a concern while safety (preventing accidents, fires, electrocutions, etc.) is more important, therefore they are operated at lower voltages. Figure 1 illustrates these three main parts of the grid. A distribution system is connected to a transmission system in a substation and the conductor that completes the connections is usually represented in electrical diagrams by nodes called **buses**.

Operators have to keep the nominal frequency (e.g., 60Hz in the Americas) and the transmission lines at their operating range (at a fixed voltage like 500kV, and with currents below a safety threshold) in order to ensure reliable operation of the grid. If there is a sudden increase in the demand of electricity, the frequency of the power grid tends to slow down, and automatic controls ramp up generation of electricity to take the frequency back to 60Hz. If there is a sudden decrease in the demand of electrical power, then the frequency of the grid tends to increase, and automatic controls then decrease generation of electrical power to reduce the frequency to the nominal level. Similarly sudden changes in electricity consumption might overload transmission lines and activate protection equipment (relays that prevent the flow of electricity through the line), and if this happens, the power is then distributed to other transmission lines.

2.1 Transmission vs. Distribution Outages

Large generation plants and the transmission network are usually referred to as the **Bulk Power System**, and this bulk power system is responsible for the reliable delivery of electricity to large areas. The bulk power system is an interconnected, redundant network that spans large regions—usually one country, but in North America there are three bulk systems: the Eastern Interconnection, the Western Interconnection, and Texas. In contrast, distribution systems are geographically smaller and their networks are mostly radial (i.e., non-redundant).

The **bulk** power system is designed and operated to satisfy the N-1 security criterion, which means that the system can lose any one of its N components (such as generators or transmission lines) and continue operating safely and serving the power supply to the customers in the large area. This operating criterion is mandatory and enforced by government entities, and therefore bulk power system operators have the incentives to make sure that their systems satisfy the N-1 criterion at any point in time, otherwise they get massive sanctions. In contrast, since distribution systems are usually non-redundant and serve customers in a regional area, they do not have to meet the same operating criterion.

The reason distribution systems do not have to meet the N-1 criterion is the scale of a system failure. A disruption in the **bulk** power grid will be the topic of national news headlines because it causes a blackout in a large part of the country

(sometimes even the whole country), while a disruption in the distribution system will usually only cause a localized outage (e.g., a neighborhood will be without electricity). Electric power in the distribution grid can also be more easily restored, while a system blackout of the bulk power system will require days of coordination in what is called **black start period.**

While distribution systems are not required to follow the N-1 criterion, there are separate criteria applied to them. For example, the hours of successful power supply to consumers as percentage of the total hours in a year is required to meet certain standard e.g. 99.999%. Other details of the distribution system will not be discussed as they go beyond the scope of this paper.

As we will show later in the paper, one of the protections embedded in the power system to prevent a bulk power outage is called **Under Frequency Load Shedding (UFLS)**, which is a mechanism where predetermined blocks of customers in the distribution system are automatically dropped from the system. This is a carefully selected procedure where electricity is not cut to safety-critical loads like Hospitals. We will show that some of the most severe MadIoT attacks will activate this protection and therefore can cause some controlled outages, but at the same time, these small outages are done in order to prevent that the bulk system goes into a cascading failure resulting in a system blackout.

2.2 Failure Analysis in the Bulk Power Grid

The power grid analysis tool we use in this paper was developed to address the limitations for modeling and analyzing cascading failures identified by the task force from the IEEE Power Engineering Society [14, 15]. As stated in these reports, most of the research in cascading failure analysis focuses on independent phenomenons, but these interactions are often ignored. In our recent work on cascading failures [33, 53–56] we have been developing a tool that captures the time interdependencies of all relevant protection equipment and stability studies in the power grid when multiple simultaneous (or quasi-simultaneous) contingencies occur. In this paper we adapt our tool to model MadIoT attacks. Before we discuss our approach in more detail, we now present related work in the analysis of failures in the power grid and discuss how our system compares to these approaches.

Cascading failure analysis has attracted a lot of attention from the research community [14, 44, 52]. There are two main approaches for studying cascading failures: stochastic models, and fine-grained simulations.

Stochastic models are used to evaluate the likelihood of a cascading event by giving us the probability of having incorrect settings for protection equipment in a given power system [26, 45]. To build these estimates, stochastic models perform a forensic analysis of previous cascading failures by looking at the properties of power systems just before they experienced a system blackout. Although these models provide a probabilistic insight of cascading events, they cannot be used to model the operation of a power system undergoing a cascade, which is particularly important when we want to understand how the system reacts to incidents in general (and cyber-attacks in particular). To understand the operation of the power system undergoing cascading failures we need to turn to detailed simulation models.

2.2.1 Power System Simulations

There are two main behaviors that we need to study when a system undergoes a failure:

1. **Transient Analysis** finds the behavior of the frequency in the power grid in the immediate aftermath of the incident. If the frequency deviates too far from 60Hz, some protection equipment will be activated. There are two options for transient analysis.

   (a) **No System Dynamics:** This is a very fast computational method where the behavior of all generators is simplified to only one generation machine. This allows us to evaluate how the frequency of the system behaves with big changes in electricity consumption. Several cascading studies use this method [35, 41]. This simplification cannot capture the frequency at every bus in the system (therefore it cannot model if a power system is partitioned into islands), nor model how each generator will react differently to cascading incidents (therefore it cannot model how the protection mechanism in each generator will activate).

   (b) **System Dynamics:** In this type of transient analysis we model all generators in the power system and all the frequencies in all the buses of the system. This is in line with one of the main objectives of a transient stability study—to determine whether the resulting angular separation between the machines in the system remains within certain bounds so that the system maintains synchronism [36]. Cascading analysis models with system dynamics are considered in [28, 34, 40].

2. **Steady-State Analysis** finds the voltages and currents of the system after all frequency equipment has tripped and can help us understand if the system ends up in a configuration where voltage protection or overcurrent protection equipment will activate. To compute these values, a power flow program uses Kirchhoff's physical laws to obtain the voltage magnitudes and phase angles at each bus of a power system. As a by-product of this calculation we can also compute real and reactive power flows in equipment such as transmission lines and transformers, as well as equipment losses [31]. There are two ways to perform steady state analysis:

(a) **DC Power Flow:** Direct Current (DC) Power Flow is a very fast way to compute voltages and phase angles. There are several cascading analysis studies that use DC Power Flow models [22, 27, 57]. DC power flow models however are approximations to AC models, and they do not show the variations on voltages that might trigger protection equipment, therefore DC methods are only valid when voltages are close to their nominal values, which rules out their use for modeling large-scale events such as MadIoT attacks.

(b) **AC Power Flow:** Alternating Current (AC) Power Flow is a more accurate (but computationally more expensive) way to analyze the steady state behavior of the power system. The only way to model voltage protection systems is with the use of AC power flow. Cascading analysis with AC power flow methods include [35, 41].

2.2.2 Power System Protections

In the previous subsection we have argued that the best practices for an accurate portrayal of power system behavior under *large-scale* events (i.e., events where voltages go beyond nominal values, and where individual generators might go beyond safety limits) is to use (1) System Dynamics for transient analysis, and (2) AC Power flow for steady-state analysis. In this section we describe how the results of our transient and steady-state simulations are used to evaluate how protection equipment in the power grid will react to changes in the operation of the system.

In particular, we model four protection mechanisms that are relevant for cascading analysis studies:

1. **Protection of Generators:** when the frequency of the system is too low or too high, the generator will be automatically disconnected from the power grid to prevent permanent damages to the generator.

2. **Under Frequency Load Shedding (UFLS):** if the frequency of the power grid is too low, controlled load shedding will be activated. As discussed before, this disconnection of portions of the distribution system is done in a controlled manner, while avoiding outages in safety-critical loads like hospitals. UFLS is activated in an effort to increase the frequency of the power grid, and prevent generators from being disconnected (as discussed in the point above).

3. **Overcurrent Protection:** if the current in a transmission line is too high, a protection relay will be triggered after time T. This activation time is based on an equation for current relays [10]. We will discuss in detail this equation when we describe our cascade outage analysis model.

4. **Over/Under Voltage Protection:** if the voltage of a bus is too low or too high, a voltage relay will be triggered after time T. This activation time depends on an equation modeling configuration thresholds and over/under voltage relay pick-up values [2].

2.2.3 Industry Practices

For day-to-day operations related to power grid failures, power operators focus on satisfying the N-1 criterion as this is the most important failure condition that is regulated and enforced by most electric regulatory agencies. Large-scale events such as a massive natural disaster, a terrorist attack, or a cyber-attack have not been a major priority for industry practices because the likelihood of these events is very small, and investment in preparing for these events has higher costs than responding to them when they happen [49].

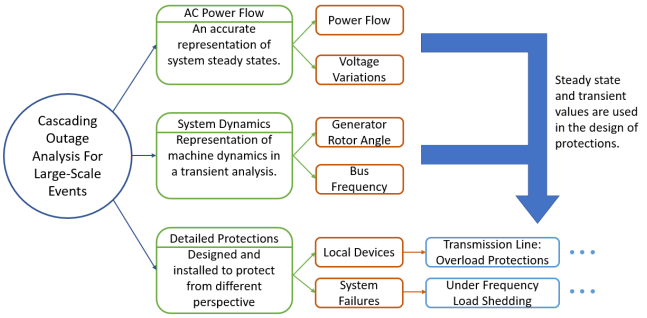


Figure 2: Analysis of Cascading Outages.

Therefore most of the industry efforts on cascading studies focus on smaller-scale events that initiate a cascade, and where the transient dynamics do not affect the cascade analysis too much. These efforts include the Transmission Reliability Evaluation of Large-Scale Systems (TRELSS) [32, 39] and the Oak Ridge-PSERC-Alaska (OPA) [18]. Similar problems have been studied by system operators like ERCOT [3]. Our tool on the other hand is designed for the study of the large disruptions in the operation of a power system like a deliberate cyber attack which can take hundreds of lines out in a short time, and therefore transient analysis has to be coupled with steady-state analysis.

The integration of (1) System dynamics, (2) AC power flow, and (3) the timing of protection equipment gives our tool a level of fidelity that goes beyond the current state-of-the-art practices [22, 27, 28, 34, 35, 35, 40, 41, 41, 57]. These three analysis techniques and their relationship are shown in Figure 2.

2.3 Contributions

Our contributions to the study of a MadIoT attacks compared to recent work [24, 47] include the following:

First, previous work considered transient and steady state simulation as separate use-cases (and in different inconsistent power systems), and as a result, the transient impacts on generators and system frequencies are not present in the power flow simulations. Therefore the predictions of cascading outages can differ between the two simulations. As we explain in Section 4.1, without the transient effect, the power flow solution will indicate a system blackout, while in reality Under Frequency Load Shedding will activate before generators start tripping and will prevent a system blackout.

Second, including the exact timing for the activation of a protection relay captures the realistic behavior of equipment in the power grid. Previous works on IoT attacks to the power grid [28, 34] do not represent the delay characteristic of protection equipment, but rather use models that appear to be based only on the immediate removal of an element after any amount of overload. Such a model violates NERC criteria for overload protection [1]. Our model is instead a discrete event simulator that does not assume that all relays will trip at the same time. In particular, we model equipment under stress, such as current overloads of 50-100% of the line rating. This model is based on the curves from manufacturers [2, 10] that relate the overload of the device to the time until it trips—e.g., if the overload of the line increases significantly, the trip time would be much shorter.

Third, we also perform the first large-scale transient analysis of MadIoT attacks on a real-world North American regional system with over 5,000 buses. This large-scale analysis shows that the most powerful MadIoT attacks can partition the bulk power system into three or more isolated islands. The power grid does not go into a system blackout, but each island will be more vulnerable to future contingencies. This is a new effect that has not been considered before.

Because by repeating the same attack conditions from previous work did not cause any blackout in our system, we introduce new variations of the MadIoT attacks where for example, the attacker systematically tries to create oscillations of demand in order to drive the system into a more vulnerable state before launching the second stage of the attack.

Finally, all our simulations are done in PowerWorld [4], which is an industry-standard transient and AC steady-state solver, as its basic building block, so the basic physics of the system are represented with industry-accepted fidelity.

These contributions are summarized in Table 1.

Table 1: Contributions

Contributions		**Our Work**	**Previous Work [24, 47]**
Simulations	Transient	PowerWorld	PowerWorld
	Steady-state	PowerWorld	Matlab
	Combined transient and steady-state analysis	Yes	No
Transient Analysis	**Under Frequency Protection**	Yes	No [47]
	Frequency in all buses	Yes	No [24]
Steady-state Analysis	**Power Flow**	AC	Not Specified
	Time for Over Current Protection	Yes	No
	Time for Voltage Protection	Yes	No
New MadIoT Attacks	**IoT Demand Increase and Decrease**	Yes	No
	IoT Repeat	Yes	No
Scale of Analysis	**Case used in Transient Simulation**	A North American regional system with over 5,000 buses	Up to WSCC 9-bus system
	Case used in Steady-state Simulation	A North American regional system with over 5,000 buses	Polish system with 3,120 Buses

3 Cascading Outage Analyzer

This section summarizes our Cascading Outage Analyzer (COA) tool. The COA model considers both steady-state and transient stability analysis in different time scales but coordinated so the transition of system stability from one steady-state operating point to another is present. The basic model checks for conditions that would trigger protective relays, and assesses the time when relays will be triggered.

The framework of the COA is described in Figure 3. The simulation has both transient and steady state parts. For each contingency, a transient simulation is run using the PowerWorld transient simulation tool. If the system reaches a stable state, then simulation results are sent to the steady state simulation as initial values, where an AC power flow is run. Based on the resulting line flows and voltage magnitudes, the timing for activating protection equipment is then computed.

If there are any new protection equipment activated from this steady state simulation, the new outage will be modeled and the next iteration of simulation will start using the PowerWorld transient simulation tool. This multi-time scale process continues until no outage occurs in both the transient and steady-state parts of the simulation, or until the transient simulation is unable to solve the problem, in which case an "algorithmic non-convergence" is declared to have occurred as a proxy to a system blackout.

We now describe how each of the four protection systems we consider are modeled.

3.1 Protection of Generators

If a mismatch between generation and load occurs, there will be a frequency deviation from the desired nominal value (if there is more load than generation, the frequency of the system will decrease, and if there is more generation than load, the frequency of the system will increase). A big frequency deviation may trigger generator under- and over frequency-protections.

Transient stability or rotor angle stability is the ability of

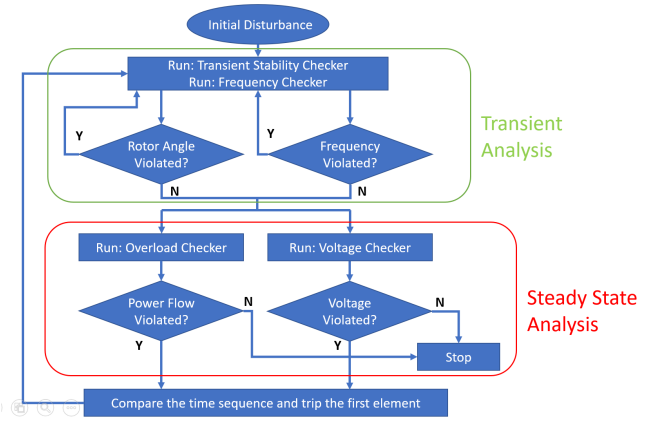


Figure 3: Overview of our Cascading Outage Analysis Tool.

the power system to remain in synchronism when subjected to large transient disturbances [37]. We choose to use time-domain simulation because the time-domain simulation takes into account the full system dynamic model and constantly checks that inter-machine rotor angle deviations lie within a specific range of values.

We use the PowerWorld transient stability solver to numerically calculate the system response after a fault. If the rotor angle deviation of a generator is bigger than a certain threshold, e.g., 100 degrees, the generator will be automatically tripped and removed from the power grid to prevent permanent damages. The disconnection of the generator won't be immediate after crossing a threshold, but it will be dependent on the amount of time that it remains in the unsafe region. We will discuss the exact configuration parameters for disconnecting a generator later in the paper.

3.2 Preventing the Tripping of Generators

When the system loses a generator or when there is a sudden increase in the load, the frequency of the power grid decreases rapidly. A countermeasure to prevent the activation of (more) under-frequency generator protections is a mechanism called Under Frequency Load Shedding (UFLS). The predominant system condition addressed by IEEE C37.117 involves the use of protective relays for under frequency shedding of connected load in the event of insufficient generation or transmission capacity within a power system. Therefore, we include UFLS along with over/under frequency generator tripping as frequency outage checkers in the COA model. Taking into consideration these protections embedded in power systems is one of the reasons we obtain different results when compared to previous work.

3.3 Overcurrent Protection

Disconnecting transmission lines because of a thermal limit violation is one of the most common events in cascading outages [53]. We trigger overcurrent protections based on the results from our steady-state results. The status and dispatch set points of units at the end of the PowerWorld transient analysis are used as starting points for the PowerWorld AC power flow simulator. An inverse-time overcurrent equation described in the Siemens SIPRO-TEC 5 Current Relay [10] is implemented in our model. The time when the over current relay trips the element is determined by equation (1),

$$T = \frac{0.14}{(\frac{I}{I_{th}})^{0.02} - 1} T_p[s], \tag{1}$$

where I_{th} is the current threshold value of the relay, and T_p is the setting value of the relay. Both values are set by the relay operator. I is the current on the monitored component such as a transmission line or a transformer. The value of T in (1) determines when the protection will be activated. It is important to understand that overloading the line past its nominal rating does not immediately result in a transmission outage. Simplified models that do not account for the detailed behavior of protection equipment are likely to consider that a line gets out of service when in reality it keeps operating (it just sags). This is another of the reasons we obtain different results from previous work.

3.4 Over/Under Voltage Protection

Another typical pattern associated with cascading outages is an under (or over) voltage problem. When the system is highly stressed, the voltage profiles of power systems may decline. Even if the AC power flow calculation converges, if a bus voltage stays below the lower limit in our simulations, a load-shedding protection mechanism will be triggered in order to return the bus voltages to their limits [53].

The bus voltages are required to be on a range for the safe operation of the connected generators. A generator may also be disconnected if the voltage of the connected bus goes out of limits for too long.

We implement in our simulator a standard inverse time characteristic equation described in ABB RXEDK 2H time over/under voltage relay [2] to find the timing for the activation of voltage protection equipment. The time duration until the under or over voltage relay trips is determined in equations (2) and (3),

$$T = \frac{k}{(\frac{U}{U_{th}}) - 1}[s], \tag{2}$$

$$T = \frac{k}{1 - (\frac{U}{U_{th}})}[s], \tag{3}$$

where k is the inverse time constant, U_{th} is the over/under voltage relay pick-up value, and U is the user defined relay operating value. The values of T in equations (2) and (3) determine when the protection will activate. As with the line overload model, over/under voltages do not immediately result in a bus outage.

4 Considerations for Modeling the Impact of IoT Attacks

This section will demonstrate the contribution of applying our cascading outage analyzer in the study of IoT demand attacks and in particular, this section will compare our results with previous work in order to show why we obtain different results. We will start our analysis with a relatively simple but standard Western System Coordinating Council (WSCC) model with 9 buses and 9 lines, as this is a model that has been used in previous work. We will also discuss in more detail some of our considerations for modeling the impact of IoT attacks. In the next section we will provide a detailed study on a model of a real-world North American system.

In this section we use the over/under frequency generation protection and Under-frequency load shedding parameters from Table 2 and Table 3. In the next section we will explain in more detail these parameters.

4.1 The Need for Combining Transient and Steady-State Simulations

Since the operation of a power system after a disturbance is a continuous process over a long time frame, a closed-loop structure of the cascading outage analyzer can better approximate the operations of the power system over various time scales after a disturbance. As previously discussed, the results and states of the system after the transient simulation are stored and set as the starting point of the steady-state simulations. The cascading outage generated from steady state simulations, if there is any, is then used as the initial condition in the transient simulation for the next loop.

Previous work considered transient and steady-state simulations as separate, and as a result, the transient impacts on generators and system frequencies are not present in the power flow simulations. Therefore the predictions of cascading outages can differ when compared to our work. Let us look at an example to see a possible inconsistency, while emphasizing the importance of a combined transient/steady-state simulation for the analysis of cascading outages caused by IoT demand attacks.

Figure 4 shows the WSCC 9-bus system considered by Soltan et al. [47]. Consider an IoT demand attack that increases all loads by 15% in the system. Now let us see what happens if a transmission line is removed if the power flow is over its rated capacity [20]. If the transient impacts of this

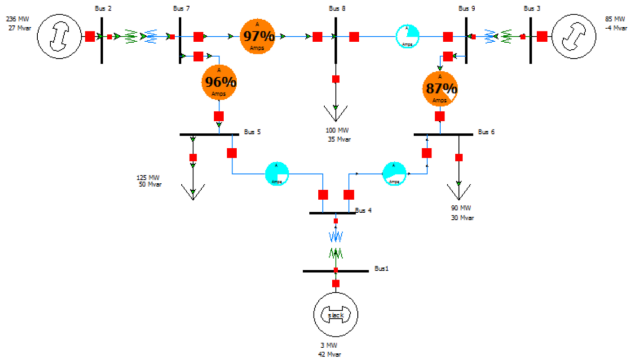


Figure 4: PowerWorld 9-bus system.

attack are not considered, the results from the steady-state power flow would indicate a line outage between bus 7 and bus 8, as highlighted with a red circle (shows the percentage of the rated capacity) in the top left corner in Figure 5.

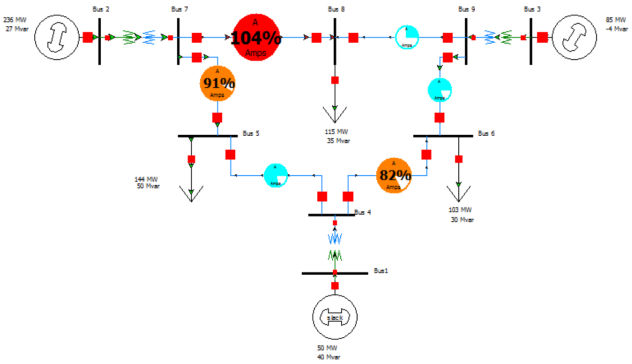


Figure 5: Power flow results of 15% of load increase.

However, because of the sudden load increase caused by the MadIoT attack, load and generation are not balanced and the frequency of the system will be affected. A frequency protection relay would disconnect a generator from the system if the frequency of the system stays lower or higher than the generator's threshold values for too long in order to prevent permanent damage to the generator. Figure 6 shows the frequency responses to the 15% load increase. We can see that the system frequency starts to decline after the attack starts (the attack starts after one second). The frequency relays then disconnect all the generators in the system two seconds after the frequency drops below the threshold of 58 Hz (table 2). Therefore, this results in a blackout in the transient simulation of the IoT demand attack. These transient stability results are different from the steady state stability study, which identifyied only one cascading line outage as discussed in the previous paragraph.

This is a motivating reason to include transient and steady state analysis together in a single simulation. Because transient and steady-state simulations are connected in a closed

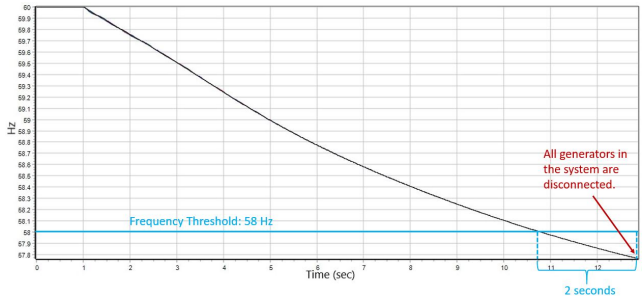


Figure 6: Frequency responses to the 15% of load increase in the transient simulation.

loop in our model, the transient solution at the end of the simulation time will be used as an initial condition for the steady-state power flow simulation. In this example, if the under frequency load shedding is not considered, which will be discussed in Section 4.2, the transient solution would include the fact that all three generators were disconnected from the system. Thus, the power flow solution would indicate a system blackout.

4.2 Under Frequency Load Shedding

Under Frequency Load Shedding (UFLS) is a countermeasure applied by bulk power system operators [5] to reduce the incidence of generator under-frequency tripping, which is a great danger to the reliable operation of the power systems. UFLS is a coordinated disconnection of small and non-critical (e.g., no Hospitals are ever disconnected) loads to prevent a large blackout.

To illustrate why it is important to consider UFLS in the simulation of IoT demand attacks, let us first take a second look at Figure 6. As observed, after the 15% load increase attack, the system frequency starts to decrease. Because there is no action that could relieve the imbalance between the increased load and unchanged generation, the system frequency declines fast until it drops below the thresholds of frequency protections at generators. Because the frequency stays below the thresholds for longer than the delay time set at the frequency protections, the generators are disconnected and there is a system blackout.

Now, let us compare the simulation results when we incorporate UFLS as defined by the parameters in Table 3. Figure 7 shows the frequency response to the 15% system demand increase attack on the WSCC 9-bus system. The system frequency declines after the IoT load increase attack starts at one second of the simulation time. The frequency of the system then reaches the first UFLS threshold at 59.3 Hz, and as a result, 5% of the system load is disconnected. However, this is not enough and the system frequency keeps declining until it reaches the second threshold: 58.9 Hz, and at that time a total of 15% of the system demand is disconnected and the frequency stops decreasing and starts to stabilize to its desired state. The system frequency reaches a new stable state and there are no generator disconnections from the system.

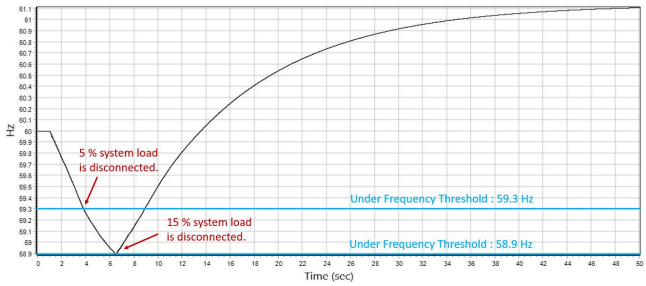


Figure 7: Frequency responses with Under Frequency Load Shedding to the 15% of load increase in the transient simulation.

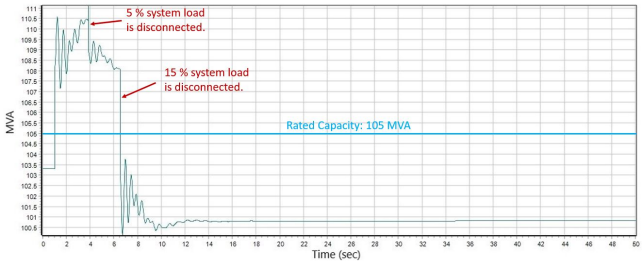


Figure 8: Power flow on the transmission line connected between bus 7 and bus 8 in the transient simulation

What is more, because of UFLS, the system load is reduced to a level where no transmission line is overloaded, and therefore there are no cascading outages. In Figure 8, we can see that the transmission line between bus 7 and bus 8 in Figure 4 is overloaded after the IoT demand increase attack begins at one second. However, the power flow on the line soon decreases following the load shedding event caused by UFLS and remains below its rated capacity at the end of the transient simulation. As discussed in Section 4.1, a power flow steady state simulation starts based on the solution of the transient simulation; the results of this new steady state stability analysis are shown in Figure 9. We can see that no line is overloaded and the combined transient and steady-state simulations end.

The example in this subsection shows that the simulation results will be significantly affected if UFLS protections are considered. In fact, by including UFLS, the closed-loop transient and steady state simulations used in this work generates a result suggesting that the system would shed some demand, but all the system transmission lines and generators will remain in operation. This result is different from the cascading line outage suggested by our steady-state simulation illustrated in Figure 5 and the complete system blackout suggested by previous work.

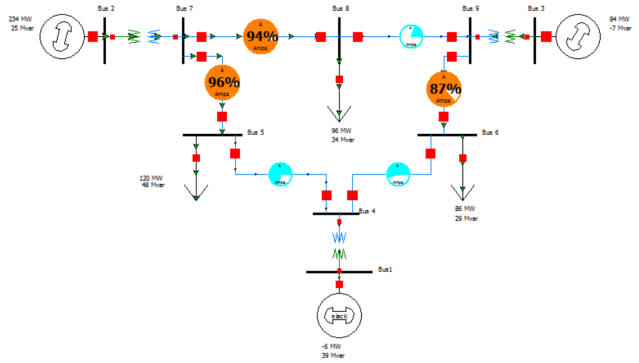


Figure 9: Power flow results after the transient simulation with UFLS.

4.3 Frequency Response Model

UFLS protections are indeed considered in some previous work [24]. However, the simplified frequency response model used by the authors is not a good fit to analyze IoT demand attacks. The system frequency responses used by Dabrowki et. al [24] model the power grid as a single large machine that represents an "aggregation" of all the synchronous generators in the system.

A synchronous machine is associated with a rotating magnetic field winding that induces alternating voltages in a armature windings of the stator. The frequency of the induced alternating voltages and of the resulting currents that flow in the stator windings when a load is connected depends on the speed of the rotor. The frequency of the stator electrical quantities is thus synchronized with the rotor mechanical speed, hence the designation "synchronous machine" [37]. When two or more synchronous machines are interconnected, the stator voltages and currents of all the machines must have the same frequency and the rotor mechanical speed of each is synchronized to this frequency. Therefore, the rotors of all interconnected synchronous machines must be in "synchronism" [37].

In contrast, the assumption of Dabrowski et. al [24] is that every generator in the system will respond to a disturbance exactly the same. In other words, the implicit assumption of this model is that all the generators in the system will always keep synchronism and respond identically. However, when the system is under a significant disturbance, generators will respond differently to the disturbance and the system will have the risk of losing synchronism in a short time after the disturbance. In some scenarios, the frequency protections will contribute to a lack of synchronism, and therefore, the frequencies at different buses will diverge from synchronism. All of this frequency diversity can not be reflected in the single machine mode [24]. A detailed discussion of why this phenomenon is important will be demonstrated in subsection 5.4, where we show how different parts of the grid start operating at different frequencies and therefore the system becomes a set of islands operating semi-independently.

4.4 Line Overloads

The line overload outage models also play an important role in understanding the impact of MadIoT attacks. Previous work [47] relied on the criteria described by Cetinay et al. [20], where a line will be removed from the system if the steady-state results indicate that the power flow on the line is greater than its rated capacity. When a transmission line is overloaded, the heat generated from the extra power flow on the line will sag the transmission line. Although it exposes the line to a possible outage from faults associated with ground element or vegetation, it does not necessarily cause any immediate real danger to the system. In fact, under an emergency, the system operator is allowed to use overloaded transmission lines for additional transmission capacity [6]. Therefore, instead of immediately removing the overloaded lines, we utilize a model that calculates the time of tripping given the overload level. The details are described in Section 3.3. The time inverse calculation in the outage protection mechanism will result in a quick tripping time for the lines that are heavily overloaded. In this way, we approximate the different actions taken at different levels of overload on transmission lines.

4.5 IoT Demand Attacks

In addition to fixed demand increase (or decrease) attacks, we also consider attacks that increase and then decrease the load. The intuition for this attack is that the first part of the attack will force automatic responses from the grid (such as UFLS) and therefore when the system starts operating with a reduced load, a reversal in the load (a big decrease) can drive the system to a potentially unstable state. After initial attack increasing the demand, the attackers will decrease the demand when they think the system frequency reverses due to UFLS and intend to overshoot the system frequency over the thresholds of generator frequency protections in the hopes of causing a generator disconnection.

This demand increase and decrease attack was studied by Dabrowski et al. [24]. However, our results will differ because of their simplification of the frequency model, as discussed in Section 4.3. In addition, if the attacker can cyclically increase and then decrease demand, it is reasonable to assume that the attacker is capable of repeating this attack. The simulation results and detailed discussions of the experiments are shown in Section 5.

5 Simulation Results in a Large Power System

The study case we use to analyze the impact of the IoT demand attacks is a large North American regional system with more than 5,000 buses, and as such it is the largest study

Table 2: Over/Under Frequency Generator Tripping. Source: Section 2.6.1 of [5].

Over Frequency Threshold	Time Delay	Under Frequency Threshold	Time Delay
60.6 Hz	9 min	59.4 Hz	9 min
61.6 Hz	30 sec	58.4 Hz	30 sec
61.8 Hz or above	0 sec	58.0 Hz	2 sec
		57.5 Hz	0 sec

done on the impact of IoT attacks on power systems. Unfortunately, because our close collaboration with the operator of this power systems we are required to maintain the confidentiality of this system and we are not allowed to share the name of the system or details of their network topology. Before we describe our simulation results we clarify our assumptions.

5.1 Assumptions

We state three main assumptions about an IoT demand attack:

1. IoT attackers have full and unlimited ability to control the compromised portion of loads;
2. The actions of attackers to increase or decrease the compromised loads are simultaneous;
3. The portion of the system demand compromised by the cyber attackers are evenly distributed at each demand connection point in the transmission system.

The third assumption is a speculation about the scalability of an IoT attack. For example, if the adversary is able to compromise one brand of air conditioner, they can systematically apply the attack to as many air conditioners as possible in the target system. Thus, if the total energy capacity of all such air conditioners is 10% of the system demand, this 10% of demand is likely to be spread to every demand connection point in the transmission system.

5.1.1 Parameters Used for Protection Equipment

There are two protections implemented in the transient simulation, namely Over/Under Frequency Generator Tripping (O/UFGT) and Under Frequency Load Shedding (UFLS). If the frequency at a bus deviates from a predefined threshold for more than a specific time period, the generator connected to that bus will be tripped, and a certain percentage of load connected to the bus will be shed. The details of O/UFGT and UFLS are shown in Table 2 and Table 3 specifically.

Since the current and voltage responses in the system are normally slower than frequency responses, the Time Inverse Overload, Time Inverse Under Voltage Load Shedding, and Time Inverse Over Voltage Generator Tripping are modeled in the steady state simulation. Each protection checker will calculate tripping times once the current flow on branches or the voltage at buses exceed the thresholds. The element (branch, generator, or load) with the shortest tripping time will be tripped as the initial conditions for the next iteration of transient simulation. The parameters of the steady state protection models described in equations (1-3) are listed in Table 4.

Table 3: Under Frequency Load Shedding. Source: Section 2.6.1 of [5].

Frequency Threshold	System Load Relief	Time Delay
59.3 Hz	5 %	0 sec
58.9 Hz	15 %	0 sec
58.3 Hz	25 %	0 sec

Table 4: Steady State Protections. Source: [53]

	Over Load	Over/Under Voltage	
		over	under
Threshold	$I_{th} = 2 \times line$ limit [amps]	$U_{th} = 1.3$ [pu]	$U_{th} = 0.8$ [pu]
Parameters	$T_p = 0.05$	$k = 0.5$	$k = 0.5$

5.2 Demand Increase Attacks

The most intuitive MadIoT attack against the power grid is a sudden increase of demand. This will attempt to overload the transmission lines and potentially cause cascading failures.

5.2.1 1% Demand Increase Attack

Previous work showed that a 1% increase attack against the Polish power grid in 2008 caused cascading failures. In their system, a 1% load increase corresponded to 210MW, requiring the adversary to compromise about 210,000 air conditioners. In our system, one percent of the load is equivalent to 822.7 MW, which would require the attacker to compromise approximately 822,000 air conditioners.

Figure 10 shows the bus frequency responses after 1% of load increase at second 1 and Figure 11 shows the power flow on branches as a percent of the branch rated capacity. We can observe that the bus frequencies shown in Figure 10, decline after the attack at second 1 except for very few buses that are connected to the region outside of the system with DC tie lines (the ones that remain at 60Hz on top of the diagram) and thereby remain less affected.

The rest of the frequencies decline from 60 Hz to 59.875 Hz in about 9 seconds and settle to a new stable state towards the end of the transient simulation. As indicated in table 2 and table 3, the system frequency doesn't violate any thresholds of

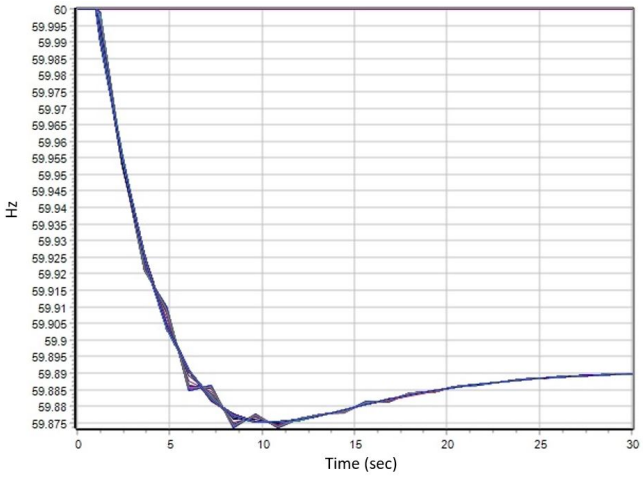


Figure 10: Frequency Response to 1% System Load Increase.

frequency protections on generators and loads. Notice that we focus our study in a short time window, since 30 seconds of transient simulation is enough to display the moving trends of the frequency in this case. In short, we can see the how the frequency is affected after the attack; however, as long as the bus frequency converges to a stable level, driving the frequency back to 60Hz can be accomplished either automatically or manually over a longer time scale.

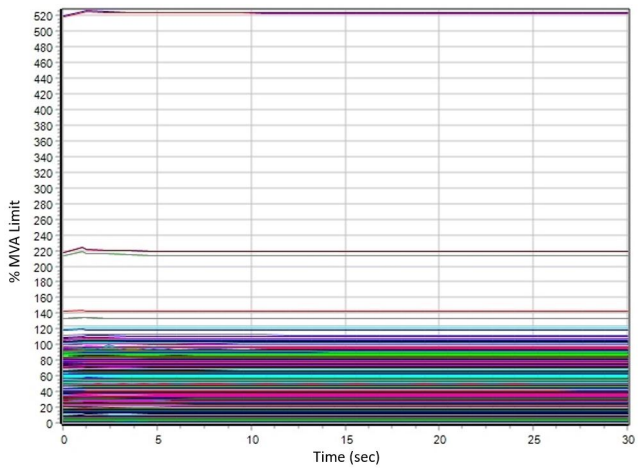


Figure 11: Branch Flow after 1% System Load Increase.

In Figure 11, we can see that the power flow of some branches slightly increases after the attack at second 1. However, no transmission line is overloaded resulting from the IoT 1% load increase attack. Note that some branches are initially overloaded before the simulation and remain unchanged during the simulation and the overload outage checker is not activated on those lines under the assumption that protection in the actual system would not have been activated under these conditions.

In summary, a 1% load increase attack does not affect our system and there is no need to activate any protection equipment as the transmission lines remain operating in their nominal values and the frequency of the system does not reach thresholds to activate any protection.

In contrast to our results, Soltan et al. [47] find that with a 1% increase in load there could be cascading outages in the summer peak of the Polish grid. We are surprised that a sudden 1% increase in load can lead to cascades in a power system. The reason for our surprise is the N-1 security criterion.

The N-1 criterion requires that electricity systems be operated to be able to withstand sudden step changes in the supply-demand balance due to outages of generation. The NERC disturbance control performance standard [8] requires any system to be able to withstand "the most severe single contingency" which may include certain common-model double outages. For ERCOT, for example, (the Power Grid of Texas) this amounts to always having 2700 MW or more of reserves to cope with a simultaneous outage of nuclear units having total production of around 2700 MW. To put that in perspective, peak load in ERCOT is around 70GW, and 1% of 70GW is 700MW, which is much smaller than the 2700MW of reserves carried in ERCOT to satisfy the N-1 criterion.

While an increase by 700MW in load due to an IoT attack (and the reaction by generation reserves) would result in somewhat different changes in transmission flows compared to the effect of a 700MW decrease in generation (and the reaction by generation reserves), we believe that it is unlikely that an increase in load of 1% would result in any unacceptably adverse conditions on the transmission system. This is because load is geographically distributed around the system, so that it is unlikely for there to be a more than a 1% increase in most transmission flows, and it is unlikely that the system is operating such that a 1% increase in current would immediately trigger the overload protection.

In the Eastern and Western Interconnections of North America, the total load is much larger (several hundred GW) but even 1% of this would only amount to slightly more than the double outage of a nuclear unit (plus it would require millions of compromised IoT devices). To summarize, the results of the Polish power grid reported by Soltan et al. [47] suggest that the system being modeled is not N-1 secure.

5.2.2 10% Demand Increase Attack

Ten percent of system load in our case study is equivalent to 8,227.3 MW, which would be equivalent to an adversary controlling over eight million air conditioners. Figure 12 shows the bus frequency responses after a 10% load increase attack at 1s and Figure 13 shows the power flow on branches as a percent of the branch rated capacity.

To better understand the variations of power flow depicted in Figure 13, let's first take a look at Figure 12. From Figure 12, we can observe that the bus frequencies plummet after the

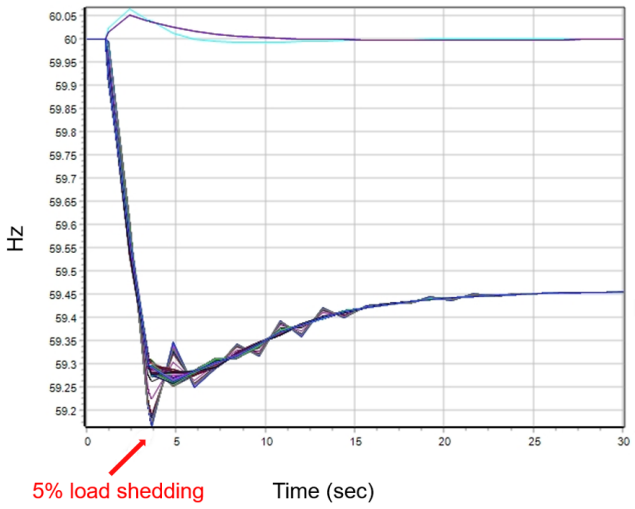


Figure 12: Frequency Response to 10% System Load Increase.

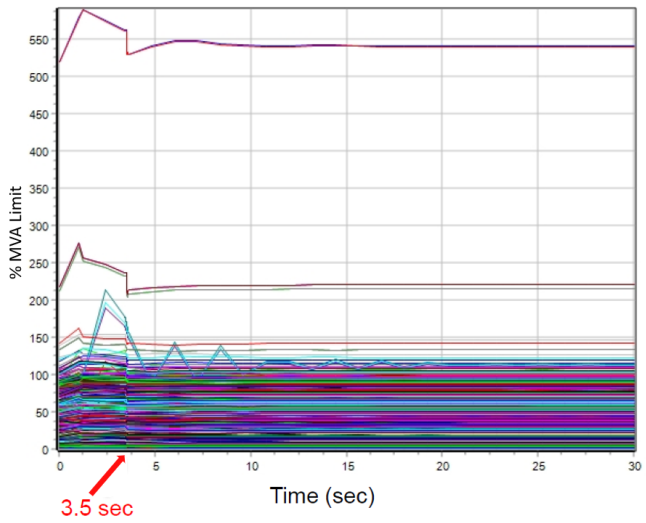


Figure 13: Branch Flow after a 10% System Load Increase.

attack begins (1s). The only lines that are not affected are the few buses that connected our power grid to another region outside the system with DC tie lines (the frequencies at the top of the figure).

In contrast to the previous 1% attack, with a 10% demand increase the power system needs to activate protection algorithms; in particular, 5% UFLS is activated at 3.5 seconds by shedding 5% of the system load. Again, as long as the bus frequency converges to a stable level, the differences between the converged value and its initial value of 60 Hz can be fixed either automatically or manually over a longer time scale.

Although the under frequency shedding has no deliberate time delay as indicated in Table 3, a 0.02 second of relay operation time is included in the simulation. Therefore, the load shedding occurs 0.02 seconds after the time frequency falls below the first UFLS threshold of 59.3 Hz.

In Figure 13, we can see that the power flows of some branches increase after the attack starts (1 sec.). However, the power flows of those branches drop to or gradually decrease to roughly their initial values after the under frequency load shedding protection is activated at 3.5 seconds. Therefore, at the end of the simulation there is no additional transmission line overloaded. Note that some branches are initially overloaded before the simulation and remain unchanged during the simulation and so, as in the previous example, the protection mechanisms for these transmission lines are not activated.

Even with the assumption of millions of compromised IoT devices to affect 10% of our load, our results show that the power grid protections to prevent generators from disconnecting from the system are effective in mitigating any further problem. The amount of UFLS is intended to reflect ERCOT standards. The Eastern and Western Interconnections may have overall lower levels of UFLS than ERCOT; however, they have much larger levels of inertia than ERCOT.

5.3 Increase and Decrease Attack

One of the characteristics of IoT attacks, is that they are highly distributed they are hard to detect. Once the load is compromised, the compromised devices are unlikely to be removed from the grid (or the Internet) in a short time after they launch the first attack. Therefore attackers can launch a sequence of attacks, the first as an attempt to drive the system to a vulnerable state, and the second to exploit that vulnerability.

In the last attack we saw how under frequency load shedding successfully prevented a cascading failure of transmission lines from a single 10% load increase attack. However, a sophisticated attacker can identify when the system frequency starts rebounding after the initial drop, and can attempt to make this trend continue by immediately decreasing electricity consumption. This can cause a frequency overshoot that may trigger the action of over-frequency protection relays on the generators and disconnect them from the power grid; creating another cycle of frequency decrease along with new load shedding etc.

A straightforward approach in this experiment is to increase the load at the first attack and decrease the same amount of load at the second attack. However, we investigate a potentially worse scenario where in the second attack, we decrease by twice the amount of the load increase in the first attack (minus the percentage of the load that the attacker loses control of after the under frequency load shedding implementation).

The result in Figure 14 shows that the frequency does overshoot after the loads decrease at second 20, however the system frequency tends to stabilize at 61.7 Hz, which happens about 10 seconds later. From Table 2, we can observe that

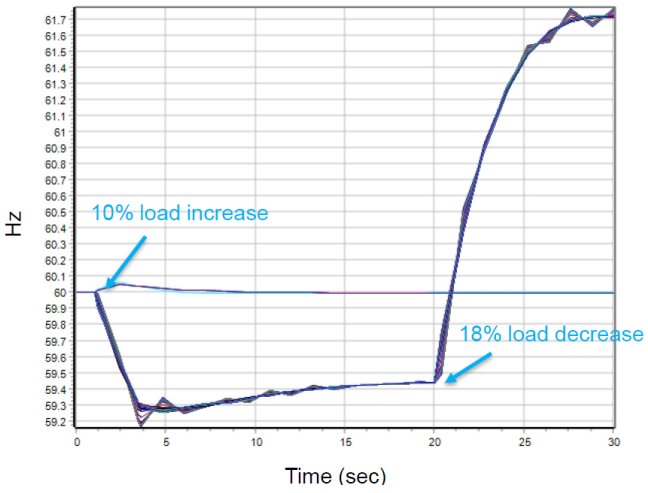


Figure 14: Frequency Response to a Cycle of Load Increase and Decrease

61.7 Hz will not cause an immediate generation trip by the frequency protections at the generators.

As mentioned in Section 5.2, a 10% system load compromised by the adversary is already a pessimistic assumption. We take this even further to 20% of the system load in this simulation to see if the IoT attack can cause a cascading result. However, we still do not observe an immediate generation trip after this demand increase and decrease attack in a system that is intended to reflect ERCOT standards for UFLS.

5.3.1 Under Frequency Load Shedding in a Repeated IoT Attack

We have explored the results of an attack "cycle" of load increase and decrease. The adversary could continue repeating this attack cycle of increasing and reducing the compromised load as long as their capabilities are not disabled by the load shedding mechanism.

The under frequency load shedding would disconnect some amount of demand each time when the IoT attack causes the frequency drop below any thresholds. Once the load is disconnected by Under-Frequency Load Shedding (UFLS) systems, the restoration of shed load is coordinated between the Independent System Operator (ISO), Transmission Service Providers (TSPs) and Distribution Service Providers (DSPs) [5]. It is fair to assume that such restoration, which requires coordination between different entities may take a relatively long time to complete. Therefore, a potential negative effect of such repeated attacks is that they can deplete the under frequency load shedding resources before they are restored, which might eventually lead to having no more UFLS protections against the attacks and will eventually cause a generator to trip.

The result in Figure 12 shows that although the system frequency needs additional measures to be brought back to its initial frequency of 60 Hz, the frequency decline caused by 10% of system load increase can be stopped by only 5% of system load shedding. In Table 3 we can see that in ERCOT, 25% of the system load is contracted as UFLS. Under this condition, the adversary needs to apply the attack at least five times to deplete the UFLS resources. What is more, additional under-frequency relays may be installed on transmission facilities with the approval of the ISO provided the relays are set at 58.0 Hz or below in the real system [5]. That means, in reality, the adversary may need to apply the attack even more times to deplete the UFLS and cause a possible system failure.

Therefore, it may take many cycles of IoT demand increase and decrease attacks to deplete the UFLS resources, and these cycles will not only deplete the resources from a defensive stand point, but also the resources available to the adversary as each activation of UFLS will remove loads controlled by the attacker. Therefore, the efficiency, even the feasibility of the approach of using up the UFLS by such repeated IoT demand attack remains unclear.

5.4 Bifurcations, and Generator Tripping

5.4.1 30% Load Increase Attack

In Section 5.3, we briefly discussed the potential threats of generator disconnections caused by over frequency protections. In this section, we extend this discussion to IoT attacks that specifically target disturbing frequency and causing generator disconnections by frequency protection. In order to observe the response of frequency protection at generators, we study the impact of a MadIoT attack consisting of a load increase or decrease by 30%.

In previous work [47], this 30% load change attack was able to disconnect all generators of the (simplified) North American Western Interconnection, causing a complete system blackout. In our system, a 30% load increase attack would require the attacker to compromise about 24 million air conditioners.

Figure 15 shows the frequency response of our system to a MadIoT attack that increases the system load by 30%. First, we can observe that due to the sudden load increase, the bus frequencies decline dramatically and some of them drop quickly below the first UFLS threshold of 59.3 Hz. At this point 5% of the system load is disconnected by UFLS.

We notice that the frequency in some buses decline at a slower rate than others and they do not reach any UFLS thresholds. For convenience, we name this set of buses Group 1. The buses with DC tie lines are again less affected, and we call this set of buses Group 2. The group of buses whose frequencies decline faster and drop below UFLS thresholds are named Group 3. The group names are indicated in Figure 15.

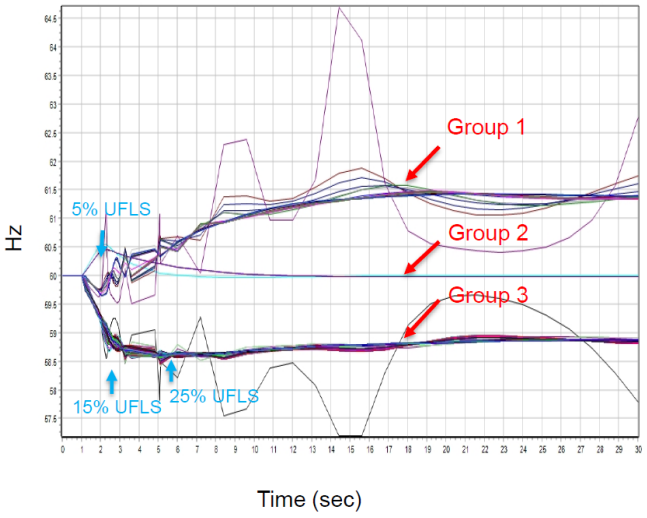


Figure 15: Frequency Response to 30% Load Increase.

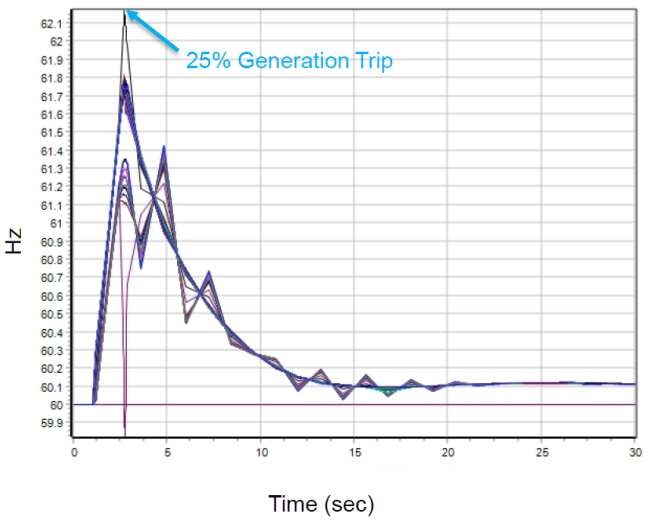


Figure 16: Frequency Response to 30% Load Decrease.

Notice that, even within a group, the frequency responses are not exactly the same. Because of the first UFLS action, the frequency deviation between buses increases. After the 5% load shed, the frequency of Group 1 starts to increase–potentially this group has more generators in their region—while the frequency of Group 3 keeps declining–indicating that this region of the grid has insufficient generation of electrical power.

Shortly afterwards, the frequency of Group 3 declines to the point where the second and third UFLS thresholds, 58.9Hz and 58.3Hz, need to be activated (at around 2.6 seconds and 5.6 seconds respectively). An additional 10% of system load is disconnected in each occasion. The frequency deviation between Group 1 and Group 3 gets larger after the two UFLSs. What is more, the frequency deviation between buses in a group, especially in Group 1, increases after the actions of UFLS.

After the three activations of UFLSs for group 3, which disconnect a total 25% of system load, the frequency decline at Group 3 is stopped. Because there is no additional load shedding, the frequency at Group 1 stops increasing as well. Thus, although the bus frequencies have not converged at the end of the simulation, they stop diverging and there is no need to activate frequency protections to disconnect generators.

5.4.2 30% Load Decrease Attack

We now study what happens if instead of increasing the load by 30%, we decrease the load by 30%. In this case we expect the frequencies in all buses to increase dramatically; furthermore, because UFLS can only be activated when the frequency is decreasing, then we know that there are no immediate protections to prevent a generator from disconnecting from the grid because of its over-frequency protections.

Figure 16 shows the system frequency response to a MadIoT attack that decreases the system load by 30%. We can see that the bus frequencies increase after the attack and a few of them go above the threshold of immediate over frequency protections at generators, which is 61.8 Hz within 5 seconds. The over frequency protections then disconnect generators, resulting in a 25% reduction of system generation. After the tripping of generators, the bus frequencies reduce and converge to a value close to 60 Hz and no more protection actions or failures are observed.

Because we model the time in which each generator is disconnected, we can see that not all of them are disconnected synchronously, as suggested in prior work, but at different times, depending on their configuration settings. When some generators are disconnected, then the frequency drops and is stabilized by the remaining generators.

Compared to the system frequency response to an IoT attack that increases the system load, we find that the bus frequencies react differently to the IoT attack that decreases the system load. In Figure 16, although the frequencies of some buses increase faster than those in some other buses, the frequencies gradually converge after 25% of the system generation is tripped. One of the conclusions we can draw from this comparison is that a quick protection reaction in big scales like the generation tripping in Figure 16 performs better than the gradual protection actions like the load shedding in Figure 15 in terms of the system frequency restoration.

We also find that the tripped generations in this simulation consist of a significant amount of wind generation. The benefit of disconnecting the wind generation or any generation that doesn't provide inertia in this condition is that the system loses less inertia after the over frequency protection action. Therefore, the system doesn't become weaker in terms of

maintaining frequency stability. This phenomenon suggests that generation that doesn't provide inertia could be included in the over frequency protection to protect the system against any following attacks targeted at disturbing the system frequency after an IoT attack.

6 Limitations

Our results also assume that all grid operators satisfy the N-1 security criterion. This is the general practice and should be expected as operators can get massive fines if they are found to be in violation of this criterion. Having said that, some blackouts have occurred because operators believe they are satisfying the N-1 criterion but a misconfigured protection device that should have been activated during an event was not activated, and this created an unanticipated N-2 event that initiated a cascading failure. As discussed in our summary of related work on cascading analysis, stochastic models can complement our approach by establishing the risk or likelihood that one of our protection devices does not work as expected and causes a series of cascading events.

We believe the type of protections considered in this study is the subset of the protections in power systems that would contribute to a cascading outage the most after a disturbance in the system. However, future work can be done to explore the impacts from other protections that are commonly equipped in the power systems e.g. differential and distance protections on buses [31]. In addition, in this study, we considered only an IoT demand attack that is evenly distributed across all the load points in the system. However, in future work, we will consider how feasible it is to compromise a large-scale set of high wattage IoT devices in a specific geographical area such that that target only a part of the system.

7 Related Work

The importance of stronger cyber security requirements in SCADA systems is highlighted by recent experiences in Ukraine. On December 23rd 2015, a third party illegally accessed the computer and Supervisory Control and Data Acquisition (SCADA) systems of three regional electricity distribution companies in Ukraine. Investigations revealed that a malware named BlackEnergy had infected the SCADA systems after successful spear phishing attacks. Seven 110 kV and twenty-three 35 kV substations were disconnected for three hours resulting in several outages that caused approximately 225,000 customers to lose power across various areas [19]. The following year, on December 17th 2016, a second power outage occurred in Ukraine and deprived part of its capital, Kiev, of power for over an hour. An assessment was made that a more advanced form of malware called "Industroyer", was used in the second cyber attack against the power grid in Ukraine [23].

While both security researchers and industry practitioners have worked on the security of the power grid for a decade, their focus has been on understanding and preventing attacks to devices in the bulk of the power grid [7, 17, 38, 48, 51], i.e., the components controlling the operation of the electrical transmission system in large geographical areas and the Supervisory Control and Data Acquisitions (SCADA) systems.

While in the U.S. the bulk power system is regulated to maintain a minimal set of cybersecurity standards [7], there is a growing push to start improving the security of systems in the distribution network. On October 19th 2017, the Federal Energy Regulatory Commission (FERC) proposed new mandatory cybersecurity controls to address the risk posed by, for example, smaller grid control centers that are typically less critical than major control centers, but which are nonetheless vulnerable to attacks [9].

Load-altering attacks have been previously studied in demand-response systems [12, 16, 21, 30, 42, 50]. Demand-response programs provide a new mechanism for controlling the demand of electricity to improve power grid stability and energy efficiency. In their basic form, demand-response programs provide incentives (e.g., via dynamic pricing) for consumers to reduce electricity consumption during peak hours. Currently, these programs are mostly used by large commercial consumers and government agencies managing large campuses and buildings, and their operation is based on informal incentive signals via as phone calls by the utility or by the demand-response provider (e.g., a company such as Enel X) asking the consumer to lower their energy consumption during the peak times. As these programs become more widespread (targetting residential consumers) and automated (giving utilities or demand-response companies the ability to directly control the load of their customers remotely) the attack surface for load altering attacks will increase.

8 Conclusions

This paper presents a study of the impacts of IoT demand attacks on power systems using the cascading outage analysis in a North American Regional Interconnection System.

From the simulation results, we show that, 1% of load increase attack does not interrupt any generator, load, or transmission line in the system. We also find that, thanks to under frequency load shedding protections, a 10% of sudden IoT load increase does not cause a cascading failure on the transmission lines.

A "frequency swing attack" is defined as a cycle of load increase and decrease attacks with the aim to push the frequency outside the safety limits of the generators. However, the frequency swing attack doesn't show an ability to cause an immediate disconnection of generators. We also discussed a possible repeated frequency swing attack and the potential impact of depleting the UFLS resources. Our analysis shows that the effectiveness of such attack would be impacted by

any additional frequency protection measures in the system, and by the diminishing resources that the adversary would have to continue the attacks.

We also considered high-impact attacks with control of 30% of the system load. The simulation results show that under a sudden IoT attack increasing 30% of the system demand, load shedding by UFLS would split the frequencies of the buses into islands of different operating regions of the grid. In contrast, a 30% decrease of the load would cause the frequency of the system to increase above the thresholds for over-frequency protections, and will result in the disconnection of some (but not all) generators. Our results show that the actions of UFLS and over frequency protection are sufficient to prevent an immediate system failure over a short time after the attack. Additional actions may be needed over a longer time scale to restore the stable operation of the system, but the main point is that a system blackout will likely not occur in this situation. In addition, we discover that including generations that are not providing inertia in the over-frequency protections would benefit the system in case of following IoT attacks targeted at disturbing the system frequency.

Our results show a different perspective on the risks of IoT attacks to the power grid and will hopefully serve as a starting point for new discussions to assess this threat. We show that while immediate cascading failures or a total system blackout will be very hard to achieve, the power system will still suffer negative consequences. First, UFLS will disconnect various consumers from the power grid. This is done to prevent further damage to the grid, but several consumers will be affected. Second, our attacks show that with millions of high-energy IoT devices, the attacker can potentially cause a bifurcation of the frequency in the power grid, forcing the grid to operated as separate islands and driving it to a more vulnerable state.

Acknowledgments

This work is supported by NSF CRISP awards CMMI-1541159 and CMMI-1925524, by a grant from the University of Texas National Security Network, and by a Defense Threat Reduction Agency award HDTRA1-14-1-0021.

References

[1] NERC Standard PRC-023-4. https://www.nerc.com/pa/Stand/Reliability%20Standards/PRC-023-4.pdf, 2015.

[2] Time over/under voltage relay and protection assemblies model rxedk 2h and raedk user manual. www.abb.com/product/us/9AAC30405217.aspx, ABB Inc, 2004.

[3] 2018 regional transmission plan scope and process. http://www.ercot.com/content/wcm/key_documents_lists/108892/2018_RTP_Scope_and_Process_draft_clean.pdf, Accessed, 2018.

[4] Powerworld simulator 20. https://www.powerworld.com/, Accessed, 2019.

[5] ERCOT nodal operating guides section 2 system operations and control requirements. www.ercot.com/content/wcm/current_guides/53525/02_030116.doc, ERCOT, 2016.

[6] ERCOT nodal operating guides section 4 emergency operations. http://www.ercot.com/content/wcm/libraries/147359/February_1__2018_Nodal_Operating_Guide.pdf, ERCOT, 2018.

[7] Cyber risk preparedness assessment table-top exercise 2012 report. May, 2013.

[8] NERC Standard BAL-002-1a. https://www.nerc.com/files/bal-002-1a.pdf, NERC, 2012.

[9] FERC sets rules to protect grid from malware spread through laptops. Washington Examiner, October,2017.

[10] Siprotec 5 distance protection and line differential protection and overcurrent protection for 3-pole tripping 7sa84, 7sd84, 7sa86, 7sd86, 7sl86, 7sj86 technical data. www.energy.siemens.com, Siemens AG, 2012.

[11] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. Sok: Security evaluation of home-based iot deployments. In *SoK: Security Evaluation of Home-Based IoT Deployments*, page 0. IEEE.

[12] Sajjad Amini, Fabio Pasqualetti, and Hamed Mohsenian-Rad. Dynamic load altering attacks against power system stability: Attack models and protection schemes. *IEEE Transactions on Smart Grid*, 9(4):2862–2872, 2018.

[13] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the mirai botnet. In *USENIX Security Symposium*, pages 1092–1110, 2017.

[14] Ross Baldick, Badrul Chowdhury, Ian Dobson, Zhaoyang Dong, Bei Gou, David Hawkins, Henry Huang, Manho Joung, Daniel Kirschen, Fangxing Li, et al. Initial review of methods for cascading failure analysis in electric power transmission systems ieee pes cams task force on understanding, prediction, mitigation and restoration of cascading failures. In *2008 IEEE Power and Energy Society General Meeting-Conversion and Delivery of Electrical Energy in the 21st Century*, pages 1–8. IEEE, 2008.

[15] Ross Baldick, Badrul Chowdhury, Ian Dobson, Zhaoyang Dong, Bei Gou, David Hawkins, Zhenyu Huang, Manho Joung, Janghoon Kim, Daniel Kirschen, et al. Vulnerability assessment for cascading failures in electric power systems. In *2009 IEEE/PES Power Systems Conference and Exposition*, pages 1–9. IEEE, 2009.

[16] Carlos Barreto, Alvaro A Cárdenas, Nicanor Quijano, and Eduardo Mojica-Nava. Cps: Market analysis of attacks against demand response in the smart grid. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 136–145. ACM, 2014.

[17] Carlos Barreto, Jairo Giraldo, Alvaro A Cardenas, Eduardo Mojica-Nava, and Nicanor Quijano. Control systems for the power grid and their resiliency to attacks. *IEEE Security & Privacy*, 12(6):15–23, 2014.

[18] Benjamin A Carreras, Vickie E Lynch, Ian Dobson, and David E Newman. Critical points and transitions in an electric power transmission model for cascading failure blackouts. *Chaos: An interdisciplinary journal of nonlinear science*, 12(4):985–994, 2002.

[19] Defense Use Case. Analysis of the cyber attack on the ukrainian power grid. *Electricity Information Sharing and Analysis Center (E-ISAC)*, 2016.

[20] Hale Cetinay, Saleh Soltan, Fernando A Kuipers, Gil Zussman, and Piet Van Mieghem. Analyzing cascading failures in power grids under the ac and dc power flow models. *SIGMETRICS Performance Evaluation Review*, 45(3):198–203, 2017.

[21] Bo Chen, Nishant Pattanaik, Ana Goulart, Karen L Butler-Purry, and Deepa Kundur. Implementing attacks for modbus/tcp protocol in a real-time cyber physical system test bed. In *Communications Quality and Reliability (CQR), 2015 IEEE International Workshop Technical Committee on*, pages 1–6. IEEE, 2015.

[22] Jie Chen, James S Thorp, and Ian Dobson. Cascading dynamics and mitigation assessment in power system disturbances via a hidden failure model. *International Journal of Electrical Power & Energy Systems*, 27(4):318–326, 2005.

[23] Anton Cherepanov. Win32/industroyer, a new threat for industrial control systems. *White paper, ESET (June 2017)*, 2017.

[24] Adrian Dabrowski, Johanna Ullrich, and Edgar R Weippl. Grid shock: Coordinated load-changing attacks on power grids: The non-smart power grid is vulnerable to cyber attacks as well. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 303–314. ACM, 2017.

[25] Tamara Denning, Tadayoshi Kohno, and Henry M Levy. Computer security and the modern home. *Communications of the ACM*, 56(1):94–103, 2013.

[26] I. Dobson. Estimating the extent of cascading transmission line outages using standard utility data and a branching process. In *2011 IEEE Power and Energy Society General Meeting*, pages 1–3, July 2011.

[27] Margaret J Eppstein and Paul DH Hines. A "random chemistry" algorithm for identifying collections of multiple contingencies that initiate cascading failure. *IEEE Transactions on Power Systems*, 27(3):1698–1705, 2012.

[28] D. Fabozzi and T. Van Cutsem. Simplified time-domain simulation of detailed long-term dynamic models. In *2009 IEEE Power Energy Society General Meeting*, pages 1–8, July 2009.

[29] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. Security analysis of emerging smart home applications. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 636–654. IEEE, 2016.

[30] Jairo Giraldo, Alvaro Cárdenas, and Nicanor Quijano. Integrity attacks on real-time pricing in smart grids: impact and countermeasures. *IEEE Transactions on Smart Grid*, 8(5):2249–2257, 2016.

[31] J Duncan Glover, Mulukutla S Sarma, and Thomas Overbye. *Power System Analysis & Design, SI Version*. Cengage Learning, 2012.

[32] R. C. Hardiman, M. Kumbale, and Y. V. Makarov. Multiscenario cascading failure analysis using trelss. In *CIGRE/IEEE PES International Symposium Quality and Security of Electric Power Delivery Systems, 2003. CIGRE/PES 2003.*, pages 176–180, Oct 2003.

[33] Bing Huang, Mohammad Majidi, and Ross Baldick. Case study of power system cyber attack using cascading outage analysis model. *IEEE PES GM, Portland OR*, 2018.

[34] S. K. Khaitan, Chuan Fu, and J. McCalley. Fast parallelized algorithms for on-line extended-term dynamic cascading analysis. In *2009 IEEE/PES Power Systems Conference and Exposition*, pages 1–7, March 2009.

[35] Daniel S Kirschen, Dilan Jayaweera, Dusko P Nedic, and Ron N Allan. A probabilistic indicator of system stress. *IEEE Transactions on Power Systems*, 19(3):1650–1657, 2004.

[36] Prabha Kundur, Neal J Balu, and Mark G Lauby. *Power system stability and control*, volume 7. McGraw-hill New York, 1994.

[37] Prabha Kundur, John Paserba, Venkat Ajjarapu, Göran Andersson, Anjan Bose, Claudio Canizares, Nikos Hatziargyriou, David Hill, Alex Stankovic, Carson Taylor, et al. Definition and classification of power system stability. *IEEE transactions on Power Systems*, 19(2):1387–1401, 2004.

[38] Yao Liu, Peng Ning, and Michael K. Reiter. False data injection attacks against state estimation in electric power grids. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 21–32, New York, NY, USA, 2009. ACM.

[39] F. Xia M. Kumbale, T. Rusodimos and R. adapa. Trelss: A computer program for transmission reliability evaluation of large-scale systems. *User's Referecne Manual*, 2, 1997.

[40] Hong Tao Ma and Badrul H Chowdhury. Dynamic simulations of cascading failures. In *2006 38th North American Power Symposium*, pages 619–623. IEEE, 2006.

[41] Shengwei Mei, Yixin Ni, Gang Wang, and Shengyu Wu. A study of self-organized criticality of power system under cascading failures based on ac-opf with voltage stability margin. *IEEE Transactions on Power Systems*, 23(4):1719–1726, 2008.

[42] Amir-Hamed Mohsenian-Rad and Alberto Leon-Garcia. Distributed internet-based load altering attacks against smart power grids. *IEEE Transactions on Smart Grid*, 2(4):667–674, 2011.

[43] Muhammad Naveed, Xiao-yong Zhou, Soteris Demetriou, XiaoFeng Wang, and Carl A Gunter. Inside job: Understanding and mitigating the threat of external device mis-binding on android. In *NDSS*, 2014.

[44] Milorad Papic, Keith Bell, Yousu Chen, Ian Dobson, Louis Fonte, Enamul Haq, Paul Hines, Daniel Kirschen, Xiaochuan Luo, Stephen S Miller, et al. Survey of tools for risk assessment of cascading outages. In *2011 IEEE Power and Energy Society General Meeting*, pages 1–9. IEEE, 2011.

[45] M. Rahnamay-Naeini, Z. Wang, N. Ghani, A. Mammoli, and M. M. Hayat. Stochastic analysis of cascading-failure dynamics in power grids. *IEEE Transactions on Power Systems*, 29(4):1767–1779, July 2014.

[46] Eyal Ronen, Adi Shamir, Achi-Or Weingarten, and Colin O'Flynn. Iot goes nuclear: Creating a zigbee chain reaction. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 195–212. IEEE, 2017.

[47] Saleh Soltan, Prateek Mittal, and H Vincent Poor. Blackiot: Iot botnet of high wattage devices can disrupt the power grid. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 15–32, 2018.

[48] Siddharth Sridhar, Adam Hahn, and Manimaran Govindarasu. Cyber–physical system security for the electric power grid. *Proceedings of the IEEE*, 100(1):210–224, 2011.

[49] Nassim Nicholas Taleb. *The black swan: The impact of the highly improbable*, volume 2. Random house, 2007.

[50] Rui Tan, Varun Badrinath Krishna, David KY Yau, and Zbigniew Kalbarczyk. Impact of integrity attacks on real-time pricing in smart grids. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 439–450. ACM, 2013.

[51] Chee-Wooi Ten, Chen-Ching Liu, and Govindarasu Manimaran. Vulnerability assessment of cybersecurity for scada systems. *IEEE Transactions on Power Systems*, 23(4):1836–1846, 2008.

[52] Marianna Vaiman, Keith Bell, Yousu Chen, Badrul Chowdhury, Ian Dobson, Paul Hines, Milorad Papic, Stephen Miller, and Pei Zhang. Risk assessment of cascading outages: Methodologies and challenges. *IEEE Transactions on Power Systems*, 27(2):631, 2012.

[53] Yezhou Wang and Ross Baldick. Cascading outage analysis using sequential outage checkers. *Modeling, Simulation, And Optimization for the 21st Century Electric Power Grid*, 2013.

[54] Yezhou Wang and Ross Baldick. Case study of an improved cascading outage analysis model using outage checkers. In *Power and Energy Society General Meeting (PES), 2013 IEEE*, pages 1–5. IEEE, 2013.

[55] Yezhou Wang and Ross Baldick. Interdiction analysis of electric grids combining cascading outage and medium-term impacts. *IEEE Transactions on Power Systems*, 29(5):2160–2168, 2014.

[56] Yezhou Wang, Chen Chen, Jianhui Wang, and Ross Baldick. Research on resilience of power systems under natural disasters—a review. *IEEE Transactions on Power Systems*, 31(2):1604–1613, 2015.

[57] Jun Yan, Yufei Tang, Haibo He, and Yan Sun. Cascading failure analysis with dc power flow model and transient stability analysis. *IEEE Transactions on Power Systems*, 30(1):285–297, 2015.

Discovering and Understanding the Security Hazards in the Interactions between IoT Devices, Mobile Apps, and Clouds on Smart Home Platforms

Wei Zhou[1], Yan Jia[2,1], Yao Yao[2,1], Lipeng Zhu[2,1],
Le Guan[3], Yuhang Mao[2,1], Peng Liu[4] and Yuqing Zhang[1,2,5*]

[1]*National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences, China*
[2]*School of Cyber Engineering, Xidian University, China*
[3]*Department of Computer Science, University of Georgia, USA*
[4]*College of Information Sciences and Technology, The Pennsylvania State University, USA*
[5]*State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, China*

Abstract

A smart home connects tens of home devices to the Internet, where an IoT cloud runs various home automation applications. While bringing unprecedented convenience and accessibility, it also introduces various security hazards to users. Prior research studied smart home security from several aspects. However, we found that the complexity of the interactions among the participating entities (i.e., devices, IoT clouds, and mobile apps) has not yet been systematically investigated. In this work, we conducted an in-depth analysis of five widely-used smart home platforms. Combining firmware analysis, network traffic interception, and black-box testing, we reverse-engineered the details of the interactions among the participating entities. Based on the details, we inferred three legitimate state transition diagrams for the three entities, respectively. Using these state machines as a reference model, we identified a set of unexpected state transitions. To confirm and trigger the unexpected state transitions, we implemented a set of phantom devices to mimic a real device. By instructing the phantom devices to intervene in the normal entity-entity interactions, we have discovered several new vulnerabilities and a spectrum of attacks against real-world smart home platforms.

1 Introduction

With the development of the Internet of Things (IoT), smart home technology has become widely used in many applications including safety and security [33], home appliances [5], home healthcare [13], etc. According to Statista research, more than 45 million smart home devices were installed in 2018, and the annual growth rate of home automation is 22% [51]. To manage the ever-increasing number of diverse smart home devices in a consolidated way, many companies have proposed their smart home platforms (e.g., Samsung SmartThings [52]). With IoT clouds playing a central role in building smart homes, real-world smart home platforms essentially engage **three (kinds of) entities** that interact with each other: an IoT cloud, smart home devices and a mobile app. Briefly speaking, the mobile app provides users with an interface to facilitate the initial setup of devices including WiFi provision. After getting Internet access, each device negotiates its login credential(s) with the IoT cloud. In this way, it can build a connection with the IoT cloud to routinely report its status and execute the received remote control commands, which are usually generated by certain home automation applications running in the IoT cloud. At the same time, the mobile app is able to monitor and control each device through the IoT cloud.

While bringing substantial convenience to our lives, smart home technology also introduces potential security hazards. Since smart home devices directly process user-generated data, once compromised, they could introduce serious consequences. For example, user privacy can be harmed [28, 42]; property can be destructed [22, 38]; life safety and psychological health are also threatened [26, 54]. Imagine a smart home which is programmed in such a way that whenever the home temperature rises to a given threshold, the windows will be automatically opened. If an attacker obtains access to a smart heater, he could easily break into the home by keeping the heater at the highest temperature [21].

Although an increasing number of research studies have focused on smart home security, we found that existing research on the insecurity of interactions (e.g. inter-operations) in smart home platforms is still quite limited. First, the existing studies usually focus on individual parts of smart home platforms. For instance, there are studies disclosing the security problems with device firmware [44, 41, 28], communication protocols [14, 50, 27], and home automation applications [23, 40, 16]. Focusing on individual parts, the revealed vulnerabilities have little to do with the interactions among the three entities engaged in a smart home platform. Second, the existing studies seem to pay most attention to classic security issues such as privacy protection [24, 57], authentication [44, 41] and permission models [25, 23, 40], and leave the potential risks of the entity-entity interactions largely uninvestigated.

*Corresponding author: zhangyq@nipc.org.cn

Third, one kind of interaction in the Samsung SmartThings platform has recently been studied in [17, 16], finding that the interaction between multiple home automation applications (i.e. IoT apps) can lead to unsafe device states. While this finding is inspiring, other essential kinds of interactions in smart home platforms have not yet been (systematically) studied in the literature. In this paper, when we use the word "interactions", we are specifically referring to the inter-operations involved among the aforementioned three entities, with a focus on high-level pairing of devices, handshaking between IoT clouds and devices, etc.

To systematically discover and understand the security hazards in the interactions involved in smart home platforms, we have analyzed several widely-used smart home platforms and conducted the following investigations in this work. First, because all the communications among the three entities are encrypted, we combined several techniques including firmware reverse-engineering and man-in-the-middle (MITM) monitoring (to break SSL) to work out the details of the interactions among the three entities. Second, based on the interactions among the three entities, we inferred three legitimate state transition diagrams for the three entities, respectively. Using the inferred state machines as a reference model, we identified unexpected state transitions in several widely-used smart home platforms. Finally, to confirm and trigger unexpected state transitions, we implemented a *phantom* device that mimics a real device. A phantom device is essentially a computer program. By instructing the phantom device to intervene in the normal interactions among legitimate smart home devices, IoT clouds, and a mobile app, we have identified several new vulnerabilities and attacks in major smart home platforms.

In summary, the main contributions are as follows:

New insights are provided: (a) Real-world smart home platforms do not strictly guard the validity of the involved state transitions. For example, we found that an IoT cloud can accept some device requests without checking whether such a request should be allowed or not in its current state. (b) The three entities can sometimes stay in unexpected state combinations, which brings potential risk. (c) IoT clouds do not always perform adequate authorization checks on interaction requests. We found that an IoT cloud sometimes simply accepts and executes sensitive device-side commands without any permission checking. (d) By carefully constructing attacks that exploit a particular combination of the above security flaws, we showed that serious new security hazards can occur. This *new* finding proves that high risk attacks are rarely caused by a single factor. Accordingly, stake holders should conduct integrated insecurity analysis on interactions among the three entities.

New hazards are discovered: (a) An adversary can remotely replace a victim's real device with a non-existing phantom device under his control. As a result, all the control commands from the victim user are exposed to the phantom device and further to the adversary, leading to privacy breaches. The adversary can also leverage the phantom device to manipulate the data to be sent to the victim user, thus deceiving or misleading the victim user. (b) An adversary can remotely take over a device. As a result, he can harvest the sensor readings to monitor the victim's home or even manipulate the smart home devices, causing data breaches and endangering the victim. (c) An adversary can remotely unbind an authorised user through a phantom device. As a result, the user can no longer control the device with his account. (d) An adversary can leverage a phantom device to mislead an IoT cloud and occupy the identity of a real device before the device is sold. When a consumer buys the device, he cannot bind the device with his account. (e) An adversary can utilize a phantom device to automatically send update requests to an IoT cloud to steal various proprietary firmware on a large scale.

The newly discovered hazards have significantly enlarged the previously-known attack surface of smart home platforms; they also provide essential new understandings about the security and privacy hazards in smart homes.

Responsible Disclosure. All the vulnerabilities described in this paper have been reported to the corresponding vendors, and they have confirmed our disclosures. We have shared the technical details with the vendors. And most of the vulnerabilities have been fixed by them.

2 Background

2.1 Terminology

To make the presentation more clear, we first define several key terminologies.

Device ID. The *Device ID* of an IoT device uniquely identifies the device. Since device IDs are used to authenticate a device, they should be kept secret at all time. The attacks discovered in this study create fake IoT devices by occupying the device ID of a real victim device.

Identity Information. By "identity information," we mean the information items whose values are used to generate (i.e. calculate) a device ID. A typical use case of identity information is as follows: a device first provides the IoT cloud it belongs to with its identity information, then the cloud generates and returns the corresponding device ID to the device. Frequently used identity information includes MAC address and device model. Since device IDs should be kept secret, identity information should also be kept secret. Unfortunately, we found that this rarely holds in practice and attackers can easily obtain device identity information.

Legitimacy Information. By "legitimacy information," we mean the information items whose values are used to conduct certain legitimacy checking of a device, but are not used to

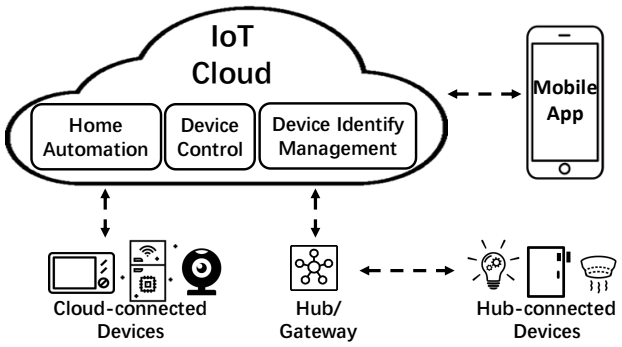


Figure 1: Smart Home Platform Architecture

generate any device IDs. We found that such information can also be easily acquired by hackers.

Phantom Devices. A phantom device is used by us to analyze the smart home interactions and to launch attacks. It is essentially a computer program that is instructed to intervene in the normal interactions among legitimate smart home devices, IoT clouds, and mobile apps.

2.2 Overview of Smart Home Platforms

The architecture shown in Figure 1 is widely adopted by major cloud-based smart home platforms. There are three key entities interacting with each other on a smart home platform: the IoT devices, the mobile app, and the IoT cloud.

The IoT cloud is the brain of a smart home platform. It is usually responsible for three kinds of services, denoted as Device Identify Management, Device Control, and Home Automation as shown in Figure 1. First, in order to ensure that only the device owner and delegated users have access to a device, the device identify management service needs to maintain a one-to-one mapping between the owner's account and the device. This binding happens at the time when the device is firstly deployed. As soon as the device is undeployed, the binding relationship should be revoked. Second, in order to allow authorized users to remotely control a device, the device control service serves as a "proxy" when users send remote commands to the device. Lastly, most smart home platforms provide home automation services, in which users can customize automation rules that define the interoperability behaviors of smart home devices. For example, a home owner can craft an automation rule that turns on the air conditioner if the indoor temperature goes above 70°F. When a thermometer detects that the temperature exceeds the specified threshold, it sends the event to an in-cloud home automation application, which then sends a command to turn on the air conditioner.

The second type of entities in a smart home platform are IoT devices. IoT devices are equipped with embedded sensors and actuators that interact with the physical world and send sensor readings to an IoT cloud. There are two typical mechanisms for devices to connect to an IoT cloud. (a) WiFi-enabled devices can connect to the Internet and thus directly communicate with the IoT cloud. We call these devices *cloud-connected devices*. (b) Energy-economic devices are not equipped with a WiFi interface to directly interact with the IoT cloud. Instead, they first connect to a hub/-gateway using energy-efficient protocols such as Z-Wave and ZigBee. Then the hub connects to the IoT cloud on behalf of the IoT devices. We call the devices connected to a hub as *hub-connected devices*. It is worth noting that the hub itself is one kind of cloud-connected device. Some platforms support both cloud-connected and hub-connected devices, while some only support one kind. The third kind of entities on a smart home platform are mobile apps. They provide users with an interface for device management (e.g., binding a device with its owner's account) and customization of the in-cloud home automation services.

Table 1: Two Types of Smart Home Platforms and Their Differences

	Platform	**Device Registration**	**Device Binding/Unbinding**
***Type I* Platform**	Alink, Joylink	*Device ID* Generated by **Cloud**	Authorization Checking Performed by **Cloud**
***Type II* Platform**	SmartThings KASA, MIJIA	Skipped	Authorization Checking Performed by **Device**

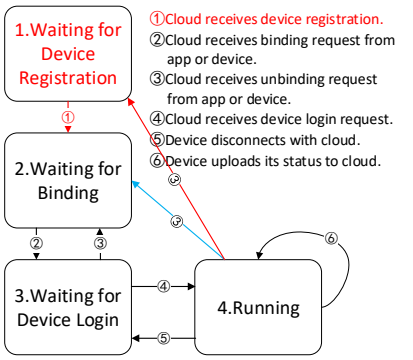
Deployment. To standardize and simplify the deployment of IoT services, smart home platform providers often provide collaborating partners with software development kits (SDKs). With SDKs, the adopting manufacturers only need to focus on device-specific initialization procedures and core application logic. Features such as over-the-air (OTA) update are also integrated in the SDKs.

2.3 Overview of the Interactions on Smart Home Platforms

In this section, we depict the interactions among the three entities during the life-cycle of a smart home device from the viewpoint of a consumer (rather than a manufacturer, supplier or retailer). To make the description more clear, the description focuses on cloud-connected devices. Hub-connected devices follow a similar model except that they use a hub as an intermediate node.

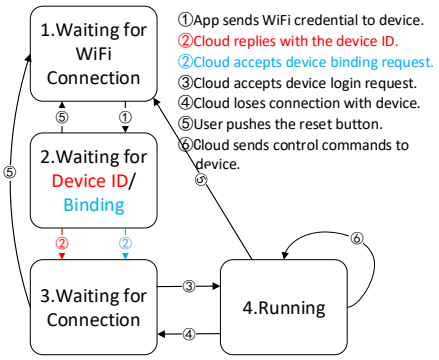
Type I vs. *Type II*: Depending on how the device ID of a device is generated, we classify the smart home platforms investigated in this work into two types. How a device ID is generated further influences device registration and device binding/unbinding. In Table 1, we list the investigated platforms for each type and the key differences between the two types. These differences lead to different attack preconditions in attacking a smart home platform.

In the following, we try to abstract an interaction model for *Type I* and *Type II* smart home platforms. When there is a difference between the two types, we also explicitly report it. In Appendix C, the complete interaction diagrams for all



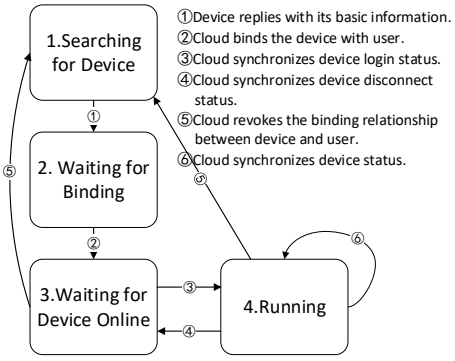
States	Description
Waiting for Device Registration	The cloud is in the initial state and waits for device registration request.
Waiting for Binding	The cloud waits for binding request from the mobile app or device.
Waiting for Device Login	After binding the device with user, the cloud waits for device login request.
Running	The cloud handles user's requests and syncs the device status.

(a) State Machine of an IoT Cloud



States	Description
Waiting for WiFi Connection	The device waits for discovery message and WiFi credential from app.
Waiting for Device ID/ Binding	The device sends register request/ binding request to the cloud, and waits for cloud response.
Waiting for Connection	Device keeps sending login request to get connection from cloud.
Running	Device keeps a connection with cloud and waits for new commands.

(b) State Machine of a Device



States	Description
Searching for Device	The mobile app keeps broadcasting discovery message to search for available device.
Waiting for Binding	The mobile app sends binding request to cloud or device, and waits for binding confirmation.
Waiting for Device Online	The mobile app waits for cloud to build connection with device.
Running	After device logs into the cloud, the mobile app receives user's instructions to the device via UI.

(c) State Machine of a Mobile App

Note: The states and transitions specific to *Type I* platforms are shown in red; the states and transitions specific to *Type II* platforms are shown in blue; and the shared states/transitions are in black.

Figure 2: High-Level State Machines of the Three Entities

the studied platforms are given.

1. Device Discovery: The life-cycle of a newly bought smart home device starts with device discovery. After the home owner (i.e. the customer) clicks the "Add Device" button on the designated mobile app, the app establishes a local connection with the device through broadcasting a discovery message or through the device's access point. Then, the device reports its basic information, such as its MAC address and device model, to the mobile app.

2. WiFi Provisioning: To access the Internet, the IoT device needs to join the same LAN as the mobile app. To obtain WiFi credentials, there are several mechanisms such as Access Point Mode [18], WiFi direct [4] and SmartConfig [29].

3. Device Registration: After device registration, the IoT device is given a unique device ID. Different types of platforms provide device IDs in different ways. For *Type I* platforms, the device sends its device identity information to the IoT cloud it belongs to. The cloud then generates a device ID and returns it to the device. The device then writes the device ID to its persistent storage. The cloud also keeps a record of the device ID for future authentication. For Type II platforms, the device's device ID is generated by the device's platform beforehand and hard-coded into the device during fabrication. Therefore, device registration is skipped.

4. Device Binding: The IoT cloud binds the device's device ID with the user account of the home owner. As a result, only authorized users can access the device via the cloud. The two types of smart home platforms adopt different device binding methods. For *Type I* platforms, the binding request is directly sent by the mobile app to the IoT cloud, which is responsible for maintaining the binding information and performing the permission checks (i.e., whether a user account should have access to a device). For *Type II* platforms, the mobile app first sends the account information to the device. The device, having the device ID and the user account, issues the binding request to the IoT cloud. It is worth noting that here the cloud unconditionally accepts the binding request from the device. This design is based on the natural assumption that the customer who physically owns a device should have full control over it.

5. Device Login: The device uses its device ID to log into the IoT cloud. Then it establishes a connection with the cloud to keep the status updated and ready itself to execute remote commands. In addition, when a device loses connection with the cloud for a long time, it tries to re-login automatically.

6. Device in Use: After successful registration and login, the device performs designed functions. Specifically, the home owner can monitor the real-time status of the device and ex-

plicitly control the device locally or remotely via the "control panel" on the mobile app.

7. Device Unbinding & Device Reset: When the home owner no longer uses the device, she can unbind or reset it. When the user requests for unbinding, for *Type I* platforms, the cloud directly erases the binding information. For *Type II* platforms, however, since the binding information is also stored on the device locally, one additional command is sent from the cloud to the device to erase the binding information.

In the life-cycle of an IoT device, although most of the time is spent in the sixth phase, i.e., the device-in-use phase, the interactions that occur during the other phases are the most complex and critical. Any oversight in these phases could lead to serious security problems and harm the normal use of devices.

2.4 State Transitions

In this subsection, we describe the state transitions inferred from our analysis of five widely-used smart home platforms (i.e., Samsung SmartThings [52], TP-LINK KASA [53], XiaoMi MIJIA [56], Ali Alink [2], and JD Joylink [31]). A smart home platform is a special kind of distributed system. In this viewpoint, the aforementioned interactions among IoT devices, mobile apps, and IoT clouds unavoidably cause state transitions. Based on our analysis of the aforementioned five platforms, we infer three state transition diagrams for the three entities, respectively. The three state machines are shown in Figure 2. In each sub-figure, an interaction with other entities (denoted by an edge) causes a state transition. The definitions for each state and each transition are annotated in the corresponding sub-figure. Note that *Type I* and *Type II* platforms behave slightly differently and we highlight the differences using different colors. Specifically, the states and transitions specific to *Type I* platforms are shown in red while the states and transitions specific to *Type II* platforms are shown in blue. The shared states and transitions are shown in black. In addition, since *Type II* platforms use hard-coded device ID, state 1 is absent in the state machine of an IoT cloud.

State Correlation. The three state machines are closely related to one another. Whenever an interaction takes place, the three entities as a whole may transfer from one 3-tuple state combination (i.e., the current state of the IoT cloud, the current state of the device, and the current state of the mobile app) to another 3-tuple. We identify all the legitimate 3-tuple state combinations and show them in the table presented in Appendix A. If a 3-tuple does not appear in the table, the corresponding state combination is illegal and might be exploited. To avoid the potential attacks, the three entities should always stay in a legitimate state combination. Unfortunately, we found that none of the investigated smart home platforms strictly maintain a three-entity state machine.

2.5 Scope of Empirical Vulnerability Analysis

Real-world cloud-based smart home platforms can be classified into two categories. The first category is the platforms *dedicated* to building a smart home (e.g., Samsung SmartThings [52]). The second category is general-purpose IoT platforms (e.g., Amazon Web Services IoT [6]) which could be customized for smart home usage. Since smart home platforms of the second category usually differ from each other in terms of device management and interaction, we leave studying common security issues with them as our future work. In addition, smart home platforms that are not cloud-based, such as HomeKit [8] and HomeAssistant [10], are out of the scope of this study, although we will discuss the implications of our research findings to platforms that are not cloud-based in Section 7.2.

In this work, we focus on five leading cloud-based smart home platforms. As mentioned earlier, they are SmartThings, KASA, MIJIA, Alink, and Joylink. To attract more cooperative manufacturers, some smart home platform providers such as Samsung, JD, and Ali make their platforms open and even open-source the corresponding device-side SDKs. Thus, the collaborative manufacturers can easily follow the documentation and assemble platform compliant devices through proper use of the SDKs. Over 200 well-known smart home device manufacturers (e.g., Philips, ECOVACS, and Media) are actually fabricating products running on these platforms [3, 30].

Some other cloud-based smart home platform providers, including TP-LINK and XiaoMi, adopt a closed "ecosystem". They fabricate smart home devices by themselves. On North America and Europe markets, TP-LINK's smart home devices, such as smart WiFi plugs and smart LED bulbs, rank in the top 10 in the category of home improvement on Amazon [19]. XiaoMi is the world's largest intelligent smart home device manufacturer. More than 85 million smart home devices have been sold under the brand of XiaoMi all over the world [34], especially in Asia-Pacific [43].

3 Threat Model and Feasibility Assessment

3.1 Threat Model

In contrast to network-based exploits (e.g., MITM) and firmware-based reverse-engineering, the adversary in our threat model seeks to exploit the design flaws in the interactions among the three entities. Therefore, we *do not* assume any forms of software bugs or protocol vulnerabilities. The targets of the attack are cloud-connected devices which directly communicate with IoT clouds. The adversary's goal is to take control of the device or to monitor/manipulate the data collected/generated by the device.

We *do* assume that the adversary has the capability to collect necessary information, including device *identity information* and *legitimacy information*. For different platforms, the difficulty levels of collecting these information items dif-

Table 2: Device Identity/Legitimacy Information

	Platform	Identity Info	Legitimacy Info
***Type I* Platform**	Alink	MAC (G), CID (P), Device Model (P)	Key (P), Sign (P)
	Joylink	MAC (G), SN (H), Device Model (P)	NULL
***Type II* Platform**	SmartThings	Device ID (H)	NULL
	KASA	Device ID (H)	MAC (G), hwID (P)
	MIJIA	Device ID (H)	TagKey (H)

P: Public information G: Guessable information H: Hard-coded information

fer. For example, for *Type II* platforms, we assume the adversary has local access to the victim device beforehand, whereas for *Type I* platforms, we do not have such an assumption. As a result, the discovered exploits may exhibit different levels of feasibility depending on which type of platform is being attacked, who is the platform provider, etc.

In the following, for both types of platforms, we analyze the feasibility of obtaining these information items case by case. The reason is that different platforms may designate and use individual information items in different ways. Correspondingly, the adversary faces different challenges in collecting them.

3.2 Prerequisites and Feasibility Assessment

In this subsection, we describe the specific identity and legitimacy information items the adversary has to obtain, and evaluate the feasibility of obtaining them in practice.

As mentioned earlier, a device is identified by a unique device ID. In essence, the discovered exploits fake a phantom device by using the victim device's device ID. Thus, the adversary needs to get the device ID of the victim device. For *Type I* platforms, given that the device ID is determined solely by the victim device's identity information, the adversary only needs to collect all the identity information.

For *Type II* platforms, the device ID is hard-coded in the victim device. So the adversary has to have local access to the victim device (e.g., connect to the same LAN or physically possess the device) to obtain the device ID. Although this seems to be a strong assumption, we note that once the hard-coded information is leaked, the victim device becomes remotely vulnerable forever.

Furthermore, some platforms use pre-configured legitimacy information (e.g., a key) as additional authentication requirements.

Depending on the way to obtain a particular identity/legitimacy information item, we classify these information items into three categories: public information (**P**), guessable information (**G**) and hard-coded information (**H**). For each platform investigated in this study, we list the needed identity/legitimacy information items plus their categories in Table 2. Note that the same information item may be used differently. For example, `MAC` addresses are used by Alink devices as identity information, but are used by KASA devices as legitimacy information.

Public information is the easiest to obtain. Information items in this category are often publicly available or can be easily inferred. For example, device model and device chip id (CID) are public information. Moreover, legitimacy information items in this category are sometimes not uniquely bound with a device but shared by multiple devices. For example, in the Alink platform, the legitimacy information is a tuple which consists of two "confidential" numerical strings, namely `Key` and `Sign`. We found that obtaining the tuple is extremely easy – a bunch of such credentials are available in the official GitHub repositories of both the Ali company[1] and the cooperative manufacturers[2].

Guessable information is the information which can be guessed by brute-force. MAC addresses are a typical kind of guessable information, because the first three bytes in a MAC address are usually fixed for a manufacturer [32]. Moreover, manufacturers often allocate a block of consecutive MAC addresses to the products of the same device model. Thus, there remains only two or three bytes for the attacker to brute-force. We detail an experiment on Alink devices in Section 6.1, in which we successfully guessed more than 7,181 valid MAC addresses. Moreover, if the adversary can be in the WiFi-range of a victim device, he can simply eavesdrop the MAC address of the device by sniffing wireless probe requests [37]. Note that this is a fundamental drawback of the WiFi protocol.

Hard-coded information is unpredictable, immutable, and inherent to a device. For example, a long device ID embedded in the device hardware is a typical kind of hard-coded information for *Type II* platforms. In addition, for some *Type I* platforms, hard-coded information (e.g., a serial number (SN)) is also incorporated in the generation of a device ID. Although hard-coded information cannot be obtained easily, it is immutable. Once this information is leaked, the victim device becomes vulnerable forever.

To get this information, the adversary needs local access to the victim device. For example, the adversary can get the device's hard-coded device ID by sniffing the device-app traffic in the device's LAN during the device discovery phase (Section 2.3). In case the adversary is using the device on behalf of the home owner, he can find the hard-coded device ID in a log maintained by the mobile app.

We now discuss the feasibility of physically accessing a victim device and the adversary's incentive to employ the discovered exploits. First, the ownership of a device can be changed if the device gets resold or decommissioned [36]. For example, increasingly popular smart home manufacturers such as Samsung and Apple provide certified refurbished devices on the on-line outlet stores or through Amazon. The

[1] `https://github.com/alibaba/AliOS-Things`

[2] `https://github.com/espressif/esp8266-alink-v1.0`

previous owner can easily extract the hard-coded credentials before re-selling the device. We have successfully obtained the device ID of a Samsung POWERbot R7040 Robot Vacuum in our experiments. If we resell or return this device, we can control this device remotely even after it is sold to another user.

Second, a recent study shows that 60% of guests would actually pay more for a vacation rental home with smart home features [15]. Thus, vacation rentals and hospitality service providers like Airbnb.com and Ziroom.com have been collaborating with smart home providers to equip an increasing number of smart home devices (e.g., smart locks, cameras and TVs) in their apartments and hotels [7]. For instance, JD has worked with Ziroom to deploy Joylink smart home devices in Ziroom rental rooms [47]. If the adversary "legitimately" rents a vacation home for one night and extracts the device ID of the home's smart lock, he could remotely open the lock at will in the future. This poses a serious threat to the safety of other tenants. We have successfully conducted such a remote hijacking attack against an Alink device (i.e., a KAADAS smart lock with model KDSLOCK001) in lab environment.

4 Analysis Methodology

The discovered exploits leverage a set of design flaws in the interactions among the three entities. This section elaborates the vulnerability analysis methodology we used to identify these design flaws.

4.1 Deciphering Communication

To protect user privacy, smart home platforms usually encrypt the communication among IoT devices, mobile apps, and IoT clouds. We must decrypt the communication traffic before we can study the interactions. This imposes significant challenge for us because some platforms are close-sourced (e.g., XiaoMi and TP-LINK).

Cloud-App Communication. A simple network sniffer confirms that most platforms adopt TLS, and mobile apps are required to verify the validity of the cloud (server) certificate. The MITM attack on the mobile app side is an obvious choice in deciphering the communication. However, to launch a MITM attack, we must replace the cloud certificate with one controlled by us. After analyzing a number of mobile apps, we found that a mobile app usually hard-codes the cloud certificate in its APK file without relying on the trust store provided by Android [39] (a.k.a. certificate pinning). If we replace the hard-coded certificate in an APK, the corresponding app would fail to run due to integrity checking. We have addressed this problem by rooting our test smartphone and installing an Xposed module[3]. This Xposed module is able to hook the certificate checking function so that we can dynamically manipulate the certificate without compromising the app integrity.

Device-App Communication. We have analyzed the APK files of a number of mobile apps and found that some platforms such as Joylink and MIJIA use a symmetric encryption algorithm to protect device-app communication. Thus, we can easily extract the communication keys by analyzing the APK files. Other platforms such as SmartThings adopt TLS for device-app communication. To deal with TLS, we have used the same method as used for deciphering cloud-app communication.

Device-Cloud Communication. Again, device-cloud communication is protected by TLS. However, we cannot easily replace the cloud certificate embedded in the firmware on a device to launch a MITM attack as is done in mobile apps. We had to perform static analysis on the device firmware. In particular, we have physically dumped the firmware images of the target devices[4]. We have manually followed the data and control flows of the cryptographic functions, and were able to locate the hard-coded certificates in the firmware images. We then replaced the hard-coded certificates with a set of certificates forged by us. However, we found that the devices enforce firmware integrity verification which denies executing any manipulated firmware. Fortunately, simple reverse engineering confirms that most devices only use the simple cyclic redundancy check (CRC) algorithm to check integrity. Therefore, we updated the CRC values to match the manipulated firmware and successfully booted the firmware images with the forged certificates. As a result, we were able to launch the MITM attack to decrypt the communication.

```
{"system": {
  "alink": "1.0", "jsonrpc": "2.0", "lang": "en",
  "sign": "3a07945eb6f453e6c0a4032c1184cc87",
  "key": "5gPFl8G4GyFZ1fPWk20m", "time": ""
 },
 "request": {
  "cid": "00000000000000000010671484", "uuid": ""
 },
 "method": "registerDevice",
 "params": {
  "model": "JIKONG_LIVING_OUTLET_00003",
  "mac": "60:01:94:A2:D5:7C","
  "version": "0.0.0;APP2.0;OTA1.0"
 },
 "id": 100
}
```

Code Listing 1: JSON Representation of Alink Device Registration Message

```
{"result": {
  "code": 1000, "msg": "success",
  "data": {
    "uuid": "D66FCB11A731CA2683A6C0DED6CD326D"
  }
 },
 "id": 100
}
```

Code Listing 2: JSON Representation of Cloud-Side Response to Alink Device Registration Message

[3] https://github.com/Fuzion24/JustTrustMe

[4] https://www.youtube.com/watch?v=KlV3_HaBpbs

4.2 Understanding the Interacting Messages

Using the aforementioned approaches, we were able to reveal plain-text network traffic among IoT devices, mobile apps, and clouds. This greatly simplified our analysis. Although different platforms adopt different communication protocols, it is a common practice that messages are encoded using the JSON (JavaScript Object Notation) format, which is quite self-explanatory. For instance, we show a message sent from a device to the cloud of the Alink platform in Listing 1. As indicated by the *method* field, this message is used to register the device. The device legitimacy information being sent includes `Sign` and `Key`. The device identity information being sent includes `CID`, `model`, and `MAC`. The respond message is shown in Listing 2. As we can see, the device ID is returned in the `uuid` field.

4.3 Phantom Devices

We investigated the interactions from three aspects. First, we tested whether each entity strictly maintains its state machine, which means an entity should only accept interacting requests acceptable in its current state. For example, as shown in Figure 2a, when an IoT cloud is working in state 4, it should deny the request from a device to bind itself to another user account. Second, we tested whether the three entities always stay in a legitimate 3-tuple state combination (see Appendix A). Third, we adjusted the parameters, especially those used in authentication, of the normal interacting requests and observed the responses. Our goals is to discover whether the receiving entity of each request conducts proper authorization checking.

To make this happen, we need to be able to craft JSON messages and send them to the receiving entities. However, we cannot arbitrarily change the requests of a real device. To cross this barrier, we created and ran a phantom device (program) that mimics a real device to assist our analysis. A phantom device is constructed as follows. Some smart home platform providers like Samsung, JD and Ali open-source their device-side SDKs and demo programs, which include the same communication logic as a real product. We simply reused them to build our phantom devices. On the other hand, XiaoMi and TP-LINK use proprietary SDKs, and we had to reverse-engineer the firmware we obtained from real devices, and implement programs to imitate the original communication functions.

With the help of phantom devices, we could arbitrarily adjust the parameters of request messages. In this way, we could trigger unexpected state transitions and manipulate/remove the authentication fields of a request to perform black-box testing against an IoT cloud. We shortly report our findings in Section 5.

The phantom devices not only facilitated interaction analysis, but also helped us figure out the relevant internal logic of an IoT cloud, which was completely opaque to us. For instance, we used a phantom device to test and confirm which device identity information is used in the generation of a device ID. Specifically, for the Alink platform, we changed the value of each field appeared in Listing 1 and recorded the corresponding returned device IDs from cloud. Finally, we compared the received device IDs to infer which fields influence the generation of a device ID. We concluded that the fields `model`, `MAC` and `CID` uniquely determine a device ID. In other words, there is a one-to-one mapping between the (`model`, `MAC`, and `CID`) tuples and device IDs in the Alink platform. We used a similar method to test all the studied platforms and the results are summarized in Table 2.

Table 3: Device Identity/Legitimacy Information

	Platform	Identified Flaws	Exploited Flaws	Applicable Attacks
***Type I* Platform**	Alink	F1.1, F1.3 F2, F3, F4	F1.1, F1.3, F2, F3, F4 F1.1, F1.3, F3 F1.1, F1.3, F3, F4 F1.1	Remote Device Hijacking Remote Device Substitution Remote Device DoS Illegal Device Occupation
	Joylink*	F1.1, F1.3 F2, F3	F1.1, F1.3 F3 F1.1	Remote Device Substitution Illegal Device Occupation
***Type II* Platform**	KASA	F1.2, F1.3, F3	F1.2, F3 F1.3, F3 F1.2	Remote Device Hijacking Remote Device Substitution Remote Device DoS
	MIJIA	F1.2, F1.3, F3	F1.2, F3 F1.3, F3 F1.2	Remote Device Hijacking Remote Device Substitution Remote Device DoS
	SmartThings†	F1.2, F1.3	F1.2	Remote Device DoS

*: Joylink platform does not support device-side unbinding request.
†: SmartThings cloud performs authorization checking on device login request.

5 Identified Design Flaws

Using the analysis methodology presented in Section 4, we have discovered four kinds of design flaws, and we have shared them with the providers of the five platforms we investigate. These design flaws are summarized in Table 3.

F1: Insufficient State Guard. We found that none of the three entities correctly guard their state machines. This could lead to severe consequences. Since IoT clouds are responsible for security-critical services such as device identify management, IoT clouds can be most affected. In the state machine of an IoT cloud (Figure 2a), when the cloud is working in state 4 (running), ideally it should only accept status upload requests (edge 6) or device unbinding requests (edge 3). Unfortunately, we found that the IoT cloud also accepts other requests. Depending on which request is accepted incorrectly when the IoT cloud is in state 4, we break down flaw F1 into three sub-flaws.

F1.1: This flaw is specific to *Type I* platforms. An attacker, having all the device identify information, can use a phantom device to send a registration request to the cloud, which is fooled to return the corresponding device ID to the attacker (Figure 3F1.1).

F1.2: This flaw is specific to *Type II* platforms. An attacker can use a phantom device to send a binding request that links the device (identified by device ID) with the attacker's account (Figure 3F1.2). Note that in *Type II* plat-

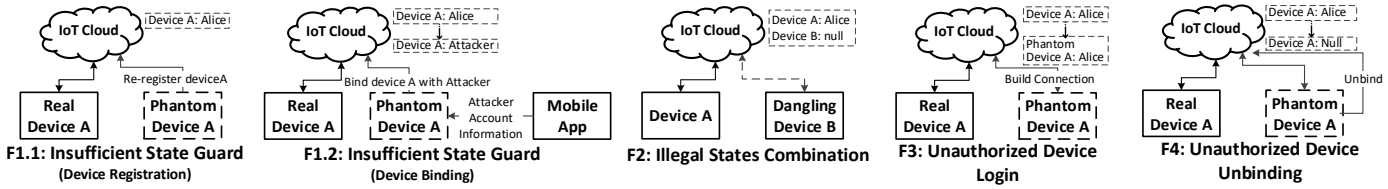


Figure 3: Identified Design Flaws

forms, the binding request is sent from the device and the cloud unconditionally accepts the binding request (see Section 2.3). As a result, the phantom device can bind the attacker's account to the victim device.

F1.3: The IoT cloud accepts device login requests even if it is in state 4. This flaw is a pre-condition of flaw F3 which we will describe shortly.

F2: Illegal State Combination. We found that the three entities sometimes stay in unexpected and illegal state combinations. One root cause is flawed synchronization among them. When an illegal state combination is exploited, security can be violated. For instance, ideally, when a user retires a smart home device, he should reset and unbind the device, making all of the three entities go back to their initial states (i.e., state combination (S1, S1, S1)). However, for *Type I* platforms, if the user unbinds the device without firstly resetting it, a connection with the cloud is still retained and the state combination actually switches to (S1, S4, S1), which is illegal based on our table in Appendix A. Since the device is in this illegal state combination, we call it a dangling device (Figure 3F2). Now, since the IoT cloud is in state 1, if the attacker remotely issues a request to register this device, the request is allowed and the cloud transfers to state 2. For the same reason, if the attacker continues to send a request to bind the device, the cloud accepts the request and transfers to state 4 (state 3 is skipped because a connection is still maintained with the victim device). At this time, if the attacker sends a control command to the device, the cloud will mistakenly forward the command to the retired device. This essentially causes a device hijacking attack.

F3: Unauthorized Device Login. A connection is maintained between the device and the IoT cloud after device login. Ideally, the cloud should only allow a login request if the request is issued from the device that is bound with the owner account. However, we found that the IoT cloud does not perform any account-based authorization check during device login. In other words, the connection is decoupled from the user account. Consequently, when the attacker uses a phantom device to login with the device ID of the victim device, the cloud is fooled into establishing a connection with the phantom device (Figure 3F3). As a necessary condition of this flaw, the cloud must accept device login requests in state 4, which is exactly what Flaw f1.3 states.

F4: Unauthorized Device Unbinding. Ideally, only the user who holds an account currently bound to a device has the privilege to unbind the device. This is true if unbinding operations are conducted on mobile apps, which indeed include user credentials in unbinding messages. Unfortunately, for *Type I* platforms, device unbinding can also be achieved on the device side. Based on our analysis, device-side unbinding commands do not included any user credentials. As a consequence, an attacker can build a phantom device to forge an unbinding request using device-side API. The binding relationship between victim user's account and the device is then revoked without the user's awareness (Figure 3F4).

6 Flaw Exploitation

Exploiting various combinations of the identified design flaws, an attacker can launch a spectrum of attacks, including remote device substitution, remote device hijacking, remote device DoS, illegal device occupation, and firmware theft. In the following, we first describe the experimental setup. Then we detail two most severe attacks revealed in this paper – how to remotely substitute and hijack a victim device, respectively. We also discuss the other attacks. In Table 3, we summarize the set of particular attacks, as well as the design flaws exploited by each attack. To visually demonstrate some of the discovered exploits, we also recorded two videos[5][6].

6.1 Experimental Setup

All the PoC (Proof of Concept) attacks were conducted within lab environment without influencing legitimate users. The tested devices include smart plugs, IP cameras, WiFi bulbs, cleaning robots and smart gateways, covering all the studied platforms. These devices are shown in Figure 4 and we also list them in Appendix B.

Obtaining Device Identity and Legitimate Information. We need device identity and legitimacy information listed in Table 2 to forge a *phantom* device. As mentioned earlier, the difficulty of obtaining an information item differs. In the following, we use an Alink device (Philips smart plug with model SPS9011A) and a TP-LINK device (WiFi Bulb with model LB110) to represent *Type I* and *Type II* devices, respectively. We describe how to obtain their identity and legitimacy information. For other platforms, similar approaches can be adopted.

[5] https://youtu.be/MayExk_PKhs
[6] https://youtu.be/fufEAtQq2_g



Figure 4: Smart Home Devices Used in Our Experiments

For the Alink device, we collected its legitimacy information (`Key` and `Sign`) very easily through the public repositories mentioned in Section 3.2. The Alink device uses `model`, `MAC`, and `CID` as its identity information. For `model` and `CID`, which are fixed for a specific device type, we extracted them from a log maintained by the corresponding mobile app. For the MAC address, we obtained it by brute-force attack. Specifically, we bought two test devices (Philips smart-plugs) and recorded the MAC address of the first one ("3C:2C:94:0B:56:69") manually. The first device served as the attacker's device in our experimental setting. Then we tried to guess the MAC address of the other device, which played the "victim device" role in our experiments. After 21,692 mutations to "3C:2C:94:0B:56:69", we "hit" the victim device's MAC address, which turned out to be "3C:2C:94:0B:AB:25".

To further demonstrate that collecting MAC addresses of real devices is not very difficult, we mutated the lower two bytes of "3C:2C:94:0B:56:69" to get 65,536 different MAC addresses. Then we used these MAC addresses with other fixed identity/legitimacy information to register the victim device. Every registration request returned us a device ID. We further wrote a probing script running on the mobile to automatically bind the returned device IDs to our test user account. If a device ID has already been bound to an account, the cloud will refuse the corresponding binding request with an error message. We found that 7,181 device IDs had already been taken by real users. It means the corresponding 7,181 MAC addresses were actively being used by real devices and they could potentially become victims in our attacks.

Ethical Consideration. In order to prevent potential influence on legitimate users, before we ran the script that tests active MAC addresses, we first evaluated the impact on a test device in lab environment. Specifically, we first simulated a legitimate user by normally binding a tested device to an account. Then we used a phantom device with the same identity and legitimacy information as the test device and got the same device ID from the cloud. We then used another account to try to bind the device as was done by the probing script. During this process, we found that the test device could still be operated normally and was not unbound.

For the TP-LINK device, since its identity information (i.e., device ID) and legitimacy information (i.e., `hwid` and MAC address) are hard-coded, we had to physically extract them. Specifically, we collected device IDs, `hwid` and MAC address by launching a MITM attack that intercepts the device-app communication.

Building Phantom Devices. With all the required information available, we implemented phantom devices using Python. As discussed in Section 4.3, when the device-side SDK is available (e.g., on platforms provided by Samsung, JD and Ali), we directly incorporated them into our program. Otherwise, our Python program mimics the behavior of a device. The behavior knowledge is obtained by reverse-engineering the firmware extracted from real devices (XiaoMi and TP-LINK). In total, we implemented five kinds of phantom devices for Samsung, Joylink, Alink, XiaoMi and TP-LINK, respectively. They were implemented by 22, 17, 14, 61 and 72 lines of code (excluding SDK functions if any), respectively.

Network Configuration. We placed the target devices and the phantom devices in two separate LANs behind NAT-enabled routers. As a result, the target devices and the phantom devices cannot communicate with each other directly. This setting resembles real-world scenarios where a remote attacker does not have access to the LAN of the victim.

6.2 Remote Device Substitution

In this subsection, we showcase how an attacker can remotely replace the victim's device with a phantom device under his control. To simplify the presentation, in the following, we use *Alice* to denote the victim/legitimate user and *Trudy* to denote the attacker. We use sequence diagrams to represent the interactions among the three entities. An attack happens when *Trudy* interferes with the normal interaction diagrams. In Appendix C, we show the normal interaction diagrams for all the studied platforms.

Attack Workflow (*Type I*). On the top of Figure 5 (above the highest dashed red line), we show the normal workflow of how *Alice* uses her IoT device on a Type I platform. After *Alice* provisions the device with a WiFi credential, the device sends its legitimacy credential and device identity information to the cloud to get registered (Step A.1). Based on the device identity information, the cloud registers the device with a device ID $\mathcal{A}$ and binds it to *Alice*'s account (Step A.2). After the device logs in (Step A.3), *Alice* can control the device with her account.

Then, the attacker, *Trudy*, kicks in as shown in the middle of the figure (between the two dashed red lines). She first lets the phantom device send the same device registration request as used in step A.2 to the cloud (Step T.1). Due to F1.1, the cloud accepts this request and registers the phantom device

with the same device ID $\mathcal{A}$, but still keeps device ID $\mathcal{A}$ bound with *Alice*. At this moment, *Alice* actually binds the phantom device and real device at the same time. Then *Trudy* could leverage the flaw F1.3 and F3 to log in a phantom device without *Alice*'s account information (Step T.2). Since the phantom device has the same device ID as the real device, the cloud disconnects the original connection with the real device and establishes a new connection with the phantom device. However, when the real device does not receive the heartbeat message for a while, it automatically logs into the cloud again and puts the phantom device offline. Now, the real device and the phantom device are in fact competing for connection with the cloud. To win the competition, *Trudy* configures the phantom device to login very frequently. As a result, the phantom device always wins. *Alice* now still appears to "control" a device through her mobile app, although this device has actually been replaced by the phantom device under the control of *Trudy*.

Attack Workflow (*Type II*). Similarly, the top part of Figure 6 (above the highest dashed red line) shows the normal workflow of how *Alice* uses her device on a *Type II* platform. After her mobile app sends her account information to the device (Step A.1), the device sends the binding request with device ID and legitimacy information, as well as the account information to the cloud (Step A.2). The cloud binds the device ID $\mathcal{A}$ to *Alice*'s account. After the device logs in the cloud (Step A.3), *Alice* can control/monitor the device with her mobile app.

In the middle of the figure (between the two dashed red lines), *Trudy* launches the remote device substitution attack. Enabled by flaws F1.3 and F3, she lets the phantom device successfully log into the cloud with the same device ID (Step T.1). At this time, the device ID $\mathcal{A}$ is still bound to *Alice*'s account. Like in the *Type I* platform, the phantom device maintains a connection with the cloud by periodically logging in. In this way, the attacker secretly substitutes *Alice*'s device with a phantom device under her own control.

Attack Consequence: Privacy Breaches. In normal operations, when *Alice* uses her mobile app to send a remote control command to the cloud, the cloud forwards the command to the "device". Unfortunately, in the remote substitution attack, the real device has been replaced by a phantom device controlled by the attacker. As a result, all the control commands from *Alice* are exposed to the phantom device and further to *Trudy*, leading to a privacy breach. For example, if *Trudy* substitutes a smart plug, he could know when *Alice* turns on/off the smart plug. This information could be used to infer whether *Alice* is at home.

Attack Consequence: Falsified Data. In normal operations, the real device updates its sensor readings to the cloud and the result is reflected in *Alice*'s mobile app. Unfortunately, in the remote substitution attack, the sensor readings are sent from the phantom device. This gives *Trudy* an opportunity to manipulate the sensor readings sent to *Alice*, thus deceiving or misleading *Alice*. For example, we tested a XiaoMi smoke alarm (model: Fire Alarm Detector) and a Alink smart lock (model: KAADAS KDSLOCK001). If the smoke alarm detects a thick smoke in the room, the smart lock will be unlocked automatically to open the window/-door. We used remote device substitution attack to manipulate the sensor readings of smoke alarm and successfully unlocked the smart lock. This leads to serious consequence because *Trudy* can enter *Alice*'s room at will.

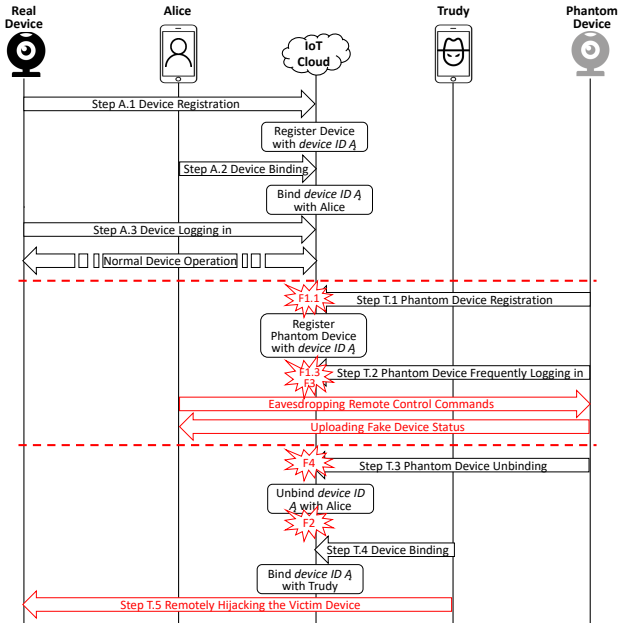


Figure 5: Remote Attacks on *Type I* Platforms

Our attack can also serve as a trigger for the flaws mentioned in previous works [17, 35]. Once a less-protected device is substituted by a phantom device and the device is in the chain of a "routine", the phantom device can further influence other data sensitive devices. For example, as mentioned in [35], the Nest Cam monitor in a house will automatically switch off when the global "away/home" state changes from "away" to "home". *Trudy* can take advantage of the substitution attack to change this state variable to disable the camera and burglarize the house without being recorded.

Stealthiness Analysis. Remote device substitution attack is highly stealthy. This is because *Alice* always sees the device to be online in her smartphone (although it is a phantom device). However, if *Trudy* feeds the phantom device with sensor readings that excessively deviate from normal, *Alice* (if she is security-savvy) might become suspicious of the dramatic change.

6.3 Remote Device Hijacking

Trudy can further remotely control *Alice*'s device by exploiting more flaws. We call this remote device hijacking attack.

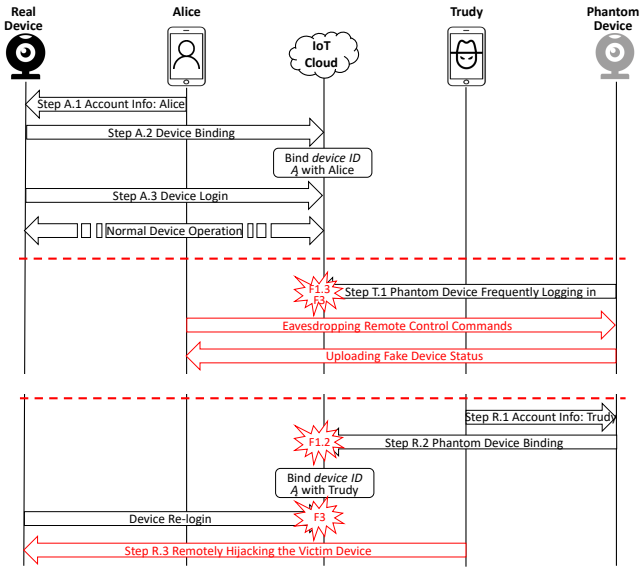


Figure 6: Remote Attacks on *Type II* Platforms

Attack Workflow (*Type I*). Continuing from Step T.2, device hijacking attack is depicted at the bottom of Figure 5 (below the lowest dashed red line). At this moment, the phantom device has already logged in the cloud. Due to flaw 4, *Trudy* is able to send a device-side unbinding request via her phantom device to the cloud (Step T.3), which puts the real device in dangling status (due to F2). Finally, *Trudy* binds the device with her account (Step T.4). As a consequence, *Alice*'s device is connected with the cloud whereas *Trudy* is able to control the device on her smartphone.

Attack Workflow (*Type II*). Unlike *Type I* platforms, to carry out a hijacking attack against *Type II* platforms, *Trudy* does not need to continue from the success of a remote substitution attack. Instead, she starts attacking from scratch, as depicted at the bottom part of Figure 6 (below the lowest dashed red line). *Trudy* starts by using her mobile app to send her account information to the phantom device (Step R.1). Next, the phantom device, which has *Alice*'s device and other legitimacy information, will send a binding request with *Trudy*'s account information (Step R.2) to the cloud. Due to F1.2, the cloud is fooled into accepting the binding request. Now, the ownership of the device has changed to *Trudy* from the view point of the cloud. The cloud then terminates the connection from the real device. However, the real device has re-connecting mechanism that continuously restores lost connection to the cloud. Due to F3, the cloud does not verify whether the account information (*Alice*) matches the current device owner (*Trudy*) or not. Therefore, the re-connecting request from the real device can be successful. At this point, *Trudy* could completely control *Alice*'s device (Step R.3).

Attack Consequence. The remote device hijacking attack allows *Trudy* to bind her account to *Alice*'s device. As a result, she can harvest the sensor readings in *Alice*'s home. She can also send remote commands to control *Alice*'s device. In our experiment, we successfully hijacked a Alink IP camera (model: RIWYTH RW-821S-ALY). As a result, we can view the victim's IP video feeds secretly, greatly threatening victim's privacy.

Stealthiness Analysis. Since *Alice*'s device has been hijacked, she can no longer talk to her device. It could raise an alert for *Alice* if she is security-savvy. Average Joe may simply regard it as a service failure and rebind his user account. It is worth mentioning that even for security-savvy users, it is not easy to trace back to the attacker. Only the IoT cloud has some clues to trace back to the attack origin.

6.4 Other Security Hazards

6.4.1 Remote Device DoS

As a basic security measure, IoT clouds only allows authorized users to control a device. If an attacker can unbind a target device from its legitimate user, the target device cannot be operated anymore, essentially leading to device denial of service (DoS) attack. To launch this attack, the attacker does not need to exploit many flaws. In particular, for *Type I* platforms, after the attacker sends the device-side unbinding command (Step T.3), as shown in Figure 5, the cloud directly revokes the binding relationship between the victim user and the device. For *Type II* platforms, as shown in Figure 6, after the attacker leverages flaw F1.2 to bind a phantom device with his account (Step R.2), the target device is unbound.

Note that since remote device DoS attacks require less flaws, the attack is applicable to more platforms. For instance, the Samsung SmartThings platform, which is immune to remote device substitution/hijacking attacks, is vulnerable to remote device DoS attacks. This is because the SmartThings platform is not subject to F3 which is essential for remote device substitution/hijacking attacks. However, F3 is not required in remote device DoS attacks. We will explain why SmartThings is not subject to F3 in Section 7.4.2.

6.4.2 Illegal Device Occupation

Although a device may be shared with multiple users, only one user account is allowed to be bound to a smart home device. If the attacker can predict the device IDs of unsold devices and use phantom devices to bind them with valid user accounts, these devices cannot be bound again after being sold. We call this attack illegal device occupation. In essence, this attack makes new devices unavailable to legitimate consumers. Note that this attack only applies to *Type I* platforms since attackers can predict device identity information. In *Type II* platforms, long and unpredictable device IDs are hard-coded in devices. Attacks can no longer learn anything about device IDs until the device is sold.

6.4.3 Firmware Theft

To protect intellectual property (IP), most IoT manufacturers employ certain tamper-resistant techniques to protect their products, including enforcing read-only property on flash chips that store proprietary firmware. However, with leaked-out firmware, the attacker can reverse-engineer it, causing IP theft and harming the corresponding manufacturers. By exploiting OTA updates available on most IoT devices, our firmware theft attack is able to bypass the aforementioned protections. By forging different kinds of phantom devices, the attacker can issue OTA update requests in bulk, and thus he can harvest hundreds of firmware images in seconds.

In the experiments, we created phantom devices to "emulate" 1,355 kinds of Alink devices, 543 kinds of Joylink devices, 118 kinds of XiaoMi devices, 23 kinds of SmartThings devices, and 18 kinds of TP-LINK devices. Eventually, we were able to collect 63 firmware images from the Alink platform, 37 from the Joylink platform, 89 from the MIJIA platform, and 16 from the KASA platform.

7 Discussion

Although we only studied five popular cloud-based smart home platforms, our findings can be generalized to other smart home devices (e.g., hub-connected devices), and have implications to platforms that are non-cloud-based. We also discuss the root causes of the identified vulnerabilities, and suggest several potential defensive approaches to mitigating the discovered exploits. Finally, we discuss the impact of the discovered exploits on commercial competitions.

7.1 Impact on Hub-Connected Devices

In this paper, we focus on cloud-connected devices. However, the discovered exploits are also applicable to hub-connected devices due to two reasons.

First, the attacker can leverage an already exploited hub to control hub-connected devices. For example, a XiaoMi smart gateway is a hub for MIJIA products. After hijacking the gateway, the attacker can further control all its connected devices. Note that Samsung SmartThings is not vulnerable to this attack because a SmartThings hub unbinds all the connected devices when the ownership of the hub is changed, which is inevitable in this attack.

Second, by forging a phantom hub-connected device, it is possible to launch firmware theft and illegal device occupation attacks. However, since the target devices are behind a hub, it is impossible to remotely hijack or substitute them.

7.2 Implications to Cloud-Free Smart Home Platforms

HomeKit. HomeKit [8] is Apple's proprietary smart home platform. Compatible devices run the HomeKit accessory protocol to directly talk to mobile apps via WiFi or Bluetooth. Moreover, using the mobile app, users can access home devices indirectly through a hub device (Apple TV, HomePod or even iPad). In this case, the cloud relays commands from the mobile app to the Apple TV or HomePod. Then the Apple TV or HomePod issue commands to home devices locally. Note outside the home LAN, there is no direct link to home devices [49]. As a result, our attacks are not applicable to HomeKit devices.

DIY Platforms. DIY smart home platforms such as Home Assistant [10] and OpenHAB [48] are open-source projects that focus on building local home automation. Due to controllable privacy and low cost (as low as the price of a Raspberry Pi), they are becoming more and more popular among DIY enthusiasts. In essence, they build private hubs that interact with different home devices. To support as many devices as possible, these platforms can be extended with components, which implement device specific logic [12]. While some devices can work by connecting to the hub locally (e.g., Philips Hue), others cannot work without relying their own cloud backends. To this end, Home Assistant classifies the smart devices into two types: devices that interact with third-party clouds, and devices that respond to events that happen within Home Assistant. For the former type, the hub serves as a proxy of other third-party clouds. For example, if a user wants to use a SmartThings device through Home Assistant, he first registers and binds the device with the SmartThings cloud. Then he needs to install a SmartThings plugin for Home Assistant to connect the device to the hub [11]. The plugin stores the user's SmartThings account token and delegates all the device requests to the SmartThings cloud.

For smart home devices relying on their own cloud backends, Home Assistant is actually transparent to the devices and thus all the exploits discovered in this paper can be applied to them. However, our attacks cannot influence devices that only work locally.

7.3 Root Cause Analysis

Some of the vulnerabilities revealed in this paper are associated with inherent design flaws of smart home platforms. Some are themselves caused by design challenges of smart homes. Therefore, some of security flaws cannot be remedied in a straightforward way.

Ownership Transfer. A natural assumption that smart home manufacturers make is that a user who physically owns a device should have full control over it. Thus, in *Type II* platforms, each device takes charge of authorization checking and sending device binding requests. This allows a legitimate user to rebind a device with another account by physically resetting it. Note that the rebind operation unbinds the previous account automatically and happens regardless of whether the device has already been unbound or not. This design directly leads to flaw F1.2, allowing an attacker to use a phantom device to remotely unbind the original user.

Device Reconnection. Network congestion may cause ran-

dom connection loss between a device and an IoT cloud. The IoT cloud mitigates the problem by allowing the device to re-login itself automatically. However, at the time of re-login, the cloud does not do any account-based authentication checking on all the platforms we have investigated except for SmartThings. This design gives raise to F3 which allows an attacker to remotely control the device without user awareness.

Cloud-Device State Inconsistency. To avoid problematic state transitions, an IoT cloud should be aware of the status of the devices it manages. Unfortunately, this is very hard to achieve in practice. In the previous work, it has shown that 22 of 24 studied devices suffer from design flaws that lead to state inconsistency [45]. For one thing, intermittent network conditions make it very difficult to keep the state of a device and the state of the IoT cloud synchronized at all time. For another, a user may reset a device by pushing the physical reset button when the device's Internet connection is lost. As a result, the device binding information is cleared on the device but not in the cloud. In all cases, the synchronization between the cloud and the device is broken. This causes flaws F1.1, F1.2, F1.3, and F2.

7.4 Mitigation

In this section, we propose several defensive design suggestions to secure smart home platforms in the first place. It should be noticed that adopting only a subset of our suggestions is not enough, because the flaws involved in the interactions are multi-faceted and tangled together (e.g., F1.3 and F3). Platform providers should consider all the potential security issues introduced by the interactions, including authentication, authorization and validity of working state machines.

7.4.1 Strict Device Authentication

We have clearly shown that existing authentication is not adequate. By violating authentication, a phantom device is indistinguishable from a victim device. To ensure that every device an IoT cloud talks to is a genuine device, we suggest that the manufacturers embed a unique client certificate into each device for high-end devices powered by Intel or ARM Cortex-A processors. In addition, the IoT cloud should always examine the client certificate before accepting any device request. For resource-restricted devices powered by a microcontroller, the manufacturers should embed a read-only random number into each device. On the cloud side, the cloud should always check whether the random number matches the other identity or legitimacy information.

Because device IDs are used by IoT clouds to identify a device, we also suggest that platform providers retrofit the device ID provisioning mechanism so that the attacker cannot easily obtain a valid device ID. Hard coding the device IDs is a bad practice because once a device ID is leaked, the corresponding device becomes vulnerable forever. The device ID of a device should be generated by the IoT cloud during registration, and the generation algorithm should use harder-to-guess information, such as user ID/passwords, random numbers, etc.

7.4.2 Comprehensive Authorization Checking

Compared with mobile-side commands, we found that most IoT clouds do not enforce strict authorization checking of device-side commands and baselessly trust arbitrarily connected devices. For *Type I* platforms, when a device talks to an IoT cloud, the user account information is absent on the device. Thus, the IoT cloud directly accepts unauthorized logins (F3) or unbind (F4) commands. For *Type II* platforms, because the device takes charge of checking the binding relationship, the cloud skips performing further authorization checking on the requests from the device.

We suggest that both the device and the IoT cloud store and maintain the binding relationship as well as perform authorization checking. Moreover, on the cloud side, the account-based authentication should be performed on every device-side request, especially for critical operations such as device login. Samsung SmartThings follows this practice and thus is not vulnerable to flaw F3. In SmartThings, devices must explicitly include user credentials for every login request. This additional credential checking prevents the target device from reconnecting to the cloud.

7.4.3 Enforcing the Validity of State Transitions

As revealed by our findings, all the tested platforms failed to enforce the validity of the involved state transitions. In order to prevent the attacker from exploiting unexpected state transitions, smart home platforms should identify and formulate every legitimate interaction request as a 3-tuple in the form of (sender entity & its state, the request message, receiver entity & its state). In addition to checking every request, the sender entity should also verify if its current state allows the request to be sent out; and the receiver entity should verify if its current state is allowed to receive the request. For instance, the IoT cloud shown in Figure 2 should only accept a device registration request when it stays in state 1. Furthermore, in order to prevent the three entities from staying out of the set of legitimate state combinations, the three entities should formally define and maintain their own state machines. In the meantime, the IoT cloud of a platform should synchronize the three entities so that they stay in a legitimate state combination. Finally, if an unrecoverable system error occurs, the three entities should roll back to their initial states immediately.

7.5 Malignant Commercial Competitions

The discovered exploits could also be leveraged by unscrupulous merchants in commercial competitions.

IP Theft. As mentioned in Section 6.4.3, a company can steal a rival's firmware and reverse-engineer it to steal pro-

prietary IP. This kind of behavior harms fair competition and hinders technology advancement.

Statistics Manipulation. By churning out hundreds of thousands of phantom devices, a malicious company could rig the number of active devices in the market. This has two implications. First, by increasing the market share of its own products, the company can present an eye-catching year-end report. Second, by increasing the number of activated devices of its rival, its rival could be overcharged by the platform provider. This is because some platform providers bill cooperative manufacturers based on the number of activated devices connected to their clouds. A unscrupulous manufacturer can use phantom devices to register a large number of non-existing devices under the name of its rivals, causing financial loss to them.

User Experience Disruption. Leveraging the illegal device occupation attack, a unscrupulous manufacturer can potentially take over a large number of its rival's in-stock products. When these products are sold, the consumers will have a terrible user experience.

8 Related Work

We review the related work on smart home security from three perspectives: device security, communication security and IoT application security.

Device Security. Device security research emphasizes the vulnerabilities of individual devices. Ling *et al.* [41] studied a smart plug system and revealed a weak authentication vulnerability. After dissecting the behavior of several IoT devices such as Phillips Hue light bulbs and Nest smoke detectors, Notra *et al.* [44] revealed that basic security mechanisms such as encryption, authentication and integrity checking are absent in these devices. Several currently available smart hubs were investigated in [20] and [55], and numerous security flaws were identified. In contrast to analyzing individual devices, our study analyzes the complex interactions among the three entities engaged in a smart home platform.

Communication Security. Communication security research emphasizes the security and privacy issues in smart home communication protocols such as BLE, ZigBee, and Z-Wave [14, 1, 50, 27]. Agosta *et al.* [1] approached the security and privacy problems involved in the key derivation algorithm adopted by the widespread Z-Wave home automation protocol. Ronen *et al.* [50] described a worm attack which has the potential of massive spread by exploiting an implementation bug in the ZigBee Light Link protocol. Researchers also demonstrated that attackers can infer private in-home activities by analyzing encrypted traffic from smart home devices [9] or by extracting features of connection-oriented application data unit exchanges [46]. Instead of focusing on a particular algorithm or protocol, this study conducts comprehensive platform-wide vulnerability analysis.

IoT Application Security. Recently, increasing numbers of researchers have paid their attention to smart home platforms, but they usually focus only on in-cloud IoT applications (i.e. home automation applications). For instance, Fernandes *et al.* [23] revealed that the capabilities implemented in the SmartThings IoT application programming framework are too coarse-grained, which allows malicious third-party IoT applications to compromise the SmartThings platform. Celik *et al.* [16] proposed *SAINT*, a static taint analysis tool to find sensitive data flows in IoT applications. The same authors further studied whether an IoT application and its environment adhere to functional safety properties. They found that 9 out of 65 SmartThings apps violate 10 out of 35 properties [17]. Kafle *et al.* [35] revealed the feasibility and severity (e.g., privilege escalation) of misuse of smart home routines. Moreover, Ding *et al.* [21] presented a tool named *IoTMon* to discover risky interaction chains among IoT applications. Our work focuses on the interactions between the participating entities engaged in a smart home platform, instead of between home automation applications.

9 Conclusions

Smart home technology is playing a more and more important role in our digital lives. To seize a greater market share, smart home platform providers shorten the time-to-market by reusing existing architectures and incorporating open-source projects without rigorous review (of the potential security and privacy issues). In this work, we conducted an in-depth analysis of five widely-used smart home platforms, and found that the complex interactions among the participating entities (i.e., devices, IoT clouds, and mobile apps), though not being systematically investigated in the literature, are vulnerable to a spectrum of new attacks, including remote device substitution, remote device hijacking, remote device DoS, illegal device occupation, and firmware theft. The discovered vulnerabilities are applicable to multiple major smart home platforms, and cannot be amended via simple security patches. Accordingly, we propose several defensive design suggestions to secure smart home platforms in the first place.

Acknowledgments

We would like to thank our shepherd William Enck and the anonymous reviewers for their helpful feedback. Wei Zhou and Yuqing Zhang was support by National Key R&D Program China (2016YFB0800700), National Natural Science Foundation of China (No.U1836210, No.61572460), Open Project Program of the State Key Laboratory of Information Security (2017-ZD-01) and in part by CSC scholarship. Peng Liu was supported by ARO W911NF-13-1-0421 (MURI), NSF CNS-1505664, NSF CNS-1814679, and ARO W911NF-15-1-0576. Le Guan was partially supported by JFSG from the University of Georgia Research Foundation, Inc.

References

[1] Giovanni Agosta, Alessio Antonini, Alessandro Barenghi, Dario Galeri, and Gerardo Pelosi. Cyber-security analysis and evaluation for smart home management solutions. In *International Carnahan Conference on Security Technology*, pages 1–6, 2016.

[2] Alibaba. Alink. `https://open.aliplus.com/docs/open/`, 2018.

[3] Alibaba. Aliyun IoT. `https://iot.aliyun.com/`, 2018.

[4] Wi-Fi Alliance. Portable Wi-Fi that goes with you anywhere. `https://www.wi-fi.org/discover-wi-fi/wi-fi-direct`, 2013.

[5] Amazon. 17 Top New Smart Home Appliances. `https://www.amazon.com/slp/smart-home-appliances/ygn94ga658qv4k8`, 2018.

[6] Amazon. AWS IoT Core. `https://aws.amazon.com/iot-core/?nc1=h_ls`, 2018.

[7] Ambient. Smart Hosting: The dos and don'ts of the ultimate Airbnb smart home. `https://www.the-ambient.com/guides/host-smart-airbnb-home-tech-217`, 2018.

[8] Apple. Apple HomeKit Developer Documentation. `https://developer.apple.com/documentation/homekit`, 2018.

[9] Noah Apthorpe, Dillon Reisman, Srikanth Sundaresan, Arvind Narayanan, and Nick Feamster. Spying on the Smart Home: Privacy Attacks and Defenses on Encrypted IoT Traffic. *arXiv preprint arXiv:1708.05044*, 2017.

[10] Home Assistant. `https://www.home-assistant.io/`, 2018.

[11] Home Assistant. Components – SmartThings. `https://www.home-assistant.io/components/smartthings/`, 2018.

[12] Home Assistant. Components. `https://www.home-assistant.io/components/`, 2019.

[13] Mesko Bertalan. Healthcare Is Coming Home With Sensors and Algorithms. `http://medicalfuturist.com/healthcare-is-coming-home/`, 2018.

[14] B.Fouladi and S.Ghanoun. Honey, I'm Home!!, Hacking ZWave Home Automation Systems. In *Black Hat USA*, 2013.

[15] BusinessWire. Smart Home Technology Can Increase Your Earning Potential. `https://www.businesswire.com/news/home/20160802005777/en/60-Percent-Guests-Pay-Vacation-Rental-Smart`, 2016.

[16] Z Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A Selcuk Uluagac. Sensitive information tracking in commodity IoT. In *Proceedings of Usenix Security Symposium*, pages 1687–1704, 2018.

[17] Z Berkay Celik, Patrick McDaniel, and Gang Tan. SOTERIA: Automated IoT safety and security analysis. In *2018 Usenix Annual Technical Conference*, pages 147–158, 2018.

[18] Chih Yung Chang, Chin Hwa Kuo, Jian Cheng Chen, and Tzu Chia Wang. Design and Implementation of an IoT Access Point for Smart Home. *Applied Sciences*, 5(4):1882–1903, 2015.

[19] Low Cherlynn. Router maker TP-LINK turns its attention to smart homes. `https://www.engadget.com/2016/08/23/tp-link-us-rebrand/`, 2016.

[20] Steven A. Christiaens. Evaluating the Security of Smart Home Hubs. Master's thesis, Brigham Young University, 2015. `https://scholarsarchive.byu.edu/etd/5631`.

[21] Wenbo Ding and Hongxin Hu. On the Safety of IoT Device Physical Interaction Control. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 832–846. ACM, 2018.

[22] Hossein Fereidooni, Jiska Classen, Tom Spink, Paul Patras, Markus Miettinen, Ahmad Reza Sadeghi, Matthias Hollick, and Mauro Conti. Breaking Fitness Records Without Moving: Reverse Engineering and Spoofing Fitbit. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 48–69, 2017.

[23] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. Security analysis of emerging smart home applications. In *Security and Privacy*, pages 636–654, 2016.

[24] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *Proceedings of Usenix Security Symposium*, pages 531–548, 2016.

[25] Earlence Fernandes, Amir Rahmati, Jaeyeon Jung, and Atul Prakash. Decentralized Action Integrity for Trigger-Action IoT Platforms. In *Proc. of NDSS*, pages 18–21, 2018.

[26] Marc Goodman. Hacking the Human Heart. `http://bigthink.com/future-crimes/hacking-the-human-heart`, 2017.

[27] Rohit Goyal, Nicola Dragoni, and Angelo Spognardi. Mind the tracker you wear: a security analysis of wearable health trackers. In *ACM Symposium on Applied Computing*, pages 131–136, 2016.

[28] Grant Hernandez, Orlando Arias, Daniel Buentello, and Yier Jin. Smart Nest Thermostat: A Smart Spy in Your Home. In *Black Hat USA*, 2014.

[29] Texas Instruments. SimpleLink Wi-Fi SmartConfig Technology. `http://www.ti.com/tool/SMARTCONFIG`, 2013.

[30] JD. JD Alpha. `https://alphadev.jd.com/`, 2018.

[31] JD. Joylink. `http://smartdev.jd.com/`, 2018.

[32] Arul John. MAC address and OUI lookup. `http://aruljohn.com/mac.pl`, 2018.

[33] Delaney John R and Colon Alex. The Best Smart Home Security Systems of 2018. `https://www.pcmag.com/article2/0,2817,2498510,00.asp`, 2018.

[34] Tour Joney. Xiaomi - the World's Largest IoT Platform. `https://xiaomi-mi.com/news-and-actions/xiaomi-the-worlds-largest-iot-platform/`, 2018.

[35] Kaushal Kafle, Kevin Moran, Sunil Manandhar, Adwait Nadkarni, and Denys Poshyvanyk. A Study of Data Store-based Home Automation. *arXiv preprint arXiv: 1812.01597*, 2018.

[36] Minhaj Ahmad Khan and Khaled Salah. IoT security: Review, blockchain solutions, and open challenges. *Future Generation Computer Systems*, 82:395 – 411, 2018.

[37] KODY. Track Wi-Fi Devices & Connect to Them Using Probequest. `https://null-byte.wonderhowto.com/how-to/track-wi-fi-devices-connect-them-using-probequest-0186137/`, 2018.

[38] Mohit Kumar. Ransomware Hijacks Hotel Smart Keys to Lock Guests Out of their Rooms. `https://thehackernews.com/2017/01/ransomware-hotel-smart-lock.html`, 2017.

[39] Ivan Kust. Securing mobile banking on Android with SSL certificate pinning. `https://infinum.co/the-capsized-eight/securing-mobile-banking-on-android-with-ssl-certificate-pinning`, 2018.

[40] Sanghak Lee, Jiwon Choi, Jihun Kim, Beumjin Cho, Sangho Lee, Hanjun Kim, and Jong Kim. FACT: Functionality-centric access control system for IoT programming frameworks. In *Proceedings of the 22nd ACM on Symposium on Access Control Models and Technologies*, pages 43–54. ACM, 2017.

[41] Zhen Ling, Junzhou Luo, Yiling Xu, Chao Gao, Kui Wu, and Xinwen Fu. Security Vulnerabilities of Internet of Things: A Case Study of the Smart Plug System. *IEEE Internet of Things Journal*, PP(99):1–1, 2017.

[42] Lily Hay Newman. Turning an Echo Into a Spy Device Only Took Some Clever Coding. https://www.wired.com/story/amazon-echo-alexa-skill-spying, 2018.

[43] AP News. Strategy Analytics: Global Smart Home Market to Hit $155 Billion by 2023. https://www.apnews.com/e7466a4bc2bd4243a8bcb3915bde8731, 2018.

[44] S Notra, M Siddiqi, H. H Gharakheili, and V Sivaraman. An experimental study of security and privacy risks with emerging household appliances. In *Communications and Network Security*, pages 79–84, 2014.

[45] TJ OConnor, William Enck, and Bradley Reaves. Blinded and Confused: Uncovering Systemic Flaws in Device Telemetry for Smart-home Internet of Things. WiSec, pages 140–150, 2019.

[46] TJ OConnor, Reham Mohamed, Markus Miettinen, William Enck, Bradley Reaves, and Ahmad-Reza Sadeghi. HomeSnitch: Behavior Transparency and Control for Smart Home IoT Devices. WiSec, pages 128–138, 2019.

[47] OFweek. JD has teamed up with Ziroom to create an smart life for tenants. https://smarthome.ofweek.com/2016-06/ART-91008-8120-29106961.html, 2016.

[48] OpenHAB. OpenHAB empowering the smart home. https://www.openhab.org/, 2018.

[49] Pocket-lint. Apple HomeKit and Home app: What are they and how do they work? https://www.pocket-lint.com/smart-home/news/apple/129922-apple-homekit-and-home-app-what-are-they-and-how-do-they-work, 2018.

[50] Eyal Ronen, Adi Shamir, Achi-Or Weingarten, and Colin OâĂŹFlynn. IoT goes nuclear: Creating a ZigBee chain reaction. In *2017 IEEE Symposium on Security and Privacy*, pages 195–212. IEEE, 2017.

[51] Statista. Smart Home Market. https://www.statista.com/outlook/279/109/smart-home/united-states, 2019.

[52] Sumsung. SmartThings Developers Documentation. https://smartthings.developer.samsung.com/docs/index.html, 2018.

[53] TP-LINK. KASA. https://www.tp-link.com/us/kasa-smart/kasa.html, 2018.

[54] Junia Valente and Alvaro A. Cardenas. Security & privacy in smart toys. In *Proceedings of the 2017 Workshop on Internet of Things Security and Privacy*, pages 19–24, 2017.

[55] Veracode. Veracode Study Reveals the Internet of Things Poses Cybersecurity Risk. https://www.veracode.com/veracode-study-reveals-internet-things-poses-cybersecurity-risk, 2015.

[56] XiaoMi. MIJIA. https://iot.mi.com/index.html, 2018.

[57] Hyunwoo Yu, Jaemin Lim, Kiyeon Kim, and Suk-Bok Lee. Pinto: Enabling Video Privacy for Commodity IoT Cameras. In *CCS*, pages 1089–1101. ACM, 2018.

A Legitimate 3-tuple State Combinations

	State of an IoT Cloud	State of a Device	State of a Mobile App
***Type I* Platform**	S1	S1 or S2	S1 or S2
	S2	S3	S2
	S3	S3	S3
	S4	S4	S4
***Type II* Platform**	S2	S1 or S2	S1 or S2
	S3	S3	S3
	S4	S4	S4

B Tested Devices and Applicable Attacks

	Tested Device	Device Model	Platform	Applicable Attacks
***Type I* Platform**	Mobile Remote HD Monitor	RIWYTH RW-821S-ALY	Alink	Remote Device Hijacking Remote Device Substitution Remote Device DoS Illegal Device Occupation
	WiFi Smart Adapter	Philips SPS9011A/93	Alink	Remote Device Hijacking Remote Device Substitution Remote Device DoS Illegal Device Occupation
	Security Smart Lock	KAADAS KDSLOCK001	Alink	Remote Device Hijacking Remote Device Substitution Remote Device DoS Illegal Device Occupation
	WiFi Smart Plug	BULL GN-Y2011	Joylink	Remote Device Substitution Illegal Device Occupation
	Smart Weighing Scale	ZK321J	Joylink	Remote Device Substitution Illegal Device Occupation
***Type II* Platform**	WiFi Plug	TP-Link HS100	KASA	Remote Device Hijacking Remote Device Substitution Remote Device DoS
	WiFi LED Bulb	TP-Link LB110	KASA	Remote Device Hijacking Remote Device Substitution Remote Device DoS
	Smart Gateway	XiaoMi Multifunctional Gateway	MIJIA	Remote Device Hijacking Remote Device Substitution Remote Device DoS
	Smoke Alarm	XiaoMi Fire Alarm Detector	MIJIA	Remote Device Hijacking Remote Device Substitution Remote Device DoS
	Cleaning Robot	Samsung POWERbot R7040	SmartThings	Remote Device DoS
	Hub	Samsung Hub 3rd Generation	SmartThings	Remote Device DoS

Table 1: Tested devices and applicable attacks

C Sequence Diagrams

We show the complete sequence diagram of interactions with concrete parameters for each of the studied platforms. Note that some essential steps are omitted because they are irrelevant to our attack. In each figure, a box corresponds to one phrase in the life-cycle of a device (Section 6.2).

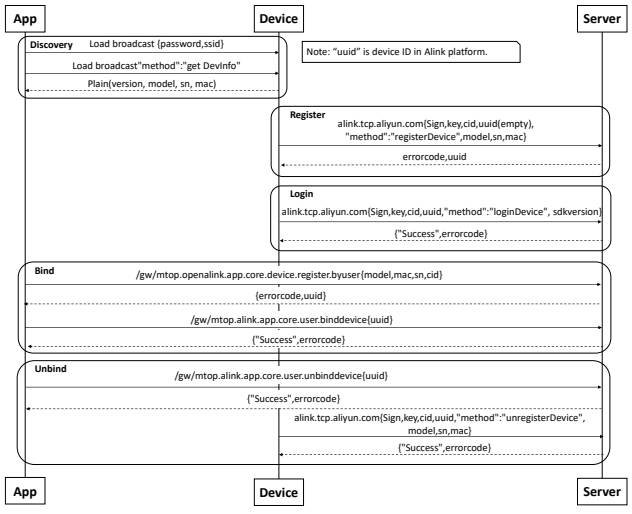


Figure 1: Sequence Diagram of the Alink Platform

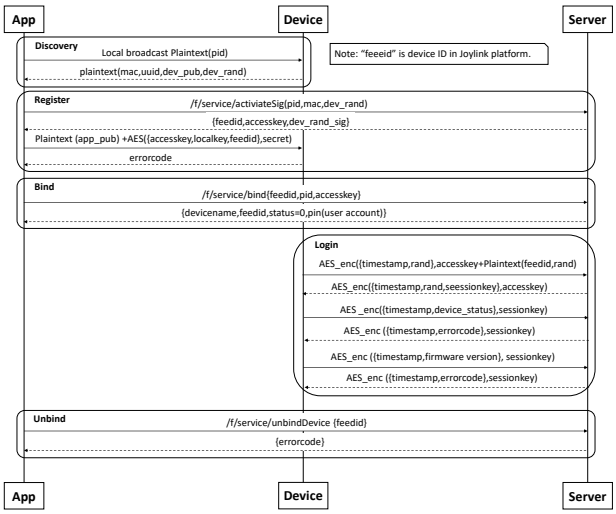


Figure 2: Sequence Diagram of the Joylink Platform

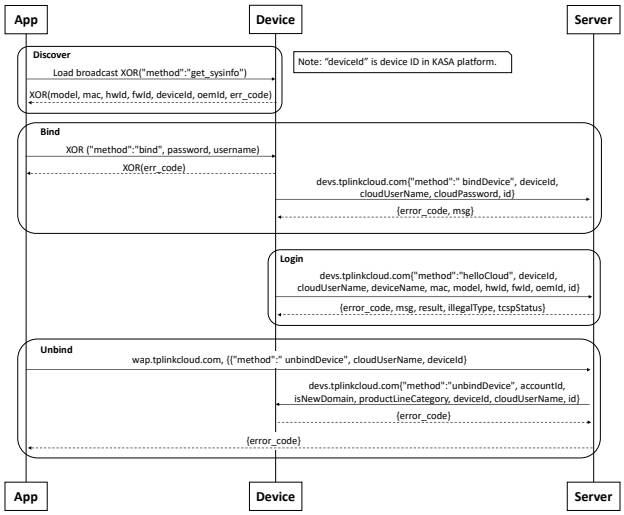


Figure 4: Sequence Diagram of the KASA Platform

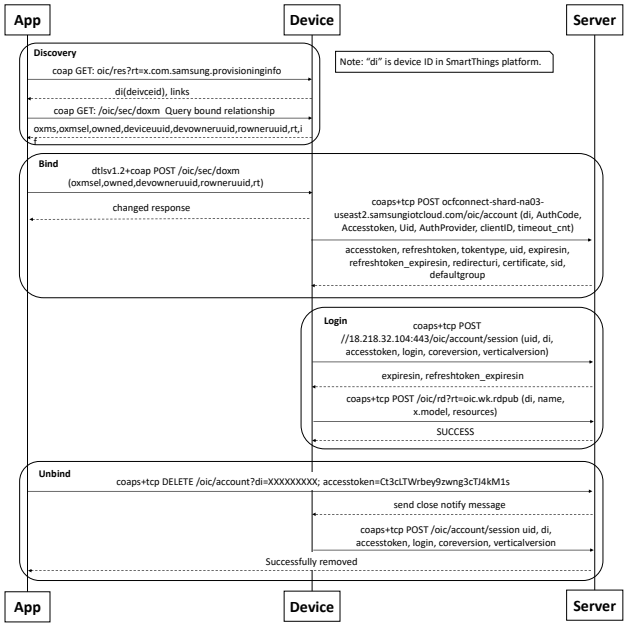


Figure 3: Sequence Diagram of the SmartThings Platform

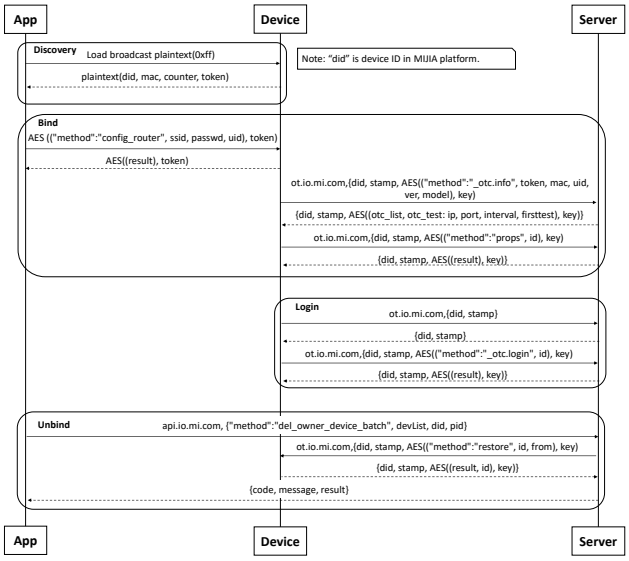


Figure 5: Sequence Diagram of the MIJIA Platform

Looking from the Mirror: Evaluating IoT Device Security through Mobile Companion Apps

Xueqiang Wang
Indiana University Bloomington

Yuqiong Sun
Symantec Research Labs

Susanta Nanda
Symantec Research Labs

XiaoFeng Wang
Indiana University Bloomington

Abstract

Smart home IoT devices have increasingly become a favorite target for the cybercriminals due to their weak security designs. To identify these vulnerable devices, existing approaches rely on the analysis of either real devices or their firmware images. These approaches, unfortunately, are difficult to scale in the highly fragmented IoT market due to the unavailability of firmware images and the high cost involved in acquiring real-world devices for security analysis.

In this paper, we present a platform that accelerates vulnerable device discovery and analysis, without requiring the presence of actual devices or firmware. Our approach is based on two key observations: First, IoT devices tend to reuse and customize others' components (e.g., software, hardware, protocol, and services), so vulnerabilities found in one device are often present in others. Second, reused components can be indirectly inferred from the mobile companion apps of the devices; so a cross analysis of mobile companion apps may allow us to approximate the similarity between devices. Using a suite of program analysis techniques, our platform analyzes mobile companion apps of smart home IoT devices on market and automatically discovers potentially vulnerable ones, allowing us to perform a large-scale analysis involving over 4,700 devices. Our study brings to light the sharing of vulnerable components across the smart home IoT devices (e.g., shared vulnerable protocol, backend services, device rebranding), and leads to the discovery of 324 devices from 73 different vendors that are likely to be vulnerable to a set of security issues.

1 Introduction

Smart home IoT devices have become favored targets for attackers [41] as much for the lack of user awareness [35] as for their poor security design [46]. As the motivation for attackers grows (e.g. IoT botnets, personal data theft), security incidents for smart home devices are only expected to increase. Securing these devices is challenging on several fronts. First, a good fraction of vendors in this space are small and medium-sized businesses that lack the budget for software quality control and security best practices, resulting in numerous insecure devices in the market. Second, many of these devices are relatively inexpensive (often less than $100) and cannot afford to have support for expensive security infrastructure, such as monitoring agents, encryption and authentication hardware, etc. Consequently, when a device is found vulnerable, there is very little incentive and capability for the vendor to release a fix. Third, high vendor fragmentation makes it hard to manage and distribute software/firmware patches.

One way to address this issue is to identify vulnerable devices before they get deployed and take appropriate measures to protect the device. Examples of such measures may include upgrading the device firmware, identifying and blocking traffic that can exploit the vulnerability, or quarantining the device completely. To identify the vulnerable devices beforehand, multiple approaches have been proposed [10, 14, 16, 17, 19, 20, 24, 25, 27, 34, 42, 47, 49, 58]. One line of research [19, 27] focused on launching an Internet-scale scan to detect trivially vulnerable devices (e.g., devices with weak passwords, certificates, and keys) that are publicly accessible. However, these approaches often cannot help identify devices with more sophisticated vulnerabilities or devices hidden behind NAT. Another line of research [10, 14, 16, 17, 20, 24, 25, 34, 42, 47, 49, 58] focused on statically and/or dynamically analyzing an IoT device or its firmware to evaluate its security. Although these approaches tend to yield more comprehensive and accurate results for individual devices, they do not scale well for a large-scale analysis. First, getting physical access to all the devices on the market is not a viable option because of restricted availability of devices in certain geographies and their prohibitively high acquisition cost. Similarly, device firmware is not always available due to the highly fragmented market that involves a lot of small integration and distribution vendors [1]. Second,

[1] Integration vendors are the ones that integrate components, tools, and SDKs, provided by OEMs. Distribution vendors simply acquire the device from an OEM and re-brand with their own before selling in the market.

even with a device or its firmware, the analysis itself is often tedious, error-prone and difficult, especially considering the "device shell" that is often put in place by the device vendors (e.g., packing, obfuscation and encryption). As a result, the market would benefit from an approach where vulnerable devices can be quickly identified at scale and the scope of analysis can be narrowed down.

Approach. In this paper, we present a platform that accelerates vulnerable device discovery and analysis without requiring access to a physical device or its firmware. Our approach is based on two observations. First, smart home IoT device vendors, especially small and medium-sized ones, often rely on same components (e.g., software built from open source projects, hardware components from common suppliers) to build their devices. Consequently, the same vulnerabilities or bad security practices often transfer from one IoT device to another. We can thus propagate vulnerability information to an *unknown* device by evaluating its similarity with devices *known* to be vulnerable. Second, similarities of devices are often reflected in their mobile companion apps, which are widely accessible. Combining these two observations enables us to build a platform that identifies vulnerable devices in a scalable way without requiring the physical devices themselves or their firmware images.

In our platform, we try to expedite the process of identifying vulnerable devices by providing two functions: (1) *app analysis*: find the characteristics of a device by analyzing its companion app, and (2) *cross-app analysis*: find device families, i.e. cluster of devices, that have similarity in some of the characteristics found in app analysis by analyzing multiple apps. Clustering helps identify apps that have a similar set of vulnerabilities based on shared components [8].

Results Overview. For our experiments, we crawled Google Play Store [3] to search for potential IoT companion apps and downloaded 3,094 of them. After filtering out some noise, we were left with a dataset of 2,081 apps (see Section 2.2 for more details). These apps were then analyzed by our platform.

First, we found the device clusters, i.e., device families, containing devices that are similar in various aspects such as software or hardware components, back-end services, and network protocols. For instance, in our analysis, we found 19 device families covering 139 apps from 122 different vendors where devices in a family shared similar software components. As another example, we found 48 different families covering 460 devices that shared similar back-end services.

Second, we tried to identify devices that are impacted by a given vulnerability using the device families already identified. In one case, we were able to discover devices from four different vendors (apps of which is estimated to be installed by more than 215,000 users) that were previously not known to be vulnerable to a software vulnerability and independently confirm the existence of the vulnerability on 45 devices from four different vendors that were previously confirmed by other sources. In another case, we were able to identify 67 devices from 16 different vendors that are impacted by a hardware security issue. In total, our platform has identified 324 potentially vulnerable devices from 73 different vendors. During the process of validation, we could reach a decision (confirm or disapprove) about 179 devices from 43 vendors, among which 164 (91.6%) are confirmed to be vulnerable.

Contributions. This paper makes the following contributions:

- It demonstrates how companion mobile apps for IoT devices can provide insights into the security of the devices themselves.
- It shows the effectiveness by using this approach to assess the security posture of IoT devices when neither the physical devices nor their firmware images are available.
- It proposes a platform to perform mobile app collection, filtering, analysis, and clustering at a large scale. It demonstrates its use by analyzing more than 2000 apps and clustering them in multiple dimensions.
- It reports the discovery of 324 devices from 73 distinct vendors that are likely to be vulnerable to a set of security issues.

2 System Design

2.1 Overview

Figure 1 presents an overview of our platform. The first component of our platform is the *IoT App Database*, which stores the companion apps of smart home IoT devices crawled from the Google Play Store [3]. The database is extended constantly by fetching more apps (e.g., when new IoT devices are on market or old apps get updated).

The apps stored in IoT App Database are then analyzed by the *App Analysis Engine*. The goal of the App Analysis Engine is to estimate the profile of an IoT device (i.e., what the device is like) based on code analysis. Specifically, the App Analysis Engine computes three things: the network interfaces of a device, the unique strings (called *imprints*) that a device may include, and code signature of the companion app. The results of App Analysis Engine are stored in the *App Analysis Database*.

A *Cross-App Analysis Engine* queries the App Analysis Database and identifies correlations across different devices in order to build a *device family*. A device family groups together different devices from different vendors based on their *similarity*. The similarity can be in terms of different dimensions (e.g., similar software, similar hardware, similar protocols, and similar cloud back-end services). The device family allows propagation of vulnerability information among similar devices. Specifically, it allows evaluation of IoT device

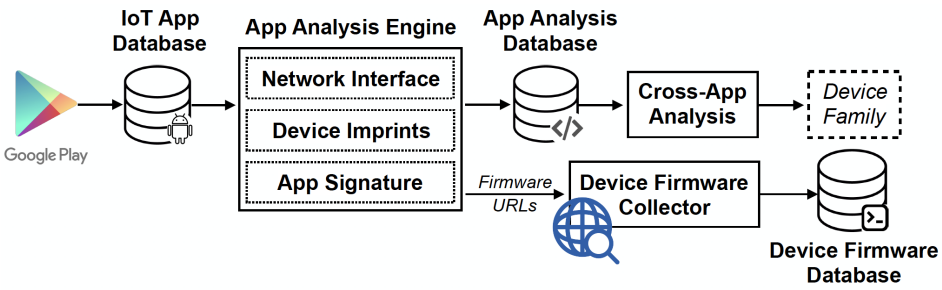


Figure 1: An overview of the platform

security from the perspective of either a device or a threat: 1) for a specific device, the similarity allows to quickly assess whether or not the device is vulnerable and if so to which vulnerabilities, and 2) for a specific vulnerability, find the set of devices on the market that might be affected by the vulnerability.

To facilitate vulnerability confirmation, our platform contains an additional component called *Device Firmware Collector*. It leverages the code analysis results output by the App Analysis Engine (e.g., Firmware URLs) as well as Internet search results to download firmware images into a *Device Firmware Database*. These firmware images later enable us to further confirm the vulnerabilities found by the Cross-App Analysis Engine. Note that the Device Firmware Collector is not an essential or required component of our platform. Rather, it is utilized as one of several means to help confirm the findings from the platform (See Section 3.2 for more details).

In the remainder of this section, we describe each component of our platform in detail.

2.2 App Collection

The first step of our platform is to gather mobile companion apps of smart home IoT devices for analysis. To achieve this goal, we crawled Google Play Store[2]. In total, we downloaded 3,094 Android apps, out of which 2,081 were included in the final dataset and analyzed by our platform.

The challenge during app collection is to identify apps that are mobile companion apps of IoT devices. To address this problem, we initialized the crawler with 281 seed apps manually selected from the online smart home products database *SmartHomeDB* [5], and used snowball sampling to collect more apps via the connections between the seed apps and other apps on Google Play (e.g., keywords, suggestions and categories). As a result, 3,094 candidate apps are initially downloaded. However, we observed that snowball sampling may sometimes introduce noise. For example, apps that manage phone camera are confused with the apps that manage home security cameras. Apps that lock phone's screen are confused with smart home locks. To eliminate such noise from the dataset, we performed filtering. The filtering is based on a clustering model (Affinity Propagation [26]) that clusters apps based on the permissions that the apps request on installation and the sensitive Android APIs that the apps may invoke at runtime. We deploy the filtering on apps that are nominated by the same seed sample and remain the largest cluster. This approach turns out to be effective: a random manual inspection of 200 apps after filtering shows that 98.5% of them are real mobile IoT companion apps. After further deduplication, 2,081 apps are left in the dataset and fed into the App Analysis Engine for analysis. Note at the first phase of the research, we worked with a relatively small dataset and focused more on validating the approach. Our platform is constantly running to collect more apps for future analysis at a larger scale.

[2]We based our analysis primarily on Android but most IoT device vendors provide mobile companion apps in both iOS and Android.

2.3 App Analysis Engine

The *App Analysis Engine* analyzes mobile companion apps collected in order to build a device profile for individual devices. Unlike previous works [12, 32, 48, 60] that focused on apps themselves, the goal here is to compute what the *device* is like, indirectly from the app. We achieved this goal by independently applying three methods: a device interface analysis that computes the network interfaces of a device, an imprint analysis that computes unique strings a device might be related to, and a fuzzy hash analysis that computes code signature of a mobile companion app. In practice, we found that the first method is more comprehensive and informative. Nevertheless, the rest two methods are still useful in filling in gaps where the first method cannot easily apply.

2.3.1 Device Interface Analysis

The device interfaces are often a good reflection of what the device is like, e.g., the protocol that the device speaks, the service that the device runs, the function that the device supports, and sometimes the hardware components in use by the device. Without directly examining a device or its firmware, we estimate the device interfaces based on analysis of its mobile

companion app, as the app and the device complement each other in their network interfaces. A peer-to-peer connection between the app and the device can benefit this estimation, as the app interfaces, in this case, are direct reflections of the device interfaces; however, this is not a necessary condition. For devices where a cloud or backend service is involved, popular backend services like Microsoft Azure IoT Hub [40] are often generic: they tend to relay the connection between the app and the device without much meddling. Such devices also work well with our approach since their app interfaces still closely reflect that of the devices. We performed a study over the online IoT device database (SmartHomeDB), and found that majority of the devices (76.3%) produced by small and medium-sized vendors support a peer-to-peer connection between the app and the device. Even large vendors like TP-Link support both cloud and peer-to-peer mode for network outage and privacy reasons. This enables us to have a good estimation of the device interfaces for many of the IoT devices, especially vulnerable ones, that are sold on the market.

We used a backward approach to compute the network interfaces of an app, starting from the network response messages that the app may receive, as these messages are information output by the device. We first identify message handling functions in the app and statically decide what the response message may look like. We then identify the request that may trigger the response. Finally, we partially instantiate and execute the app code to reconstruct the request [12, 56]. Figure 2 shows an example of the request and response extracted from the mobile companion app *com.Zengge.LEDWifiMagicColor* of Zengge Wi-Fi Bulb. With many different pairs of requests and responses (e.g., with UDP/48899, TCP/5577 of the bulb and also the cloud server **.magichue.net*), we obtain a good estimation of the device interfaces.

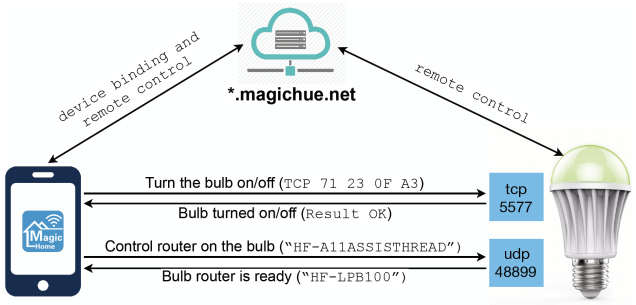


Figure 2: Interfaces of Zengge Wi-Fi Bulb

Response Extraction. We rely on symbolic execution [33] to estimate what the response messages from a device may look like, without actually running the device. We first built a Control Dependency Graph (CDG) and Data Dependency Graph (DDG) of a mobile companion app using Soot [51]. We then start from standard network receiving functions in Android (e.g., <*java.net.DatagramSocket: void receive(java.net.DatagramPacket)*>) and forward execute the mobile companion app symbolically. Whenever we encounter a branch that is dependent on the content of a response message (e.g., fields of the response are checked against a value), we capture the check as a symbolic constraint and fork the execution. After all executions terminate, the conjunction of the symbolic constraints is stored as a "description" of the response message. In order for response messages from two devices to be similar, they have to satisfy the same set of symbolic constraints.

One practical issue is to decide when to terminate a symbolic execution. In our experience, we found that a valid response from the device (i.e., the response passes checks performed by the app) often triggers state changes of the app. Such state changes could be either UI element changes (e.g., updating device status displayed to the user) or modifications to the local registry (e.g., storing device information to configuration files, shared preferences or databases). To confirm this heuristic, we randomly sampled 200 response handling procedures that exist in 179 apps from our app set and evaluated manually the impact of valid responses. Among these responses, 162 of them had an influence on UI elements, and 76 of them resulted in modifications to the local registry (with some overlapping cases where responses changed both); only eight of them would not trigger such changes, but the app stored response content (e.g., login token) in global variables. This study shows that state changes can be a good approximation for the termination of valid response handling. We thus mark such state changes as the point where we terminate the symbolic execution and produce the conjunction of the symbolic constraints. In addition, we supplement this method with the observation that invalid responses are discarded quickly by the apps (i.e., within few lines of code). We thus also set a threshold on the number of procedures to execute before we terminate the execution. Utilizing these two heuristics, we could produce a small but meaningful set of constraints that closely describe a valid response that an IoT device may produce.

Pairing Request and Responses. The next step is to identify the request sent by the app that will trigger the response from the device. In many cases, the request is straightforward to identify: it co-locates with the response message handling functions. In other cases, however, it is trickier as the request can be located in a different procedure or class, especially when the communication between app and device is asynchronous. In these cases, static code analysis can be limited in identifying the matching request.

Fortunately, we observe that a matching pair of request and response often share a large code base of their handling functions (i.e., classes and methods used to process the request and response). Such similarities are reflected in the stack at runtime. To confirm this observation, we examined the paired requests and responses for the same set of 200

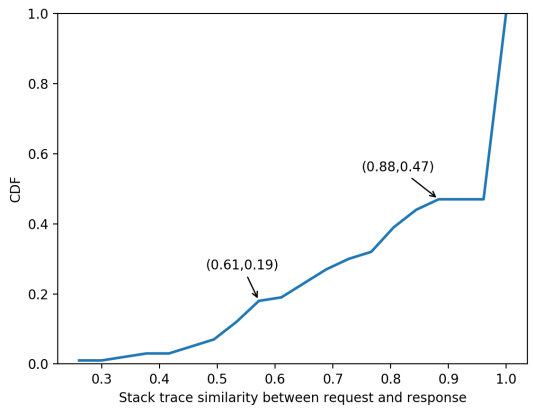


Figure 3: Cumulative Distribution Function (CDF) of request and response similarity

response handling procedures, and evaluated the similarities between stack traces of the responses and the requests. Figure 3 shows the Cumulative Distribution Function (CDF) of the Jaccard Similarity: 81% pairs of request and response share over 61% of their stack frames, and more than half (53%) of the request-response pairs have over 88% frames in common. For unpaired requests and responses, the similarity reduces to almost zero. Thus, by recording and comparing the execution stack of the app when the app is making requests (i.e., via concrete execution) and processing responses (i.e., via program dependence graph), we can pinpoint with good accuracy, among multiple request sending functions, the one that most likely will trigger the target response. As an example, Figure 4 shows the stack traces of a request and response that are used by Chuango Wi-Fi alarm system. The request and response are matched based on the common stack frames (e.g., those triggered by the same user click) despite being located in different classes that run in different threads.

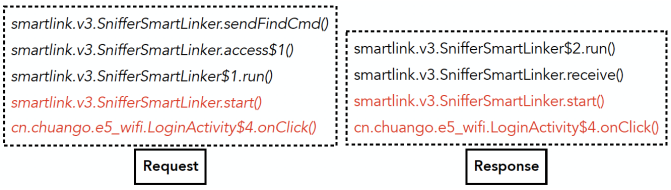


Figure 4: A matching asynchronous request and response in *cn.chuango.e5_wifi*

Request Reconstruction. After identifying a matching pair of request and response, the next step is to reconstruct the request string. Unlike responses, requests are produced by the app. Therefore, we may reconstruct a complete request string as compared to a set of symbolic constraints for responses. A number of techniques have been developed for reconstructing program values via program slicing and execution [12, 29, 48, 56]. We adopted the *Instantiated Partial Execution (IPE)* technique developed in Tiger [12] in our platform. The advantage of using IPE is that it evaluates and instantiates variables to concrete values if they are found to be irrelevant to the request string thereby dramatically reducing the number of paths need to be explored. In addition, IPE also caches outputs of code slices and reuses the results if applicable, further reducing the analysis complexity. By using IPE, we were able to reduce the time needed to reconstruct a request to under a minute.

The result produced by device interface analysis is a set of request and response pairs. The requests are fully or partially[3] reconstructed request strings and responses are sets of symbolic constraints. A device is said to have a similar interface as another device if they both accept similar requests and output similar responses.

2.3.2 Imprints Analysis

Device imprints (i.e., unique strings) found in an app can help correlate different devices. We are particularly interested in imprints that show up in the communication between the app and the device, as they are indicative of the uniqueness of the *device*. In contrast, there are also app imprints, such as app developer emails or special class names, that identify an app or library. However, they are less indicative of the device.

Table 1: Examples of device imprints

Type	Imprints	Device
device keywords	"20140930073702357" (dir. name in firmawre)	Homeboy Wi-Fi Security Camera
	"0622707c-da97-4286-cafe-"* (UUID of the device family)	SensingTEK Cameras
cert and comm. keys	"Ztwy518518puy518" (AES key)	Zhongteng Smart Home Devices
user & pwd	"P0rtal@123!" (account pwd)	Pro1 Thermostats
special URLs	"qjg7ec".internetofthings .ibmcloud.com (MQTT *orgID*)	Max Smart Home Devices

Inspired by previous work done by Costin et al. [16] that directly extracts imprints from embedded firmware images, we also focus on four types of device imprints: device (backdoor) keywords, certificates and keys, non-trivial usernames and passwords, and special URLs. The method we used to identify imprints is simple: we build a Data Dependence Graph of an app and check backward from network APIs to find constant strings in the app that affect parameters of those APIs. Note that these APIs are used to communicate with the device. In other words, we only use unique strings as imprints if they are related to the device (i.e., they are part of either requests to or responses from the device). A parser later decides which category the constant strings fall into and

[3] Certain requests require user input (e.g., login request). In these cases, we partially reconstruct the request with <NONE> string replacing the missing user input.

whether or not they are commonly seen (e.g., *admin* for both username and password is ignored). Table 1 shows an example of a few imprints we collected in our dataset. When two apps have the same imprints (and both imprints affect the communication with devices), it serves as a strong indicator of the similarity between devices. For instance, by using imprint *"OBJ-000165-PBKMW"*, we were able to correlate *VStarcam* and *OUSKI* IP Cameras (the latter is later confirmed to be a rebranding of the former).

Although imprints can serve as strong evidence of correlation, imprint analysis as a method is less applicable in general since many times imprints of a device do not manifest themselves in the app. For example, we were not able to spot the existence of any magic keyword, like the "*xmlset_roodkcableoj28840ybtide*" (i.e., edit by 04882 joel backdoor in reverse) keyword used by a number of devices for debugging purposes reported by Constin et al. [16]. This makes sense since the magic keyword is built into the firmware images for debugging purposes, and device debugging is generally not a critical functionality required for customer facing apps. However, it highlights the limitation of imprint analysis, and the reason why we need a fully fledged device interface analysis.

2.3.3 Fuzzy Hash Analysis

Another method we used is to assess code similarity via fuzzy hash. Similar mobile companion apps often indicate similar devices. We thus compute *ssdeep* of objects found in an app, including classes, libraries, and other types of resources (e.g., texts), and compare the results across apps. The benefit of using fuzzy hash as compared to traditional hashing algorithm (e.g., *SHA1*) is that we can relate objects that are similar but are not exactly the same. Through this way, we were able to identify a few similar devices. For example, the companion apps of *CHITCO* and *EDUP* smart switches are found to have 50.7% objects matched with 80/100 similarity, and these two devices are later confirmed to share similar software. Note, however, similarities between devices do not necessarily mean similarities in the apps. We observed in many cases that similar devices have different apps (e.g., apps are developed independently), and therefore cause failures to fuzzy hash analysis. Code similarity is more useful for identifying obvious correlations as well as for cases where other analysis methods have some difficulties to apply (e.g., for native libraries).

2.3.4 Modularity

A special consideration we made while building the App Analysis Engine is the modularity of the analysis. The reason we took this extra step as compared with generating analysis result per app is to accommodate the *modular similarity* that often appear across IoT devices. It is common that IoT device vendors, especially smaller ones, comprise their products from a number of existing modules on market, such as hardware components from common suppliers, software built from open source projects, binary driver code for protocols and etc. For example, the *HiFlying* Wi-Fi module is used by a number of vendors to manage Wi-Fi connectivity for their devices. Thus it is important for our analysis to be modular as well, in order to track device similarities and detect vulnerability propagation at a finer granularity of individual device components (Refer to Section 2.4 and Section 3.3 for more details of the components that we can track).

We based our design on the observation that device components are often managed by different code modules in the app (e.g., class, package). Taking the previous HiFlying Wi-Fi module as an example, devices such as *BeSMART* thermostat that uses the module often have two separate classes, *com.hiflying.smartlink.v3.SnifferSmartLinkerSendAction* and *com.besmart.thermostat.MyHttp*, for handling Wi-Fi connection and user interaction over HTTP, respectively. We thus infer such modularity from the app (e.g., based on class hierarchy and invocation stack) and apply the above analysis method on individual modules.

2.4 Cross-App Analysis Engine

The analysis results output by the App Analysis Engine are stored into the *App Analysis Database*, which is then queried by the *Cross-App Analysis Engine*. The Cross-App Analysis Engine is designed to detect modular similarities between different devices. In particular, the comparison is made to detect four types of similarities: similar software components, similar hardware components, similar protocol, and similar backend services.

Similar Software Components. Similar device interfaces, especially application interfaces, are indicative of strong connections between software components of different devices. For example, we were able to correlate 72 different smart home IoT devices from 16 distinct vendors that might have used the same version of GoAhead web server[4]. Such correlation is powerful, as in many times security weaknesses manifest them in software and security weakness found in one device can directly impact the security of others. For example, we were able to identify seven previously unreported devices that are vulnerable to a known vulnerability, as detailed in Section 3.3.

Another interesting phenomenon detected is device rebranding. In the smart home IoT industry, smaller vendors sometimes do not develop their own products. Instead, they customize IoT devices from OEMs and resell with their own branding. As reflected in the app analysis results, rebranded devices have almost identical device interfaces across multi-

[4]GoAhead is a simple web server specifically designed for embedded devices.

ple modules as the original OEM devices. Although device rebranding itself is not an issue, it complicates the security practices in firmware update and patching. In some cases, for example as shown in Section 3.3, a vulnerability is inherited by the rebranded devices from the OEM but the security patch that fixes the vulnerability is not.

Similar Hardware Component. Smart home IoT devices may be built upon similar hardware (e.g., Wi-Fi module). Such similarities in hardware components are sometimes reflected in device companion apps due to the need for the app to configure or interact with the hardware component. Due to the specialty of the hardware, such device-app interfaces can be unique, allowing a strong correlation of different devices using the same hardware. For example, we found that two Wi-Fi modules with a known security weakness of credential leakage are potentially being used by 166 devices from 35 different device vendors. The total downloads of these apps together are over 278,000 times.

Similar Protocol and Backend Service. A specific protocol often has its own request and response format. Similarly, a specific backend service often exposes standard APIs. Cross-App Analysis Engine can detect similarities in network interfaces and thus correlate devices that speak the same protocol or speak with the same backend service, even if such protocols or backend services are not documented. For example, we found that 39 different devices from 11 vendors are very likely to speak the SSDP protocol, which was known to be vulnerable as a reflector for DDoS attacks. As another example, we found that 32 devices from 10 vendors relied on the same cloud service to manage their devices, and the cloud service has a reported security weakness that allows attackers to take full control of the IoT devices by device ID and password enumeration.

Future Work. There are additional dimensions that security evaluation of an IoT device can benefit from similarity analysis. For example, previous works [16, 28] have shown that same developers or sub-contractor may follow a similar way of coding thus having the same set of bad security practices or vulnerabilities built into their devices. Similarly, the same development toolchain (e.g., compiler) may transform code in a similar way that leads to the same set of security issues [7, 52, 54]. As a future work, we plan to extend our analysis to cover more dimensions of similarities in order to obtain a more accurate and complete evaluation of smart home IoT devices.

2.5 Device Firmware Collector

Our platform features an additional component called *Device Firmware Collector* which enriches the *Device Firmware Database* through downloading firmware images of devices corresponding to the apps being analyzed. The purpose of the firmware images is to help us confirm the findings from the cross-app analysis phase. In our current platform, we collect device firmware in two ways. First, we utilize the firmware downloading links that are embedded in the mobile companion apps. As IoT devices are usually headless (i.e., no keyboard or screen for user interaction), they often deploy firmware updates via the companion apps. As a result, links are sometimes built into the app by the vendor. Such links are often special URLs that can be extracted through imprint analysis. Second, we follow the app pages on Google Play, which often direct to device vendors, to crawl potential firmware files. Specifically, we used Google Custom Search API to programmatically search through vendor websites for firmware image files.

For the files collected, we filter out non-firmware files by checking their format using *Binwalk* [2]. Binwalk is a well-known firmware unpacking tool which extracts various data from a binary blob through pattern matching. Once a file is decided to be a firmware, a special effort is made to correlate firmware version with app version. As we will discuss in Section 3.3, this helps us to decide at which version a particular vulnerability is fixed and whether or not that fix has an impact on the app.

Note that not all device firmware could be downloaded. Even for the ones that we collected, there is still a considerable amount of firmware encrypted or obfuscated that renders the analysis difficult. This is a limitation yet to overcome in vulnerability confirmation, as discussed in Section 4.2.

3 Dataset and Results

3.1 Dataset and Platform Statistics

Dataset. In total, our dataset comprises of 2,081 apps collected through the method described in Section 2.2. The average size of the apps is 13MB (Min. 23KB and Max. 142MB). These apps spread globally (271 languages) and have a total download exceeding 1.2 billion. The apps cover 1,345 different device vendors and, by our estimate, about 4,720 different device models. We note that this dataset is still incomplete: by comparing certain types of devices in our dataset (e.g., IP camera) against online lists of devices [13, 21–23] of the same type, we estimate the dataset to cover ∼5-20% of the total IoT companion apps.

Testbed and App Processing. Our app analysis platform ran on a 4 Core, 3.33GHz Ubuntu 16.04 server with 16 GB RAM and 1TB hard drive. The Android emulator is compiled from Android Open Source Project, AOSP 4.4.4.

In total, our platform needed ∼68.3 hours to process the 2,081 apps, with an average processing time of 118.2 seconds per app. In our experiment, we set the maximum processing time to 10 minutes and the majority of the apps are processed successfully within this time frame. The platform was not able to fully analyze 73 (3.5%) apps within the timeout win-

dow and therefore only partial analysis results are available for them. In addition, 43 (2.1%) apps were not analyzed by the platform because the tool we used (i.e., Soot) to build CDG and DDG failed to handle the app bytecode during interpretation. Overall, about 98% of the apps were either fully or partially analyzed.

One practical concern is the obfuscation of the app and its impact on the analysis. As reported in previous study, the majority (85.8%) of the device companion apps are produced by the standard tool in Android SDK (i.e., Proguard) [53], which mangles the apps by renaming classes, methods and fields. While Proguard introduces skews to the fuzzy hash analysis, it does not affect our main analysis method (i.e., network interface analysis) since it does not obfuscate network APIs, data-flow and control-flow. Another concern is the packing of the app—some developers use packers to encrypt their code, which would also have an impact on the network interface analysis. However, consistent with observations made by prior research [53], packers are often seen in malware, and less adopted by benign apps. In our dataset, only a handful of apps used commercial packers. Currently, we did not apply any special processing to these apps. There is an orthogonal line of research on developing better unpacking tools(e.g., DexHunter [59] and PackerGrind [57]) and our platform can be supplemented by these tools.

Table 2: IoT device families

Type	Number of Families	Covered Apps	Covered Vendors
Software	19	139	122
Rebranding	28	156	104
Hardware	14	61	51
Protocol	40	271	210
Backend	48	460	422

Device Family. Table 2 shows all the device families detected via our cross-app analysis. For example, we were able to identify 19 distinct device families covering 122 different vendors and 139 apps that were using similar software within the family. As another example, we were able to detect 14 distinct device families covering 51 different vendors that were using similar hardware components within the family. Note that these families are not mutually exclusive; a device might share software components with one device and hardware components with another. The largest device family we identified includes 31 device vendors and the smallest device family includes only 2 device vendors. Figure 5 shows a more intuitive illustration of the device family map.

3.2 Results Validation

Our platform is solely based on code analysis of mobile companion apps, without requiring the physical devices or their firmware images. This is the key to a large-scale security analysis of smart home IoT devices. However, the drawback

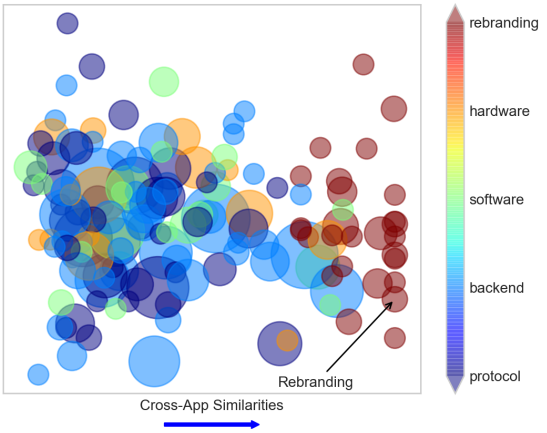


Figure 5: Device family map. Circle size indicates the number of device vendors in the family (the largest circle covers 31 vendors, while the smallest covers two).

of such approach is the accuracy of the result: the output from this analysis (e.g., a family of devices impacted by a particular vulnerability) is a conjecture that points to potential security issues that need to be validated with real devices.

In this paper, we validate and report some of the results we obtained from our analysis to demonstrate the value of the approach. We took a hybrid validation route, taking into consideration practical limitations such as the budget. First, we try to acquire the real device and test it in a local environment (Figure 6 shows the devices we purchased for validation). Second, if we do not have the device, we try to simulate, or in some cases statically analyze, device firmware stored in the Device Firmware Database (built from the method discussed in Section 2.5). Third, if neither of the first two methods applies, we search through online reports including vendor manuals and websites, bug reporting forums, IoT hacking communities and so on. Fourth, we work with the vendor and request their help in validating the results. We primarily used the second and third method, as the first method is very expensive and the fourth method is often a black hole (i.e., no responses from vendor). Upon confirming our findings, we also try to estimate the impact of the finding by searching the online presence of the device on *Shodan* [4].

Ethics. Testing vulnerabilities and scanning real-world devices often bring up serious ethical concerns. In our study, we pay special attention to not cross legal and ethical boundaries. For both real and simulated devices, we evaluate the device in a local network that only allows outbound connections. The device is brought offline immediately after the experiment to avoid being exploited and used as a bot. To evaluate the impact of a particular security issue, we collect data from existing results of *Shodan*, instead of scanning vulnerable



Figure 6: Smart home IoT devices for vulnerability validation

devices directly. In this way, we don't introduce extra network scans. Most importantly, we release our findings to all affected vendors, and refrain from including the real name of any device that is still un-patched or under investigation.

3.3 Results Overview

We present our findings from the perspective of threats, by showing how many smart home IoT devices are potentially impacted by a given vulnerability or security weakness. However, it is also possible to look at the findings from a device's perspective, i.e., for a specific device, what kind of vulnerability or security weakness it may suffer from. The results are encouraging: we identified 324 device models from 73 vendors that are potentially vulnerable to a number of security issues. For the devices that we can confirm or disapprove, about 91% are confirmed to be vulnerable. The total number of users of these devices is estimated to be over 11.1 million.

3.3.1 Vulnerable Software

To demonstrate how software vulnerabilities propagate across devices, we applied our analysis to five high profile vulnerabilities (shown in Table 3) that were reported in GoAhead web server which many smart home IoT devices utilize to provide a web-based interface. These vulnerabilities range from authentication bypass to backdoor account to remote code execution. We started from mobile companion app *object.liouzx.client* of *NEO Coolcam* IP Camera, which was known to be vulnerable to these vulnerabilities, and utilized the cross-app analysis to identify devices that might be similar in their software. Since these are relatively old vulnerabilities (reported in 2017), we expected fewer results. However, in total we still identified 72 device models belonging to 16 distinct vendors that share similar software as the vulnerable device. To validate the results, we utilized the methodology discussed in Section 3.2. We were able to confirm through online reports that 45 device models from four vendors are indeed vulnerable. Since these results are already publicly disclosed, we included them in Table 3. Additionally, we confirmed through manual firmware analysis that six device models from three IP camera vendors, *Vendor A*, *Vendor B* and *Vendor C*, are also impacted by these vulnerabilities. We have informed those vendors about the vulnerabilities but no patch is released yet. Furthermore, we confirmed through real device that one baby monitor device from *Vendor D* is also impacted by the vulnerabilities. Vendor D has asked us to refrain from including their names until further investigation is done on their side. In total, we confirmed the existence of the vulnerabilities on 52 device models from eight different device vendors, with *seven device models from four vendors newly discovered*.

While validating the results, we also encountered one case where our platform mistakenly flagged a device as vulnerable. The analysis results output by the platform show that three device models produced by *KGUARD* have very similar interfaces with other vulnerable devices. However, our manual validation on the real device as well as emulated firmware shows that the *KGUARD* devices are not vulnerable. We inspected the firmware, and found that the software configuration of *KGUARD* is indeed very similar to the vulnerable devices. Specifically, we found that 26 out of 31 CGI programs and web pages are in common. But the vulnerable code was removed. Our hypothesis is that since these devices are relatively new on market (i.e., after the vulnerability was reported), *KGUARD* may have customized the software configuration and patched the vulnerability before releasing the product to the market.

We also want to highlight two observations we made during the process of results validation. First, although the vulnerabilities are old, we are still seeing a large set of devices to be potentially vulnerable. A Shodan search shows that there are potentially 58,456 devices running in the wild still being vulnerable and the total number of app downloads is more than 282,000 times. This is the result of a dataset with merely $\sim$2K apps. With a large-scale analysis covering more device models and vendors, the problem can be even worse. This demonstrates a common issue in the smart home IoT market: the market is highly fragmented and many smaller vendors never bother taking care of the device after selling the device.

Another observation to highlight is the difficulty to validate the results, or rather the general challenge of evaluating the security of an IoT device. We were *not* able to validate findings on 17 device models from seven out of the 16 vendors, despite that the cross-app analysis tells us they might also be vulnerable. The validation was not successful for a number of reasons: some devices are not targeting U.S. market therefore we could not easily acquire, some devices do not provide firmware download therefore we could not evaluate, for devices that we could download firmware the encryption and packing render analysis difficult. In addition, the collaboration with device vendors have been very difficult. Many times,

Table 3: IoT devices impacted by vulnerable software and device rebanding

CVE	Impacted Vendor	Device Models	Validation Method	Mobile App	App Downloads	Confirmation Status
CVE-2017-8221 CVE-2017-8222 CVE-2017-8223 CVE-2017-8224 CVE-2017-8225	IP Camera Vendor A	4	Firmware	App A	100,000+	Newly Discovered
	IP Camera Vendor B	1	Firmware	App B	5,000+	
	IP Camera Vendor C	1	Firmware	App C	100,000+	
	Baby Monitor Vendor D	1	Device	App D	10,000+	
	Instar	2	Reports	camviewer.mobi.for_instar	1,000+	Independently Reconfirmed
	VStarcam	35	Reports	vstc.GENIUS.client	1,000+	
	Sricam	6	Reports	object.shazx1.client.yi object.smartmom.client	55,000+	
	Conceptronic	2	Reports/Firmware	camviewer.p2pwificam.client	10,000+	
	KGUARD	3	Device	object.kguard.client	10,000+	FP
	7 Vendors	17	N/A	7 apps	146,000+	Pending Confirmation

our email request about potential security issues went into a black hole (i.e., no responses ever received from the vendor). We believe this is also an artifact of market fragmentation as smaller vendors tend to care less about the security of their products.

3.3.2 Device Rebranding

Investigation of a more recent vulnerability, *CVE-2018-11560*, leads to another interesting finding of device rebranding. This vulnerability was initially reported in Insteon IP Camera 2864-222 (firmware 1.4.1.9), where the embedded web server on the device had a missing bounds check when parsing CGI parameters, resulting in a stack buffer overflow. We used the companion app of Insteon IP camera as the input to our Cross-App Analysis Engine to detect if any other devices might be vulnerable to the same vulnerability. To our surprise, we found that almost identical device interfaces are provided by a major IP camera vendor, *Foscam*. We initially suspected that the same web server might be used by both vendors, but later through research we found that Insteon IP camera 2864-222 is actually a rebranded version of Foscam IP Camera FI8918W—it is based on the exact same hardware and software but with a different brand name. Not surprisingly, early versions of Foscam IP camera also suffer from the same vulnerability, but no one has reported that.

The interesting part, however, is that Foscam actually patched the vulnerability *before* the vulnerability was reported in Insteon IP camera. We examined the firmware history of Foscam IP camera and found that the vulnerable code was shipped in over eight Foscam firmware versions before Jul. 2017, impacting at least 15 Foscam models. In firmware updates (2.x.1.120) of Jul. 2017, the vulnerability was patched. However, this patch never made it to the Insteon IP camera until the vulnerability was reported in 2018. We contacted Foscam about this issue, but their response neither confirmed nor denied the finding. Instead, we were advised to update to the newest version of firmware. This highlights another interesting issue about smart home IoT devices. Due to the fragmented market, smaller IoT vendors sometimes do not develop their own products. Instead, they customize IoT devices from OEMs and resell with their own branding. This complicates the security management of the product and puts customers in danger, as vulnerabilities in upstream vendors tend to propagate to a broader set of downstream vendors but security patches are not. Indeed, a Shodan search with the IP camera fingerprints (e.g., server type, time stamps) shows that although Foscam released patches as early as Jul. 2017, there are still 30.7% (10,210 out of 33,230) devices that are not patched to the secure version.

Additionally, our analysis shows that re-branding is indeed not uncommon. With a dataset of ∼2K apps, we identified 27 other re-branded device families not including Foscam example. Examples of these devices include smart plugs from *Bayit* and *Orvibo*, Wi-Fi sockets from *CHITCO* and *EDUP* and so on. Further validation is needed to confirm if these devices inherit any vulnerabilities from upstream vendors.

3.3.3 Vulnerable Hardware

Different IoT device vendors may rely on a common hardware module (e.g., Wi-Fi, Bluetooth), which, if vulnerable, could impact multiple devices. The challenge, however, is that IoT device vendors often do not publicize the hardware components in use. As a result, it is often difficult to decide if a device is vulnerable due to a vulnerable hardware component without tearing apart the physical device or unpacking the firmware to examine the driver code.

Through cross-app analysis, we identified a total of *166 devices belonging to 35 different vendors* that are potentially impacted by two recent security weaknesses found in hardware. In one example, a recent study [36] demonstrated that Hi-Flying Wi-Fi module (HF-LPB100, HF-LPT100, HF-LPB200) can be leveraged by an adversary to steal home network Wi-Fi credentials. The Hi-Flying Wi-Fi module is a self-contained 802.11b/g/n module used by a number of IoT devices to provide wireless interfaces. As an important feature, the module supports credential (e.g., SSID, password) provisioning from device companion app to IoT device via *SmartLink*. As reported by the study [36], the provisioning process may leak Wi-Fi credentials: an adversary could passively listen to the traffic and gather the home Wi-Fi network credentials without much effort. Through our cross-app analysis, we identified that 26 apps, covering 108 devices from 21

vendors are potentially impacted by this security weakness. These apps have been downloaded more than 158,000 times. In another example, ESP8622, a low-cost Wi-Fi microchip that appears in many cheap IoT devices (e.g., Wi-Fi controller, smart plug), was reported to have a similar vulnerability in its *ESP-Touch* provisioning protocol. In our analysis, we identified that 21 apps covering 58 devices from 14 distinct vendors are potentially impacted by the security weakness. In total, these apps have been downloaded more than 120,000 times.

Among the devices flagged by the platform, we were able to confirm that 67 devices from 16 vendors are indeed impacted by the security weaknesses (43 devices from eight vendors are confirmed through vendor response. 24 devices from eight vendors are confirmed through firmware emulation, real device or online reports.). Through vendor response, we were also able to identify that seven devices from two vendors were mistakenly flagged by the platform as vulnerable (i.e., ∼9% false positive rate). We manually examined the two apps to analyze the reason for the false positive. For one case, 14 devices supported by the *Revogi* app were flagged by the platform as potentially vulnerable. However, four of them (Power Plug SOW324, Power Strip SOW321 and SOW323, and Smart Light LTW311) were not actually using the vulnerable hardware. The issue was due to the imprecision of the static analysis performed by the platform. Since the app supports multiple devices from the same vendor, the code modules that control individual devices are not clearly distinguishable (i.e., some modules are shared across devices but others are independent). As a result, the platform was not able to attribute the network interfaces that correspond to the vulnerable hardware to a specific device. Instead, the platform outputs all the devices supported by the app as potentially vulnerable. For another case, three devices supported by the *smanos* app were flagged by the platform by mistake. The devices were found not to be using the vulnerable hardware, but the code module and the corresponding network interfaces that control the hardware was included in the app. This may be due to that the app developer built the app upon some open source templates that contain the hardware module, or maybe the device vendor changed their hardware configuration during the device development process, but the app code was never cleaned up. Nevertheless, the IPE method used by the platform is guided by static analysis to construct network interfaces as long as a code snippet is reachable from an Android *activity*, even though that activity may never be actually triggered by the real device.

3.3.4 Vulnerable Protocol

Similar to hardware components, IoT device vendors often do not publicize the protocols that a device speaks. These protocols range from more open and standard ones such as UPnP, mDNS and SSDP to proprietary ones such as TDDP[5] used for debugging, penetrating private networks and various other purposes. Not knowing which protocol a device can speak creates a great security challenge of managing the device, especially when the protocols are found to be vulnerable or can be leveraged by an adversary to launch attacks.

Through cross-app analysis, we can identify devices that speak the same protocol, thus may suffer from similar security problems. For example, previous research [37] showed that SSDP protocol can be abused by adversaries in order to launch DDoS attacks. SSDP queries such as "ssdp:all" and "upnp:rootdevice" may result in a response size orders of magnitude larger, thus if openly accessible to the Internet may serve as a reflector to amplify requests sent by the attacker. Through cross-app analysis, we identified *39 devices from 11 different vendors* that speak SSDP, despite that few of them clearly documented the protocol that their devices speak. As a result, once these devices are activated in the environment where a firewall is not configured to block incoming queries, they may act as reflectors for DDoS attacks. It's difficult to tell the exact number of devices that are exposed and vulnerable, but the total app downloads (over 10.2 million) indicate that a massive number of devices could possibly be harnessed by attackers.

We validated the results output by the platform. In total, we were able to confirm that 18 devices from six vendors are indeed speaking the SSDP protocol. One device, *Bixi* gesture controller, was mistakenly flagged by the platform. The case with *Bixi* gesture controller is interesting: the device itself does not speak SSDP, but its companion app does, therefore causing false positive for the platform. The reason is that the gesture controller is a device that allows users to control other devices via gesture. It does not speak SSDP but relies on its companion app to use SSDP to discover subsidiary devices for it to control. In this case, the network interface of the app is not an exact mirror of the device interface, causing false positives in the platform.

3.3.5 Vulnerable Backend Service

IoT devices may rely on the same IoT cloud backend service to relay command and control (e.g., to penetrate private home networks). When the backend service contains a security weakness, multiple IoT devices using the same service are impacted at the same time. However, without detailed knowledge of the registered customers of the cloud service, many of these impacted devices are left vulnerable until the problems are independently discovered.

Our cross-app analysis can help address this issue. In a particular case, the security weakness was initially reported on DeepSec 2017 [38], where an IoT cloud backend service is found to be using very short device IDs (i.e., only six digits) to register IoT devices. Consequently, any IoT device that is using the service to relay commands and control is vulnerable to device ID and password enumeration attacks. A successful

[5] TDDP stands for TP-Link Device Debug Protocol.

attack may enable attackers to authenticate to the device and abuse the device as a bot. We used the vulnerable device reported in DeepSec, *Yoosee*, as the seed for cross-app analysis and found *32 devices from 10 different vendors* also rely on the same vulnerable backend to relay command and control. While it is hard to estimate the actual number of devices in the wild that are vulnerable, the total amount of downloads of these apps is over 226,000 times.

Among the 32 devices flagged by the platform, we were able to confirm that 12 devices from seven vendors are indeed sending requests to the specific backend server, and the device IDs are indeed enumerable (i.e., 6-digits). We also found that four devices from one vendor, namely *secrui*, were mistakenly flagged by the platform. The reason is similar to the "dead code" issue we encountered while validating results for devices with vulnerable hardware: we found that *secrui* app embedded a self-contained app *com/jwkj* that talks to the problematic backend server and thus the app interfaces exhibit similarity with those that are vulnerable. However, the embedded app was never actually executed nor did the devices supported by *secrui* app actually talk to the problematic backend server.

3.4 Accuracy of Results

In total, the platform flagged 324 devices from 73 vendors as potentially vulnerable, and we were able to confirm that 164 devices from 38 vendors are indeed vulnerable. This accounts for roughly *50.6%* of all the devices flagged by the platform. During the process of validation, we were also able to identify that 15 devices from 5 vendors were mistakenly flagged by the platform as vulnerable. This accounts for *8.4%* of all the devices that we could either confirm or disapprove (i.e., false positives). Table 4 enumerates the reasons for the false positives and the number of instances of each reason. The first reason for the false positive is the existence of the patch. After vulnerabilities were disclosed, vendors may patch the device. In this case, the app-device interface may stay largely the same, but the device is no longer vulnerable. This is a fundamental limitation of the approach, as the platform is designed to only extract information from the app, not the device. Thus, if the patch does not have any impact on the app, the approach cannot differentiate a vulnerable device from a patched one. The second reason for false positive is the “dead code” inside of the apps. Sometimes the apps may contain code that was *not* actually being used by the device (legacy code, code adopted from elsewhere without cleaning and etc.). Statically, it is difficult to decide if the code will ever be triggered and executed at runtime. Our platform currently may mistakenly include analysis results from such "dead code" if the "dead code" exhibit similarities with other vulnerable devices, thus causing false positives. The third reason for the false positive is the imprecision of the static analysis. Currently, the static analysis techniques used by the platform are not precise enough to attribute network interfaces to individual devices if a single app supports multiple devices and these devices share much common control logic inside of the app. This issue, as well as the “dead code” issue listed above, are not a fundamental problem with the approach. Rather, they are an artifact of the static analysis techniques we used to analyze the apps in the platform. We are currently working on improvements to the techniques to improve the accuracy and precision of the static analysis. Finally, we encountered an exception case to our approach where the app interface is not an exact reflection of the device interface (i.e., the *Bixi* gesture controller). However, due to the nature of the device (i.e., a device that controls other devices), it is uncommon among the IoT devices. We are exploring the Google Play store to identify if there are any more devices of the similar kind.

Table 4: Reasons for the false positives

Reason	# of Devices	# of Vendors
Existence of the patch	3	1
Dead code inside of the apps	7	2
Fail to attribute interfaces to individual devices	4	1
Difference in device and app interface	1	1

4 Discussion

4.1 Miscellaneous Findings

During the process of analyzing the interfaces between apps and devices, we have some interesting observations, which are presented here.

Confusing Trust Model. We observed that IoT device developers sometimes have a confusing, if not conflicting, trust assumption regarding the local environment that their devices will run in. On one hand, they seem to assume that the local environment (i.e., consumer’s home network) is not trustworthy. They apply encryption and authentication to protect the communication between the app and the device. On the other hand, they place an excessive amount of trust on the app, e.g. they would embed the encryption keys and authentication credentials into the app. In such a scenario, an adversary within the local environment can easily bypass the protections that the device developer built, as long as the adversary has access to the Google Play Store and has a basic knowledge of reverse engineering an app. As an example, TP-Link Smart Plug (HS110) accepts commands from its mobile companion app (and potentially anywhere else from within the LAN) without authentication. The vendor seems to be concerned about local threats to this design and, therefore, encrypts the communication. However, the encryption key in use (i.e., integer 171 XOR message) is simple and static, and most importantly built into the app. Anyone with access to the app can thus forge the

communication easily. This problem is also reported by [50]. Another example is the D-Link water sensor. D-Link water sensor requires authentication from its mobile companion app. However, the credentials used to authenticate the app is fixed (i.e., not configurable by user) and built into the app. These examples highlight the confusing mindset of many IoT device developers and the lack of general understanding of security. While in this paper we do not intend to give solutions to the problem, we believe a more standard architecture developed with security in mind can help limit the freedom offered to developers thus improving the security.

"Convenient" Provisioning. Smart home IoT devices are often headless—they do not provide direct user interfaces (e.g., touch screen, keyboard). As a result, they often rely on mobile companion apps to provision the credentials of home Wi-Fi network, in order for them to join the network. Our observation through studying the device interfaces is that the provisioning method is evolving from more user interactive approaches (e.g., AP Mode, WPS and out-of-band channels such as Bluetooth) to a more automated and hands-off approach where users do not need to do anything except providing the credentials. This presumably provides convenience, but many times at the cost of security. These newer methods such as *Smart Config* [30] and *Sound Wave*[6] often artificially create a side channel between the app and the device, and rely on these channels to transfer Wi-Fi credentials. Unfortunately, these side channels are publicly observable therefore allowing the credentials to be leaked. In addition, even without considering the openness of side channels, securing a side channel can often be much more difficult than normal means of communication. This highlights the long-lasting problem of balancing usability with security.

4.2 Limitations and Future Work

The major limitation of the approach discussed in this paper is the accuracy of the analysis results. As we based our analysis solely on mobile companion apps, we are inherently limited to the information we can obtain from the app, and sometimes the information we can obtain may not be an accurate reflection of the device. For example, a device may have patched a vulnerability and the patch did not change the device interfaces at all. In this case, our analysis will still output the device as potentially vulnerable since our platform would have no clue about the existence of the patch by just inspecting the app. This, however, is a trade-off we have to make in order to study IoT device security at scale. We believe a multi-stage solution can help address this limitation where the first stage (i.e., our platform) narrows down the scope of analysis by identifying the potentially vulnerable devices, and the second stage automates the vulnerability confirmation with more targeted but rigorous analysis, e.g., dynamic/static analysis of firmware, device fuzzing.

Another limitation of the approach is that the network interface analysis can be rendered less effective in scenarios where IoT backend servers or cloud significantly decouple device interfaces from app interfaces. An example is the Google and Amazon devices where much of the management is done through the cloud. In this case, our approach can glean less information about the device software. However, information such as the Wi-Fi credential provisioning module and the backend services in-use are still available in the app, which allows the platform to predict security issues of these components.

This work could also benefit greatly from an automatic vulnerability collection system. Currently, this is a manual process: we manually collect the vulnerabilities and impacted devices that were reported publicly. We then propagate the vulnerability information to more devices through our platform. An automatic vulnerability collection system can help label the initial seed devices as well as evaluating security from a device's perspective (i.e., to find the set of security issues that a given device may have).

Another aspect to improve on is the dimension and granularity of the similarity analysis, as mentioned in Section 2.4. Further improvements to the App Analysis Engine may allow the platform to detect similarities in finer components of a device software stack (e.g., web server, PHP interpreter, web application, OS, driver) as well as other dimensions (e.g., similar developer, similar development toolchain). This would enable us to track vulnerability propagation more comprehensively and accurately. We leave the refinement of the App Analysis Engine for future work.

The general methodology, i.e., utilizing companion app analysis to study the device, also enables a number of interesting applications that we plan to explore for the future work. For example, from app analysis, we could potentially tell what types of sensors are on a device and what types of network traffic a device may produce. This would allow a home security gateway, which is shipped as a default component of many Wi-Fi routers on the market, to enforce an accurate protection profile and detect anomalous behaviors of IoT devices in real time. As another example, instead of similarities, the Cross-App Analysis Engine in our platform can detect differences between devices. Such differences may enable a more accurate fingerprinting method of the device.

5 Related Work

IoT device vulnerabilities. IoT devices are affecting increasing number of users in every aspect of their life. Meanwhile, various studies revealed that firmware of many devices is filled with vulnerabilities. For instance, Paleari [44, 45] reported that D-Link DIR-645 routers expose critical web pages to unauthenticated remote attackers, allowing them to extract

[6] Wi-Fi credentials are encoded in the sound wave and sent out directly by the phone. This method is being used by devices such as *360* and *Securenet* IP Cameras.

root credentials and take full control of the device; also multiple web interfaces are affected by stack-buffer overflow that leads to remote code execution. Cui et al. [18] found that attackers may inject malicious firmware modifications to a device while it's updated. With the vulnerabilities and the huge amount of insecure devices [31], there arise a series of large-scale attacks, such as Mirai [35], BASHLITE [1], etc. To better understand the events, researchers have conducted comprehensive study on both features of the known vulnerabilities (and malware) [15] and their propagation [6, 39].

Various analysis approaches have been proposed to identify vulnerable IoT devices. For instance, with a network scanner (i.e., *nmap*), Cui et al. [19] found that over 13% devices are publicly accessible by default credentials. Similarly, using an Internet-wide scanning, Heninger et al. [27] showed that a large amount of TLS and SSH servers on embedded devices are affected by weak certificates and keys. Online services, *Shodan* [4] for example, allow security researchers to identify vulnerable online web services and devices. Such works are effective to find devices with known vulnerabilities and evaluate their impacts, but in many cases fail to catch problems that also appear in other devices.

Further, researchers are utilizing different techniques [16, 20, 24, 25, 47, 49, 55] to statically identify vulnerabilities in the device firmware. A majority of the approaches fall into the broader category of *vulnerability search*: derive *signatures* from known vulnerabilities, and then use them to search in other firmware images. To name a few, Costin et al. [16] conducted a large-scale study by scanning 32k firmware images with simple signatures (e.g., certificates, unique keyword), which is difficult to cover vulnerabilities that are not bound to specific strings. To address the problem, following works that collect robust features from such as I/O behavior of the image binary [47] and control flow graphs [24, 25] were proposed. Instead of extracting signatures from firmware image, Xiao et al. [55] presented an approach that discovers unknown vulnerabilities based on the study of existing security patches. In addition, Davidson et al. [20] built a symbolic execution framework on top of KLEE [9] for detecting vulnerabilities in MSP430 microcontroller family, which is difficult to scale as it needs to customize for specific architectural features of an IoT device. Similarly, Shoshitaishvili et al. [49] showed how symbolic execution and other techniques (e.g., program slicing) work to find authentication bypass vulnerabilities in a firmware image. Corteggiani et al. [14] improved symbolic execution of firmware by incorporating source code semantics, etc. While these approaches are effective, they rely on the static analysis of the firmware images, and therefore are limited in cases where the images are not publicly available or cannot be unpacked. In contrast, our work focuses on finding potential vulnerabilities using analysis of the IoT companion apps, which turns to be scalable, especially for IoT devices of smaller companies.

Another effective approach is dynamic firmware analysis [10, 17, 42, 58]. Zaddach et al. [58] performed dynamic analysis by forwarding I/O access from an emulator to the actual hardware, and further, Muench et al. [42] presented how to orchestrate execution between multiple testing environments. Koscher et al. [34] used an FPGA bridge to allow the emulator full and real-time access to the hardware. While these approaches are accurate, they are not applicable to large-scale analysis because of the lack of budget to obtain the device, and the required effort to figure out the hardware interfaces. To address the problem, Costin et al. [17] built a QEMU-based emulation framework to discover vulnerabilities in web interfaces of an IoT device; Chen et. al [10] presented a full system emulation tool, FIRMADYNE, for Linux-based firmware in order to identify vulnerabilities. These works, however, also rely on feasibility of the firmware, and tend to be affected by the heterogeneous architectures of the firmware. Given the difficulty of fuzzing IoT devices directly [43], Chen et al. [11] proposed a testing method to detect memory corruptions in the device with the assistance from app analysis. Similarly, it requires presence of the physical device, and also fuzzing each individual device is time-consuming. Again, our work is more scalable since it is only based on static analysis of the companion apps, and leverages cross-app similarities as an indicator to find other potentially vulnerable devices.

Mobile app analysis. Dozens of static and dynamic techniques have been presented to analyze mobile apps. Among them, most related to our work are those [12, 32, 48, 60] proposed to collect runtime values in an Android app. These techniques may serve different purposes, e.g., collecting developer credentials [60], harvesting obfuscated/encrypted values for malware detection [48], extracting application imprints from network request [12] and reconstructing format of a protocol [32]. However, they have a basic idea in common: extracting parts of the application code (i.e., slices) that are related to the target (e.g., APIs, variables), and generating target value by only executing the slices. In this study, we designed the request construction with a similar *Instantiated Partial Execution* (IPE) as in [12]. Another related work is *Autoprobe* [56], which collects request probes from the malware, and then using the probes to fingerprint a remote malware server. *Autoprobe* is not applicable in our settings for several unique challenges. For instance, requests of an IoT device would not be triggered automatically because of the absence of the device; mobile companion apps often serve multiple devices, and thus it's difficult to pair the request and response and collect telemetry for each individual device. In our work, the interface analysis engine leverages several techniques that not only triggered the request, but also conducted a modularity analysis after locating the request/response pair.

6 Conclusion

In this paper, we present a platform to accelerate vulnerable device discovery in smart home IoT device market. Different

from previous approaches that examine real IoT devices or firmware images, our platform analyzes mobile companion apps of devices to indirectly detect device similarity and vulnerability propagation across devices, thus making it practical for large-scale analyses. By analyzing 2,081 mobile companion apps, our platform was able to discover 324 devices from 73 vendors that are potentially vulnerable to a number of security issues, out of which 164 devices from 38 vendors are confirmed to be indeed vulnerable.

Acknowledgments

We are grateful to the anonymous reviewers for their insightful comments that greatly helped us improve the paper. In particular, we want to thank our shepherd, Adwait Nadkarni, for his guidance and constructive feedback about this paper. This work is supported in part by NSF CNS-1527141, 1618493, 1801432, 1838083 and ARO W911NF1610127.

References

[1] Bashlite. https://github.com/anthonygtellez/BASHLITE.

[2] Binwalk: Firmware analysis tool. https://github.com/ReFirmLabs/binwalk.

[3] Google play. https://play.google.com/store/apps?hl=en.

[4] Shodan: the search engine for the internet of things. https://www.shodan.io.

[5] Smarthomedb. https://www.smarthomedb.com.

[6] ANTONAKAKIS, M., APRIL, T., BAILEY, M., BERNHARD, M., BURSZTEIN, E., COCHRAN, J., DURUMERIC, Z., HALDERMAN, J. A., INVERNIZZI, L., KALLITSIS, M., ET AL. Understanding the mirai botnet. In *USENIX Security Symposium* (2017), pp. 1092–1110.

[7] BAPTISTE, D. Vulnerability in compiler leads to backdoor in software. https://2018.zeronights.ru/wp-content/uploads/materials/04-Vulnerability-in-compiler-leads-to-stealth-backdoor-in-software.pdf, 2018.

[8] BAYER, U., COMPARETTI, P. M., HLAUSCHEK, C., KRUEGEL, C., AND KIRDA, E. Scalable, behavior-based malware clustering. In *NDSS* (2009), vol. 9, Citeseer, pp. 8–11.

[9] CADAR, C., DUNBAR, D., ENGLER, D. R., ET AL. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI* (2008), vol. 8, pp. 209–224.

[10] CHEN, D. D., WOO, M., BRUMLEY, D., AND EGELE, M. Towards automated dynamic analysis for linux-based embedded firmware. In *NDSS* (2016).

[11] CHEN, J., DIAO, W., ZHAO, Q., ZUO, C., LIN, Z., WANG, X., LAU, W. C., SUN, M., YANG, R., AND ZHANG, K. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. *Proc. 2018 NDSS, San Diego, CA* (2018).

[12] CHEN, Y., YOU, W., LEE, Y., CHEN, K., WANG, X., AND ZOU, W. Mass discovery of android traffic imprints through instantiated partial execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), ACM, pp. 815–828.

[13] CORPORATION, T. R. Thermostat manufacturers. https://www.thermostat-recycle.org/thermostat_manufacturers.

[14] CORTEGGIANI, N., CAMURATI, G., AND FRANCILLON, A. Inception: system-wide security testing of real-world embedded systems software. In *27th {USENIX} Security Symposium ({USENIX} Security 18)* (2018), pp. 309–326.

[15] COSTIN, A., AND ZADDACH, J. Iot malware: Comprehensive survey, analysis framework and case studies. *BlackHat USA* (2018).

[16] COSTIN, A., ZADDACH, J., FRANCILLON, A., BALZAROTTI, D., AND ANTIPOLIS, S. A large-scale analysis of the security of embedded firmwares. In *USENIX Security Symposium* (2014), pp. 95–110.

[17] COSTIN, A., ZARRAS, A., AND FRANCILLON, A. Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security* (2016), ACM, pp. 437–448.

[18] CUI, A., COSTELLO, M., AND STOLFO, S. When firmware modifications attack: A case study of embedded exploitation. *Proc. 2018 NDSS, San Diego, CA* (2013).

[19] CUI, A., AND STOLFO, S. J. A quantitative analysis of the insecurity of embedded network devices: results of a wide-area scan. In *Proceedings of the 26th Annual Computer Security Applications Conference* (2010), ACM, pp. 97–106.

[20] DAVIDSON, D., MOENCH, B., RISTENPART, T., AND JHA, S. Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security Symposium* (2013), pp. 463–478.

[21] DIRECTORY, I. G. Ip cameras. https://directory.ifsecglobal.com/ip-cameras-code004823.html.

[22] EBAY. Smart bulbs. https://www.ebay.com/b/Smart-Bulbs/20706/bn_72322334.

[23] EBAY. Wi-fi smart plugs. https://www.ebay.com/b/Wi-Fi-Smart-Plugs/185061/bn_118504145.

[24] ESCHWEILER, S., YAKDAN, K., AND GERHARDS-PADILLA, E. discovre: Efficient cross-architecture identification of bugs in binary code. In *NDSS* (2016).

[25] FENG, Q., ZHOU, R., XU, C., CHENG, Y., TESTA, B., AND YIN, H. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 480–491.

[26] FREY, B. J., AND DUECK, D. Clustering by passing messages between data points. *science 315*, 5814 (2007), 972–976.

[27] HENINGER, N., DURUMERIC, Z., WUSTROW, E., AND HALDERMAN, J. A. Mining your ps and qs: Detection of widespread weak keys in network devices. In *USENIX Security Symposium* (2012), vol. 8, p. 1.

[28] HOLCOMBE, J. Soho network equipment. https://www.securityevaluators.com/wp-content/uploads/2017/07/soho_techreport.pdf, 2017.

[29] HU, G., YUAN, X., TANG, Y., AND YANG, J. Efficiently, effectively detecting mobile app bugs with appdoctor. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 18.

[30] INSTRUMENTS, T. Simplelink wi-fi smartconfig. http://www.ti.com/sitesearch/docs/universalsearch.tsp?searchTerm=SmartConfig.

[31] INTERNET CENSUS. Port scanning /0 using insecure embedded devices. https://seclists.org/fulldisclosure/2013/Mar/166.

[32] KIM, J., CHOI, H., NAMKUNG, H., CHOI, W., CHOI, B., HONG, H., KIM, Y., LEE, J., AND HAN, D. Enabling automatic protocol behavior analysis for android applications. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies* (2016), ACM, pp. 281–295.

[33] KING, J. C. Symbolic execution and program testing. *Communications of the ACM 19*, 7 (1976), 385–394.

[34] KOSCHER, K., KOHNO, T., AND MOLNAR, D. {SURROGATES}: Enabling near-real-time dynamic analyses of embedded systems. In *9th {USENIX} Workshop on Offensive Technologies ({WOOT} 15)* (2015).

[35] KREBSONSECURITY. New mirai worm knocks 900k germans offline. https://krebsonsecurity.com/2016/11/new-mirai-worm-knocks-900k-germans-offline/.

[36] LI, C., CAI, Q., LI, J., LIU, H., ZHANG, Y., GU, D., AND YU, Y. Passwords in the air: Harvesting wi-fi credentials from smartcfg provisioning. In *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks* (2018), ACM, pp. 1–11.

[37] MAJKOWSKI, M. Stupidly simple ddos protocol (ssdp) generates 100 gbps ddos. https://blog.cloudflare.com/ssdp-100gbps/, Jun. 2017.

[38] MARTIN, B., AND BRAUNLEIN, F. Deepsec 2017 talk: Next-gen mirai botnet – balthasar martin & fabian braunlein. https://blog.deepsec.net/deepsec-2017-talk-next-gen-mirai-botnet-balthasar-martin-fabian-braunlein/, Sep. 2017.

[39] MARZANO, A., ALEXANDER, D., FONSECA, O., FAZZION, E., HOEPERS, C., STEDING-JESSEN, K., CHAVES, M. H., CUNHA, Í., GUEDES, D., AND MEIRA, W. The evolution of bashlite and mirai iot botnets. In *2018 IEEE Symposium on Computers and Communications (ISCC)* (2018), IEEE, pp. 00813–00818.

[40] MICROSOFT. Azure iot hub. https://azure.microsoft.com/en-us/services/iot-hub/.

[41] MIKHAIL KUZIN, YAROSLAV SHMELEV, V. K. New trends in the world of iot threats. https://securelist.com/new-trends-in-the-world-of-iot-threats/87991/.

[42] MUENCH, M., NISI, D., FRANCILLON, A., AND BALZAROTTI, D. Avatar 2: A multi-target orchestration platform. In *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)* (2018), vol. 18, pp. 1–11.

[43] MUENCH, M., STIJOHANN, J., KARGL, F., FRANCILLON, A., AND BALZAROTTI, D. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS 2018, Network and Distributed Systems Security Symposium, 18-21 February 2018, San Diego, CA, USA* (2018).

[44] PALEARI, R. Multiple vulnerabilities on d-link dir-645 devices. http://roberto.greyhats.it/advisories/20130801-dlink-dir645.txt.

[45] PALEARI, R. Unauthenticated remote access to d-link dir-645 devices. http://roberto.greyhats.it/advisories/20130227-dlink-dir.txt.

[46] PASCU, L. The iot threat landscape and top smart home vulnerabilities in 2018.

https://www.bitdefender.com/files/News/CaseStudies/study/229/Bitdefender-Whitepaper-The-IoT-Threat-Landscape-and-Top-Smart-Home-Vulnerabilities-in-2018.pdf.

[47] PEWNY, J., GARMANY, B., GAWLIK, R., ROSSOW, C., AND HOLZ, T. Cross-architecture bug search in binary executables. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015), IEEE, pp. 709–724.

[48] RASTHOFER, S., ARZT, S., MILTENBERGER, M., AND BODDEN, E. Harvesting runtime values in android applications that feature anti-analysis techniques. In *NDSS* (2016).

[49] SHOSHITAISHVILI, Y., WANG, R., HAUSER, C., KRUEGEL, C., AND VIGNA, G. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS* (2015).

[50] STROETMANN, L. Reverse engineering the tp-link hs110. https://www.softscheck.com/en/reverse-engineering-tp-link-hs110/.

[51] VALLÉE-RAI, R., CO, P., GAGNON, E., HENDREN, L., LAM, P., AND SUNDARESAN, V. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers* (2010), IBM Corp., pp. 214–224.

[52] WANG, X., ZELDOVICH, N., KAASHOEK, M. F., AND SOLAR-LEZAMA, A. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 260–275.

[53] WERMKE, D., HUAMAN, N., ACAR, Y., REAVES, B., TRAYNOR, P., AND FAHL, S. A large scale investigation of obfuscation use in google play. In *Proceedings of the 34th Annual Computer Security Applications Conference* (2018), ACM, pp. 222–235.

[54] XIAO, C. Malware xcodeghost infects 39 ios apps, including wechat, affecting hundreds of millions of users. https://unit42.paloaltonetworks.com/malware-xcodeghost-infects-39-ios-apps-including-wechat-affecting-hundreds-of-millions-of-users/.

[55] XIAO, F., SHA, L.-T., YUAN, Z.-P., AND WANG, R.-C. Vulhunter: a discovery for unknown bugs based on analysis for known patches in industry internet of things. *IEEE Transactions on Emerging Topics in Computing* (2017).

[56] XU, Z., NAPPA, A., BAYKOV, R., YANG, G., CABALLERO, J., AND GU, G. Autoprobe: Towards automatic active malicious server probing using dynamic binary analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 179–190.

[57] XUE, L., LUO, X., YU, L., WANG, S., AND WU, D. Adaptive unpacking of android apps. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)* (2017), IEEE, pp. 358–369.

[58] ZADDACH, J., BRUNO, L., FRANCILLON, A., BALZAROTTI, D., ET AL. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *NDSS* (2014).

[59] ZHANG, Y., LUO, X., AND YIN, H. Dexhunter: toward extracting hidden code from packed android applications. In *European Symposium on Research in Computer Security* (2015), Springer, pp. 293–311.

[60] ZHOU, Y., WU, L., WANG, Z., AND JIANG, X. Harvesting developer credentials in android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks* (2015), ACM, p. 23.

All Things Considered: An Analysis of IoT Devices on Home Networks

Deepak Kumar[‡] Kelly Shen[†] Benton Case[†] Deepali Garg[◁]
Galina Alperovich[◁] Dmitry Kuznetsov[◁] Rajarshi Gupta[◁] Zakir Durumeric[†]
[†]*Stanford University* [◁]*Avast Software* [‡]*University of Illinois Urbana-Champaign*

Abstract

In this paper, we provide the first large-scale empirical analysis of IoT devices in real-world homes by leveraging data collected from user-initiated network scans of 83M devices in 16M households. We find that IoT adoption is widespread: on several continents, more than half of households already have at least one IoT device. Device types and manufacturer popularity vary dramatically across regions. For example, while nearly half of North American homes have an Internet-connected television or streaming device, less than three percent do in South Asia where the majority of devices are surveillance cameras. We investigate the security posture of devices, detailing their open services, weak default credentials, and vulnerability to known attacks. Device security similarly varies geographically, even for specific manufacturers. For example, while less than 17% of TP-Link home routers in North America have guessable passwords, nearly half do in Eastern Europe and Central Asia. We argue that IoT devices are here, but for most homes, the types of devices adopted are not the ones actively discussed. We hope that by shedding light on this complex ecosystem, we help the security community develop solutions that are applicable to today's homes.

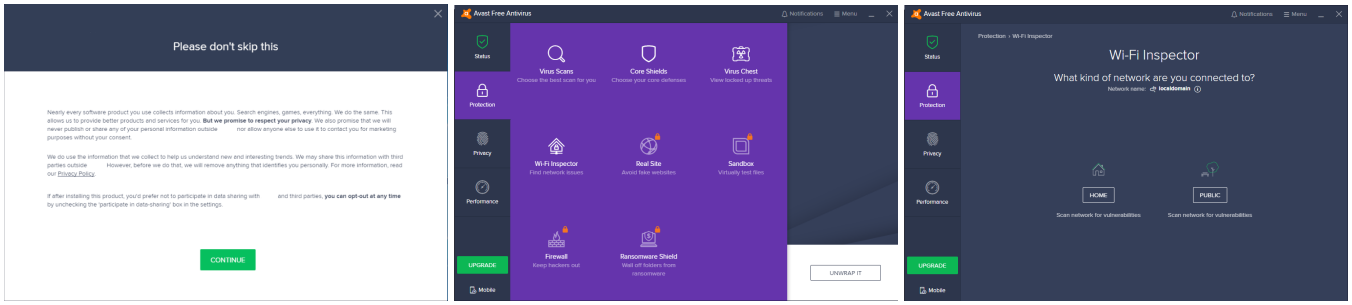
1 Introduction

The weak security posture of many popular IoT devices has enabled attackers to launch record-breaking DDoS attacks [4], compromise local networks [43, 57], and break into homes [22, 41]. However, despite much attention to IoT in the security community [22, 23, 29, 33, 55], there has been little investigation into what devices consumers are adopting and how they are configured in practice. In this work, we provide a large-scale empirical analysis of 83M IoT devices in 16M real-world homes. We partner with Avast Software, a popular antivirus company, whose consumer security software lets customers scan their local network for IoT devices that support weak authentication or have remotely exploitable vulnerabilities. Leveraging data collected from user-initiated network scans in 16M households that have agreed to share data for research and development purposes, we describe the current landscape of IoT devices and their security posture.

IoT devices are widespread. More than half of households have at least one IoT device in three global regions and in North America more than 70% of homes have a network-connected device. Media devices like smart televisions are most common in seven of eleven global regions, but there is significant variance otherwise. For example, surveillance cameras are most popular in South and Southeast Asia, while work appliances prevail in East Asia and Sub-Saharan Africa. Home assistants are present in more than 10% of homes in North America but have yet to see significant adoption in other markets. There is a long tail of 14K total manufacturers, but surprisingly we find that 90% of devices worldwide are produced by only 100 vendors. A handful of companies like Apple, HP, and Samsung dominate globally, but there also exist a set of smaller vendors with significant regional adoption. For example, Vestel, a Turkish manufacturer, is the third largest media vendor in North Africa and the Middle East, but has negligible broader adoption.

A surprising number of devices still support FTP and Telnet with weak credentials. In Sub-Saharan Africa, North Africa, the Middle East, and Southeast Asia, around half of devices support FTP and in Central Asia, nearly 40% of home routers use Telnet. Similar to the regional differences in device type and manufacturer popularity, there are dramatic differences in the use of weak credentials. For example, while less than 15% of devices with FTP allow weak authentication in Europe and Oceania, more than half do in Southeast Asia and Sub-Saharan Africa. Interestingly, this is not entirely due to manufacturer preference. While less than 20% of TP-Link home routers allow access to their administration interface with a weak password in North America, nearly half do in Eastern Europe, Central Asia, and Southeast Asia. About 3% of homes in our dataset are externally visible and more than half of those have a known vulnerability or weak password.

Our results indicate that IoT is not a security concern of the



(a) **Data Sharing Consent** (b) **WiFi Inspector Drawer** (c) **WiFi Inspector Initiation**

Figure 1: **WiFi Inspector**—WiFi Inspector allows users to scan their local network for insecure IoT devices. Data sharing back to Avast for research purposes is an explicit part of the installation process, and presented to the user in plain English. For ease of reading, we duplicate the text shown in panel (a) in Appendix A.

future, but rather one of today. We argue that there already exists a complex ecosystem of Internet-connected embedded devices in homes worldwide, but that these devices are of different than the ones considered by most recent work. We hope that by shedding light on the types of devices consumers are purchasing, we enable the security community to develop solutions that are applicable to today's homes.

2 Methodology and Dataset

Our study leverages several network vantage points, including data collected from Avast, a passive network telescope, and active Internet-wide scans. In this section, we discuss these datasets and the role they play in our analysis.

2.1 WiFi Inspector

Avast Software is a security software company that provides a suite of popular antivirus and consumer security software products like *Avast Free Antivirus*. Avast software is sold on a freemium model: the company provides a free basic version of their product and charges for more advanced versions. Avast estimates that their software runs on 160 M Windows and 3 M Mac OS computers, and makes up approximately 12% of the antivirus market share [45].

As of 2015, all antivirus products from Avast include a tool called *WiFi Inspector* that helps users secure IoT devices and other computers on their home networks. WiFi Inspector runs locally on the user's personal computer and performs network scans of the local subnet to check for devices that accept weak credentials or have remotely exploitable vulnerabilities. Scans can also be manually initiated by the end user. WiFi Inspector alerts users to security problems it finds during these scans and additionally provides an inventory of labeled IoT devices and vulnerabilities in the product's main interface (Figure 1). We next describe how WiFi Inspector operates:

Network Scanning To inventory the local network, WiFi Inspector first generates a list of scan candidates from entries in the local ARP table as well through active ARP, SSDP, and mDNS scans. It then probes targets in increasing IP order over ICMP and common TCP/UDP ports to detect listening services.[1] Scans terminate after the local network has been scanned or a timeout occurs. After the discovery process completes, the scanner attempts to gather application layer data (e.g., HTTP root page, UPnP root device description, and Telnet banner) from listening services.

Detecting Device Types To provide users with a human-readable list of hosts on their network, WiFi Inspector runs a classification algorithm against the application-and transport-layer data collected in the scan. This algorithm buckets devices into one of fourteen categories:

1. Computer
2. Network Node (e.g., home router)
3. Mobile Device (e.g., iPhone or Android)
4. Wearable (e.g., Fitbit, Apple Watch)
5. Game Console (e.g., XBox)
6. Home Automation (e.g., Nest Thermostat)
7. Storage (e.g., home NAS)
8. Surveillance (e.g., IP camera)
9. Work Appliance (e.g., printer or scanner)
10. Home Voice Assistant (e.g., Alexa)
11. Vehicle (e.g., Tesla)
12. Media/TV (e.g., Roku)
13. Home Appliance (e.g., smart fridge)
14. Generic IoT (e.g., toothbrush)

[1] WiFi Inspector scans several groups of TCP/UDP ports: common TCP ports (e.g., 80, 443, 139, 445); TCP ports associated with security problems (e.g., 111, 135, 161); common UDP ports (e.g., 53, 67, 69); and ports associated with services that provide data for device labeling (e.g., 20, 21, 22). When hosts are timely in responding, the scanner will additionally probe a second set of less common ports (e.g., 81–85, 9971). In total, the scanner will target up to 200 ports depending on host performance. The scanner will identify devices so long as they are connected to the network.

Protocol	Field	Search Pattern	Device Type Label	Confidence
DHCP	Class ID	`(?i)SAMSUNG[- :_]Network[- :_]Printer`	Printer	0.90
UPnP	Device Type	`.*hub2.*`	IoT Hub	0.90
HTTP	Title	`(?i)Polycom - (?:SoundPoint IP )?(?:SoundStation IP )?`	IP Phone	0.85
mDNS	Name	`(?i)_nanoleaf(?:api|ms)?\._tcp\.local\.`	Lighting	0.90

Table 1: **Example Device Classification Rules**—Our device labeling algorithm combines a collection of 1,000 expert rules and a supervised classifier, both of which utilize network and application layer data. Here, we show a few examples of these expert rules, which provide 60% coverage of devices in a random sample of 1,000 devices.

We consider devices in the latter eleven categories to be IoT devices for the remainder of this work. Because the classifier greatly affects the results of this work, we describe the algorithm in detail in Section 2.2.

Manufacturer Labeling To generate a full device label, WiFi Inspector combines device type with the device's manufacturer (e.g., Nintendo Game Console). Avast determines manufacturer by looking up the first 24 bits of each device's MAC address in the public IEEE Organizationally Unique Identifier (OUI) registry [32]. We note that at times, the vendor associated with a MAC address is the manufacturer of the network interface rather than the device. For example, MAC addresses associated with some Sony Playstations belong to either FoxConn or AzureWave, two major electronic component manufacturers, rather than Sony. In this work, we manually resolve and document any cases that required grouping manufacturers together.

Checking Weak Credentials WiFi Inspector checks for devices that allow authentication using weak credentials by performing a dictionary-based attack against FTP and Telnet services as well as web interfaces that use HTTP basic authentication. When possible, WiFi Inspector will also try to log into HTTP-based administration interfaces that it recognizes. The scanner attempts to log in with around 200 credentials composed of known defaults (e.g., `admin`/`admin`) and commonly used strings (e.g., `user`, `1234`, `love`) from password popularity lists, leaks, vendor and ISP default lists, and passwords checked by IoT malware. WiFi Inspector immediately notifies users about devices with guessable logins.

Checking Common Vulnerabilities In addition to checking for weak credentials, WiFi Inspector checks devices for vulnerability to around 50 recent exploits that can be verified without harming target devices (e.g., CVE-2018-10561, CVE-2017-14413, EDB-ID-40500, ZSL-2014-5208, and NON-2015-0211). Because there is bias towards more popular manufacturers in these scans, we do not provide ecosystem-level comparisons between different vulnerabilities.

2.2 Device Identification Algorithm

A significant portion of our work is based on identifying the manufacturers and types of IoT devices in homes. We describe the algorithm that Avast has developed in this section:

Classifier WiFi Inspector labels device type (e.g., computer, phone, game console) through a set of expert rules and a supervised classification algorithm, both of which run against network and application layer data. Classification is typically possible because manufacturers often include model information in web administration interfaces as well as in FTP and Telnet banners [4]. Additionally, devices broadcast device details over UPnP and mDNS [14]. WiFi Inspector uses expert rules—regular expressions that parse out simple fields (e.g., telnet banner or HTML title)— to label hosts that follow informal standard practices for announcing their manufacturer and model. This approach, while not comprehensive, reliably identifies common devices [4, 21]. WiFi Inspector contains approximately 1,000 expert rules that are able to identify devices from around 200 manufacturers. We show a sample of these rules in Table 1. However, these rules only identify 60% of devices from a random sample of 1,000 manually-labeled devices. To categorize the remaining devices, WiFi Inspector leverages an ensemble of four supervised learning classifiers that individually classify devices using network layer-data, UPnP responses, mDNS responses, and HTTP data. Therefore, when identifying a device, WiFi Inspector first tries the expert rules, and in the case of no match, next applies the ensemble of four supervised classifiers.

The network classifier is built using a random forest, which aggregates the following network features of a device:

1. MAC address
2. Local IP address
3. Listening services (i.e., port and protocol)
4. Application-layer responses on each port
5. DHCP `class_id` and hostname

The UPnP, mDNS and HTTP classifiers leverage raw text responses. The classifier treats each response as a bag-of-words representation, and uses TF-IDF to weight words across all responses. This representation is fed as input to a Naïve Bayes classifier.

Training and Evaluation To train the supervised algorithm, Avast collected data on approximately 500K random devices from real-world scans. 200K of these were manually classified through an iterative clustering/labeling process, where experts clustered devices based on network properties

Classifier	Coverage	Accuracy	Macro F1
Supervised Ensemble	0.91	0.95	0.78
Network	0.89	0.96	0.79
UPnP	0.27	0.91	0.37
mDNS	0.05	0.94	0.25
HTTP	0.14	0.98	0.23
Final Classifier	0.92	0.96	0.80

Table 2: **Device Classifier Performance**—Our final classifier combines the supervised classifier and expert rules, and achieves 92% coverage and 96% accuracy against a manually labeled set of 1,000 devices.

Region	Homes		Devices	
North America	1.24 M	(8.0%)	9.2 M	(11.1%)
South America	3.2 M	(20.9%)	18 M	(21.6%)
Eastern Europe	4.2 M	(27.2%)	18.8 M	(22.6%)
Western Europe	2.9 M	(19.1%)	15 M	(18.0%)
East Asia	543 K	(3.5%)	3 M	(3.7%)
Central Asia	107 K	(0.7%)	500 K	(0.6%)
Southeast Asia	813 K	(5.3%)	3.6 M	(4.3%)
South Asia	824 K	(5.3%)	6.6 M	(7.7%)
N. Africa, Middle East	1.2 M	(7.5%)	6.1 M	(7.3%)
Oceania	124 K	(0.8%)	680 K	(0.8%)
Sub-Saharan Africa	266 K	(1.7%)	1.8 M	(2.2%)

Table 3: **Regional Distribution of Homes**—The 15.5M homes and 83M devices in our dataset are from geographically diverse regions. Because this breakdown is representative of Avast market share rather than organic density of homes and devices, we limit our analysis to within individual regions.

and labeled large clusters, winnowing and re-clustering until all devices were labeled. The remaining 300K devices were labeled using the expert rules. To tune model parameters, we performed five-fold cross-validation across the original training set. However, because the initial clustering was used to help identify devices in the clustering/labeling step, the dataset is not used for validation. Instead, Avast curated a validation set of 1,000 manually labeled devices, whose labels were never used for training. The final classifier achieves 96% accuracy and 92% coverage with a 0.80 macro average F1 score (Table 2). We mark devices we cannot classify as "unknown".

2.3 Avast Dataset

Avast collects aggregate data about devices, vulnerabilities, and weak credentials from WiFi Inspector installations of consenting users for research and development purposes. Users are informed about this data collection in simple English when they install the product (Figure 1) and can opt out at any time. We worked with Avast to analyze *aggregate data* about the types of devices in each region. No individual records or personally identifiable information was shared with our team. Although WiFi Inspector supports automatic vulnerability scans, we only use data from user-initiated scans in this paper so that we can guarantee that users knowingly scanned their networks. In addition, we exclude scans of public networks by only analyzing networks that were marked as home networks in Windows during network setup. We detail the ethical considerations and our safeguards in Section 2.6.

We specifically analyze data about devices found in scans run between December 1–31, 2018 on Windows installations. This dataset consists of data about 83 M devices from 15.5 M homes spanning 241 countries and territories, and 14.3 K unique manufacturers. For installations with multiple scans during this time period, we use the latest scan that found the maximum number of devices. We aggregate each country into 11 regions, defined by ISO 3166-2 [56]. As shown in Table 3, WiFi Inspector is more popular in Europe and South America than in North America. Because of this market share, as well as significant regional differences in IoT deployment, we discuss regions separately.

Threats to Validity While WiFi Inspector is installed in a significant number of homes, the dataset is likely colored by several biases. First, the data is predicated on users installing antivirus software on their computers. There is little work that indicates whether users with antivirus software have more or less secure practices. Second, we only analyzed data from installations on Windows machines due to differences between Mac and Windows versions of the software. This may skew the households we study to different socioeconomic groups or introduce other biases. Third, WiFi Inspector *actively notifies* users about problems it finds. As a result, users may have patched vulnerable hosts, changed default passwords, or returned devices to their place of purchase. This may skew our results to indicate that homes included in this study are more secure than in practice.

2.4 Network Telescope

While WiFi Inspector scans can identify the types of devices present in home networks, the data does not provide any insight into whether devices have been compromised. To understand whether devices are infected and scanning to compromise other devices (e.g., as was seen for Mirai [4]), we consider the IP addresses scanning in a large network telescope composed of approximately 4.7 million IP addresses. We specifically analyze the traffic for a 24 hour period on January 1, 2019 for scan activity using the methodology discussed by Durumeric et al. [17]: we consider an IP address to be scanning if it contacts at least 25 unique addresses in our telescope on the same port within a 480 second window. In total, we observe 1.7 M scans from a total of 529 K unique IP addresses from 1.4 billion packets during our measurement

period. Of the 500,716 homes scanned by WiFi Inspector on this day, 1,865 (0.37%) were found scanning on the network telescope.

2.5 Internet-Wide Scanning

We further augment the WiFi Inspector data with data collected from Internet-wide scans performed by Censys [16] to understand whether the vulnerabilities present on gateways (i.e., home routers) could be remotely exploitable. Similarly to our network telescope data, we investigate the intersection between Censys and Avast data for a 24-hour period on January 30, 2019 to control for potential DHCP churn. We also check whether devices that accept weak credentials for authentication present login interfaces on public IP addresses. We discuss the results in Section 4.

2.6 Ethical Considerations

WiFi Inspector collects data from inside users' homes. To ensure that this data is collected in line with user expectations, we only collect statistics about homes where the user explicitly agreed to share data for research purposes. This data sharing agreement is not hidden in a EULA, but outlined in simple English. We show the dialogue where users acknowledge this in Figure 1. We note that this is an explicit *opt-out* process. The data sharing agreement is the last message shown to the user before the main menu, meaning users do not need to wait and remember to turn off data collection at a later time.

In order to keep up to date information on the devices in a home, WiFi Inspector runs periodic, automated scans of the local network. Automated scans do not perform any vulnerability testing or password weakness checks; they only identify devices through banners and MAC addresses. We limit our analysis to homes where a user explicitly *manually initiated* a network scan.

To protect user privacy and minimize risk to users, Avast only shared aggregate data with our team. This data was aggregated by device manufacturer, region, and device type. The smallest region contained over 100,000 homes. We never had access to data about individual homes or users; no personally identifiable information was ever shared with us. Avast did not collect any additional data for this work, nor did they change the retention period of any raw data. No data beyond the aggregates in this paper will be stored long term.

Internally, Avast adheres to a strict privacy policy: all data is anonymized and no personally identifiable information is ever shared with external researchers. All handling of WiFi Inspector data satisfies personal data protection laws, such as GDPR, and extends to data beyond its territorial scope (i.e., outside of the European Union). Specific identifiers like IP addresses are deleted in accordance with GDPR and only collected when explicitly necessary for the security function of the product.

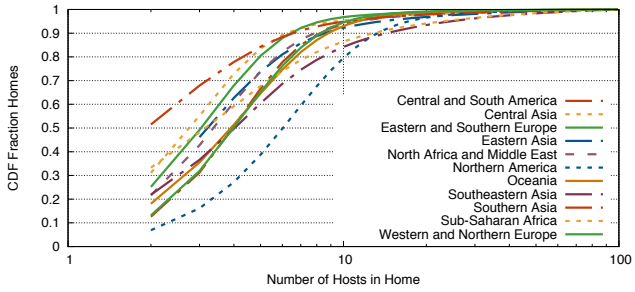


Figure 2: **Devices per Region**—There is significant variance in device usage across regions. The largest presence is in North America, where homes have a median seven hosts. Conversely, homes in South Asia have a median two hosts. The number of devices per home starts at two as all homes require at least one computer and one router to be included.

3 IoT in Homes

It is vital that the security community understands the types of IoT devices that consumers install and their respective regional distributions given their increasing security and privacy implications. In this section, we provide one of first large-scale analyses of these devices based on scans from 15.5 M homes.

The presence of IoT devices varies by region. For example, while more than 70% of homes in North America have an IoT device, fewer than 10% of homes in South Asia do (Figure 2). Media devices (i.e., smart TVs and streaming devices) are the most common type of device in seven of the eleven regions, in terms of both presence in homes (2.5%–42.8%) and total number of devices (16.6%–59.0%). Four regions differ: surveillance devices are most common in South and Southeast Asia, while work appliances are most common in East Asia and Sub-Saharan Africa. We show the most popular devices in each region in Table 4.

Despite differences in IoT popularity across regions, there are strong correlations between regions for the *types* of devices that are popular.[2] In other words, the most popular types of devices are similar across regions. Still, certain pairs of regions differ. For example, homes in all Asian regions are least similar to homes in North America. On the other hand, homes in geographically similar regions (e.g., South Asia and Southeastern Asia) are highly correlated, even when they differ from the global distribution. The fact that distinct regions

[2]To quantify the preference for difference types of devices across regions, we leverage a Spearman's rank correlation test across each pairwise region, taking the rank ordered list of device types for each region as input (Table 5). Per Cohen's guidelines, we find all regions rank ordered distributions are strongly correlated (>0.7 coefficient) with p-values < 0.05 [11], indicating little change in the rank order of device type distributions across regions.

	IoT	Media/TV		Work Appl		Gaming		Voice Asst.		Surveil.		Storage		Automat.		Wearable		Other IoT	
Region	Homes	Homes	Devices	H	D	H	D	H	D	H	D	H	D	H	D	H	D	H	D
North America	71%	42.8	44.9	32.7	28.0	16.0	12.0	9.5	7.5	3.9	3.7	2.7	1.7	2.3	1.9	0.2	0.1	0.4	0.2
South America	34.4%	20.5	51.7	7.5	24.0	4.3	9.8	0.1	0.3	4.6	13.3	0.3	0.6	0.0	0.1	0.0	0.1	0.1	0.2
Eastern Europe	25.7%	16.8	50.2	6.0	23.6	2.7	7.6	0.2	0.6	2.5	14.0	1.2	3.4	0.1	0.4	0.0	0.1	0.0	0.0
Western Europe	57.2%	40.2	59.0	14.0	18.9	7.5	9.2	1.8	2.3	3.8	5.6	2.5	3.2	1.3	1.6	0.0	0.0	0.0	0.0
East Asia	30.8%	12.2	25.8	14.9	44.5	6.3	12.1	0.9	1.6	2.2	9.1	3.1	6.5	0.1	0.2	0.1	0.2	0.0	0.1
Central Asia	17.3%	13.5	54.2	1.6	12.0	0.6	2.4	0.0	0.2	2.4	30.3	0.2	0.8	0.0	0.0	0.0	0.1	0.0	0.0
Southeast Asia	21.7%	9.0	25.4	7.5	31.2	1.0	2.7	0.2	0.5	7.8	37.0	0.9	2.7	0.1	0.2	0.1	0.3	0.0	0.0
South Asia	8.7%	2.5	16.6	2.7	24.2	0.4	2.4	0.1	0.8	4.1	54.5	0.2	1.1	0.0	0.2	0.0	0.2	0.0	0.0
N. Africa, M. East	19.1%	9.4	35.7	5.1	26.2	1.8	6.4	0.1	0.3	5.2	28.5	0.7	2.4	0.0	0.2	0.0	0.2	0.0	0.1
Oceania	49.2%	30.7	46.6	19.8	25.9	10.1	12.7	3.2	4.2	3.0	5.3	3.5	4.3	0.7	0.9	0.1	0.2	0.0	0.0
Sub-Saharan Africa	19.7%	6.9	21.7	10.9	49.9	2.5	7.1	0.1	0.4	2.8	18.0	0.8	2.3	0.1	0.3	0.1	0.3	0.0	0.1

Table 4: **IoT in Homes**—We show the percent of households that have one or more of each type of IoT device and the percent of devices (in gray) in each region that are of a certain type. For example, 42.8% of homes in North America have at least one media device and 44.9% of North American IoT devices are media devices. For the presence of any IoT device, we only report the percent of homes with an IoT device.

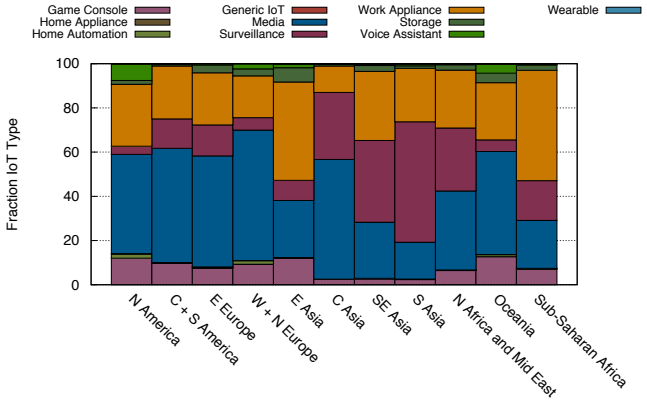


Figure 3: **IoT Device Distribution by Region**—IoT device type distributions vary between different geographic regions. For example, Surveillance devices are most prevalent in Asia, whereas Home Automation devices only appear in North America and Europe.

	N. America	S. America	E. Europe	W. Europe	East Asia	Central Asia	SE Asia	South Asia	N. Africa, ME	Oceania	S-S Africa
North America	–	81	88	92	88	76	77	81	87	93	86
South America	81	–	87	85	90	85	88	87	90	90	92
E. Europe	88	87	–	95	95	93	93	94	98	98	96
W. Europe	92	85	95	–	90	88	83	87	92	95	89
East Asia	88	90	95	90	–	90	93	92	93	98	99
Central Asia	76	85	93	88	90	–	93	90	94	90	93
Southeast Asia	77	88	93	83	93	93	–	99	95	96	95
South Asia	81	87	94	87	92	90	99	–	97	92	95
N. Africa, Middle East	87	90	98	92	93	94	95	97	–	96	95
Oceania	93	90	98	95	98	90	96	92	96	–	96
Sub-Saharan Africa	86	92	96	89	99	93	95	95	95	96	–

Table 5: **Regional Similarities**—We calculate the similarity regions by computing the Spearman's rank correlation test over each region's rank order list of most popular types of devices. We show the most similar region (green) and least similar region (red) by row. Correlation coefficients presented are out of 100. In all cases, p-values were < 0.05.

have unique preferences for device types points to deeper differences between regions, making it harder to reason about IoT in aggregate and more challenging to generalize findings from one region to others.

We also considered the relative popularity of types of devices within each region. Even in areas with similar rank order popularity, the proportion of device types in those regions varies (Figure 3). We compute a pairwise proportion test across each region to quantify the differences between regions and find that nearly all regions have varying proportions of IoT device types, except when a device type accounts for fewer than 1% of devices. We discuss each region below.

3.1 North America

North America has the highest density of IoT devices of any region: 71.8% of homes have an IoT device compared to the global median of 40.2%. Similar to other regions, media devices (e.g., TVs and streaming boxes) and work appliances account for the most devices in North American homes. Nearly half of homes have one media device and one third have a work appliance (Table 4). Media devices are also the most prolific, accounting for 44.9% of IoT devices in North America. In contrast, work appliances only account for 28% of devices (Table 4). There is a long tail of manufacturers that produce media devices in the U.S., and the most popular vendor, Roku, only accounts for 17.4% of media devices (Table 11). Second most popular is Amazon (10.2%). In

contrast, there are only a handful of popular work appliance vendors—HP is the most common and accounts for 38.7% of work appliances in North America.

Though popular in every region, a considerably higher number of homes in North America contain a game console. This is one of the reasons that a smaller fraction of IoT devices are media-related than in Western and Northern Europe. There are three major vendors of game consoles: Microsoft (39%), Sony (30%),[3] and Nintendo (20%).

North America is the only region to see significant deployment of home voice assistants like Amazon Echo [3] and Google Home [25]. Nearly 10% of homes now have a voice assistant and the device class accounts for 7.5% of IoT devices in the region. Two thirds of home assistants are Amazon produced, the remaining one third are Google devices. North America is also one of the only region to see automation devices, which are present in 2.5% of homes. There are four major manufacturers in this space, Nest[4] (44.2%), Belkin (15.1%), Philips (14.4%), and Ecobee (9.8%). These vendors sell products such as the Nest Thermostat [42], Wemo smart plug [5], Philips Hue Smart Lights [46], and the Ecobee Smart Thermostat [19].

The relative ranking of IoT device type popularity generally does not change as more IoT devices are added to North American homes. To quantify this, we calculate the Spearman rank correlation for each pairwise set of homes based on the number of devices and observe only slight deviations from the overall regional distribution. As more devices are added to the network, the correlation coefficients for North America hover between 0.98–1.0, indicating minimal change. Despite minimal change in the relative ranking of IoT device types, we note that the fraction of each device type does vary as more IoT devices are added to the home. For example, for homes with one IoT device, voice assistants make up only 3.9% of all devices, down from 7.3% across all homes. Game consoles are also more popular in homes with only one IoT device, up from 13.9% to 16.5%.

3.2 Central and South America

South American homes are the least similar to North America of any region (Table 5). While the most common types of IoT devices in both regions are media devices (51.7% vs 44.9%) and work appliances (24% vs 28%), significantly fewer South American homes have an IoT device (34% vs 71%) and there are significantly more surveillance devices: 13.3% vs 3.7% of devices (Table 4). Prior research uncovered that there is an increased reliance on surveillance devices in Brazil and surrounding regions to deter violence [27, 34], which may offer one explanation. The only other device type we commonly see are game consoles (9.8% of devices). No other class appears in more than a fraction of a percent of homes.

The vendor distribution of media devices in Central and South America differs from the global distribution. Two vendors appear in the top 5 for this region that do not appear in any other region. First is Arcadyan, a Taiwanese company that primarily manufactures cable boxes in this category, and is often found in LG Smart TVs. The second is Intelbras, a Brazilian company that manufactures DVRs and smart video players. Intelbras accounts for 11% of the surveillance cameras in the region, though they are third to Hikvision and Dahua.

3.3 Europe

Eastern and Western Europe are both most similar to Oceania, primarily due to the three regions sharing a similar fraction of storage devices (Table 4). Still, the regions vary in terms of their IoT usage: 57.2% of Western European homes have at least one IoT device, compared to 25.7% in Eastern European homes.

Manufacturers in Western Europe are similar to the global distribution with a handful of exceptions. Sagemcom and Free, two French companies that sell media boxes and IP cameras, are the first and third largest media vendors in Western Europe, accounting for 15.7% and 9.3% of all devices compared to 5.7% and 3.2% globally. The markets of both companies are highly localized, as 99% of their devices in our dataset are located in Western and Northern Europe. In other device categories, such as work appliances, game consoles, and home automation, there is limited variance from the global distribution. Outside of North America and Oceania, Western Europe is the only other region where more than 1% of homes have a home automation device.

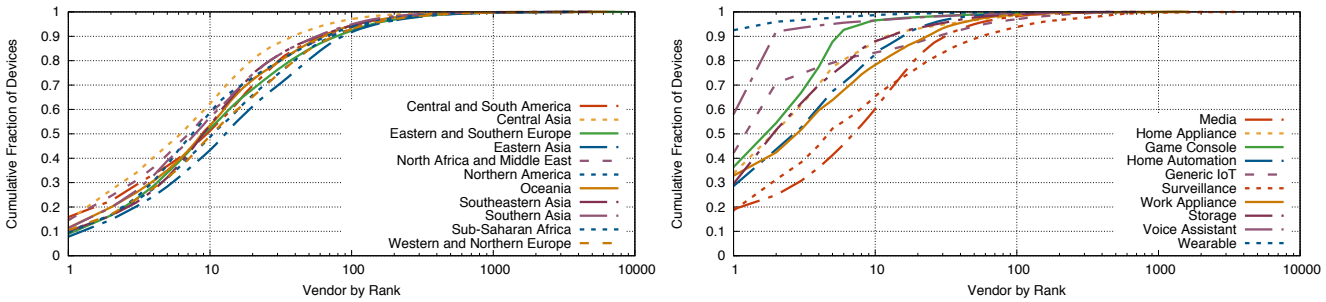
There are significantly more surveillance devices in Eastern Europe than Western Europe (14% versus 5.6% of devices). Eastern Europe is also unlike most other regions in that its rank ordered device type distribution changes as more IoT devices are added over time. For homes with one IoT device, surveillance devices only make up 5.3% of all IoT devices, but this changes drastically for homes with 3 IoT devices, where the number of surveillance devices shoots up to 13.8%. The fraction of surveillance devices continually increases as more IoT devices are added to Eastern European homes. In homes with 10 IoT devices, surveillance devices are the most popular device, accounting for 42.7% of all devices.

3.4 Asia

We analyze the four regions (East, Central, South, and Southeast) of Asia separately as they have different IoT profiles. For example, surveillance devices make up 54.5%, 37%, and

[3] Sony PlayStation devices are split across three vendors in this distribution primarily due to their network cards being manufactured by two third party vendors, Azurewave (11.6%) and Foxconn (9%).

[4] A classification error misclassifies Nest products as mobile devices. We manually correct this in our analysis since Nest does not sell mobile devices.



(a) **Vendors per Region**

(b) **Vendors per Device Type**

Figure 4: **IoT Vendors per Region and Device Type**—There is a long tail of IoT manufacturers worldwide. However, in all regions, 100 vendors account for more than 90% of devices and 400 vendors account for 99%. In contrast, some device types are almost entirely dominated by one or two vendors. For example, Amazon and Google produce 91.9% of voice assistants and Hikvision produces 18.6% of surveillance devices.

30.3% of devices in South, Southeast, and Central Asia (Figure 3), whereas only 9.1% of devices are surveillance related in East Asia. This is not due to a large number of homes with cameras, but rather that other types of IoT devices are sparse. For example, only 9% of S.E. Asian Homes and 2.5% of South Asian homes contain a media device whereas more than 40% homes in North America and Western Europe do. Similar to other regions, Hikvision is the most prevalent vendor of surveillance devices in S.E. Asia and South Asia, making up 25.8% and 34.7% of surveillance devices in each region respectively. Unlike other regions, a private[5] vendor accounts for 15.5% of all surveillance devices in Southern Asia.

East and Central Asia are more similar to Eastern Europe and Africa than they are to South and Southeast Asia. East Asia, for example, is most similar to Sub-Saharan Africa because its largest device type is work appliances, which make up 44.5% of the devices in the region. Central Asia more closely follows Eastern Europe with media devices accounting for 54.2% of devices. All Asian regions do have one thing in common: they are all the least similar to North American homes, indicating fundamental differences in IoT device usage between the Asian countries and North America.

3.5 Africa and Middle East

The North Africa, Middle East (combined) region is most similar to Eastern Europe. Media devices are the most prevalent, appearing in 9.4% of homes and accounting for 35.7% of devices. Again, we observe a local media vendor with a large presence: Vestel, a Turkish TV manufacturer, is the third largest media vendor after Samsung and LG. Surveillance devices make up 28.5% of their overall devices, and appear in 5.2% of homes. Sub-Saharan Africa is distinct in that work appliances are most popular (50% of devices). 11% homes in the region have at least one work appliance. The most popular vendor is HP (33.6%), followed by a long tail of other manufacturers.

[5]Private vendors are ones that have paid an additional fee to IEEE to keep their MAC address mapping off of the public OUI list.

3.6 Oceania

Oceania ranks third to North America and Western Europe in terms of fraction of homes that contain an IoT device (49.2% of homes). Similar to other regions, the most popular device type in the region are media devices, which are found in 30.7% of homes. This is followed by work appliances (19.8% of homes) and gaming consoles (10.1% of homes). Oceania is one of the only regions that contains home automation devices, appearing in 0.7% of homes in our dataset. Similar to North America and Western Europe, Oceania has a moderate number of voice assistant devices, which appear in 3.2% of homes and account for 4.2% of all devices. Unlike North America and Western Europe, homes in Oceania contain many networked storage devices. They account for 4.3% of all devices, which is most similar to homes in Eastern Europe and East Asia.

3.7 IoT Device Vendors

While we find devices from 14.3K unique vendors, 90% of all devices globally are manufactured by 100 vendors (Figure 4a). Globally, there are 4,157 vendors (29%) that only appear in one home. Unlike device type distributions, which are consistent across region, vendor distributions vary heavily across device type (Figure 4b). Some device types are dominated by a small handful of vendors. For example, Amazon and Google account for over 90% of voice assistant devices globally. Other device types like media devices and surveillance devices are split across many vendors. Media devices are the most heterogeneous by vendor: the top 10 vendors only account for 60% of devices.

Device Type	Mean Correlation	Top-10 Mean Correlation
Game Console	**0.43**	**0.49**
Voice Assistant	**0.23**	**0.26**
Home Automation	**0.98**	**0.98**
Surveillance	0.07	**0.28**
Work Appliance	0.04	**0.22**
Storage	0.05	-0.03
Media	0.04	0.09
Router	0.01	0.02
Mobile Device	0.01	0.03

Table 6: **Vendor Correlation by Device Type**—We show the mean correlation in rank ordered vendor distributions per device type across every pair of regions across all vendors as well as the top 10 vendors in each category. The correlations in bold are statistically significant, and indicate consistency in vendors for these device types across all regions in our dataset.

Regional differences in vendor preferences may cause the observed variance in vendor distributions across device types. To measure this, we compute the pairwise Spearman's correlation for each vendor distribution across every pair of regions (e.g. vendor distribution for voice assistants in North America vs. East Asia). We then aggregate[6]over device type by taking the average correlation across each pair of regions (Table 6).

We observe that device types dominated by a handful of vendors globally (Figure 4b) show moderate to strong correlations across all regions, indicating stability in popular vendors across geographic areas. For example, game consoles are dominated by three major players (Microsoft, Sony, Nintendo) in almost every region across the world. In contrast, there are a number of device types, such as media and storage devices, for which there are no correlations across region, even when looking only at the top 10 vendors. This indicates that for these device types, regions have differing vendor preferences. This result aligns with our investigation of individual regions, where we observed many regions prefer local media vendors that are less prevalent in the global distribution.

4 Home Security

Beyond understanding the landscape of IoT devices, we investigate the security profile of devices in homes, including devices that allow weak authentication, the security profile of home routers, and the presence of homes that exhibit scanning behavior on a large darknet.

[6]We note that correlation coefficients are not additive, so to aggregate we convert the respective correlation r-values to z-values using a Fisher's Z transform [13], take the average of the Z values, and convert back to an r-value. In addition, we could only compare rank order for vendors who appeared in all 11 regions in the dataset. There were three device categories (wearables, home appliances, generic IoT) for which no vendors appeared in all regions; we could not compute correlations in these cases.

Port	Service	Devices	Port	Service	Devices
1900	UPnP	46.2%	139	SMB	10.6%
80	HTTP	45.7%	8443	HTTPS Alt.	9.5%
5353	mDNS	39.2%	8009	HTTP Alt.	9.3%
8080	HTTP Alt.	26.9%	445	SMB	8.7%
443	HTTPS	21.1%	7676	Custom	8.2%
9100	JetDirect	19.5%	49152	–	7.9%
515	LPR	16.5%	21	FTP	7.8%
631	IPP	11.8%	5000	UPnP	7.8%
554	RTSP	11.8%	23	Telnet	7.1%
8008	HTTP Alt.	11.1%			

Table 7: **Popular IoT Services**—We show the common open ports in IoT devices in our dataset. The most popular protocols are related to device discovery (UPnP, mDNS) and device administration (HTTP, HTTPS).

Credential	%
admin/admin	88.3%
admin/	5.9%
Administrator/	1.4%
sysadm/sysadm	0.9%
root/	0.7%
root/root	0.4%
user/	0.4%
meo/meo	0.3%
admin/password	0.3%
admin/ttnet	0.3%
other	1.0%

(a) Weak FTP Credentials

Credential	%
admin/admin	35.6%
root/xc3511	16.0%
vodafone/vodafone	10.4%
guest/guest	7.8%
admin/1234	7.5%
root/hslwificam	3.9%
root/vizxv	3.7%
root/oelinux123	2.2%
admin/4321	1.8%
	1.6%
other	9.5%

(b) Weak Telnet Credentials

Table 8: **Most Popular Weak FTP and Telnet Credentials**—`admin/admin` accounts for the 88.3% and 35.6% of the weak FTP and Telnet credentials.

Many IoT devices act as embedded servers: 67.5% of devices provide at least one TCP- or UDP-based service. Many of these services are not surprising—network printers necessarily run services like IPP. However, we also note that devices commonly support older protocols like Telnet (7.1% of IoT devices) and FTP (7.8%). The most common protocol is Universal Plug and Play (UPnP), which is prevalent on 46.2% of devices. We also observe HTTP and mDNS on nearly half of devices. We show the top protocols in Table 7.

4.1 Weak Device Credentials

WiFi Inspector identifies devices that allow authentication with weak default credentials by attempting to log in to FTP and Telnet services with a small dictionary of common default credentials (Section 2). We find that 7.1% of IoT devices and 14.6% of home routers support one of these two protocols.

Region	FTP										Telnet						HTTP
	All IoT		Work Appl.		Surveillance		Router		Storage		All IoT		Surveillance		Router		TP-Link
	Vuln	Sup	Vuln	Sup	Vuln	Sup	Vuln	Sup	Vuln	Sup	Vuln	Sup	Vuln	Sup	Vuln	Sup	Vuln
North America	20.8	5.4	23.4	16.7	6.4	4.6	5.0	4.6	3.2	27.0	0.5	4.8	5.8	9.9	1.3	5.3	16.8
South America	39.0	7.4	42.0	27.8	13.1	2.9	11.9	9.3	4.8	25.9	4.9	8.6	18.9	16.6	1.6	13.2	42.3
Eastern Europe	31.6	9.9	40.7	30.9	9.8	5.8	16.2	12.6	6.6	31.2	3.0	8.9	9.3	19.4	2.3	20.9	48.9
Western Europe	14.7	6.5	23.6	19.9	7.2	5.1	4.4	7.4	5.5	26.4	1.0	4.2	8.1	7.5	2.1	3.3	23.6
East Asia	36.0	17.3	41.5	32.0	6.9	5.5	4.4	7.5	12.2	36.7	0.4	13.8	4.7	13.0	0.9	19.9	23.8
Central Asia	29.5	3.0	64.2	10.2	9.9	2.7	53.9	15.7	3.8	35.1	4.9	6.7	6.4	16.1	7.3	37.6	47.3
Southeast Asia	50.4	7.4	59.5	25.4	7.4	1.4	21.0	14.8	5.8	37.7	3.6	12.1	6.3	12.4	2.0	18.1	43.7
South Asia	33.7	13.4	38.6	36.6	5.4	2.4	6.8	11.1	4.2	35.4	2.9	14.6	7.6	13.7	0.9	19.3	21.4
Oceania	14.7	9.2	16.2	29.9	5.0	4.2	28.2	13.4	6.7	25.0	0.7	7.8	5.7	14.8	0.9	17.1	19.9
N. Africa, M. East	44.6	9.8	53.4	30.4	7.5	2.6	33.7	23.9	8.2	25.9	4.8	11.1	10.5	17.3	1.7	26.6	24.0
Sub-Saharan Africa	55.3	15.4	61.5	27.2	10.8	5.1	23.6	12.5	10.1	35.4	1.1	12.0	5.2	14.1	1.6	20.9	25.4

Table 9: **Weak Default Credentials by Region and Device Type**—We show the weak FTP and Telnet device population by region and device type, highlighting both the fraction of devices that support (Sup) each protocol as well as the fraction that are vulnerable with weak default credentials (Vuln). Some regions have a higher fraction of devices with weak credentials—in the largest case, 50% of FTP devices in Southeast Asia and 4.9% of all Telnet devices in Central Asia are weak. We further observe that the likelihood of having weak FTP credentials is correlated to weak Telnet credentials, indicating that the presence of weak credentials may be linked to weaker security posture in the region overall.

Of those, 17.4% exhibit weak FTP passwords and 2.1% have weak Telnet passwords. In both cases, `admin/admin` is most common and accounts for 88% of weak FTP and 36% of weak Telnet credentials (Table 8). The credential is used by FTP devices from 571 vendors and from 160 Telnet vendors.

Regions vary in terms of vulnerable IoT device populations. In the smallest case, 14.7% of FTP devices in Western Europe support weak default credentials while more than 55% of FTP devices in Sub-Saharan Africa that are weak. A similar, though not as drastic range exists for Telnet. North America has the smallest vulnerable population of Telnet devices (0.5%), Central Asia and South America share the largest vulnerable Telnet population (4.9% of all IoT Telnet devices), primarily because of their reliance on surveillance devices, which have the weakest Telnet profile of all IoT devices.

Nearly all of the IoT devices that support FTP are work appliances (76%), storage (9.1%), media (7.6%), and surveillance devices (5.1%). Media and surveillance devices appear in the list due to their global popularity—unlike storage and work appliances where 29% and 23% of devices support FTP respectively, only 1% of media devices and 4% of surveillance devices support FTP. This aligns with the business need for work and storage devices to facilitate user file transfer, and also explains why there is little variance in the types of devices that support FTP across regions.

Storage devices are the device type most likely to support FTP, though only a small fraction of them use weak credentials. There are two regional exceptions—East Asia and Sub-Saharan Africa (Table 9), which exhibit 12.2% and 10.1% of storage devices with weak credentials respectively. We observe this is primarily due to one vendor, ICP Electronics, which has a large market presence in the two regions: 12.1% and 10.1% of storage devices in East Asia and Sub-Saharan Africa respectively. 74% of ICP storage devices exhibit weak default credentials.

A surprising number of home routers also support FTP (10.2%). TP-Link is responsible for the most routers with weak FTP credentials (Table 10)—regions that rely on TP-Link routers thus have a higher rate of devices with weak FTP credentials. Of all TP-Link devices across all regions, 9.3% offer an open FTP port, and 62.8% of those devices are protected by weak credentials.

Unlike FTP, there is little reason for any IoT devices to support Telnet in 2019. Yet, we find both that surveillance devices and routers consistently support the protocol. Surveillance devices have the weakest Telnet profile, with 10.7% of surveillance devices that support Telnet exhibiting weak credentials. This aligns with anecdotal evidence that suggests that these kinds of devices are easy to hack [4].

4.2 Home Routers

Nearly every home in our dataset has a home router. Similar to most types of IoT devices, there are regional differences and a long tail of vendors globally (Table 9). In total, we see home routers from 4.8 K vendors. TP-Link is the most popular manufacturer globally (15% of routers) and is the top provider in five regions: South America, Central Asia, Eastern Europe, South Asia, and Southeast Asia. Arris is the most popular router vendor in North America (16.4%)—likely because popular ISPs like Comcast supply Arris routers to customers. Huawei is the most popular vendor in Sub-Saharan and North Africa, accounting for 19.8% and 25.6% of all routers respectively.

Vendor	% Open	% Weak	% of Weak
Ricoh	92.1%	71.2%	29.8%
Kyocera	91.7%	97.1%	26%
HP	7.3%	92.4%	24.5%
Sharp	89.4%	94.2%	6.4%
Canon	2.7%	79.3%	2.1%

(a) Work Appliance (FTP)

Vendor	% Open	% Weak	% of Weak
TP-Link	9.3%	62.8%	55.9%
Technicolor	22.9%	20.4%	9.6%
ZTE	9.9%	37.5%	9.5%
MicroTik	46.9%	13.0%	5.3%
D-Link	16.2%	10.9%	3.9%

(b) Router (FTP)

Vendor	% Open	% Weak	% of Weak
D-Link	38.9%	6.1%	33.0%
Huawei	13.6%	4.8%	18.7%
TP-Link	15.0%	1.4%	12.6%
Zyxel	53.5%	2.9%	12.1%
Intelbras	12.7%	26.4%	7.1%

(c) Router (Telnet)

Table 10: **Weak Vendors by Device Type**—We show the vendors that exhibit weak default credentials across each device type in our dataset sorted by the fraction of weak devices they contribute to their respective device types. For example, 71.2% of Ricoh printers that support FTP also support weak default credentials, and these make up 29.8% of all weak work appliances.

Weak FTP/Telnet Credentials More than 93% of routers have HTTP administration interfaces on port 80. We also find that many routers support DNS over UDP (66.5%), UPnP (63.4%), DNS over TCP (42.1%), HTTPS (42.2%), SSH (19.7%), FTP (10.8%), and Telnet (14.6%). Of the devices that support FTP and/or Telnet, 12% have weak FTP and 1.6% have weak Telnet credentials. 1.2% of *all* routers exhibit a weak FTP credential and 0.2% exhibit of all routers have a weak Telnet credential. For FTP, TP-Link routers had the weakest profile: 55.3% of their routers with an open FTP port exhibited a weak credential. For Telnet, D-Link routers were the weakest—6% of all open routers had a weak credential, and 35.3% of all D-Link routers had an open Telnet port. We show a breakdown by region in Table 10.

Weak HTTP Administration Credentials WiFi Inspector attempts to login to the HTTP interfaces for devices from a small number of common vendors, including TP-Link—the most common router manufacturer. In our dataset, there are 3.8 M TP-Link home routers, of which 82% have an HTTP port open to the local network. WiFi Inspector was able to check for weak default credentials on 2.5 M (66%) of the devices with HTTP. Overall, 1.2 M (30%) of TP-Link routers exhibit weak HTTP credentials. Nearly all (99.6%) use `admin/admin`. The number of TP-Link routers with guessable passwords varies greatly across regions (Table 9). For example, only 6% of TP-Link routers in North America have weak passwords while around 45% do in South and Central Asia, and East and South Europe.

External Exposure To understand whether routers with weak default credentials are also exposed on the public Internet, we joined the WiFi Inspector dataset with Internet-wide scan data from Censys [16] for devices on a single day—January 30, 2019.[7] A small number of home routers host publicly accessible services: 3.4% expose HTTP, 0.8% FTP, 0.7% Telnet, and 0.8% SSH. Open gateways are primarily located in three regions—Central America (29.3%), Eastern Europe (20.6%), and Southeast Asia (17.2%). Of routers that are externally exposed, we find that 51.2% of them are exposed with a vulnerability—far higher than the fraction non-externally available routers in our dataset with a weakness or vulnerability (25.8%). The most popular router vendor in these regions is TP-Link, which is also the vendor responsible for the most externally exposed routers (19.7%). We note this is not simply because TP-Link is the largest vendor—a proportion test across regions shows that TP-Link routers appear in the set of externally exposed routers at a higher rate than that of non-externally exposed routers.

[7] We perform this analysis for January because of GDPR restrictions on Avast data.

4.3 Scanning Homes

While scan data can provide insight into the vulnerability of hosts, it typically does not indicate whether hosts have been compromised. We analyzed the homes from WiFi Inspector that were seen performing vulnerability scans in a large network telescope (Section 2) on January 1, 2019 to better understand infected devices. Of the 500.7 K homes that WiFi Inspector collected data from that day, 1,865 (0.37%) homes were found to be scanning for vulnerabilities. Scans most frequently target TCP/445 (SMB, 26.7% homes) followed by TCP/23 (Telnet, 11.3%), TCP/80 (HTTP, 10.7%), and TCP/8080 (HTTP, 9.4%). In addition to checking credentials, WiFi Inspector also checks devices for a handful of recent, known vulnerabilities (CVEs, EDBs, and others). 1,156 (62%) of scanning homes contained at least one known vulnerability—conversely, 7.2 M (46.8%) non scanning homes in our dataset contain at least one known vulnerability. To test the differences between these populations, we used a proportions t-test at a confidence interval of 95%. We observe that the two sets are statistically significantly different (p-value: $2.31 * 10^{-39}$), indicating that scanning homes have a higher vulnerability profile than homes globally. This trend also holds for the number of vulnerable devices in scanning homes (9.7%) compared to homes globally (5.7%). Unfortunately, we were unable to determine why homes without known vulnerabilities were seen scanning. This is likely due to devices being compromised through means outside of our measurement vantage point, for example, vulnerabilities that we do not test for.

Although the overall vulnerability profile of devices in scanning homes is higher, this is not true of all specific vul-

nerabilities. Of the 25 vulnerabilities observed in scanning homes, 17 appeared at a ratio that was not statistically significantly different than devices globally. The remaining eight vulnerabilities were statistically significantly different, though six appear at a *smaller* rate in scanning homes than globally. The two vulnerabilities that appeared at a higher rate in scanning homes were both related to EternalBlue—a leaked NSA exploit targeting SMB on Windows that was primarily responsible for the WannaCry outbreak that impacted millions of Windows devices in 2017 [44]. Specifically, we identify 5.2% of devices within scanning homes that are vulnerable to EternalBlue, and further, 1.3% of devices in scanning homes are *already compromised*, and communicating through a backdoor. This additionally explains some fraction of the SMB scanning we observed on the darknet, as machines compromised via EternalBlue often scan for other hosts running vulnerable SMB servers. We note that although these homes contain vulnerable devices, we cannot claim that they are scanning as a result of these devices—for one, we do not have full vulnerability coverage, and two, it is an outstanding challenge to attribute device behavior from our vantage point. Still, the presence of any scanning homes in general indicates a threat landscape larger than simply publicly accessible devices, and one that should be considered by the security community.

5 Discussion

Recent security research has focused on new home IoT devices, such as smart locks and home automation. Our results suggest that while these devices are growing in importance in western regions, they are far from the most common IoT devices around the world. Instead, home IoT is better characterized by smart TVs, printers, game consoles, and surveillance devices—devices that have been connected to our home networks for more than a decade. Furthermore, these are the kinds of devices that still support weak credentials for old protocols: work appliances are the device type with the highest fraction of weak FTP credentials; surveillance devices are the worst for telnet credentials. Improving the security posture of these devices remains just as important as ensuring that new technologies are secure—our home networks are only as secure as their weakest link.

There are some immediate next steps. As outlined in Section 4, much of the devices that support weak credentials are manufactured by a handful of popular vendors across all regions (Table 10). The security community can start addressing these challenges by encouraging the largest offending vendors to adopt better security practices. On the policy end, law enforcement and legal entities have started to provide legal disincentives for weak security practices. In light of the Mirai attacks, the U.S. Federal Trade Commission has prompted legal action against D-Link [12] for putting U.S. consumers at risk.

A larger question remains on how to address the long tail of vendors. As described in Section 3, regions often have vastly different preferences for vendors across device types. As a result, working to improve the security of devices based solely on the global distribution may inadvertently leave smaller regions with divergent preferences less secure.

Finally, it is not immediately clear how to measure the impact of compromise on home security. In our work, we measured the prevalence of scanning, though this is just one indication of compromise. Furthermore, we only observed 0.37% of homes scanning; amounting to only 1.8 K homes on a single day. In spite of all the data collected within homes in this paper, we could not effectively identify why certain homes were compromised. Researchers have proposed systems to enable auditing of home IoT setups [55, 58], but there is still more work to be done.

6 Related Work

Our work build on research from a number of areas, primarily in home network measurement and IoT security.

Home Network Measurement Early research in home network measurement primarily focused on debugging networks—projects like Netalyzer [35] were conceived to enable users to debug their home Internet connectivity [9,15,49]. A number of follow on papers leveraged Netalyzer-like scans to investigate the state of devices in homes [1, 9, 14], as well to try and understand the implications of a connected home on user behavior [10].

Most similar to our work is presented by Grover et al. who installed home routers with custom firmware in 100 homes across 21 countries to measure the availability, infrastructure, and usage of home networks [26, 53]. Their work focuses on the network properties of home networks on aggregate, and also is able to measure networks continuously based on their position in the network. Our work instead focuses on the *devices* behind the NAT in their ubiquity and their security properties, with particular attention spent on IoT devices.

Recent work has built off of network scanning to enable rich device identification. Feng et al. built a system that leverages application layer responses to perform device identification without machine learning, similar to our hand curated expert rules [21]. This work has built off a number of papers that leverage banners and other host information to characterize hosts [6, 20, 39, 50, 51]. Other rule based engines have been used in other work on active, public scan data based on probing for application banners [4, 16].

Home IoT Security Home IoT security has been of recent interest to researchers in light of its growing security and privacy implications, from the systems level up through the application layer. Ma et al. investigated the rise of the Mirai botnet [4], which was largely composed of IoT devices compromised due to weak credentials and used to launch

massive DDoS attacks. This is not isolated to only attackers—researchers have been breaking the home IoT devices since their conception [8, 31, 38, 47, 59]. Notably, Fernandes et al. outlined a number of challenges in Samsung SmartThings devices, from their access control policy to their third-party developer integration [22]. In response, researchers have built systems to enable security properties in home IoT, such as information flow tracking and sandboxing [23, 33], improving device authentication [54], and enabling auditing information [55, 58]. Most recently, Alrawi et al. synthesized the security of home IoT devices into a SoK, where they present a systematization of attacks and defense on home IoT and outline how to reason about home IoT risk [2].

Internet-Wide IoT Scanning There has been a wealth of recent work that has used Internet-wide scanning for security analysis, including analyzing embedded systems on the public Internet (e.g., [4, 7, 18, 24, 28, 30, 36, 37, 40, 48, 52, 60]). In contrast to these works, we focus on devices inside of homes that are not visible through Internet-wide scanning.

7 Conclusion

In this paper, we conducted the first large-scale empirical analysis of IoT devices on real-world home networks. Leveraging internal network scans of 83M IoT devices in 16M homes worldwide, we find that IoT devices are widespread. In several regions, the majority of homes now have at least one networked IoT device. We analyzed the types and vendors of commonly purchased devices and provided a landscape of the global IoT ecosystem. We further analyzed the security profile of these devices and networks and showed that a significant fraction of devices use weak passwords on FTP and Telnet, are vulnerable to known attacks, and use default HTTP administration passwords that are left unchanged by users. We hope our analysis will help the security community develop solutions that are applicable to IoT devices already in today's homes.

8 Acknowledgements

The authors thank Avast's WiFi Inspector team and the backend data team for their support and insight. The authors also thank Renata Teixeira and David Adrian.

References

[1] B. Agarwal, R. Bhagwan, T. Das, S. Eswaran, V. N. Padmanabhan, and G. M. Voelker. Netprints: Diagnosing home network misconfigurations using shared knowledge. In *9th USENIX Networked Systems Design and Implementation Conference*, 2009.

[2] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose. SoK: security evaluation of home-based iot deployments. In *40th IEEE Symposium on Security and Privacy*, 2019.

[3] Amazon. All things alexa. https://www.amazon.com/Amazon-Echo-And-Alexa-Devices.

[4] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, et al. Understanding the Mirai botnet. In *26th USENIX Security Symposium*, 2017.

[5] Belkin. WeMo smart plug. https://www.belkin.com/us/p/P-F7C063/.

[6] B. Bezawada, M. Bachani, J. Peterson, H. Shirazi, I. Ray, and I. Ray. Behavioral fingerprinting of iot devices. In *2nd ACM Workshop on Attacks and Solutions in Hardware Security*, 2018.

[7] A. Bonkoski, R. Bielawski, and J. A. Halderman. Illuminating the security issues surrounding lights-out server management. In *7th USENIX Workshop on Offensive Technologies*, 2013.

[8] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *25th Networking and Distributed Systems Security Symposium*, 2018.

[9] M. Chetty, D. Haslem, A. Baird, U. Ofoha, B. Sumner, and R. Grinter. Why is my internet slow?: making network speeds visible. In *29th SIGCHI Conference on human factors in computing systems*, 2011.

[10] M. Chetty, J.-Y. Sung, and R. E. Grinter. How smart homes learn: The evolution of the networked home and household. In *9th International Conference on Ubiquitous Computing*, 2007.

[11] J. Cohen. Statistical power analysis for the behavioral sciences, 1998.

[12] F. T. Commission. FTC charges D-Link put consumers' privacy at risk due to the inadequate security of its computer routers and cameras. https://www.ftc.gov/news-events/press-releases/2017/01/ftc-charges-d-link-put-consumers-privacy-risk-due-inadequate.

[13] D. M. Corey, W. P. Dunlap, and M. J. Burke. Averaging correlations: Expected values and bias in combined pearson rs and fisher's z transformations. *The Journal of general psychology*, 125(3), 1998.

[14] L. DiCioccio, R. Teixeira, M. May, and C. Kreibich. Probe and pray: Using UPnP for home network measurements. In *13th International Conference on Passive and Active Network Measurement*, 2012.

[15] L. DiCioccio, R. Teixeira, and C. Rosenberg. Measuring home networks with homenet profiler. In *14th International Conference on Passive and Active Network Measurement*, 2013.

[16] Z. Durumeric, D. Adrian, A. Mirian, M. Bailey, and J. A. Halderman. A search engine backed by Internet-wide scanning. In *22nd ACM Conference on Computer and Communications Security*, 2015.

[17] Z. Durumeric, M. Bailey, and J. A. Halderman. An Internet-wide view of Internet-wide scanning. In *23rd USENIX Security Symposium*, 2014.

[18] Z. Durumeric, E. Wustrow, and J. A. Halderman. ZMap: Fast Internet-wide scanning and its security applications. In *22nd USENIX Security Symposium*, 2013.

[19] Ecobee. Ecobee 4. https://www.ecobee.com/ecobee4/.

[20] X. Feng, Q. Li, Q. Han, H. Zhu, Y. Liu, J. Cui, and L. Sun. Active profiling of physical devices at internet scale. In *25th International Conference on Computer Communication and Networks*, 2016.

[21] X. Feng, Q. Li, H. Wang, and L. Sun. Acquisitional rule-based engine for discovering internet-of-things devices. In *27th USENIX Security Symposium*, 2018.

[22] E. Fernandes, J. Jung, and A. Prakash. Security analysis of emerging smart home applications. In *37th IEEE Symposium on Security and Privacy*, 2016.

[23] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash. Flowfence: Practical data protection for emerging iot application frameworks. In *25th USENIX Security Symposium*, 2016.

[24] B. Ghena, W. Beyer, A. Hillaker, J. Pevarnek, and J. A. Halderman. Green lights forever: Analyzing the security of traffic infrastructure. In *8th USENIX Workshop on Offensive Technologies*, 2014.

[25] Google. Google home. https://store.google.com/au/product/google_home.

[26] S. Grover, M. S. Park, S. Sundaresan, S. Burnett, H. Kim, B. Ravi, and N. Feamster. Peeking behind the NAT: an empirical study of home networks. In *13th ACM Internet Measurement Conference*, 2013.

[27] F. HALAIS. Spectacle and surveillance in brazil. https://www.opendemocracy.net/opensecurity/flavie-halais/spectacle-and-surveillance-in-brazil.

[28] M. Hastings, J. Fried, and N. Heninger. Weak keys remain widespread in network devices. In *ACM Internet Measurement Conference*, 2016.

[29] W. He, M. Golla, R. Padhi, J. Ofek, M. Dürmuth, E. Fernandes, and B. Ur. Rethinking access control and authentication for the home internet of things. In *27th USENIX Security Symposium*, 2018.

[30] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *21st USENIX Security Symposium*, 2012.

[31] G. Hernandez, O. Arias, D. Buentello, and Y. Jin. Smart nest thermostat: A smart spy in your home. *Black Hat USA*, 2014.

[32] IEEE. Registration authority. https://standards.ieee.org/products-services/regauth/oui/index.html.

[33] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, A. Prakash, and S. J. Unviersity. Contexlot: Towards providing contextual integrity to appified iot platforms. In *24th Network and Distributed Systems Security Symposium*, 2017.

[34] M. M. Kanashiro. Surveillance cameras in brazil: exclusion, mobility regulation, and the new meanings of security. *Surveillance & Society*, 2008.

[35] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson. Netalyzr: illuminating the edge network. In *10th Internet Measurement Conference*, 2010.

[36] M. Kührer, T. Hupperich, J. Bushart, C. Rossow, and T. Holz. Going wild: Large-scale classification of open DNS resolvers. In *15th ACM Internet Measurement Conference*, 2015.

[37] M. Kührer, T. Hupperich, C. Rossow, and T. Holz. Exit from hell? reducing the impact of amplification ddos attacks. In *23rd USENIX Security Symposium*, 2014.

[38] D. Kumar, R. Paccagnella, P. Murley, E. Hennenfent, J. Mason, A. Bates, and M. Bailey. Skill squatting attacks on Amazon Alexa. In *27th USENIX Security Symposium*, 2018.

[39] M. Miettinen, S. Marchal, I. Hafeez, N. Asokan, A.-R. Sadeghi, and S. Tarkoma. Iot sentinel: Automated device-type identification for security enforcement in iot. In *37th International Conference on Distributed Computing Systems (ICDCS)*, 2017.

[40] A. Mirian, Z. Ma, D. Adrian, M. Tischer, T. Chuenchujit, T. Yardley, R. Berthier, J. Mason, Z. Durumeric, J. A. Halderman, et al. An internet-wide view of ics devices. In *14th Annual Conference on Privacy, Security and Trust (PST)*. IEEE, 2016.

[41] A. Muravitsky, V. Dashchenko, and R. Sako. Iot hack: how to break a smart home again. https://securelist.com/iot-hack-how-to-break-a-smart-home-again/84092/.

[42] Nest Labs. Nest thermostat. https://nest.com/thermostats/.

[43] L. H. Newman. An elaborate hack shows how much damage iot bugs can do. https://www.wired.com/story/elaborate-hack-shows-damage-iot-bugs-can-do/.

[44] L. H. Newman. The ransomware meltdown experts warned about is here. https://www.wired.com/2017/05/ransomware-meltdown-experts-warned/.

[45] OPSWAT. Windows anti-malware market share report. https://metadefender.opswat.com/reports/anti-malware-market-share.

[46] Philips. Philips hue. https://www2.meethue.com/en-us.

[47] E. Ronen, A. Shamir, A.-O. Weingarten, and C. O'Flynn. IoT goes nuclear: Creating a ZigBee chain reaction. In *38th IEEE Symposium on Security and Privacy (SP)*, 2017.

[48] N. Samarasinghe and M. Mannan. Tls ecosystems in networked devices vs. web servers. In *International Conference on Financial Cryptography and Data Security*, 2017.

[49] M. A. Sánchez, J. S. Otto, Z. S. Bischof, and F. E. Bustamante. Trying broadband characterization at home. In *14th International Conference on Passive and Active Network Measurement*, 2013.

[50] A. Sarabi and M. Liu. Characterizing the internet host population using deep learning: A universal and lightweight numerical embedding. In *18th ACM Internet Measurement Conference*, 2018.

[51] Z. Shamsi, A. Nandwani, D. Leonard, and D. Loguinov. Hershel: single-packet os fingerprinting. In *6th ACM SIGMETRICS Conference*, 2014.

[52] D. Springall, Z. Durumeric, and J. A. Halderman. FTP: The forgotten cloud. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2016.

[53] S. Sundaresan, S. Burnett, N. Feamster, and W. De Donato. Bismark: A testbed for deploying measurements and applications in broadband access networks. In *19th USENIX Annual Technical Conference*, 2014.

[54] Y. Tian, N. Zhang, Y.-H. Lin, X. Wang, B. Ur, X. Guo, and P. Tague. Smartauth: User-centered authorization for the Internet of things. In *26th USENIX Security Symposium*, 2017.

[55] Q. Wang, W. U. Hassan, A. Bates, and C. Gunter. Fear and logging in the internet of things. In *25th Networking and Distributed Systems Symposium*, 2018.

[56] Wikipedia. ISO-3166-2. https://en.wikipedia.org/wiki/ISO_3166-2.

[57] O. Williams-Grut. Hackers stole a casino's database through a thermometer in the lobby fish tank. https://www.businessinsider.com/hackers-stole-a-casinos-database-through-a-thermometer-in-the-lobby-fish-tank-2018-4.

[58] J. Wilson, R. S. Wahby, H. Corrigan-Gibbs, D. Boneh, P. Levis, and K. Winstein. Trust but verify: Auditing the secure Internet of things. In *15th Annual International Conference on Mobile Systems, Applications, and Services*, 2017.

[59] G. Zhang, C. Yan, X. Ji, T. Zhang, T. Zhang, and W. Xu. Dolphinattack: Inaudible voice commands. In *24th ACM Conference on Computer and Communications Security*, 2017.

[60] J. Zhang, Z. Durumeric, M. Bailey, M. Liu, and M. Karir. On the mismanagement and maliciousness of networks. In *Network and Distributed System Security Symposium*, 2014.

A Data Sharing Policy

The first panel in Figure 1 presents users with a text blurb about WiFi Inspector's data sharing policy. For ease of reading, we have copied that text below here:

> Nearly every software product you use collects information about you. Search engines, games, everything. We do the same. This allows us to provide better products and services for you. But we promise to respect your privacy. We also promise that we will never publish or share any of your personal information outside Avast, nor allow anyone else to use it to contact you for marketing purposes without your consent.
>
> We do use the information that we collect to help us understand new and interesting trends. We may share this information with third parties outside Avast. However, before we do that, we will remove anything that identifies you personally. For more information, read our Privacy Policy.
>
> If after installing this product, you'd prefer not to participate in data sharing with Avast and third parties, you can opt-out at any time by unchecking the "participate in data-sharing" box in the settings.

B Device Landscape

	Routers		Gaming		Automation		Storage		Surveillance		Work		Assistant		Media	
N. America	16.4	Arris	39.2	Microsoft	44.2	Nest	24.9	W Digital	12.1	Hikvision	38.8	HP	63.2	Amazon	17.4	Roku
	8.1	Cisco	19.7	Nintendo	15.1	Belkin	14.1	Synology	7.3	Dahua	10.3	FoxConn	32.0	Google	10.2	Amazon
	5.2	Sagemcom	11.6	Azurewave	14.4	Phillips	5.9	Seagate	6.3	D-Link	8.4	Amazon	1.7	Unknown	9.9	Samsung
	4.6	Actiontec	9.4	Sony	9.8	ecobee	3.9	ICP	5.8	Suga	8.0	Epson	0.8	StreamUnlimited	5.9	Apple
	4.3	TP-Link	9.0	FoxConn	2.7	Enphase	3.0	WD	5.3	Flir	7.5	Canon	0.4	Apple	5.8	Google
S. America	22.2	TP-Link	43.7	Microsoft	33.5	Philips	25.0	W Digital	20.8	Hikvision	29.2	HP	39.1	Google	26.0	Samsung
	7.7	Arris	13.6	Sony	13.0	Belkin	14.7	Sagemcom	16.3	Dahua	18.0	Epson	27.5	Amazon	13.6	Arcadyan
	7.0	Technicolor	10.7	Azurewave	12.1	–	13.1	Synology	8.4	–	9.0	FoxConn	6.2	–	7.5	Google
	6.5	Huawei	9.6	FoxConn	5.9	SMA	9.7	D-Link	8.2	Intelbras	7.1	Brother	3.7	TI	6.3	LG
	4.6	Mitrastar	6.6	Nintendo	4.7	Enphase	8.5	Seagate	4.0	Cisco	5.7	Samsung	3.2	Dell	5.0	Intelbras
East Asia	12.9	NEC	45.9	Nintendo	49.0	Philips	37.2	Synology	28.5	Hikvision	13.4	Canon	56.2	Google	8.6	Panasonic
	11.9	Buffalo	21.9	Sony	7.0	Belkin	13.4	Buffalo	10.5	Dahua	11.1	Epson	32.6	Amazon	7.5	Amazon
	8.4	TP-Link	8.9	FoxConn	4.8	Belkin	12.1	ICP	8.6	Dahua	10.6	Moimstone	2.1	Xiaomi	6.9	FoxConn
	5.5	EFM	8.0	Azurewave	4.2	Gongjin Elec	8.8	I-OData	5.0	Panasonic	9.3	FoxConn	0.7	TCL	6.3	Google
	4.4	Huawei	4.9	Microsoft	4.2	SMA	8.2	QNAP	2.4	Bilian	9.2	HP	0.7	Onkyo	5.9	Sony
Central Asia	49.5	TP-Link	22.8	Microsoft	11.1	Fn-Link	37.4	Synology	43.2	Hikvision	23.7	HP	21.3	Amazon	37.2	Samsung
	16.6	Huawei	20.9	FoxConn	11.1	Cambridge	14.0	D-Link	16.2	Dahua	10.0	Yealink	17.0	Amazon	28.6	LG
	6.4	Cambridge	17.7	Azurewave	11.1	TP-Link	13.5	W Digital	11.0	Cisco	9.4	Canon	6.4	D-Link	6.9	FoxConn
	5.3	D-Link	12.5	Sony	–	–	7.7	ICP	6.2	Cisco	7.5	Epson	4.3	M-Cube	–	–
	3.0	ZTE	10.0	Liteon	–	–	4.1	QNAP	3.2	ICP	6.9	XEROX	4.3	TI	–	–
East Europe	23.9	TP-Link	37.3	Microsoft	40.3	Philips	26.7	Synology	20.6	Hikvision	27.7	HP	44.9	Google	30.8	Samsung
	7.3	ZTE	14.7	Sony	25.1	Philips	15.9	W Digital	18.7	Dahua	10.8	FoxConn	23.7	Amazon	17.0	LG
	7.1	Huawei	13.2	FoxConn	5.4	SMA	14.0	Sagemcom	12.0	Cisco	7.1	Canon	7.6	Amazon	5.4	FoxConn
	6.6	D-Link	11.0	Azurewave	3.2	eQ-3	9.7	ICP	4.3	Cisco	5.6	Epson	2.4	TI	4.7	Google
	3.8	Asus	9.5	Nintendo	3.2	Murata	7.6	QNAP	3.4	ICP	4.9	Samsung	2.3	Telemedia	3.3	Neweb
West Europe	18.0	Sagemcom	30.6	Microsoft	33.1	Philips	38.7	Synology	37.1	Free	39.0	HP	48.6	Amazon	15.7	Sagemcom
	16.1	Free	22.5	Nintendo	17.7	Alertme.com	17.7	W Digital	8.0	Hikvision	11.6	Canon	37.2	Google	14.1	Samsung
	5.7	AVM	14.9	Sony	6.1	eQ-3	7.2	ICP	7.0	Hikvision	9.2	FoxConn	6.4	Apple	9.3	Free
	5.2	Huawei	11.5	FoxConn	5.7	Hager	5.7	Technicolor	6.3	Dahua	9.0	Epson	0.7	Apple	8.4	Google
	3.8	TP-Link	8.3	Azurewave	4.8	SMA	4.5	QNAP	5.1	D-Link	4.1	Brother	0.6	Telemedia	6.2	Google
South Asia	24.2	TP-Link	64.9	Microsoft	26.3	Philips	20.1	W Digital	34.3	Hikvision	33.1	HP	44.8	Google	17.1	FoxConn
	7.4	Huawei	8.7	FoxConn	24.1	SMA	14.5	Synology	18.4	Dahua	16.6	Canon	33.8	Amazon	16.9	Samsung
	7.4	D-Link	5.7	Azurewave	14.0	Matrix	14.5	Synology	18.4	Dahua	8.1	FoxConn	2.7	HP	8.3	LG
	7.3	Tenda	3.6	Sony	1.3	Espressif	10.6	Seagate	3.0	Cisco	6.0	Epson	2.5	Dell	6.1	Google
	2.7	Haier	2.0	Nintendo	1.3	Xiaomi	10.3	WD	2.1	ICP	3.6	Ricoh	1.8	Intel	5.5	Neweb
S.E. Asia	18.9	TP-Link	44.6	Microsoft	34.7	Inspur	36.4	Synology	24.7	Hikvision	15.4	HP	49.1	Google	19.7	Samsung
	14.3	Huawei	11.6	Nintendo	18.9	Philips	19.4	W Digital	17.2	Dahua	13.9	FoxConn	21.7	Amazon	10.8	FoxConn
	12.0	ZTE	11.5	FoxConn	18.6	Rf-Link	8.6	ICP	4.8	Cisco	9.7	Epson	2.7	TI	10.6	ZTE
	5.3	Fiberhome	10.2	Azurewave	8.2	SMA	7.5	QNAP	4.0	ICP	9.5	Canon	2.6	HP	10.5	LG
	4.3	Mikrotic	6.5	Sony	2.0	Belkin	6.6	D-Link	3.8	PLUS	7.3	Ricoh	2.3	Dell	4.1	Neweb
Oceania	19.3	Technicolor	43.7	Microsoft	30.3	Philips	21.0	Synology	16.8	Hikvision	23.5	HP	85.3	Google	17.7	Google
	15.4	Huawei	15.0	Nintendo	20.3	Belkin	15.9	HyBroad	13.9	Dahua	19.3	FoxConn	8.0	Amazon	12.2	Roku
	12.1	Sagemcom	11.1	FoxConn	16.4	Lifi	13.1	W Digital	3.7	D-Link	14.1	Epson	1.3	Apple	10.0	Apple
	7.6	TP-Link	10.3	Azurewave	10.1	Enphase	9.5	ICP	3.4	Baichuan	10.2	Canon	1.3	Apple	8.6	Samsung
	4.7	Netcomm	9.3	Sony	6.2	SMA	6.5	Seagate	3.0	Yealink	6.5	Brother	0.6	Liteon	6.8	Sonos
N. Africa, ME	25.6	Huawei	26.0	Microsoft	27.3	Philips	29.1	Askey	19.5	Hikvision	29.4	HP	27.6	Google	20.9	Samsung
	23.2	TP-Link	18.7	FoxConn	10.6	SMA	19.2	W Digital	15.3	Dahua	9.7	FoxConn	21.3	Amazon	17.2	LG
	8.4	ZTE	16.6	Sony	8.1	Lifi	9.7	ICP	5.4	Cisco	7.2	Canon	1.9	Dell	5.4	Vestel
	6.1	D-Link	12.2	Azurewave	3.2	Sercomm	9.1	Synology	4.3	Topwell	4.3	Samsung	1.9	Apple	3.8	Sagemcom
	4.7	Zyxel	7.7	Liteon	2.7	ZTE	7.7	VTech	4.0	ICP	3.9	Konika	1.8	HP	2.7	Apple
S-S Africa	19.7	Huawei	40.7	Microsoft	21.1	SMA	24.7	Synology	39.0	Hikvision	33.6	HP	33.8	Google	24.1	Samsung
	12.0	TP-Link	14.5	FoxConn	17.6	TI	19.2	W Digital	16.3	Dahua	8.5	Canon	28.8	Amazon	7.4	LG
	8.1	Ubiquiti	13.9	Sony	10.8	Philips	10.1	ICP	2.8	Cisco	8.4	Yealink	7.3	HP	7.4	LG
	6.7	Mikrotic	9.7	Azurewave	3.9	HP	9.3	QNAP	2.2	ICP	6.3	FoxConn	2.9	Dell	5.8	Apple
	6.5	D-Link	8.4	Nintendo	2.9	Hager	7.8	Seagate	1.7	PLUS	5.3	Ricoh	2.2	Apple	5.2	Sagemcom

Table 11: **Most Popular Vendor per Region per Device Type**—We show the five most popular vendors per device type across the eleven regions in our dataset. We excluded two device types, wearable and home appliances, as they were barely present in our dataset and splitting up their vendor distribution by region provided only a handful of devices in each region.

Region	FTP								Telnet			
	Work Appliance		Storage		Surveillance		Home Router		Surveillance		Home Router	
N. America	35.3	HP	40.1	ICP	49.8	Axis	63.8	TP-Link	42.9	Dahua	45.2	TP-Link
	13.9	Ricoh	25.2	W Digital	13.4	Vivotek	9.6	ZTE	22.7	PLUS	40.5	Zyxel
	9.3	FoxConn	10.0	QNAP	7.9	Trendnet	8.0	Mikrotic	9.3	Metrohm	4.4	–
	8.4	Kyocera	6.7	TP-Link	4.6	D-link	4.3	–	4.8	–	4.1	Belkin
	7.9	Sharp	5.5	WD	2.9	Creston	2.0	T&W	3.7	Cisco	0.7	Intelbras
S. America	39.6	Ricoh	24.2	W Digital	43.3	Vivotek	40.3	Technicolor	51.4	PLUS	44.5	Intelbras
	25.3	HP	20.2	ICP	30.6	Axis	28.5	TP-Link	10.3	Cisco	21.6	Huawei
	22.8	Kyocera	12.9	QNAP	6.9	Level One	11.1	D-Link	9.8	Metrohm	12.0	BluCastle
	3.3	Sharp	8.1	Cisco	6.5	D-Link	7.4	Mikrotic	8.8	Dahua	5.9	TP-Link
	1.2	Xerox	7.3	La Cie	2.2	Trendnet	4.7	Cameo	3.3	Ralink	5.7	Loopcomm
East Asia	39.9	Ricoh	49.0	I-O	44.3	Vivotek	62.4	TP-Link	43.8	PLUS	45.8	NEC
	17.6	Sharp	25.4	ICP	27.8	Axis	14.1	I-O	11.9	Metrohm	18.6	Hitron
	8.6	HP	7.9	QNAP	8.7	Logitec	6.9	DrayTek	10.8	Dahua	15.6	Huawei
	7.4	Kyocera	7.7	EFM	4.3	Imi	2.9	corega	9.2	ICP	4.9	Buffalo
	5.6	Xerox	1.7	inXtron	3.5	Buffalo	1.8	Mikrotic	4.3	Cisco	4.7	TP-Link
Central Asia	66.4	HP	66.7	ICP	39.1	Axis	92.6	TP-Link	36.0	PLUS	52.3	D-Link
	9.3	FoxConn	33.3	QNAP	17.4	Zhongxi	1.9	Huawei	16.9	Dahua	35.2	Huawei
	11.5	Kyocera	–	–	13.0	Ezvis	1.5	ZTE	11.2	–	8.8	Cambridge
	3.5	Ricoh	–	–	13.0	Vivotek	1.5	Mikrotic	10.1	Metrohm	2.4	TP-Link
	3.1	Xerox	–	–	8.7	–	1.1	Asus	5.6	iStor	0.6	Eltex
East Europe	42.4	Kyocera	53.1	ICP	31.6	Axis	45.8	TP-Link	35.0	PLUS	60.5	D-Link
	25.9	Ricoh	18.2	QNAP	20.3	Ezvis	14.6	ZTE	26.5	Dahua	18.9	Huawei
	23.6	HP	12.5	W Digital	12.5	Vivotek	11.6	Technicolor	12.3	Metrohm	8.6	TP-Link
	3.7	Sharp	3.0	WD	9.0	Zhongxi	8.3	Mikrotic	5.0	Cisco	2.8	Zyxel
	2.4	FoxConn	1.7	La Cie	4.3	D-Link	7.5	Sagemcom	2.5	iStor	1.8	ZTE
West Europe	27.9	Kyocera	49.2	ICP	49.7	Axis	40.8	TP-Link	35.5	PLUS	65.6	Zyxel
	22.2	HP	17.1	W Digital	11.0	Vivotek	20.6	Arcadyan	20.9	Dahua	15.1	TP-Link
	18.8	Ricoh	8.5	QNAP	10.8	Advance Vision	12.4	Technicolor	14.0	Metrohm	13.2	ZTE
	9.5	Sharp	4.1	WD	4.7	D-Link	6.9	AVM	5.8	iStor	0.9	–
	3.5	FoxConn	3.8	Synology	3.9	Hikvision	4.7	Mikrotic	5.0	–	0.8	Winstars
South Asia	51.4	HP	32.4	W Digital	61.5	Matrix	34.7	ZTE	42.1	PLUS	40.3	Smartlink
	19.6	Ricoh	17.6	QNAP	11.5	Axis	26.4	TP-Link	18.4	Dahua	34.7	D-Link
	10.5	Canon	14.7	WD	10.3	D-Link	12.4	D-Link	11.3	Metrohm	5.4	Huawei
	5.6	Kyocera	14.7	ICP	3.8	3DSP	6.8	Fiberhome	8.8	–	2.9	Fida
	4.2	FoxConn	8.8	–	2.6	CardioMEMS	3.0	Binatone	4.6	Cisco	2.6	Zyxel
S.E. Asia	46.6	Ricoh	39.9	ICP	45.1	Vivotek	62.3	TP-Link	45.7	PLUS	36.6	Huawei
	19.5	HP	25.4	QNAP	39.6	Axis	16.6	Mikrotic	16.2	Metrohm	24.6	Zyxel
	6.3	Sharp	11.6	W Digital	3.0	–	7.6	DrayTek	12.6	Dahua	12.3	TP-Link
	6.3	Kyocera	6.5	WD	2.4	Matrix	3.3	Sagemcom	5.3	Cisco	7.5	ZTE
	4.4	Xerox	4.3	I-O	1.2	Level One	1.8	D-Link	5.1	–	5.0	RicherLink
Oceania	21.1	Kyocera	65.1	ICP	30.8	Axis	57.6	TP-Link	35.8	PLUS	91.4	TP-Link
	18.3	HP	15.1	W Digital	30.8	–	32.4	NetComm	18.9	Dahua	2.7	D-Link
	17.9	Ricoh	11.6	QNAP	30.8	Ezvis	3.6	D-Link	13.2	Metrohm	1.4	ZTE
	14.3	Xeros	2.3	–	7.7	Adaptive Recognition	1.8	Billion	9.4	–	0.9	NetComm
	8.7	Sharp	2.3	Cisco	0.0	UTC F&S	1.8	Billion	1.1	–	0.9	–
N. Africa, ME	35.1	Kyocera	58.7	ICP	34.8	Axis	81.9	TP-Link	48.7	PLUS	38.9	TP-Link
	24.1	HP	18.5	QNAP	19.1	Vivotek	5.7	ZTE	16.3	Metrohm	34.2	Zyxel
	23.7	Ricoh	11.4	W Digital	10.3	D-Link	4.8	Askey	11.8	Dahua	19.0	Huawei
	5.6	Sharp	4.9	WD	3.4	Level One	1.7	Boca	4.9	iStor	2.6	D-Link
	5.0	FoxConn	1.6	Xerox	2.9	SMD	0.8	Cameo	3.6	Cisco	1.1	AirTies
S-S Africa	32.1	HP	43.5	ICP	72.5	Axis	30.7	TP-Link	43.4	PLUS	60.0	Zyxel
	28.7	Kyocera	16.5	QNAP	16.7	Vivotek	28.0	D-Link	16.9	Dahua	17.4	Huawei
	26	Ricoh	11.8	W Digital	2.9	Hikvision	22.3	Mikrotic	14.0	Metrohm	7.3	TP-Link
	4.0	FoxConn	7.1	Xerox	2.0	Netcore	6.6	ZTE	5.9	–	4.9	Fida
	3.0	Sharp	5.9	Seagate	2.0	Bosch	4.2	Billion	4.4	iStor	2.2	ZTE

Table 12: **Vendors with Weak FTP and Telnet Credentials by Region**—We show the top five vendors in each device type by region that exhibit weak FTP or Telnet credentials. In most cases, a small handful of vendors are responsible for most of the weak devices.

KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities

Wei Wu[1,2,3⋆], Yueqi Chen[2], Xinyu Xing[2], and Wei Zou[1,3]

[1]{CAS-KLONAT[†], BKLONSPT[‡]}, Institute of Information Engineering, Chinese Academy of Sciences, China

[2]College of Information Sciences and Technology, Pennsylvania State University, USA

[3]School of Cyber Security, University of Chinese Academy of Sciences, China
{wuwei, zouwei}@iie.ac.cn, {yxc431, xxing}@ist.psu.edu

Abstract

Automatic exploit generation is a challenging problem. A challenging part of the task is to connect an identified exploitable state (exploit primitive) to triggering execution of code-reuse (*e.g.,* ROP) payload. A control-flow hijacking primitive is one of the most common capabilities for exploitation. However, due to the challenges of widely deployed exploit mitigations, pitfalls along an exploit path, and ill-suited primitives, it is difficult to even manually craft an exploit with a control-flow hijacking primitive for an off-the-shelf modern Linux kernel. We propose KEPLER to facilitate exploit generation by automatically generating a "single-shot" exploitation chain. KEPLER accepts as input a control-flow hijacking primitive and bootstraps any kernel ROP payload by symbolically stitching an exploitation chain taking advantage of prevalent kernel coding style and corresponding gadgets. Comparisons with previous automatic exploit generation techniques and previous kernel exploit techniques show KEPLER effectively facilitates evaluation of control-flow hijacking primitives in the Linux kernel.

1 Introduction

Software bugs may have extremely serious consequences, especially for the OS kernel, where they could be fatal to the reliability and security of the entire OS because of the higher privilege that the kernel resides in and the abundance of hardware resources that the kernel has direct control of. Kernel bugs can lead to data leakage, privilege escalation and even persistant attacks [33]. One straightforward solution to minimize consequences of kernel bugs is to immediately patch all of the kernel bugs reported via mail lists and kernel fuzzers [72] [58] [52] [37]. In practice, considering the lack of manpower to patch all bugs timely, vendors typically prioritize their efforts to patch the bugs with more severe security implications after assessing their exploitability. With the deployment of various kernel mitigations, the exploitability of bugs has been obviously weakened but still hard to decide. Despite the undecidability of the general exploitability problem, sometimes a carefully crafted exploit could serve as a constructive proof of exploitability.

Capable of proving exploitability by generating working exploits from a vulnerability Proof-of-Concepts (PoC), automatic exploit generation is a preferred choice for exploitability assessment because its soundness and efficiency [7] [3] [9] [64] [5] [55] [30] [75]. More importantly, automatically generating concrete exploits could not only help exploitability evaluation, but also let a user to gain advantages in adversarial settings (e.g. Capture-The-Flag competitions) by scoring fast. Last but not least, these generated exploits could potentially help defender-side to evaluate the effectiveness of proposals of new kernel mitigation.

The common workflow of automatic exploit generation systems are similar. In general we can divide them into the following two steps: ❶*exploit primitive identification* and ❷*exploit primitive evaluation*. In the first step, they search for pre-defined exploit primitive ("exploitable" states) based on the crashing path triggered by a PoC input. In the second step, after pinpointing an exploit primitive, they add exploit constraints and perform *constraint solving* to generate a concrete input to exercise a predefined exploit technique (*e.g.,* `ret2libc` attack).

However, in order to generate working exploit for a control-flow hijacking primitive, there remains to be the following three challenges in the process of *exploit primitive evaluation* which limits the capability of automatic exploit generation techniques to target a complex real-world system such as the Linux kernel.

Challenge 1, exploit mitigation. Exploit mitigations are designed and introduced to reduce attack surface and raise the bar of exploitation. For a modern Linux kernel, many new hardware features [38] [11], compiler-assisted instrumentation [22] [40], sensitive data objects protection [12] [65] [21] [13] and virtualization-based hypervisor [49] [51] have

⋆The main part of the work was done while studying at Pennsylvania State University.

†Key Laboratory of Network Assessment Technology, CAS

‡Beijing Key Laboratory of Network Security and Protection Technology

been introduced as exploit mitigations. As a consequence, many kernel exploit techniques are no longer effective [36] [61] [41] [39] [77], despite the fact that heavier enforcement such as control-flow integrity (CFI) [2] [16] [79] [78] is still *not* widely-adopted by major Linux releases perhaps due to performance concerns.

Challenge 2, exploit path pitfall. Side effects of exploit primitives could terminate exploitation in middle. Memory corruption, occurred along with an exploit primitive, can frustrate the attempt to trigger the primitive the second time, because an *exploit path* could unavoidably contain instructions triggering an unexpected termination of exploitation. Such termination can be a kernel panic of invalid memory access or an infinite loop in a kernel thread.

Challenge 3, ill-suited exploit primitive. Lack of stack pivoting gadget [54] which is a vital step to perform ROP attack and insufficient control over general registers can make an exploit primitive ill-suited. Although some strong primitives have been proven exploitable and even Turing complete [35], there is still a gap between ill-suited exploit primitive and the requirement of mounting a certain exploit technique.

Considering the three challenges, it could be quite difficult to even manually craft an exploit with a control flow hijack primitive. To address the above challenges, we propose KEPLER, an exploit primitive evaluation framework for real-world Linux kernel vulnerabilities. KEPLER employs a novel exploit technique designed for Linux kernel which reduces exploitation of a control-flow hijacking primitive to constructing a classical overflow in kernel stack. The exploit technique exposes less constraints over the quality of an exploit primitive and availability of stack pivoting gadget than previous exploit techniques and could still bypass currently widely deployed kernel mitigations while previous approches could not. Starting from a possibly ill-suited *control-flow hijacking primitive* (CFHP), KEPLER overcomes the lack of stack pivoting gadget and manages to build a "single-shot" exploitation chain to bootstrap existing *Turing complete* exploit techniques such as return oriented programming (ROP) [56], with the ability to bypass mainstream kernel mitigations as well as detouring exploit path pitfalls.

To achieve this, KEPLER leverages a carefully designed code-reuse template for control-flow hijacking primitives to bypass widely-deployed mitigations in Linux kernel. KEPLER first enhances an exploit primitive to satisfy a minimal requirement of controlling dual registers (*e.g.,* rip and rdi for x86-64) at the same time. Starting from the primitive, KEPLER generates an exploit which sequentially executes a chain of five gadgets. The design of the code-reuse template involves several insights about Linux kernel coding style. Specifically, KEPLER reuses those stack-based invocations of kernel I/O channel functions to leak and smash kernel stack of current process and execute arbitrary user-supplied ROP payload. KEPLER leverages *blooming gadget* to enhance control over necessary registers for a control-flow hijacking primitive. KEPLER uses a *bridging gadget* to combine the practice of leaking kernel stack canary and smashing kernel stack into a single shot and thus prevent unexpected kernel panic.

To generate exploits for each CFHP against arbitrary kernel binary, KEPLER operates in the following two phase: first, KEPLER statically analyzes the kernel binary for five categories of candidate gadgets. Then KEPLER starts kernel symbolic execution from the CFHP, and performs a *Depth First Search* (DFS) based gadget stitching algorithm over candidate gadgets.

Our evaluation of KEPLER shows that it is powerful for exploit primitive evaluation. To highlight the effectiveness of KEPLER, we compare KEPLER with existing exploit hardening/generation tools (*e.g.,* Q [60], fuze [75]) and KEPLER outperforms all of them in terms of generating effective kernel exploits under modern mitigation settings in Linux kernel.

This research work makes the following contribution:

- **Kernel single-shot exploitation**. We present a code-reuse exploit technique which converts a single ill-suited control-flow hijacking primitive into arbitrary ROP payload execution under various constraints posed by modern Linux kernel mitigations and the primitive itself. The proposed technique exploits prevalent kernel coding style and corresponding gadgets and thus is hard to defeat. Our approach to calculate exploitation chain is automatable because the gadget stitching problem could be cast as a search problem over a search space of reasonable size. In addition, the "single-shot"[1] nature of this technique makes it suitable for the vulnerabilities prone to unexpected termination because it avoids stressing a control-flow hijacking primitive for multiple times.

- **Semi-automatic exploit generator for Linux kernel**. We implement KEPLER using a set of tools including IDA SDK, QEMU/KVM and angr. Starting from a user-supplied control flow hijack primitive, KEPLER analyzes the Linux kernel binary, tracks down useful kernel gadgets, and automatically generates many gadget chains for launching "single-shot" exploitation and bypassing kernel mitigations. It requires no kernel source code and can be applied to stripped kernel images. KEPLER applies to modern Linux kernels; our evaluation uses version 4.15.0 which was the latest as of the time of our evaluation.

- **Practical impacts**. We systematically evaluate the effectiveness and efficiency of KEPLER using 16 real-world kernel vulnerabilities and 3 recently-released CTF challenges. Given a kernel control-flow hijacking primitive, we show that KEPLER generally could generate tens of

[1] The proposed exploit technique requires a lot of analysis effort, but with respect to the precondition for launching the attack, it requires only a single control-flow hijacking primitive.

thousands of distinct exploitation chains with the ability to bypass kernel mitigations and perform successful exploitation. We show that KEPLER can output the first working exploit for a kernel vulnerability in less than about 50 wall-clock minutes.

2 Background and Related Work

Exploit primitives [6] [47] [55] are machine states that violate security policies at various level and indicate an attacker could get extra capabilities beyond the normal functionality provided by the original program. A `control-flow hijacking primitive` (`CFHP`) is a machine state that potentially deviates from the legal control-flow graph. In the context of symbolic analysis, a control-flow hijacking primitive is usually identified by applying a heuristic which queries the backend constraint solver to check whether the number of possible control flow jump target is beyond a threshold when the control flow jump target contains symbolic bytes. An arbitrary memory write primitive is a machine state that an attacker could modify arbitrary kernel memory on his will. Similarly, an arbitrary memory leak primitive is a machine state which allows an attacker to dump data content at arbitrary kernel address. Sometimes the primitive does not allow an attacker to modify/leak data at arbitray kernel address (*e.g.,* a stack info leak which only dump several bytes on kernel stack [68]), they are referred as restricted memory write/leak primitive.

As is described above, this research work mainly focuses on two aspects – ❶ facilitating exploit primitive evaluation for even ill-suited exploit primitives and bypassing widely-deployed kernel mitigation mechanisms by designing a new exploit technique. ❷ developing a tool to automate the newly proposed kernel exploitation approach. As a result, the works most relevant to ours include those pertaining to exploit primitive identification, exploit primitive evaluation and kernel exploit techniques/mitigations. In the following, we describe the existing works in these three directions and discuss their difference from ours.

2.1 Exploit Primitive Identification

To assist the process of finding a useful exploit primitive (*e.g.,* control-flow hijacking primitive and memory write/leak primitive), there is a rich collection of research works. For example, using preconditioned symbolic execution and concolic execution techniques, Brumley *et al.* develop `AEG` as well as `mayhem` to identify control-flow hijacking primitives for further exploitation. [3] [9] [7]. Shoshitaishvili *et al.* develop a cyber reasoning system `Mechanical Phish` [62]. It is built on angr [64] [69] [63] and performs fuzzing and symbolic tracing for the PoC to identify exploit primitives. To efficiently explore state space for exploit primitives in Linux kernel, Wu *et al.* propose an automatic technique that utilizes under-context fuzzing along with partial symbolic execution to explore `CFHP` and memory write primitive for UAF bugs [75]. To construct better exploit primitives with the capability of out-of-bound access, Heelan *et al.* utilize regression tests to obtain the knowledge of how to perform heap layout manipulation [30]. To obtain better exploit primitives for stack Use-Before-Initialization vulnerabilities, Lu *et al.* propose a deterministic stack spraying approach as well as an exhaustive memory spraying technique [46].

In this work, we do not focus on facilitating primitive identification. Rather, we assume an exploit primitive is already identified and our research endeavor centers around subsequent primitive evaluation phase.

2.2 Exploit Primitive Evaluation

In the primitive evaluation phase, a security analyst or an automatic exploit generation system tries suitable exploit techniques for the seemingly exploitable states identified before.

Initially, without considering Data Execution Prevention, such systems use straightforward techniques such as `ret2stack-shellcode` and `ret2libc` with a `CFHP` [3] [9] [62]. Taking $W \oplus X$ into account, Schwartz *et al.* propose Q [60] to facilitate exploitation with a `CFHP` by automatically constructing a `ROP` chain. Our work addresses the problem of `ROP` bootstrapping without stack pivoting gadget and is orthogonal to automatic `ROP` chaining techniques [33] [60] [66] [24] [57] because we do not tackle the problem of `ROP` payload construction.

With the facilitation of forward and backward taint analysis, Mothe *et al.* devise a technical approach to craft working exploits for simple vulnerabilities in user-mode applications [48]. Utilizing symbolic execution, Repel *et al.* craft exploits with single memory write primitive (*e.g.,* unsafe unlink and lookaside list corruption) for those heap overflow vulnerabilities residing in the userland applications [55]. To facilitate primitive evaluation of kernel Use-After-Free exploitation, Xu *et al.* propose two memory collision mechanisms [77] to unleash `CFHP`s.

Recently, some research works take `CFI` into consideration for primitive evaluation [29] [8] [59] [23] [31] [32] [35]. For example, Ispoglou *et al.* propose block oriented programming (`BOP`) [35] to facilitate evaluation of repeatable arbitrary memory write primitives by proving the Turing completeness under `CFI` along with common userspace mitigations. `BOP` assumes the existence of a dispatcher gadget. `BOP` also automates exploit generation. As a repeatable arbitrary memory write primitive is almost "god-mode" of kernel exploitation, our work facilitates primitive evaluation for those weaker exploit primitives (*e.g.,* non-repeatable and ill-suited `CFHP`) in real-world .

In this work, we also develop a tool for facilitating primitive evaluation. However, this research work is fundamentally different from the aforementioned works in at least one of the following aspects. First, without assuming a perfect ex-

ploit primitive (*e.g.,* unlimited number of invocations of an arbitrary memory write primitive), we can faciliate exploit primitive evaluation for those usually ignored exploit primitive (*e.g.,* ill-suited primitives and primitive that can only be triggered once). Second, rather than dealing with applications in the user space, our tool targets the Linux kernel where exploitation typically involves complicated operations and sophisticated security mitigation mechanisms are generally enabled. Third, rather than generating one single exploit for a target vulnerability, our tool automatically explores many possible exploitation chains and output various working exploits.

2.3 Kernel Exploit Techniques/mitigations

Initially, a `CFHP` in the kernel can directly execute shellcode in user-space because there is no isolation between user and kernel space (*e.g.,* `ret2usr`). Supervisor Mode Execution Prevention (`SMEP`) [38] prevents kernel from executing userspace code. An attacker can use code-reuse attack. To set stack pointer to controlled payload, she uses the prevalent "pivot-to-userspace" gadget to pivot stack to userspace [44]. With adoption of Supervisor Mode Access Prevention (`SMAP` [11]), an attacker can no longer rely on a fake stack in userspace because userspace memory access is forbidden except during I/O channel functions. Because there is usually none intra-kernel stack pivoting gadget for a Linux kernel, an attacker usually chooses to disable `SMAP` by flipping corresponding bits in the `cr4` register [41]. However, the "`cr4`-flipping" attack typically rely on double `CFHP`s [41] and not suitable for a *none re-triggerable* `CFHP`. In addition, a virtualization-based hypervisor can detect `cr4` register modification by inspecting a `vmexit` and thus mitigate such exploitation [49] [51]. `Ret2dir` [39] attack sprays the `physmap` region by calling syscall `mmap`, as the direct mapped physical memory is marked as executable previously, diverting a `CFHP` to land on physmap led to arbitary shellcode exectuion. However, with a kernel patch applied, these `physmap` pages are no longer executable.

To enforce CFI policy for the kernel, several kernel `CFI` solutions [16] [70] [26] have been proposed, however, these mitigations are not broadly adopted by major Linux release version such as CentOS, Ubuntu and Debian.

Data-only kernel exploit techniques directly use a memory write primitive to modify sensitive kernel data objects such as process credentials, page tables and virtual dynamic shared object (vdso) [36]. However, mitigations have been proposed [17] [67] and deployed [40] to prevent these low-hanging fruits.

Note some memory write primitive can be transformed to a control-flow hijacking primitive by overwriting and invoking a code pointer in kernel's data section or in the heap [19].

Kernel address space layout randomization (KASLR) is widely deployed in order to present the attacker an unpredictable attack surface. However, due to its lack of timely re-randomization and coarse-grained nature (only randomizing section base address), an attacker does not even need an arbitrary read primitive [45] [71] to bypass KASLR. With a hardware side channel [34] [28], he can infer the coarse memory layout without leaking any kernel memory content. Despite kernel Page Table Isolation (`KPTI` [13]) removes some side channels with extra overhead, he can also uses a restricted memory leak primitive to infer the coarse memory layout [68]. In the default setting of Linux kernel, knowing coarse memory layout is enough for various exploit techniques. (Kernel) Code diversification/randomization [14] [15] [27] [74] [42] [53] could significantly raise the bar of code-reuse exploitation by thwarting an attacker from locating useful gadget.

3 Assumptions and Threat Model

Our threat model consists of a modern Linux kernel protected by widely-deployed mitigations with a known vulnerability.

Mitigation setting. Similar to recent major Linux release versions, we make the following assumptions of kernel mitigations. A kernel has enabled `SMEP` and `SMAP` [11] protection to prevent direct userspace access in kernel execution. A kernel has enabled stack canary to protect return address over stack for all functions containing local variable [22]. A kernel has enabled protection to prevent direct modification of sensitive kernel data objects including process credential [40] and page table [17]. A kernel has enabled `KASLR`. A kernel has enabled `KPTI` [13] protection. A kernel has been protected by virtualization-based hypervisor which prevents unauthorized modification of `cr4` regsiter [49]. A kernel has set physmap pages as non-executable. A kernel has enabled `STATIC_USERMODEHELPER`. The option routes all `call_usermodehelper()` calls through a guard binary that can properly filter the requested userland programs to be run by the kernel [43]. However, A kernel does `not` enable a `CFI` enforcement such as `RAP` [70] because of performance concerns.
Available Exploit Primitives. We assume there is a PoC which triggers the vulnerability and leads to a control-flow hijacking primitive (`CFHP`). We assume the `CFHP` is already identified with the PoC through either manual analysis or dynamic analysis such as symbolic tracing, thus finding exploit primitives is orthogonal to our work and we can focus on evaluating the `CFHP`. We assume a restricted memory leak primitive to help infer coarse kernel memory layout (*e.g.,* getting the base address of code section `.text` and `physmap` region). We do `not` assume the existence of an arbitrary memory write primitive which could write to arbitrary kernel memory address. We do *not* assume the existence of an arbitrary memory leak primitive that can dump arbitrary kernel memory content. Under the threat model, the content in arbitrary memory address such as the stack canary value of arbitrary kernel thread usually remains secret because the coarse kernel memory layout information does not reveal the stack canary value of a kernel stack. We also assume the location of current kernel stack

remains secret although a restricted memory leak primitive might help leak a pointer to current stack under some specific vulnerability context.

4 Motivating Example

```
1 struct ip_mc_socklist {
2   struct ip_mc_socklist *next_rcu;
3   struct ip_mreqn multi;
4   unsigned int sfmode;
5   struct ip_sf_socklist *sflist;
6   struct rcu_head rcu;
7 };
```

(a) Definition of struct `ip_mc_socklist`. Its first member `next_rcu` is unmanageable because the PoC uses a heap spray technique which does not allow us to control first `QWORD` of struct `ip_mc_socklist`.

```
1  void ip_mc_drop_socket(struct sock *sk){
2    struct inet_sock *inet = inet_sk(sk);
3    struct ip_mc_socklist *iml;
4    // inet->mc_list is a dangling pointer
5    while ((iml = inet->mc_list) != NULL) {
6      // iml is alias of the dangling pointer
7      inet->mc_list = iml->next_rcu;
8      // queuing a rcu_head for execution in
           the future
9      kfree_rcu(iml, rcu);
10   }
11 }
12 void rcu_reclaim(struct rcu_head *head){
13   head->func(head); // control-flow hijack
14 }
15 void rcu_do_batch(...){
16   struct rcu_head *next, *list;
17   while (list) {
18     next = list->next; // next rcu is
           unmanageable
19     rcu_reclaim(list);
20     list = next;
21   }
22 }
```

(b) Tailored source code pertaining to the `CFHP`. function `ip_mc_drop_socket` repeat invoking `kfree_rcu()` which queues a `rcu` task for asynchronous execution until `inet->mc_list` is `NULL`. The site pertaining to the `CFHP` is in function `rcu_reclaim`.

Table 1: A control-flow hijacking primitive in kernel UAF vulnerability CVE-2017-8890.

We illustrate the challenges in evaluating a `CFHP` on x86-64 with CVE-2017-8890 [50], a recent vulnerability in the Linux kernel.

4.1 Vulnerability and Exploit Primitive

The root cause of the bug is an Use-After-Free over an object `ip_mc_socklist` defined in Table 1a. As is shown in Table 1b, the UAF bug results in a dangling pointer `inet->mc_list` and `*iml` become an alias of the dangling pointer in line 5. In line 7, there is an UAF access by dereferencing `iml->next_rcu`. In line 9, `kfree_rcu(iml)` queues a callback denoted by `iml->rcu`. Function `rcu_do_batch` handles the previously queued callback when the CPU gets a chance to process the `rcu` callback list, thus triggers another UAF access to the `ip_mc_socklist` object in `rcu_reclaim()`. The loop from line 5 to line 10 will continue and add another callback if `iml->next_rcu` is not `NULL`.

The `CFHP` is due to an asynchronous `kfree_rcu` call over the dangling pointer `*iml`. To be specific, by manipulating the value in `iml->rcu_head` through a proper heap spray, a security analyst can get a `CFHP` later in `rcu_reclaim()` because function `kfree_rcu()` (line 9) is designed to queue a `rcu` callback denoted by `iml->rcu_head`. Function `rcu_do_batch` will be executed in the future, it will iterate over a list of `rcu_head` added by `kfree_rcu()`. The statement pertaining to the `CFHP` (line 13) allows the attacker to control `rip` (`head->func`) through heap spraying. At the time of the control-flow hijacking, register `rdi` (`head`) points to a controllable memory region.

4.2 Challenges of Crafting Working Exploit

However, developing an exploit with the aforementioned `CFHP` is quite difficult because of the following challenges.

4.2.1 Challenge 1: Exploit Mitigations

Widely deployed kernel mitigations frustrate a large amount of straight forward exploit techniques. With the presence of `SMEP/SMAP` protection, it is impossible to directly launch a traditional `ret2user`/`pivot2usr` attack. The `ret2dir` attack [39] is also not suitable because the `physmap` region is no longer executable [10]. With the write-protection over sensitive data such as process credential and page table , it is not possible to direct overwrite these data to escalate priviledge by converting the `CFHP` into a memory write primitive.

A security analyst may think of the "`cr4`-flipping" attack which usually requires two `CFHP`s: one for flipping the `cr4` register and the other to launch `ret2user`/`pivot2usr`. However, this is often infeasible because virtualization based hypervisor could easily detect the behavior of flipping `cr4` register by inspecting a `vmexit` [49] [51]. Even if there is not protection over `cr4` register, leveraging the "`cr4`-flipping" attack or a similar exploit technique relying multiple `CFHP` with this `CFHP` still faces the following challenge.

4.2.2 Challenge 2: Exploit Path Pitfall

Attempting to leverage the "`cr4`-flipping" exploit technique, a security analyst would expect two `CFHP`s to disable `SMEP/SMAP` and pivoting to userspace respectively. However, such a exploit technique could be imfeasible because of the exploit path pitfall after the first `CFHP`.

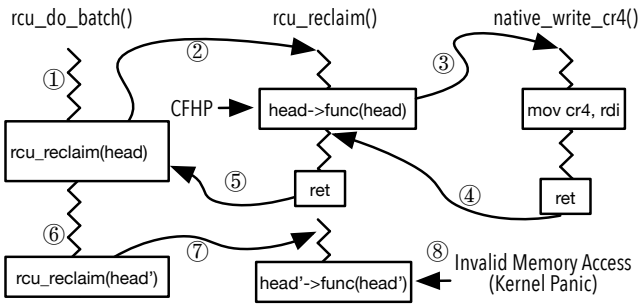


Figure 1: Demonstration of an exploit path pitfall in the motivating example. After applying the first CFHP to overwrite cr4 register with native_write_cr4(), rcu_reclaim(head′) is invoked but head′ is unmanageable and panics the kernel.

As is shown in Figure 1, after using the CFHP in rcu_reclaim to disable SMAP protection by invoking the function native_write_cr4() to zero the corresponding bits in cr4 register, an attacker encounters an unexpected termination: in previous execution, the loop from line 5 to line 10 in Table 1b queues another rcu callback denoted by head′ (iml->next_rcu ->rcu) which is not under our control and causes a kernel panic.

This kernel panic attributes to an invalid memory access. The heap spray technique used by the PoC does not allow an attacker to control first QWORD of iml (an ip_mc_socklist object) because the first QWORD of an freed chunk becomes heap-metadata, thus iml->next_rcu becomes unmanageable.

A straight-forward solution to tackle the invalid memory access is adding extra constraints to ensure all related memory accesses are valid. However, constraint solving for complex program usually incurs high cost (both cpu time and memory) [4] and could fail because of constraint conflicts [5]. Instead of devoting extra resource to handle these pitfalls, an ideal exploit path should effectively detour them. As a result of lacking control of iml->next_rcu, the exploit path pitfall can not be simply prevented by techniques like adding constraints over the sprayed data and thus unavoidably panic the kernel. Although it is possible to tackle the problem by further tuning the PoC [55] [75] and obtain a better exploit primitive, the showcased exploit path pitfall already hinders the evaluation of the CFHP.

In addition, an exploit path pitfall could also be attributed to un-successful heap spray. Due to the undeterminacy of kernel execution, there is not any heap spray technique with 100% success ratio. To get multiple CFHPs for a UAF vulnerability, an attacker may need to do multiple rounds of heap spraying and trigger a vulnerability multiple times, which could dramastically decrease the success ratio of the entire exploitation.

Even if two control-flow hijack primitive is available to the attacker, it could still be very difficult for him to bypass the mitigation combination of SMAP as well as virtualization-based hypervisor, because the attempt to modify cr4 register (in order to turn off SMAP and pivot kernel stack to userspace) could be easily prevented.

4.2.3 Challenge 3: Ill-suited Exploit Primitive

Facing the infeasibility of popular kernel exploit techniques, it is nature for a security analyst to use generic code-reuse technique such as ROP [56] as a second resort.

Stack pivoting is a vital step [54] for a ROP attack especially when an attacker does not control the contents on the stack (*e.g.,* the CFHP does not result from a stack overflow). In userspace, many heap exploits relying on a stack pivoting gadget (*e.g.,* function swapcontext() and setcontext() in libc) to bootstrap a ROP attack.

It is however difficult in the target kernel to pivot stack pointer to a memory region under our control with this CFHP. The reason behind is two-fold. First, as is mentioned before, we can not simply pivot to a userspace with a traditional gadget such as xchg eax,esp; ret because of SMAP. Second, there is not a suitable stack pivoting gadget in a Linux kernel binary for this CFHP. Considering register rdi points to controllable area with this primitive, it would be great to have a gadget to overwrite rsp with a controllable memory address, such as gadget with form xchg r**,rsp; ret, mov rsp,[r**]; jmp rxx and mov rsp,r**; ret for consecutive payload [39]. Unfortunately, similar traditional stack-pivoting gadget does not exist or contains unavoidable exploit path pitfall (*e.g.,* gadget xchg rsp, r14 ; jmp rsp could successfully pivot the stack pointer but inevitably panics the kernel) in our investigation across various modern linux kernel images.

Although previous works have demonstrated the power of code-reuse attack, mounting a traditional ROP attack for the CFHP seems surprisingly difficult because of the lack of stack pivoting gadget.

Without the capability of performing ROP attack, one may think of reusing other kernel functions. Unfortunately, there is also a problem of insufficient control over general registers because only rdi points to a memory region under control and other registers are not in control initially. We need to enhance this CFHP by controlling more general registers and perform subsequent exploitation.

5 Overview

To tackle the three challenges exposed by the motivating example, a security analyst needs to design a new exploit technique to turn a CFHP into a more exploit-friendly machine state based on the vulnerability context and his prior experience. Due to the lack of a ready-to-use exploit technique, he could expect a lot of debugging and manual efforts to explore possible exploit paths and improve his prototype exploit during the exploit development process and such practice could

be extremely time-consuming and even fruitless.

In the following, we discuss the considerations that go into the design of KEPLER as well as the high level design of this framework to facilitate CFHP evaluation.

5.1 Requirements for Design

To support evaluation of a CFHP with working exploits, KEPLER should receive as input a state representing a CFHP and it should be able to find an exploit path towards priviledge escalation and output corresponding exploit. To achieve the above ultimate goal, KEPLER should adopt an exploit technique which satisfies the following requirements.

First, an exploit technique is able to bypass all the widely-deployed mitigations enabled in the threat model. *Second*, taking potential exploit path pitfalls into consideration, an exploit technique should depend on only one control-flow hijacking primitive and detour these pitfalls to prevent an unexpected termination and make exploitation more reliable. The form of an exploit technique would be ideally similar to a "magic gadget[2]" which is previously mentioned in user-space exploitation [18], especially in the context of adversarial scenarios like Capture-The-Flag cyber competition. *Third*, an exploit technique should benefit from time-tested exploit technique such as ROP by efficiently bootstrapping traditional code-reuse attack in absence of stack pivoting gadget with an ill-suited CFHP. *Last but not least*, an exploit technique should be suitable for the automation framework. On one hand, it should be hard-to-defeat and not depend on any special code or feature which could be easily eliminated. On the other hand, the exploit generation phase should be easily automated - it should be a well-defined search problem over a search space of reasonable size.

5.2 High Level Design

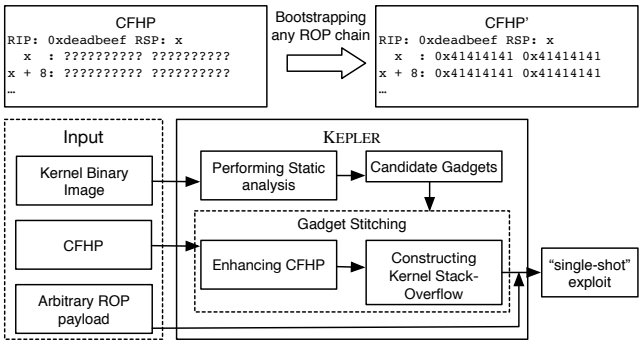


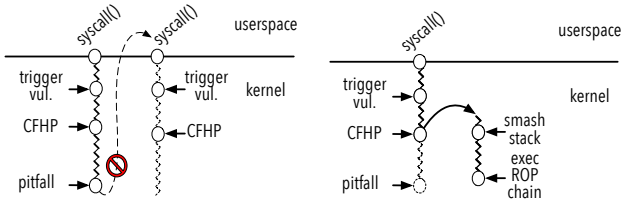
Figure 2: Overall of KEPLER's design.

[2]The term "magic gadget" means given a CFHP, one can instantly succeed in exploitation (*e.g.,* getting a shell) by diverting control-flow to such gadget, thus boost exploit development and gain advantages in a game.

To satisfy the requirements mentioned above, we design KEPLER to facilitate exploit primitive evaluation. KEPLER automatically generates an exploitation chain to *bootstrap* any kernel ROP chain with a single CFHP through "single-shot" exploitation.

Figure 2 shows how KEPLER automates the analysis task necessary to leverage a CFHP to produce an exploit in the presence of aforementioned challenges. Given a kernel state snapshot representing the CFHP, KEPLER enhances its power to construct a kernel stack-overflow by symbolically stitching several types of candidate gadget identified by static analysis on the kernel binary image.

As is mentioned before, our basic idea is to *bootstrap* a traditional ROP attack with a CFHP in Linux kernel. At the high level, we achieve this by a "single-shot" exploitation chain which transforms a function pointer corruption based primitive (CFHP) into a stack-overflow based primitive (CFHP′) as is shown in Figure 2.



(a) Exploitation by envoking a control-flow hijacking primitive twice.

(b) Exploitation with a single control flow hijack primitive.

Figure 3: A comparison of two exploitation approaches; a known approach triggers a vulnerability twice but blocked by an exploit path pitfall and the other triggers the vulnerability only once.

Although there is not any gadget in Linux kernel which allows an attacker to directly escalate priviledge, The proposed "single-shot" exploitation is similar to "magic gadget" mentioned above in a sense that it only requires a single CFHP and could reliably achieve the goal of arbitrary code execution in kernel context. To be specific, as is shown in Figure 3b, the "single-shot" exploitation chain could finish exploitation with a single CFHP and thus is able to circumvent an exploit path pitfall after the return of CFHP which could cause an unexpected termination otherwise. We can benefit from a stack-overflow based CFHP because it allows us to place arbitrary ROP payload on current kernel stack without any stack pivoting gadget. Given the scarcity (or non-existence) of intra-kernel stack pivoting gadget, we argue that constructing a kernel stack overflow is the most generic approach to perform a kernel ROP attack.

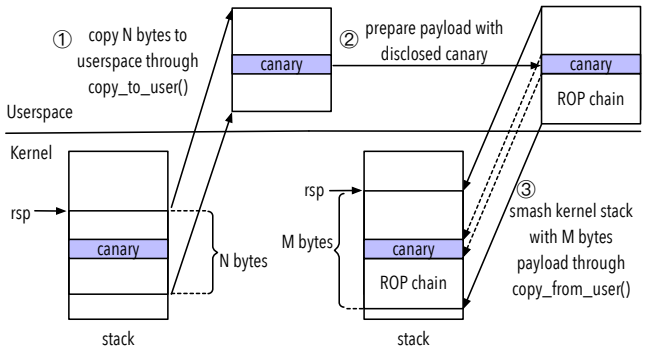


Figure 4: An overview of "single-shot" exploitation which discloses kernel stack canary and then smashes the kernel stack with arbitrary user-supplied ROP payload.

6 Design

In this section, we describe the exploit technique adopted by KEPLER and the insights behind the exploit tehcnique. As is mentioned before, KEPLER uses a CFHP to construct a stack overflow and bootstraps arbitrary ROP payload.

Our "single-shot" exploitation technique builds on two key ideas of *breaking isolation with I/O functions* and *improving exploit success ratio with a single CFHP*.

First, we can break isolation between kernel-space and user-space by abusing kernel I/O functions. The insight behind is such data channels are born to bypass SMAP which prohibits user-space access because SMAP is explicitly and temporarily disabled during execution of these functions. Figure 4 illustates a practice of reusing I/O functions to construct kernel stack overflow by first leaking kernel stack canary with copy_to_user and then smashing kernel stack with copy_from_user.

Second, "single-shot" exploitation can be achieved through stitching various kernel function gadgets. We can enhance register control for a CFHP with *blooming gadget* - a prevalent family of function gadgets in Linux kernel. We can detour exploit path pitfalls with a *bridging gadget*.

6.1 Constructing Stack Overflow

There is a family of prevalent stack smashing gadgets inside Linux kernel, we observe they could greatly aid constructing stack-overflow via taking intrinsic *short return path* triggered by a page fault. However, such gadgets can not be directly used because initial CFHP does not have enough register control and the presence of a stack canary. We address the two problem in Section 6.2 and 6.3.

6.1.1 Looking into Stack-Smashing Gadget

We present stack-smashing gadgets which relies on functions that serve as data channel between user-space and kernel-space. The gadget could aid exploitation by transporting payload of arbitrary length from user-space to kernel stack, without assuming the stack location is already known.

```
1  static long bsg_ioctl(struct file *file, unsigned
       int cmd, unsigned long arg){
2    struct sg_io_v4 hdr;
3    ...
4    if (copy_from_user(&hdr, uarg, sizeof(hdr)))
5      return -EFAULT; // short return
6    ...
7  }
```

(a) Source code.

```
1  ...
2    mov rdi,rsp
3    call <_copy_from_user>
4    test rax,rax
5    je 0xffffffff813d6ce4
6    mov rax,0xfffffffffffffff2
7    jmp <epilogue>
8  ...
9  <epilogue>:
10   mov rcx,QWORD PTR [rsp+0xa0]
11   xor rcx,QWORD PTR gs:0x28
12   jne <__stack_chk_fail>
13   add rsp,0xa8
14   pop rbx
15   ret
```

(b) Assembly code.

Table 2: The kernel code fragment of an stack-smashing gadget that could smash kernel stack with carefully crafted payload.

As is named after, copy_from_user(void* dst, void* src, unsigned long length)[3] is a heavily used I/O kernel function which migrates data from the user to kernel space. Recall that SMAP prevents kernel code from accessing user-space address, and to temporarily bypass the restriction, copy_from_user uses a special instruction STAC to set AC flag in EFLAGS register before accessing user-space memory, thus allows the subsequent instructions to explicitly access user-space memory. Once the copy is finished, instruction CLAC is executed to re-enable SMAP.

As is specified in Linux kernel implementation, the function copy_from_user() takes as input three arguments - dst, src and length - which indicate the destination, source and length of the data that need to be copied from the user to kernel space.

From the perspective of an attacker, a kernel stack overflow could be caused if he lets the CFHP jump to the site right before the invocation of copy_from_user (line 2 in Table 2b) and the machine state satisfies the following three requirements:

[3]The security of copy_from_user has been improved by adding extra checks during the development of Linux Kernel. For example, upon failure during copy, set the dst memory region after the successfully copied bytes to zero to prevent uninitialized use.

❶ parameter `dst` (*e.g.,* `rdi`) points to current kernel stack, ❷ parameter `src` (*e.g.,* `rsi`) points to any user-space address so that its content is controllable by an attacker, ❸ parameter `length` (*e.g.,* `rdx`) is greater than the size of current stack frame to cause a kernel stack overflow.

An interesting observation is that most (91% in Linux 4.15) invocations of this function set destination `dst` to address of a variable on kernel stack and thus will copy user data into a kernel stack. If control-flow is hijacked to a invoking site of this function (*e.g.,* line 2 in Table 2b), an attacker could abuse such coding style to satisfy the requirement ❶ above because the code snippet help set `rdi` to current stack frame. However requirements ❷ and ❸ are still waiting to be satisfied given the initial `CFHP` does not imply any control over register `rdi` and register `rsi`. We will address this issue with blooming gadget in Section 6.3.

6.1.2 Choosing Short Return Path

The prevalent error handling code which is introduced to make kernel code more robust also provides a short return path for an attacker. An attacker could benefit from such a short return path after overflowing current stack frame.

As is depicted in line 4-5 in Table 2a, function `bsg_ioctl` will directly return if return value of `copy_from_user` is *not* zero. Our statistic indicates the function `copy_from_user()` has been invoked at 671 sites in Linux 4.15. Among all these invocation sites, more than 99% contain the fault handling implementation.

The insight of prioritizing short return path after stack smash is to prevent un-expected kernel panic as well as avoid the complexity of resolving extra data dependency in an error-prone and long normal return path of the function containing tens of basic blocks.

To take a short return path, `copy_from_user` must return a non-zero value as is shown in Table 2a. Reviewing the source code of kernel function `copy_from_user`, we have identified 3 different situations which will force the function to return a non-zero value, 1) incurring an integer overflow when calculating *src*+*length*, 2) neither `src+length` nor `src` residing in user-space, 3) encountering an unresolvable page fault during copy. For the first two situations, the function `copy_from_user()` performs sanity check and returns a non-zero value without actually copying data to the kernel. For the last situation, the function migrates data to the kernel and pads with zeros the bytes failing to be copied.

6.1.3 Triggering Page Fault during `copy_from_user`

To force `copy_from_user` returning a non-zero value as well as successfully copying the `ROP` payload from user-space to kernel stack, we trigger page fault after copying enough payload according to the last condition described above.

We illustrate a representative example in Figure 5. We map two adjacent pages ($p1$ and $p2$) in the user-space and

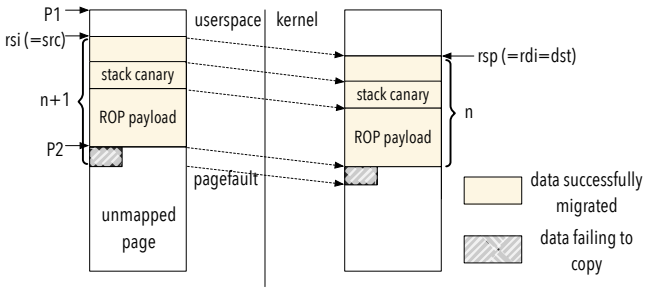


Figure 5: An example where `copy_from_user` triggers a page fault when copying user data to kernel stack.

then unmap the second one. We fill the end of the page $p1$ with the actual payload including a stack canary and a `ROP` chain. We will discuss how to leak stack canary in Section 6.2. Through the technical approaches mentioned in Section 6.3, assume we have already obtained the control over registers `rsi` and `rdx` pertaining to the second and third parameters of `copy_from_user` respectively. Leveraging the control over registers, we manipulate the values in these registers. More specifically, we set `rsi` and `rdx` to $p1 + \text{PAGE_SIZE} - n$ and $n+1$ respectively, with n representing the length of payload actual copied. When the function attempts to copy the last byte, it failes to access the content at $p2$ and triggers a page fault because the page $p2$ is not mapped into the memory. Eventually `copy_from_user` returns a positive number 1 because one byte is not successfully copied.

6.2 Bypassing Stack Canary

As is mentioned earlier, to prepare a working payload for stack smash, an attacker has to know the value of kernel stack canary which remains secret in our threat model. We consider the presence of a strong kernel stack canary setting where stack canary is enabled for all functions containing a local variable.

We will first introduce two kinds of prevalent gadgets, then we discuss how to pair them to dump kernel stack memory.

6.2.1 Exposing Stack-disclosure Gadget and Auxiliary Function

To leak stack canary, an intuitive way is to construct an info leak of its value to user-space through an official data channel such that `SMAP` is not violated. In the following we introduce stack-disclosure gadget which is twin gadget of stack-smash gadget as well as auxiliary function prologue gadget.

Stack-disclosure gadget. Function `copy_to_user()` is widely used in the Linux kernel codebase to copy kernel memory into user-space. In Linux kernel 4.15, our statistic indicates this function has been invoked at 594 sites. Of all these invocations, 82% are used for copying data from kernel

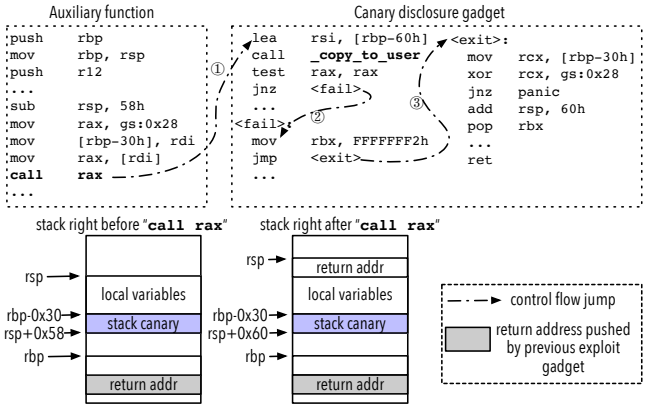


Figure 6: An example of canary disclosure gadget and its corresponding auxiliary gadget.

stack to user-space. Since this naturally establish a channel to migrate data from kernel stack to user-space, we could exploit the characteristic of this kernel function to disclose the canary on kernel stack.

Auxiliary function. To successfully return from a stack-disclosure gadget and continue exploitation, we use auxiliary function to create a similar stack frame as the stack-disclosure gadget and transfer the control-flow to stack-disclosure gadget with an indirect call after the function prologue.

Auxiliary function should have stack canary protection and contain a controllable indirect call after its own function prologue which establishes a stack frame. Its layout of stack frame could be paired with a stack-disclosure gadget to form a "complete" stack frame and pass the stack canary check by putting a valid stack canary on the stack.

6.2.2 Disclosing Canary on Kernel Stack

By diverting the control-flow to a call site of `copy_to_user()`, we are closer to successfully disclose stack contents by satisfying the following four requirements. ❶, the registers should be set properly as parameters for `copy_to_user`, ❷, the kernel should not panic during the path caller function returns, ❸, the caller function of `copy_to_user` checks stack canary before return, ❹, the return address on stack must be set properly to continue the rest of the exploitation.

To tackle the first requirement, we leverage *blooming gadget* described in Section 6.3. For the second requirement, we could *trigger a page fault* and take a *short return path* similar to the technique described in Section 6.1.2. For the last two requirements, we pair stack-disclosure gadget with auxiliary function to generate a valid stack frame.

The key insight behind using auxiliary function to pair with stack-disclosure gadget is reusing the canary generated by the prologue of auxiliary function. A pair of them should have the same number of saved registers, the same canary location and stack size of 8 bytes difference.

```
1 static void aliasing_gtt_unbind_vma(struct
      i915_vma *vma) {
2     ...
3     vma->vm->clear_range(vma->vm, vma->node.start,
          vma->size);
4     ...
5 }
```

Table 3: The kernel code fragment (a blooming gadget) that could enhance the control over multiple general registers.

To elucidate the rationale behind the pairing, we take for example the routine of canary disclosure in Figure 6. We re-direct a `CFHP` to auxiliary function, After the prologue of auxiliary function which saves registers and establishes a stack frame, the target of indirect call `call rax` is set to the stack disclosure gadget in ①. Then content of current stack frame is copied to user-space by `copy_to_user`, a page fault is triggered to force non-zero return value of `copy_to_user`, as result *short return path* is taken in ②. Before the function returns, stack canary sanity check is performed ③, because the auxiliary function put a valid stack canary in current stack frame, the canary check is successfully passed and return to the caller of auxiliary function.

6.3 Putting them together: "Single-shot" Exploitation

It remains challenging to use an ill-suited `CFHP` to first disclose stack canary and then smashing kernel stack. The reason behind is a `CFHP` in practice may have limitations in the following two aspects. First, difficulty in combining aforementioned two building blocks with a single `CFHP`, second, lack of register control. "Single-shot" exploitation uses a *blooming gadget* to amplify control over other registers and a *bridging gadget* to combine the two actions sequentially.

6.3.1 Augmenting `CFHP` with Blooming Gadget

To enhance a `CFHP` with the ability to control more registers, we introduce a family of *blooming gadgets*. The use of blooming gadget is inspired by `COOP` [59] which exploits a series of type confusions C++ program. Although Linux is written in C, its code heavily exhibits feature of object-oriented programming. The "self" object is usually passed as the first argument of function through `rdi`. And oftentimes the function contains'' indirect call using function pointer that resides in the object passed as parameter. We could let the `CFHP` to land at these functions to abuse type confusion.

We illustrate one such blooming gadget in Table 3. Kernel function `aliasing_gtt_unbind_vma()` contains an indirect call with three parameters calculated by dereferencing the function's first parameter `*vma`. Assume we have a `CFHP` with

```
1 void regcache_mark_dirty(struct regmap *map) {
2     map->lock(map->lock_arg); // the 1st
          control-flow hijack
3     map->cache_dirty = true;
4     map->no_sync_defaults = true;
5     map->unlock(map->lock_arg); // the 2nd
          control-flow hijack
6 }
```

Table 4: The source code of a bridging gadget – the kernel code fragment that could spawn multiple CFHPs.

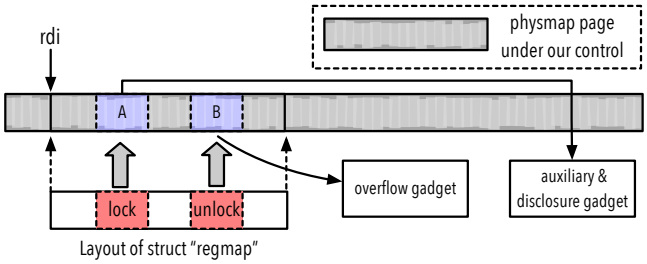


Figure 7: Memory layout after using physmap spray [39] to allocate physmap pages with data under our control.

control over rdi, we can get an augmented CFHP which controls rdi, rsi, and rdx at the same time at line 3 in Table 3.

Note a blooming gadget works only if rdi is controllable at beginning. We found this requirement is easy to fullfil in practice. Our insight is that a CFHP usually has one register potentially controllable - either the register is fully controllable or the register points to a heap area under control. Such primitive can be turned into a CFHP with rdi control easily through a single gadget which ends with an indirect call. A worst case happens where a CFHP implies none of potentially controllable registers. Fortunately, we are still able to leverage uninitialized or controllable data on kernel stack as well as a common ROP gadget. For example, we could use the gadget `add rsp, 0x68; pop rdi; ret` to gain control over rdi if rsp+0x68 and rsp+0x70 is under our control.

6.3.2 Spawning Multiple CFHPs with Bridging Gadget

As is demonstrated in the motivating example in Section 4, an exploitation practice depending on re-triggerable CFHP is not reliable because of exploit path pitfalls. We use bridging gadget - a family of kernel functions with multiple controllable indirect calls - to spawns two CFHPs and combine canary leak and stack smash into a single shot.

For example, function regcache_mark_dirty shown in Table 4 is such a bridging gadget which contains two indirect calls, map->lock in line 2 and map->unlock in line 5.

As we can observe from the kernel gadget shown in Table 4, the function pointers tied to these two indirect calls are enclosed in a data object referred by the first argument

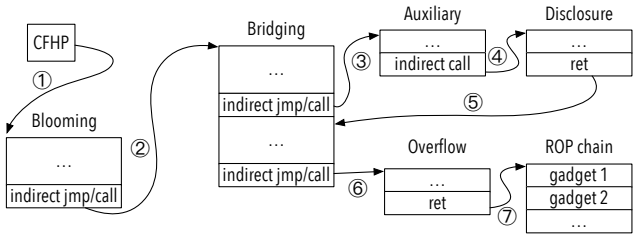


Figure 8: An illustration of how KEPLER stitches various kernel gadgets for ultimate exploitation.

of the function regcache_mark_dirty(). Recall that the first argument of a function is specified by the general register rdi, and we can usually obtain the control over that register using technique described in Section 6.3.1. As a result, in order to obtain control over both function pointers, we could first employ ret2dir [39] to allocate physmap pages and carefully crafted a data object accordingly. Then, we could refer the register rdi to a proper spot and set rip to the entry site of the gadget shown in Table 4. As is shown in Figure 7, assume the data carefully crafted in the spots of A and B represent the address of the auxiliary gadget together with a disclosure gadget responsible for leaking stack canary as well as the entry address of the gadget pertaining to stack smashing respectively. Then, by executing the bridging gadget shown in Table 4, we could first leak canary using the first indirect call. After the return of the call to copy_to_user(), there is no operations between the consecutive indirect calls that impose additional constraint to the second function pointer. Therefore, we could perform the stack smashing using the second function pointer without involving unexpected termination.

7 Implementation

Using IDA Pro SDK [20] and angr [64], we implemented KEPLER with about 8,000 lines of Python code. KEPLER is an automated tool that tracks down the aforementioned exploitation gadgets and chains them for exploitability assessment. Figure 8 depicts how these gadgets are concatenated. While previous sections have discussed the basic building blocks to perform an "single-shot" exploitation, the exploitation chain could not be determined once and for all with static analysis because uniqueness of each CFHP and different gadget combinations bring about the variation of the exploitation context. For example, the consecutive exploitation gadgets might no longer obtain the control over related registers with a different initial CFHP.

To address this problem, we developed our tool to assess each of the gadget chains potentially useful for kernel exploitation. More specifically, we follow the guidance of the exploitation chain construction shown in Figure 8, and design our tool to perform a depth-first exhaustive search which explores all the possible combinations of exploitation gad-

gets. When performing the depth-first search: starting from the rip hijack site, KEPLER symbolically executes the gadget chain that the search algorithm explores. To determine whether a gadget chain is useful for exploitation, our tool checks the memory access and deems a gadget chain useless if that exploitation chain attempts to access the user space or an unmapped kernel memory region. In addition, KEPLER examines the control over the registers at two critical sites – one at the entry of the disclosure gadget and the other at the entry of the overflow gadget. We implemented KEPLER to deem a gadget chain useless and terminate symbolic execution earlier if it has no control over the registers rdi and rdx at the first checking site or has no control over rsi and rdx at the second checking site. The reason behind this implementation is that, after executing bridging and auxiliary gadgets, we might lose the control over the registers needed for disclosure and overflow gadgets. With the check right before symbolically executing the two gadgets, we can quickly determine the usefulness of the gadget chain in exploitation, terminate unnecessary symbolic execution and thus save the computation resources.

In the process of the assessment of the exploitation chain, KEPLER symbolically executes each exploitation gadget. For some of them, they might carry a large number of basic blocks and even infinite loops. This could significantly influence the efficiency our tool and even incur the state explosion problem. To avoid these issues, for each path in an exploitation gadget, we set KEPLER to explore at most 20 basic blocks. In addition, we developed KEPLER to concretize each symbolic address. To be more specific, we set up a kernel page under our control in the physmap region and then concretize each symbolic address with a non-overlapping address of that memory page. In this work, we implemented this concretization mechanism by simply extending ControlledData – one of the symbolic address concretization strategies of angr.

For each gadget chain that passes the assessment, KEPLER further performs constraint solving to generate payload accordingly. Technically, this can be easily done by using angr. However, the Z3 solver used in angr consumes memory exhaustively and generally does not release the memory used for constraint solving even after the completion of computation. To address this problem, KEPLER partitions symbolic execution and constraint solving into two different processes. In this way, KEPLER could terminate the memory-intensive process every time the constraint solving is completed and thus free the memory for consecutive computation.

As is described in Section 6.1, the payload smashed to the stack contains the stack canary disclosed as well as a sequence of addresses indicating an ROP chain that performs actual exploitation. With respect to the stack canary, we employ a separated thread in the user-space to rapidly retrieve the canary – whenever it is disclosed to the user-space – and then make it ready for stack smashing. Regarding the ROP chain used in this work, we simply choose the ROP payload commonly used for privilege escalation. In Appendix, we specify the ROP payload used in this work. It invokes kernel functions commit_creds() and prepare_kernel_cred() to obtain the root privilege. Note that the construction of an ROP payload is out of scope of this paper. There are many commonly-adopted ROP payloads, which can be naturally hooked with our new kernel exploitation technique.

8 Case Study and Evaluation

In this section, we demonstrate our new exploit technique and evaluate our automated tool KEPLER using real-world kernel vulnerabilities and some recently-released CTF challenges. To be specific, we compare KEPLER with various kernel exploitation techniques to show it is an effective exploit technique, we also compare KEPLER with automatic exploit generation systems to highlight its power in evaluating exploitability with a CFHP. In addition, we show the efficacy and efficiency of KEPLER in facilitating exploitation chain construction.

8.1 Setup

We first randomly selected 3 recently released CTF challenges as well as 16 real-world kernel vulnerabilities archived between 2016 and 2017. Then, we successfully assembled these vulnerabilities in a mainline Linux kernel 4.15.0 by inserting them into the kernel code or reverting their patch accordingly. In this work, we evaluate our tool KEPLER by using this single Linux kernel, and demonstrate the effectiveness of "single-shot" exploitation by launching exploitations against the inserted vulnerabilities.

As is summarized in Table 5, the vulnerabilities inserted cover various types such as Use-After-Free and Out-Of-Bound (OOB) read/write etc. It should be noted that the CVEs selected are a little bit unbalanced – with more in 2017 and less in 2016. On the one hand, this is because there are more than 2× of kernel vulnerabilities reported in 2017 than those in 2016 [1]. On the other hand, this is because some components in Linux kernel experience significant overhaul since 2016 and we have difficulty of re-enabling the corresponding vulnerabilities in a new kernel image.

In order to run and evaluate KEPLER, we also assembled and configured a testbed which has a 32-core Intel(R) Xeon(R) Platinum 8124M CPU and 256GB of memory. For each vulnerability, we then used this testbed to run 28 concurrent workers which symbolically explore the kernel code space and track down useful exploitation chains in parallel.

8.2 Effectiveness of "single-shot" exploitation

By searching the Internet, we gathered 10 exploits pertaining to the vulnerabilities inserted. As is shown in Table 5, these

ID	Vulnerability type	Public exploit	Q	FUZE	KEPLER	G1	G2	G3	G4	First chain (min)	Total time (hour)	Total # of exploitation chains
CVE-2017-16995	OOB readwrite	✓†	✗	✗	✓	41	114	27	201	45	37	29788
CVE-2017-15649	use-after-free	✓	✗	✓	✓	29	79	25	280	16	28	60207
CVE-2017-10661	use-after-free	✗	✗	✗	✓	28	78	30	301	17	25	49070
CVE-2017-8890	use-after-free	✗	✗	✗	✓	21	88	23	304	17	18	50471
CVE-2017-8824	use-after-free	✓	✗	✓	✓	63	101	35	306	50	70	164898
CVE-2017-7308	heap overflow	✓	✗	✗	✓	31	91	30	241	14	47	110176
CVE-2017-7184	heap overflow	✓	✗	✗	✓	31	95	31	254	24	37	93752
CVE-2017-6074	double-free	✓	✗	✗	✓	18	79	31	308	16	15	31436
CVE-2017-5123	OOB write	✓†	✗	✗	✓	40	86	27	311	14	39	113466
CVE-2017-2636	double-free	✗	✗	✗	✓	18	89	29	289	29	19	26372
CVE-2016-10150	use-after-free	✗	✗	✗	✓	34	84	25	293	52	34	88499
CVE-2016-8655	use-after-free	✓†	✗	✓†	✓	18	109	32	260	15	17	47413
CVE-2016-6187	heap overflow	✗	✗	✗	✓	22	85	32	301	17	21	51954
CVE-2016-4557	use-after-free	✗	✗	✗	✓	21	80	21	295	16	37	40889
CVE-2017-17053	use-after-free	✗	✗	✗	✗	-	-	-	-	-	-	-
CVE-2016-9793	integer overflow	✗	✗	✗	✗	-	-	-	-	-	-	-
TCTF-credjar	use-after-free	✓†	✗	✗	✓	35	89	25	292	25	14	82913
0CTF-knote	uninitialized use	✗	✗	✗	✓	21	89	33	318	17	36	40923
CSAW-stringIPC	OOB read&write	✓†	✗	✗	✓	35	88	25	289	17	33	84414

Table 5: The comparison of exploitability as well as performance of KEPLER. G1, G2, G3 and G4 represent the blooming gadget, bridging gadget, auxiliary and disclosure gadget pair, and stack-smash gadget. The "first chain" column indicates the time spent on pinpointing the first exploitation chain. The "total time" column specifies the total amount of time spent on finding all useful exploitation chains. † symbol represents the cases where the exploits could only bypass major mitigations (*e.g.,* SMAP and SMEP) and fail to bypass others under our threat model. ✓and ✗ symbols indicate the existence and non-existence of a working exploit.

publicly available exploits perform exploitation through various approaches and therefore demonstrate different capability in bypassing kernel mitigations. Among these exploits, we found there are only 5 of them demonstrating the ability to perform exploitation under our aforementioned threat model. In comparison with the working exploits generated by KEPLER, publicly available exploits demonstrate much weaker exploitability (with 5 vs 17 cases). To some extent, this implies existing exploitation approaches highly rely upon the quality of the target vulnerability and corresponding CFHP, whereas our approach KEPLER could utilize prevalent kernel function and gadgets to explore exploitable machine states and thus escalate the exploitability for a CFHP. However, previous exploit technique Q [60] could not generate working exploit because Q rely on a stack pivoting gadget while its gadget discovery phase return none of working pivoting gadget[4]. FUZE could only generate exploit for 3 cases because it evaluate exploitability of a CFHP simply with two straight forward exploit technique: pivot-2-usr and "cr4-flipping". The former does not bypass SMAP and the latter only works when at least two CHFPs is available.

Even for the vulnerabilities against which both public exploits and ours demonstrate the same capabilities in bypassing mitigations, we argue that our approach still exhibits stronger exploitability. This is because the public exploits circumvent mitigations by manipulating control registers with two CFHPs, as is discussed in Section 2.3, this practice can be easily restricted by virtualization extension. For the two vulnerabilities CVE-2017-17053 and CVE-2016-9793 for which our approach fails to derive working exploits, we manually examine their execution traces leading to the kernel panic. We find that the failure results from the following fact. In order to take the control over rip prior to exploitation, both of these vulnerabilities require an exploit to access the data in the user space. This violates the protection of SMAP. KEPLER restricts any operations that violate our threat model and output a failure if none of the exploitation chains could avoid such violation.

[4]The result related to Q in our evaluation is based on inference of its design instead of running its tool because we were not able to get the source code of Q.

8.3 Effectiveness and Efficiency of Our Tool

Our experiment utilizes KEPLER to explore the aforementioned kernel image with the vulnerabilities inserted. In this process, we exhaustively search gadget chains useful for exploitation and mitigation circumvention. In Table 5, we show the total number of useful exploitation chains identified as well as the total amount of time spent on finding these gadget chains. As we can observe, KEPLER could automatically pinpoint tens of thousands of unique kernel gadget chains to perform exploitation without triggering kernel protections. Since we implement KEPLER to perform gadget chain exploration in parallel, we also discover that these gadget chains could typically be identified within 50 hours. These observations together imply that KEPLER could diversify the ways of performing kernel exploitation in an efficient fashion. Given that some commercial security products pinpoint kernel exploita-

tion by using the patterns of exploits, the ability to diversify exploitation has the potential to assist an adversary to bypass the detection of commercial security products.

From Table 5, we also observe that, for different vulnerabilities, KEPLER generates different number of gadget chains useful for exploitation. This can be attributed to the following fact. In the process of gadget chain identification and assessment, KEPLER starts gadget assessment from different machine states and contexts. For some vulnerabilities, the machine states and contexts do not provide us with sufficient control over some registers and memory regions. Under this circumstance, the availability of useful kernel gadgets would vary and thus influence the total number of generated exploits.

In Table 5, we also depict the time spent on finding the first kernel gadget chain useful for exploitation. As we can observe, KEPLER could quickly output an useful exploitation chain in less than about 50 wall-clock minutes (and the corresponding CPU-core time is roughly 1400 minutes given the prototype system uses 28 concurrent workers). This implies KEPLER has the potential to be used as a tool to quickly derive a working exploit without too many human efforts. Last but not least, Table 5 also shows the total number kernel gadgets in different categories. As we can observe, there are typically tens of gadgets in each categories. This means that one cannot simply block our exploitation approach by eliminating a small number of kernel gadgets. In Section 9, we will further discuss the defense of our exploitation approach.

9 Discussion and Future Research

In this section, we discuss some plausible defence mechanisms against our "single-shot" exploitation chain. Also, we elaborate why they are not effective nor suitable for preventing the proposed attack. Following our discussion and analysis, we then provide some suggestions for the future research.

Plausible Defense Mechanisms. To defend against the exploit chain mentioned above, one straightforward reaction is to eliminate the gadgets that must be used in kernel exploitation. However, as we have already demonstrated and discussed in Section 8, the tool we develop could enrich the choices of the gadgets needed for exploitation. This means that, following this potential solution, Linux developers would inevitably introduce significant amount of kernel code changes and it is difficult to guarantee these changes would not bring about negative influence upon Linux kernel execution. Other security mitigation could also be used as potential defense mechanisms. For example, there have already been a rich collection of research works on control and data flow integrity protection (e.g. [2] [78] [79] [25] [16] [26]). In addition, randomizing stack canary per-function call [73] could idealy prevent our exploit technique because it discourage the effort to fake stack frame and leak stack canary with `copy_to_user`. Integrating and enabling them in Linux kernel, they could easily fail the attack mentioned above. Unfortunately, these techniques usually incur unacceptable overhead (e.g., [16] has an average overhead of 13%) or sometimes rely upon hardware features to reduce their overhead (e.g., [25]). As a result, they are barely used as a practical, general defense solution in popular release version of Linux kernel.

Possible Future Research. Looking ahead, we suggest the future research could be conducted from two aspects. From the perspective of automatic exploit primitive evaluation, we believe there is an emerging need to invent technique to systematically evaluate various exploit primitives, expecially for those weak exploit primitives. In practice, theory and techniques should be proposed to facilitate deriving better exploit primitive with a initially weak exploit primitive. From the perspective of defense, on one hand, we believe there remains the need to design lightweight control-flow enforcements for Linux kernel. On the other hand, instead of manually overhauling kernel code, one could augment GCC with the ability to eliminate the exploitation gadgets at compilation time.

10 Conclusion

We show it is generally challenging to generate exploits with a control-flow hijacking primitive in the Linux kernel under a realistic threat model, while there are a lot of research efforts in identifying exploit primitives and facilitating exploit generation with various exploit primitives. We propose KEPLER, a framework to facilitate evaluation of control-flow hijacking primitives which leverages a novel "single-shot" exploitation to convert a control-flow hijacking primitive into a classic stack overflow and thus bootstrap traditional code-reuse attack against modern Linux kernel. In comparison with previous automatic exploit generation and exploit hardening techniques, we show that KEPLER outperforms other exploit techniques and is able to generate thousands of exploit chains for a control-flow hijacking primitive in Linux kernel despite the challenges of widely-deployed security mitigations, exploit path pitfalls and ill-suited exploit primitives. Following the experimental results, we safely conclude that KEPLER can significantly facilitate evaluating control-flow hijacking primitive in the Linux kernel.

11 Availability

We release the source code of KEPLER, a kernel embeded with vulnerabilities and generated gadget chains for research and education purposes [76].

12 Acknowledgements

We would like to thank our shepherd Stephen McCamant and anonymous reviewers for their help and comments. The IIE authors were partially supported by the Stategic Priority Research Program of the CAS (XDC02040100,

XDC02030200, XDC02020200), the National Key Research and Development Program of China (2016YFB0801004, 2016QY071405, 2018YFB0803602, 2016QY06X1204), the Key Foundation of Beijing Committee of Science and Technology (Z181100002718002), the Key Laboratory of Network Assessment Technology of Chinese Academy of Sciences and Beijing Key Laboratory of Network Security and Protection Technology. The PSU authors were partially supported by IST seed grant. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements.

References

[1] Linux kernel vulnerability statistics, 2018. https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12nd ACM conference on Computer and communications security (CCS)*, 2005.

[3] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley. Automatic exploit generation. *Communications of the ACM*, 57, 2014.

[4] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), 2018.

[5] T. Bao, R. Wang, Y. Shoshitaishvili, and D. Brumley. Your exploit is mine: Automatic shellcode transplant for remote exploits. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P)*, 2017.

[6] S. Bratus, M. Locasto, M. Patterson, L. Sassaman, and A. Shubina. Exploit programming: From buffer overflows to weird machines and theory of computation. {*USENIX; login:*}, 2011.

[7] D. Brumley, P. Poosankam, D. X. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 29th IEEE Symposium on Security and Privacy (S&P)*, 2008.

[8] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*, 2015.

[9] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Proceedings of IEEE Symposium on Security and Privacy (SP), 2012*, 2012.

[10] K. Cook. x86/mm: Always enable config_debug_rodata and remove the kconfig option, 2016. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=9ccaf77cf05915f51231d158abfd5448aedde758.

[11] J. Corbet. Supervisor mode access prevention, 2012. https://lwn.net/Articles/517475/.

[12] J. Corbet. Post-init read-only memory, 2015. https://lwn.net/Articles/666550/.

[13] J. Corbet. A page-table isolation update, 2018. https://lwn.net/Articles/752621/.

[14] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, 2015.

[15] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz. It's a trap: Table randomization and protection against function-reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[16] J. Criswell, N. Dautenhahn, and V. Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, 2014.

[17] L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi. Pt-rand: Practical mitigation of data-only attacks against page tables. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS)*, 2017.

[18] david942j. The one-gadget in glibc, 2018. https://david942j.blogspot.com/2017/02/project-one-gadget-in-glibc.html.

[19] dong-hoon you. New reliable android kernel root exploitation techniques, 2016. http://powerofcommunity.net/poc2016/x82.pdf.

[20] C. Eagle. *The IDA Pro Book (Second edition)*. no starch press, 2011.

[21] J. Edge. Extending the use of ro and nx, 2011. https://lwn.net/Articles/422487/.

[22] J. Edge. "strong" stack protection for gcc, 2014. https://lwn.net/Articles/584225/.

[23] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*.

[24] A. Follner, A. Bartel, H. Peng, Y.-C. Chang, K. Ispoglou, M. Payer, and E. Bodden. Pshape: Automatically combining gadgets for arbitrary method execution. In *International Workshop on Security and Trust Management*, 2016.

[25] X. Ge, W. Cui, and T. Jaeger. Griffin: Guarding control flows using intel processor trace. *ACM SIGOPS Operating Systems Review*, 51(2), 2017.

[26] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-grained control-flow integrity for kernel software. In *Proceedings of 2016 IEEE European Symposium on Security and Privacy (Euro S&P)*, 2016.

[27] J. Gionta, W. Enck, and P. Larsen. Preventing kernel code-reuse attacks through disclosure resistant code diversification. In *Proceedings of the 2016 IEEE Conference on Communications and Network Security (CNS)*, 2016.

[28] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. Prefetch side-channel attacks: Bypassing SMAP and kernel

ASLR. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[29] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, 2014.

[30] S. Heelan, T. Melham, and D. Kroening. Automatic heap layout manipulation for exploitation. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[31] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. Automatic generation of data-oriented exploits. In *Proceedings of the 24nd USENIX Security Symposium (USENIX Security)*, 2015.

[32] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*, 2016.

[33] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of the 18th USENIX Security Symposium (USENIX Security)*, 2009.

[34] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space aslr. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, 2013.

[35] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, 2018.

[36] itsZN. Bypassing smep using vdso overwrites, 2015. https://itszn.com/blog/?p=21.

[37] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin. Razzer: Finding kernel race bugs through fuzzing. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*, 2019.

[38] M. Jurczyk and G. Coldwind. SMEP: What is it, and how to beat it on windows, 2011. http://j00ru.vexillium.org/?p=783.

[39] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis. ret2dir: Rethinking kernel isolation. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, 2014.

[40] S. Knox. Real-time kernel protection(rkp), 2016. https://www.samsungknox.com/en/blog/real-time-kernel-protection-rkp.

[41] A. Konovalov. Exploiting the linux kernel via packet sockets, 2017. https://googleprojectzero.blogspot.com/2017/05/exploiting-linux-kernel-via-packet.html.

[42] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis. Compiler-assisted code randomization. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P)*, 2018.

[43] G. Kroah-Hartman. Introduce static_usermodehelper to mediate call_usermodehelper, 2017. https://patchwork.kernel.org/patch/9519063/.

[44] Lexfo. Cve-2017-11176: A step-by-step linux kernel exploitation, 2018. https://blog.lexfo.fr/cve-2017-11176-linux-kernel-exploitation-part4.html#stack-pivoting.

[45] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *Proceedings of 27th USENIX Security Symposium (USENIX Security)*, 2018.

[46] K. Lu, M. Walter, D. Pfaff, and S. Nürnberger and Wenke Lee and Michael Backes. Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS)*, 2017.

[47] M. Miller. Modeling the exploitation and mitigation of memory safety vulnerabilities. In *Breakpoint*, 2012.

[48] R. Mothe and R. R. Branco. Dptrace: Dual purpose trace for exploitability analysis of program crashes. In *Black Hat USA Briefings*, 2016.

[49] J. Nakajima and S. Grandhi. Kernel protection using hardware based virtualization. In *The Linux Foundation events*, 2017.

[50] National Vulnerability Database. Cve-2017-8890 detail, 2017. https://nvd.nist.gov/vuln/detail/CVE-2017-8890.

[51] M. Oh. Detecting and mitigating elevation-of-privilege exploit for cve-2017-0005, 2017. https://cloudblogs.microsoft.com/microsoftsecure/2017/03/27/detecting-and-mitigating-elevation-of-privilege-exploit-for-cve-2017-0005/.

[52] S. Pailoor, A. Aday, and S. Jana. Moonshine: Optimizing os fuzzer seed selection with trace distillation. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[53] M. Pomonis, T. Petsios, A. D. Keromytis, M. Polychronakis, and V. P. Kemerlis. Kernel protection against just-in-time code reuse. *ACM Transactions on Privacy and Security (TOPS)*, 22(1), 2019.

[54] A. Prakash and H. Yin. Defeating rop through denial of stack pivot. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 111–120, 2015.

[55] D. Repel, J. Kinder, and L. Cavallaro. Modular synthesis of heap exploits. In *ACM SIGSAC Workshop on Programming Languages and Analysis for Security (PLAS)*, 2017.

[56] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):2, 2012.

[57] J. Salwan. Ropgadget, 2012. https://github.com/JonathanSalwan/ROPgadget.

[58] S. Schumilo, C. Aschermann, and R. Gawlik. kafl: Hardware-assisted feedback fuzzing for os kernels. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*, 2017.

[59] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications.

In *In Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P)*, 2015.

[60] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Security Symposium (USENIX Security)*, 2011.

[61] M. Seaborn and T. Dullien. Exploiting the dram rowhammer bug to gain kernel privileges, 2015. https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html.

[62] Y. Shoshitaishvili, A. Bianchi, K. Borgolte, A. Cama, J. Corbetta, F. Disperati, A. Dutcher, J. Grosen, P. Grosen, A. Machiry, et al. Mechanical phish: Resilient autonomous hacking. *IEEE Security & Privacy*, 16(2):12–22, 2018.

[63] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS)*, 2015.

[64] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK:(state of) the art of war: Offensive techniques in binary analysis. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*, 2016.

[65] K. A. Shutemov. pagemap: do not leak physical addresses to non-privileged userspace, 2015. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=ab676b7d6fbf4b294bf198fb27ade5b0e865c7ce.

[66] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 34 IEEE Symposium on Security and Privacy (S&P)*, 2013.

[67] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee. Enforcing kernel security invariants with data flow integrity. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS)*, 2016.

[68] spender. wait for kaslr to be effective, 2017. https://grsecurity.net/~spender/exploits/wait_for_kaslr_to_be_effective.c.

[69] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS)*, 2016.

[70] P. Team. Rap: Rip rop, 2015. https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf.

[71] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[72] D. Vyukov. Syzkaller, 2015. https://github.com/google/syzkaller.

```
int i=0;
unsigned long *p=(unsigned long*)PAYLOAD_START;
p[i++]=0; // padding
p[i++]=0; // canary location
p[i++]=0; // padding for saved registers
...
// priviledge escalation
p[i++]=POPRDI; // pop rdi ; ret
p[i++]=0;
p[i++]=PREPARE_KERNEL_CREDS;
p[i++]=POPRDXRET; // pop rdx ; ret
p[i++]=COMMIT_CREDS;
p[i++]=MOV_RDI_RAX_JMP_RDX; // mov rdi, rax ;
    jmp rdx
// sleep for 60 minutes
p[i++]=POPRDI; // pop rdi ; ret
p[i++]=1000 * 60 * 60;
p[i++]=MSLEEP;
```

Table 6: The kernel ROP payload that performs privilege escalation.

[73] Z. Wang, X. Ding, C. Pang, J. Guo, J. Zhu, and B. Mao. To detect stack buffer overflow with polymorphic canaries. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 243–254. IEEE, 2018.

[74] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello. Shuffler: Fast and deployable continuous code re-randomization. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[75] W. Wu, Y. Chen, J. Xu, X. Xing, W. Zou, and X. Gong. Fuze: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[76] ww9210. kepler-cfhp, 2018. https://github.com/ww9210/kepler-cfhp.

[77] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[78] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, 2013.

[79] M. Zhang and R. Sekar. Control flow integrity for cots binaries. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security)*, 2013.

Appendix

As is discussed in Section 6, we utilize a series of kernel gadgets to bypass kernel mitigations. After that, we redirect the control flow of the Linux kernel to a universal ROP payload. By using that payload,

```
// copy more payloads to the kernel stack
// prepare auguments for copy_from_user()
p[i++]=POPRAX; // pop rax ; ret
p[i++]=POPRSI; // pop rsi ; ret
p[i++]=0xffffffff81254a99; // mov rdi, rsp ;
    call rax
p[i++]=POPRAX;
p[i++]=0x1000;
p[i++]=0xffffffff81a04201; // sub rdi, rax ;
    mov rax, rdi ; ret
p[i++]=POPRSI;
p[i++]=STAGE_TWO_ROP_PAYLOAD;
p[i++]=POPRDX; // pop rds ; ret
p[i++]=0x1040;
p[i++]=COPY_FROM_USER; // copy_from_user()

// substract rsp to the first gadget
p[i++]=POPRAX;
p[i++]=0x1040;
p[i++]=0xffffffff81a04201; // sub rdi, rax ;
    mov rax, rdi ; ret
p[i++]=POPR12; // pop r12 ; ret
p[i++]=0xffffffff810001cc; // ret
p[i++]=0xffffffff81c01688; // mov rsp, rax ;
    push r12 ; ret
```

Table 7: The kernel ROP payload that copies an ROP payload to the current stack frame and then subtracts the stack pointer to execute the ROP payload.

we demonstrate the exploitability of a kernel vulnerability. In Table 6, we show an ROP payload used in this work. As is specified, it first performs privilege escalation. Then, it sets the Linux kernel to fall into asleep for a long time by using the kernel function `msleep()`.

Considering Linux kernel might perform inline permission checks and we need to execute an ROP payload with an arbitrary length, we further utilize the ROP payload like the one shown in Table 7 to address this payload length issue.

PeX: A Permission Check Analysis Framework for Linux Kernel

Tong Zhang*
Virginia Tech

Wenbo Shen†
Zhejiang University

Dongyoon Lee
Stony Brook University

Changhee Jung
Purdue University

Ahmed M. Azab‡
Samsung Research America

Ruowen Wang‡
Samsung Research America

Abstract

Permission checks play an essential role in operating system security by providing access control to privileged functionalities. However, it is particularly challenging for kernel developers to correctly apply new permission checks and to scalably verify the soundness of existing checks due to the large code base and complexity of the kernel. In fact, Linux kernel contains millions of lines of code with hundreds of permission checks, and even worse its complexity is fast-growing.

This paper presents PeX, a static Permission check error detector for LinuX, which takes as input a kernel source code and reports any missing, inconsistent, and redundant permission checks. PeX uses KIRIN (Kernel InteRface based Indirect call aNalysis), a novel, precise, and scalable indirect call analysis technique, leveraging the common programming paradigm used in kernel abstraction interfaces. Over the interprocedural control flow graph built by KIRIN, PeX automatically identifies all permission checks and infers the mappings between permission checks and privileged functions. For each privileged function, PeX examines all possible paths to the function to check if necessary permission checks are correctly enforced before it is called.

We evaluated PeX on the latest stable Linux kernel v4.18.5 for three types of permission checks: Discretionary Access Controls (DAC), Capabilities, and Linux Security Modules (LSM). PeX reported 36 new permission check errors, 14 of which have been confirmed by the kernel developers.

1 Introduction

Access control [38] is an essential security enforcement scheme in operating systems. They assign users (or processes) different access rights, called permissions, and enforce that only those who have appropriate permissions can access critical resources (e.g., files, sockets). In the kernel, access control is often implemented in the form of *permission checks* before the use of *privileged functions* accessing the critical resources.

Over the course of its evolution, Linux kernel has employed three different access control models: Discretionary Access Controls (DAC), Capabilities, and Linux Security Modules (LSM). *DAC* distinguishes privileged users (a.k.a., root) from unprivileged ones. The unprivileged users are subject to various permission checks, while the root bypasses them all [4]. Linux kernel v2.2 divided the root privilege into small units and introduced *Capabilities* to allow more fine-grained access control. From kernel v2.6, Linux adopted *LSM* in which various security hooks are defined and placed on critical paths of privileged operations. These security hooks can be instantiated with custom checks, facilitating different security model implementations as in SELinux [41] and AppArmor [3].

Unfortunately, for a new feature or vulnerability found, these access controls have been applied to the Linux kernel code in an ad-hoc manner, leading to *missing*, *inconsistent*, or *redundant* permission checks. Given the ever-growing complexity of the kernel code, it is becoming harder to manually reason about the mapping between permission checks and privileged functions. In reality, kernel developers rely on their own judgment to decide which checks to use, often leading to over-approximation issues. For instance, *Capabilities* were originally introduced to solve the "super" root problem, but it turns out that more than 38% of *Capabilities* indeed check `CAP_SYS_ADMIN`, rendering it yet another root [5].

Even worse, *there is no systematic, sound, and scalable way to examine whether all privileged functions (via all possible paths) are indeed protected by correct permission checks*. The lack of tools for checking the soundness of existing or new permission checks can jeopardize the kernel security putting the privileged functions at risk. For example, DAC, CAP and LSM introduce hundreds of security checks scattered over millions of lines of the kernel code, and it is an open problem to verify if all code paths to a privileged function encounter its corresponding permission check before reaching the function. Given the distributed nature of kernel development and the significant amount of daily updates, chances are that some

*This work was started when Tong Zhang interned at Samsung Research America, mentored by Wenbo Shen and Ahmed M. Azab.
†Corresponding author.
‡Now at Google.

parts of the code may miss checks on some paths or introduce the inconsistency between checks, weakening the operating system security.

This paper presents PeX, a static permission check analysis framework for Linux kernel. PeX makes it possible to soundly and scalably detect any missing, inconsistent and redundant permission checks in the kernel code. At a high level, PeX statically explores all possible program paths from user-entry points (e.g., system calls) to privileged functions and detects permission check errors therein. Suppose PeX finds a path in which a privileged function, say `PF`, is protected (preceded) by a check, say `Chk` in one code. If it is found that any other paths to `PF` bypass `Chk`, then it is a strong indication of a missing check. Similarly, PeX can detect inconsistent and redundant permission checks. While conceptually simple, it is very challenging to realize a sound and precise permission check error detection at the scale of Linux kernel.

In particular, there are two daunting challenges that PeX should address. First, Linux kernel uses indirect calls very frequently, yet its static call graph analysis is notoriously difficult. The latest Linux kernel (v4.18.5) contains 15.8M LOC, 247K functions, and 115K indirect callsites, rendering existing precise solutions (e.g., SVF [43]) unscalable. Only workaround available to date is either to apply the solutions unsoundly (e.g., only on a small code partition as with K-Miner [22]) or to rely on naive imprecise solutions (e.g., type-based analysis). Either way leads to undesirable results, i.e., false negatives (K-Miner) or positives (type-based one).

For a precise and scalable indirect call analysis, we introduce a novel solution called *KIRIN* (Kernel InteRface based Indirect call aNalysis), which leverages kernel abstraction interfaces to enable precise yet scalable indirect call analysis. Our experiment with Linux v4.18.5 shows that KIRIN allows PeX to detect many previously unknown permission check bugs, while other existing solutions either miss many of them or introduce too many false warnings.

Second, unlike Android which has been designed with the permission-based security model in mind [2], Linux kernel does not document the mapping between a permission check and a privileged function. More importantly, the huge Linux kernel code base makes it practically impossible to review them all manually for the permission check verification.

To tackle this problem, PeX presents a new technique which takes as input a small set of known permission checks and automatically identifies all other permission checks including their wrappers. Moreover, PeX's dominator analysis [31] automates the process of identifying mappings between permission checks and their potentially privileged functions as well. Our experiment with Linux kernel v4.18.5 shows that starting from a small set of well-known 3 DAC, 3 Capacities, and 190 LSM checks, PeX automatically (1) identifies 19, 16, and 53 additional checks, respectively, and (2) derives 9243 pairs of permission checks and privileged functions.

The contributions of this paper are summarized as follows:

Table 1: Commonly used permission checks in Linux.

Type	*Total #*	*Permission Checks*
DAC	3	generic_permission, sb_permission, inode_permission
Capabilities	3	capable, ns_capable, avc_has_perm_noaudit
LSM	190	security_inode_readlinkat, security_file_ioctl, etc..

- **New Techniques**: We proposed and implemented PeX, a static permission check analysis framework for Linux kernel. We also developed new techniques that can perform scalable indirect call analysis and automate the process of identifying permission checks and privileged functions.
- **Practical Impacts**: We analyzed DAC, Capabilities, and LSM permission checks in the latest Linux kernel v4.18.5 using PeX, and discovered 36 new permission check bugs, 14 of which have been confirmed by kernel developers.
- **Community Contributions**: We will release PeX as an open source project, along with the identified mapping between permission checks and privileged functions. This will allow kernel developers to validate their codes with PeX, and to contribute to PeX by refining the mappings with their own domain knowledge.

2 Background: Permission Checks in Linux

This section introduces DAC, Capabilities, and LSM in Linux kernel. Table 1 lists practically-known permission checks in Linux. Unfortunately, the full set is not well-documented.

2.1 Discretionary Access Control (DAC)

DAC restricts the accesses to critical resources based on the identity of subjects or the group to which they belong [36,46]. In Linux, each user is assigned a user identifier (uid) and a group identifier (gid). Correspondingly, each file has properties including the owner, the group, the `rwx` (read, write, and execute) permission bits for the owner, the group, and all other users. When a process wants to access a file, DAC grants the access permissions based on the process's uid, gid as well as the file's permission bits. For example in Linux, `inode_permission` (as listed in Table 1) is often used to check the permissions of the current process on a given inode. More precisely speaking, however, it is a wrapper of `posix_acl_permission`, which performs the actual check.

In a sense, DAC is a coarse-grained access control model. Under the Linux DAC design, the "root" bypasses all permission checks. This motivates fine-grained access control scheme—such as Capabilities—to reduce the attack surface.

2.2 Capabilities

Capabilities, since Linux kernel v2.2 (1999), enable a fine-grained access control by dividing the root privilege into small sets. As an example, for users with the `CAP_NET_ADMIN` capability, kernel allows them to use `ping`, without the need to grant the full root privilege. Currently, Linux kernel v4.18.5 supports 38 Capabilities including `CAP_NET_ADMIN`,

CAP_SYS_ADMIN, and so on. Functions capable and ns_capable are the most commonly used permission checks for Capabilities (as listed in Table 1). Both determine whether a process has a particular capability or not, while ns_capable performs an additional check against a given user namespace. They internally use security_capable as the basic permission check.

Capabilities are supposed to be fine-grained and distinct [4]. However, due to the lack of clear scope definitions, the choice of specific Capability for protecting a privileged function has been made based on kernel developers' own understanding in practice. Unfortunately, this leads to frequent use of CAP_SYS_ADMIN (451 out of 1167, more than 38%), and it is just treated as yet another root [5]; grsecurity points out that 19 Capabilities are indeed equivalent to the full root [1].

2.3 Linux Security Module (LSM)

LSM [51], introduced in kernel v2.6 (2003), provides a set of fine-grained pluggable hooks that are placed at various security-critical points across the kernel. System administrators can register customized permission checking callbacks to the LSM hooks so as to enforce diverse security policies. The latest Linux kernel v4.18.5 defines 190 LSM hooks. One common use of LSM is to implement Mandatory Access Control (MAC) [8] in Linux (e.g., SELinux [40, 41], AppArmor [3]). MAC enforces more strict and non-overridable access control policies, controlled by system administrators. For example, when a process tries to read the file path of a symbolic link, security_inode_readlink is invoked to check whether the process has read permission to the symlink file. The SELinux callback of this hook checks if a policy rule can grant this permission (e.g., allow domain_a type_b:lnk_file read). It is worth noting that the effectiveness of LSM and its MAC mechanisms highly depend on whether the hooks are placed *correctly* and *soundly* at all security-critical points. If a hook is missing at any critical point, there is no way for MAC to enforce a permission check.

3 Examples of Permission Check Errors

This section illustrates different kinds of permission check errors, found by PeX and confirmed by the Linux kernel developers. We refer to those functions, that validate whether a process (a user or a group) has proper permission to do certain operations, as *permission checks*. Similarly, we define *privileged functions* to be those functions which only a privileged process can access and thus require permission checks.

3.1 Capability Permission Check Errors

Figure 1 shows real code snippets of Capability permission check errors in Linux kernel v4.18.5. Figure 1a shows the kernel function scsi_ioctl, in which sg_scsi_ioctl (Line 7) is safeguarded by two Capability checks, CAP_SYS_ADMIN and CAP_SYS_RAWIO (Line 5). To the contrary, scsi_cmd_ioctl in Figure 1b calls the same function sg_scsi_ioctl (Line 8) without any Capability check. These two functions share three similarities. First, both of them are reachable from the userspace by ioctl system call. Second, both call sg_scsi_ioctl with a userspace parameter, void __user *arg. Last, there is no preceding Capability check on all possible paths to them (though scsi_ioctl performs two checks).

```
1  int scsi_ioctl(struct scsi_device *sdev, int cmd,
   ↪  void __user *arg)
2  {
3    ...
4    case SCSI_IOCTL_SEND_COMMAND:
5      if (!capable(CAP_SYS_ADMIN) ||
       ↪  !capable(CAP_SYS_RAWIO))
6        return -EACCES;
7      return sg_scsi_ioctl(sdev->request_queue, NULL,
       ↪  0, arg);
8    ...
9  }
```

(a) sg_scsi_ioctl (Line 7) is called **with** CAP_SYS_ADMIN and CAP_SYS_RAWIO capability checks (Line 5). arg is user space controllable.

```
1  int scsi_cmd_ioctl(struct request_queue *q, ...,
   ↪  void __user *arg)
2  {
3    ...
4    case SCSI_IOCTL_SEND_COMMAND:
5      ...
6      if (!arg)
7        break;
8      err = sg_scsi_ioctl(q, bd_disk, mode, arg);
9      break;
10   ...
11   return err;
12 }
```

(b) sg_scsi_ioctl (Line 8) is called **without** capability checks. arg is user space controllable.

```
1  int sg_scsi_ioctl(struct request_queue *q, struct
   ↪  gendisk *disk, fmode_t mode, struct
   ↪  scsi_ioctl_command __user *sic)
2  {
3    ...
4    err = blk_verify_command(req->cmd, mode);
5    ...
6    return err;
7  }
8
9  int blk_verify_command(unsigned char *cmd, fmode_t
   ↪  mode)
10 {
11   ...
12   if (capable(CAP_SYS_RAWIO))
13     return 0;
14   ...
15   return -EPERM;
16 }
```

(c) sg_scsi_ioctl calls blk_verify_command, which checks CAP_SYS_RAWIO capability.

Figure 1: Capability check errors discovered by PeX.

The kernel is supposed to sanitize userspace inputs and check permissions to ensure that only users with appropriate permissions can conduct certain privileged operations. As SCSI (Small Computer System Interface) functions manipulate the hardware, they should be protected by Capabilities. At first glance, scsi_ioctl seems to be correctly protected (while scsi_cmd_ioctl misses two Capability checks).

However, delving into sg_scsi_ioctl ends up with a different conclusion. As shown in Figure 1c, sg_scsi_ioctl calls blk_verify_command, which in turn checks CAP_SYS_RAWIO. Considering all together, scsi_ioctl checks CAP_SYS_ADMIN once but CAP_SYS_RAWIO "twice", leading to a *redundant* permission check. On the other hand, scsi_cmd_ioctl checks

```
1  static int do_readlinkat(int dfd, const char __user
   ↪  *pathname, char __user *buf, int bufsiz)
2  {
3    ...
4    error = security_inode_readlink(path.dentry);
5    if (!error) {
6      touch_atime(&path);
7      error = vfs_readlink(path.dentry, buf, bufsiz);
8    }
9    ...
10 }
```

(a) Kernel LSM usage in system call `readlinkat`. `vfs_readlink` (Line 7) is protected by `security_inode_readlink` (Line 4). Both `pathname` and `buf` (Line 1 and Line 7) are user controllable.

```
1  int ksys_ioctl(unsigned int fd, unsigned int cmd,
   ↪  unsigned long arg)
2  {
3    ...
4    error = security_file_ioctl(f.file, cmd, arg);
5    if (!error)
6      error = do_vfs_ioctl(f.file, fd, cmd, arg);
7    ...
8  }
9
10 int xfs_readlink_by_handle(struct file *parfilp,
   ↪  xfs_fsop_handlereq_t *hreq)
11 {
12   ...
13   error = vfs_readlink(dentry, hreq->ohandle, olen);
14   ...
15 }
```

(b) Kernel LSM usage in system call `ioctl`. It calls `security_file_ioctl` (Line 4) to protect `do_vfs_ioctl` (Line 6). `hreq->ohandle` and `olen` are also user controllable.

Figure 2: LSM check errors discovered by PeX.

only `CAP_SYS_RAWIO`, resulting in a *missing* permission check for `CAP_SYS_ADMIN`. In particular, PeX detects this bug as an *inconsistent* permission check because the two paths disagree with each other, and further investigation shows that one is redundant and the other is missing.

3.2 LSM Permission Check Errors

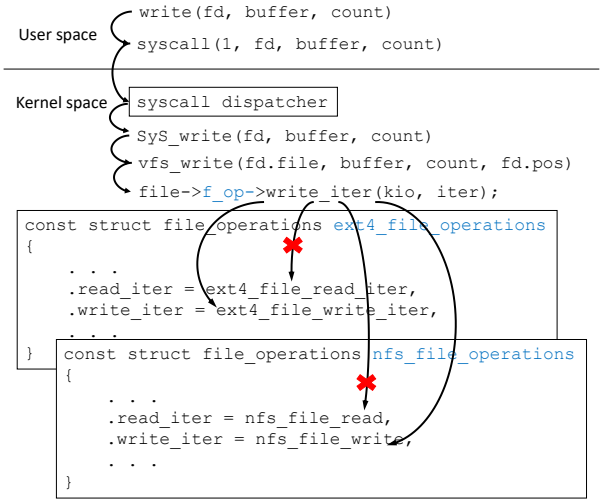
The example of LSM permission check errors is related to how LSM hooks are instrumented for two different system calls `readlinkat` and `ioctl`.

Figure 2a shows the LSM usage in the `readlinkat` system call. On its call path, `vfs_readlink` (Line 7) is protected by the LSM hook `security_inode_readlink` (Line 4) so that a LSM-based MAC mechanism, such as SELinux or AppArmor, can be realized to allow or deny the `vfs_readlink` operation.

Figure 2b presents two sub-functions for the system call `ioctl`. Similar to the above case, `ioctl` calls `ksys_ioctl`, which includes its own LSM hook `security_file_ioctl` (Line 4) before `do_vfs_ioctl` (Line 6). This is proper design, and there is no problem so far. However, it turns out that there is a path from `do_vfs_ioctl` to `xfs_readlink_by_handle` (Line 10), which eventually calls the same privileged function `vfs_readlink` (see Line 7 in Figure 2a and Line 13 in Figure 2b). While this function is protected by the `security_inode_readlink` LSM hook in `readlinkat`, that is not the case for the path to the function going through `xfs_readlink_by_handle`. The problem is that SELinux maintains separate 'allow' rules for `read` and `ioctl`. With the *missing* LSM `security_inode_readlink` check, a user only with

```
1  struct file_operations {
2      ...
3      ssize_t (*read_iter) (struct kiocb *, struct
       ↪  iov_iter *);
4      ssize_t (*write_iter) (struct kiocb *, struct
       ↪  iov_iter *);
5      ...
6  }
```

(a) The Virtual File System (VFS) kernel interface.



(b) VFS indirect calls in Linux kernel.

Figure 3: Indirect call examples via the VFS kernel interface.

the 'ioctl allow rule' may exploit the `ioctl` system call to trigger the `vfs_readlink` operation, which should only be permitted by the different 'read allow rule'.

The above two Capability and LSM examples show how challenging it is to ensure correct permission checks. There are no tools available for kernel developers to rely on to figure out whether a particular function should be protected by a permission check; and, (if so) which permission checks should be used.

4 Challenges

This section discusses two critical challenges in designing static analysis for detecting permission errors in Linux kernel.

4.1 Indirect Call Analysis in Kernel

The first challenge lies in the frequent use of indirect calls in Linux kernel and the difficulties in statically analyzing them in a scalable and precise manner. To achieve a modular design, the kernel proposes a diverse set of abstraction layers that specify the common *interfaces* to different concrete implementations. For example, Virtual File System (VFS) [12] abstracts a file system, thereby providing a unified and transparent way to access local (e.g., `ext4`) and network (e.g., `nfs`) storage devices. Under this kernel programming paradigm, an abstraction layer defines an interface as a set of indirect function pointers while a concrete module initializes these pointers with its own implementations. For example, as shown in Figure 3a, VFS abstracts all file system operations in a *ker-*

nel interface `struct file_operations` that contains a set of function pointers for different file operations. When a file system is initialized, it initializes the VFS interface with the concrete function addresses of its own. For instance, Figure 3b shows that `ext4` file system sets the `write_iter` function pointer to `ext4_file_write_iter`, while `nfs` sets the pointer to `nfs_file_write`.

However, kernel's large code base challenges the resolution of these numerous function pointers within kernel interfaces. For example, the kernel used in our evaluation (v4.18.5) includes 15.8M LOC, 247K functions, and 115K indirect callsites. This huge code base makes existing precise pointer analysis techniques [23–25, 35, 43] unscalable. In fact, Static Value Flow (SVF) [43], i.e., the state-of-the-art analysis that uses flow- and context-sensitive value flow for high precision, failed to scale to the huge Linux kernel. That is because SVF is essentially a whole program analysis, and its indirect call resolution thus requires tracking all objects such as functions, variables, and so on, making the value flow analysis unscalable to the large-size Linux kernel. In our experiment of running SVF for the kernel on a machine with 256GB memory, SVF was crashed due to an out of memory error[1].

Alternatively, one may opt for a simple "type-based" function pointer analysis, which would scale to Linux kernel. However, the type-based indirect call analysis would suffer from serious imprecision with too many *false* targets, because function pointers in the kernel often share the same type. For example, in Figure 3a, two function pointers `read_iter` and `write_iter` share the same function type. Type based pointer analysis will even link `write_iter` to `ext4_file_read_iter` falsely, which may lead to false permission check warnings.

PeX addresses this problem with a new kernel-interface aware indirect call analysis technique, detailed in §5.

4.2 The Lack of Full Permission Checks, Privileged Functions, and Their Mappings

The second challenge lies in soundly enumerating a set of permission checks and inferring correct mappings between permission checks and privileged functions in Linux kernel.

Though some commonly used permission checks for DAC, Capabilities, and LSM are known (Table 1), kernel developers often devise custom permission checks (wrappers) that internally use basic permission checks. Unfortunately, the complete list of such permission checks has never been documented. For example, `ns_capable` is a commonly used permission check for Capabilities, but it calls `ns_capable_common` and `security_capable` in sequence. It is the last `security_capable` that performs the actual capability check. In other words, all the others are "wrappers" of the "basic" permission check `security_capable`. This example shows how hard it is for a permission check analysis tool to keep up with all permission checks.

To make matters worse, Linux kernel has no explicit documentation that specifies which privileged function should be protected by which permission checks. This is different from Android [2], which has been designed with the permission-based security model in mind from the beginning. Take the Android `LocationManager` class as an example; for the `getLastKnownLocation` method, the API document states explicitly that permission `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION` is required [7].

Unfortunately, existing *static* permission error checking techniques are not readily applicable in order to address these problems. Automated LSM hook verification [44] works only with clearly defined LSM hooks, which would miss many wrappers in the kernel setting. Many other tools require heavy manual efforts such as user-provided security rules [20, 56], authorization constraints [33], annotation on sensitive objects [21]. These manual processes are particularly error-prone when applied to huge Linux code base. Alternatively, some works such as [18, 32] rely on *dynamic* analysis. However, such run-time approaches may significantly limit the code coverage being analyzed, thereby missing real bugs.

Moreover, all of above existing works cannot detect permission checks soundly. Their inability to recognize permission checks or wrappers leads to missing privileged functions or false warnings for those that are indeed protected by wrappers. Since the huge Linux kernel code base makes it practically impossible to review them all manually, reasoning about the mapping is considered to be a daunting challenge.

In light of this, PeX presents a novel static analysis technique that takes as input a small set of known permission checks to identify their basic permission checks and leverages them as a basis for finding other permission check wrappers (§6.2). In addition, PeX proposes a dominator analysis based solution to automatically infer the mappings between permission checks and privileged functions (§6.3).

5 KIRIN Indirect Call Analysis

PeX proposes a precise and scalable indirect call analysis technique, called KIRIN (Kernel InteRface based Indirect call aNalysis), on top of the LLVM [27] framework. KIRIN is inspired by two key observations: (1) almost all (95%) indirect calls in the Linux kernel are originated from kernel interfaces (§4.1) and (2) the type of a kernel interface is preserved both at its initialization site (where a function pointer is defined) and at the indirect callsite (where a function pointer is used) in LLVM IR. For example in Figure 3b, the kernel interface object `ext4_file_operations` of the type `struct file_operations` is statically initialized where `ext4_file_write_iter` is assigned to the field of `write_iter`. For the indirect call site `file→f_op→write_iter`, one can identify that `f_op` is of the type `struct file_operations` and

[1] SVF internally uses LLVM SparseVectors to save memory overhead by only storing the set bits. However, it still blows up both the memory and the computation time due to the expensive insert, expand and merge operations.

```
@ext4_file_operations = dso_local local_unnamed_addr
    ↪ constant %struct.file_operations {
%struct.module* null,
i64 (%struct.file*, i64, i32)* @ext4_llseek,
i64 (%struct.file*, i8*, i64, i64*)* null,
i64 (%struct.file*, i8*, i64, i64*)* null,
i64 (%struct.kiocb*, %struct.iov_iter*)*
    ↪ @ext4_file_read_iter,
i64 (%struct.kiocb*, %struct.iov_iter*)*
    ↪ @ext4_file_write_iter,
```

(a) LLVM IR of `ext4_file_operations` initialization.

```
%25 = load %struct.file_operations*,
    ↪ %struct.file_operations** %f_op, align 8
%write_iter.i.i = getelementptr inbounds
    ↪ %struct.file_operations,
    ↪ %struct.file_operations* %25, i64 0, i32 5
%26 = load i64 (%struct.kiocb*, %struct.iov_iter*)*,
    ↪ i64 (%struct.kiocb*, %struct.iov_iter*)**
    ↪ %write_iter.i.i, align 8
%call.i.i = call i64 %26(%struct.kiocb* nonnull
    ↪ %kiocb.i, %struct.iov_iter* nonnull %iter.i) #10
```

(b) LLVM IR of callsite `file→f_op→write_iter` in `vfs_write`.

Figure 4: Indirect callsite resolution for `vfs_write`.

infer that `ext4_file_write_iter` is one of potential call targets. Based on this observation, PeX first collects indirect call targets at kernel interface initialization sites (§5.1) and then resolves them at indirect callsites (§5.2).

5.1 Indirect Call Target Collection

In Linux kernel, a kernel interface is often defined in a `C struct` comprised of function pointers (§4.1): e.g., `struct file_operations` in Figure 3a. Many kernel interfaces (`C structs`) are *statically* allocated and initialized as with `ext4_file_operations` and `nfs_file_operations` in Figure 3b. Some interfaces may be *dynamically* allocated and initialized at run time for reconfiguration.

For the former, KIRIN scans all Linux kernel code linearly to find all statically allocated and initialized `struct` objects with function pointer fields. Then, for each `struct` object, KIRIN keep tracks of which function address is assigned to which function pointers field using an offset as a key for the field. For instance, Figure 4a shows the LLVM IR of statically initialized `ext4_file_operations`. KIRIN finds that the kernel interface type is `struct file_operations` (Line 1), and `ext4_file_write_iter` is assigned to the 5th field `write_iter` (Line 7). Therefore, KIRIN figures out that `write_iter` may point to `ext4_file_write_iter`, not `ext4_file_read_iter` (even though they have the same function type).

For the rest dynamically initialized kernel interfaces, KIRIN performs a data flow analysis to collect any assignment of a function address to the function pointer inside a kernel interface. KIRIN's field-sensitive analysis allows the collected targets to be associated with the individual field of a kernel interface.

```
struct usb_driver* driver =
    ↪ container_of(intf->dev.driver, struct
    ↪ usb_driver, drvwrap.driver);
retval = driver->unlocked_ioctl(intf,
    ↪ ctl->ioctl_code, buf);
```

(a) C code of a `container_of` usage, followed by an indirect call.

```
#define container_of(ptr, type, member) ({          \
    void *__mptr = (void *)(ptr);                   \
    ((type *)(__mptr - offsetof(type, member))); })
%unlocked_ioctl = getelementptr inbounds i8*, i8**
    ↪ %add.ptr76, i64 3
```

(b) Original `container_of` and the LLVM IR for the callsite.

```
#define container_of(ptr, type, member) ({          \
    type* __res;                                    \
    void* __mptr = ((void *)((void*)(ptr) -         \
    ↪ offsetof(type, member)));                     \
    memcpy(&__res, &__mptr, sizeof(void*));         \
    (__res);})
%unlocked_ioctl = getelementptr inbounds
    ↪ %struct.usb_driver, %struct.usb_driver* %20, i64
    ↪ 0, i32 3
```

(c) Modified `container_of` and the LLVM IR for the callsite.

Figure 5: Fixing `container_of` missing struct type problem.

5.2 Indirect Callsite Resolution

KIRIN stores the result of the above first pass in a key-value map data structure in which the key is a pair of kernel interface type and an offset (a field), and the value is a set of call targets. At each indirect callsite, KIRIN retrieves the type of a kernel interface and the offset from LLVM IR, looks up the map using them as a key, and figures out the matched call targets. For example, Figure 4b shows the LLVM IR snippet in which an indirect call `file→f_op→write_iter` is made inside of `vfs_write`. When an indirect call is made (Line 4), KIRIN finds that the kernel interface type is `struct file_operations` (Line 1) and the offset is 5 (Line 2). In this way, KIRIN reports that `ext4_file_write_iter` (assigned at Line 7 in Figure 4a) is one of potential call targets that are indirectly called by dereferencing `write_iter`.

When applying KIRIN to Linux kernel, we found in certain callsites, the kernel interface type is not presented in the LLVM IR, making their resolution impossible. For example, the macro `container_of` is commonly used in order to get the starting address of a `struct` object by using a pointer to its own member field. Figure 5a shows an example of using `container_of` (Line 1). It calculates the starting address of `usb_driver` through its own member `drvwrap.driver`. Based on the address, the code at Line 2 makes an indirect call by using a function pointer `unlocked_ioctl` that is another member of the `struct usb_driver` object.

Figure 5b shows the original macro `container_of` (Lines 1-3) and resulting LLVM IR (Line 4). The problem of this macro is that it involves a pointer manipulation, the LLVM IR of which voids the `struct` type information, i.e., the second argument of the macro. To solve this problem, KIRIN redefines `container_of` in a way that the `struct` type is preserved in the LLVM IR (on which KIRIN works), as in Figure 5c (Lines 1-5). This adds back the kernel interface type

`struct.usb_driver` in the LLVM IR (Line 6), thereby enabling KIRIN to infer the correct type of `driver` and resolve the targets for `unlocked_ioctl`.

Our experiment (§7.2) shows that KIRIN resolves 92% of total indirect callsites for `allyesconfig`. PeX constructs a more sound (less missing edges) and precise (less false edges) call graph than other existing workarounds (e.g., [22]).

6 Design of PeX

Figure 6 shows the architecture of PeX. It takes as input kernel source code (in the LLVM bitcode format) and common permission checks (Table 1), analyzes and reports all detected permission check errors, including missing, inconsistent, and redundant permission checks. In addition, PeX produces the mapping of permission checks and privileged functions, which has not been formally documented.

At a high-level, PeX first resolves indirect calls with our new technique called KIRIN (§5). Next, PeX builds an augmented call graph—in which indirect callsites are connected to possible targets—and cuts out only the portion reachable from user space (§6.1). Based on the partitioned call graph, PeX then generates the interprocedural control flow graph (ICFG) where each callsite is connected to the entry and the exit of the callee [17]. Then, starting from a small set of (user-provided) permission checks, PeX automatically detects their wrappers (§6.2). After that, for a given permission check, PeX identifies its potentially privileged functions on top of the ICFG (§6.3), followed by a heuristic-based filter to prune obviously non-privileged functions (§6.4). Finally, for each privileged function, PeX examines all user space reachable paths to it to detect any permission checks error on the paths (§6.5). The following section describes these steps in detail.

6.1 Call Graph Generation and Partition

PeX generates the call graph leveraging the result of KIRIN (§5), and then partitions it into two groups.

User Space Reachable Functions: Starting from functions with the common prefix `SyS_` (indicating system call entry points), PeX traverses the call graph, marks all visited functions, and treats them as user space reachable functions. The user reachable functions in this partition are investigated for possible permission check errors.

Kernel Initialization Functions: Functions that are used only during booting are collected to detect redundant checks. The Linux kernel boots from the `start_kernel` function, and calls a list of functions with the common prefix `__init`. PeX performs multiple call graph traversals starting from `start_kernel` and each of the `__init` functions to collect them.

Other functions such as IRQ handlers and kernel thread functions are not used in later analysis since they cannot be directly called from user space. The partitioned call graph serves as a basis for building an interprocedural control flow graph (ICFG) [31] used in the inference of the mapping between permission checks and privileged functions (§6.3).

6.2 Permission Check Wrapper Detection

Sound and precise detection of permission check errors requires a complete list of permission checks, but they are not readily available (§4.2). One may name some commonly used permission checks, as in Table 1. However, they are often the wrapper of basic permission checks, which actually perform the low-level access control, and even worse there could be other wrappers of the wrapper.

PeX solves this by automating the process of identifying all permission checks including wrappers. PeX takes an incomplete list of user-provided permission checks as input. Starting from them, PeX detects basic permission checks, by performing the *forward call graph slicing* [26, 37, 45] over the augmented call graph. For a given permission check function, PeX searches all call instructions inside the function for the one that passes an argument of the function to the callee. In other words, PeX identifies the callees of the permission check function which take its *actual* parameter as their own *formal* parameter. Similarly, PeX then conducts *backward call graph slicing* [26, 37, 45] from these basic permission checks to detect the list of their wrappers. PeX refers to only those callers that pass permission parameters as wrappers, excluding other callers just using the permission checks.

Figure 7 shows an example of the permission check wrapper detection. Given a known permission check `ns_capable` (Lines 10-13), PeX first finds `security_capable` (Line 4) as a basic permission check, and then based on it, PeX detects another permission check wrapper `has_ns_capability` (Lines 14-20). Note that the parameter `cap` is passed from both the parents `ns_capable_common` and `has_ns_capability` to the child `security_capable`; and the result of `security_capable` is returned to them. Our evaluation (§7.3) shows that based on 196 permission checks in Table 1, PeX detects 88 wrappers.

6.3 Privileged Function Detection

It is important to understand the mappings between permission checks and privileged functions for effective detection of any permission check errors therein. However, the lack of clear mapping in Linux kernel complicates the detection of permission check errors (§4.2).

To address this problem, PeX leverages an interprocedural *dominator* analysis [31] that can automatically identify the privileged functions protected by a given permission check. PeX conservatively treats all targets (callees) of those call instructions, that are dominated by each permission check (§6.2) on top of the ICFG (§6.1), as its *potential* privileged functions. The rationale behind the dominator analysis is based on the following observation: since there is no single path that allows the dominated call instruction to be reached without visiting the dominator (i.e., the permission check),

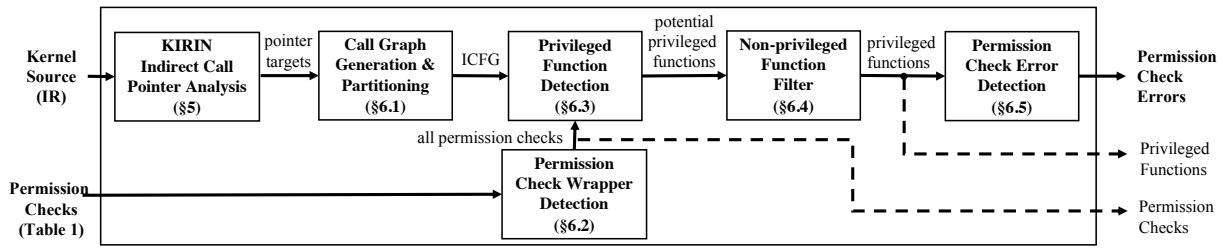


Figure 6: PeX static analysis architecture. PeX takes as input kernel source code and permission checks, and reports as output permission check errors. PeX also produces mappings between identified permission checks and privileged functions as output.

```
1  static bool ns_capable_common(struct user_namespace
   ↪  *ns, int cap, bool audit)
2  {
3      ....
4      capable = audit ?
       ↪  security_capable(current_cred(), ns, cap) :
5          security_capable_noaudit(current_cred(), ns,
           ↪  cap);
6      if (capable == 0)
7          return true;
8      return false;
9  }
10 bool ns_capable(struct user_namespace *ns, int cap)
11 {
12     return ns_capable_common(ns, cap, true);
13 }
14 bool has_ns_capability(struct task_struct *t,
15                        struct user_namespace *ns, int cap)
16 {
17     ...
18     ret = security_capable(__task_cred(t), ns, cap);
19     ...
20 }
```

Figure 7: Permission check wrapper examples.

Algorithm 1 Privileged Function Detection

INPUT:
pcfuncs - all permission checking functions
OUTPUT:
pvfuncs - privileged functions

```
1:  procedure PRIVILEGED FUNCTION DETECTION
2:      for f ← pcfuncs do
3:          for u ← User(f) do
4:              CallInst ← CallInstDominatedBy(u)   ▷ Inter-procedural analysis, for full program path
5:              callee ← getCallee(CallInst)
6:              pvfuncs.insert(callee)
7:          end for
8:      end for
9:      return pvfuncs
10: end procedure
```

the callee is likely to be the one that should be protected by the check on all paths [2].

Algorithm 1 shows how PeX uses the dominator analysis to find potential privileged functions `pvfuncs` for a given list of permission check functions `pcfuncs`. For each permission check function `f` (Line 2), PeX finds all users of `f`, i.e., the callsite invoking `f` (Line 3). For each user (callsite) `u`, PeX performs interprocedural dominator analysis over the ICFG to find all dominated call instructions (Line 4). All their callees are then added to `pvfuncs` (Lines 5-6).

Note that the call graph generated by KIRIN (§5) has resolved most of the indirect calls, which allows PeX to perform—on top of the resulting ICFG—more sound privileged function detection. For example, our experiment (§7.3) shows that KIRIN can identify `ecryptfs_setxattr` (reachable via indirect calls over the ICFG) as a privileged function and detect its missing permission check bug (Table 6, LSM-17). Note that if some other unsound workaround such as [22] had been used, this bug could not have been detected.

[2] This does not necessarily mean that the permission check dominates all call instructions of ICFG which invoke the resulting privileged function. As long as some call instructions are dominated by the check, their callees are treated as privileged functions.

6.4 Non-privileged Function Filter

The conservative approach in §6.3 may lead to too many potential privileged functions. In this step, PeX applies heuristic-based filters to prune out false privileged functions. In the current prototype, the filter contains a set of kernel library functions which are not privileged functions, e.g., `kmalloc`, `strcmp`, `kstrtoint`. Though PeX is currently designed to avoid false negatives (and thus leverages a small set of library filters only), one can add more aggressive filters to purge more false privileged functions. With releasing PeX, we expect a good opportunity for the kernel development community to contribute to the design of non-privileged function filters where domain knowledge is helpful.

6.5 Permission Check Error Detection

This last step is validating the use of privileged functions to detect any potential permission check errors. For a given mapping between a permission check and a privileged function, PeX performs a backward traversal of the ICFG, starting from the privileged functions with the corresponding permission check in mind. Note that PeX validates every possible path to each privileged function of interest.

Algorithm 2 shows PeX's permission check error detection algorithm. Recall that PeX treats user reachable kernel functions and kernel initialization functions separately and detects different forms of errors (§6.1). Lines 2-12 shows how PeX detects missing, redundant, and inconsistent checks in user reachable kernel functions. For each privileged function `f` (Line 5) in a mapping, PeX finds all possible paths `allpath` from user entry points to that privileged function `f` over the ICFG (Line 6). Line 7-18 checks each path `p` for the preceding permission check function, the lack of which should be reported as a bug. If the call to the privileged function (`pvcall`) is not preceded by the corresponding permission check func-

Algorithm 2 Permission Check Error Detection

```
INPUT:
    pc − pv - permission check function to privileged function mapping
    pcfuncs - all permission check functions
    kinitfuncs - kernel init functions
1:  procedure PERMISSION CHECK ERROR DETECTION
2:      for pair ← pc − pv do
3:          pvfuncs ← pair.pv                          ▷ privileged functions
4:          pcfunc ← pair.pc                           ▷ permission check functions
5:          for f ← pvfuncs do
6:              allpath ← getAllPathUseFunc(f)         ▷ get all user reachable paths that
    call the privileged function f
7:              for p ← allpath do
8:                  pvcall ← PrivilegeFunctionCallInPath(p)
9:                  if pvcall not Preceded by pcfunc then
10:                     if pvcall not Preceded by any pcfuncs then
11:                         report(p)                  ▷ Report missing checks
12:                     else
13:                         report(p)                  ▷ Report inconsistent check
14:                     end if
15:                 else if pvcall Preceded by multiple same pcfunc then
16:                     report(p)                      ▷ Report redundant checks
17:                 end if
18:             end for
19:         end for
20:     end for
21:     for f ← kinitfuncs do
22:         if f uses any pcfuncs then
23:             report(f)                  ▷ Report unnecessary checks during kernel boot
24:         end if
25:     end for
26: end procedure
```

tion (`pcfunc`) and any other check functions (those in `pcfuncs`) over a given path `p`, then PeX reports a missing check (Lines 6-7). And if `pvcall` is preceded not by the corresponding check (`pcfunc`) but other check in `pcfuncs`, PeX reports an inconsistent check. Finally, if PeX discovers that `pvcall` is indeed preceded by `pcfunc` checks but multiple times, then it reports a redundant check (Lines 15-17). Besides, Lines 21-25 shows how PeX detects redundant checks in kernel initialization functions. As `kinitfuncs` includes a conservative list of functions that can only be executed during booting (thus obviating the need of any checks), all detected permission checks are marked as redundant (Lines 22-24).

7 Implementation and Evaluation

PeX was implemented using LLVM [27]/Clang-6.0. It contains about 7K lines of C/C++ code. Clang was modified to preserve the kernel interface type at allocation/initialization sites by using an *identified struct* type instead of using unnamed *literal struct* type. We also automated the generation of the single-file whole vmlinux LLVM bitcode `vmlinux.bc` using `wllvm` [13]. This avoids building each kernel module separately or changing kernel build infrastructures, as observed in prior kernel static analysis works [22, 49]. We evaluated PeX on the latest stable Linux kernel v4.18.5. In summary, KIRIN resolves 86%–92% of indirect callsites depending on its compilation configurations. PeX reported 36 permission check errors warnings to the Linux community, 14 of which have been confirmed as real bugs.

Table 2: Input Statistics for Kernel v4.18.5.

	defconfig	**allyesconfig**
# of yes(=y) config	1284	9939
# of compiled LOC	2,414,772	15,881,692
vmlinux size	481 MB	3.8 GB
vmlinux.bc size	387 MB	3.3 GB
# of total functions	42,264	247,465
# of syscall entries	857	1,027
# of init functions	1,570	9,301
# of indirect callsites (ICS)	20,338	115,537

Table 3: Indirect Call Pointer Analysis.

	defconfig			**allyesconfig**		
	KIRIN	**TYPE**	**KM**	**KIRIN**	**TYPE**	**KM**
% of ICS resolved	86	100	1	92	100	na
# of avg target	3.6	10K	3.6	6.2	81K	na
analysis time (min)	1	1	9,869	6.6	1	na

7.1 Evaluation Methodology

We evaluated PeX with two different kernel configurations: (1) `defconfig`, the (commonly-used) default configuration, and (2) `allyesconfig` with all non-conflict configuration options enabled. The use of `allyesconfig` not only stress-tests PeX (including KIRIN) with a larger code base than `defconfig`, but also covers the majority of kernel code, allowing PeX to detect more bugs. In addition, we used 3 DAC, 3 Capabilities, and 190 LSM permission checks(Table 1) as input permission checks, from which PeX finds other wrappers. For the non-privileged function filter, we collected 1827 library functions from `lib` directory in the kernel source code. All experiments were carried out on a machine running Ubuntu 16.04 with two Intel Xeon E5-2650 2.20GHz CPU and 256GB DRAM.

7.2 Evaluation of KIRIN

We compared the effectiveness and efficiency of KIRIN with type-based approach and SVF-based K-Miner approach.

K-Miner [22] works around the scalability problem in SVF by analyzing the kernel on a per system call basis, rather than taking the entire kernel code for analysis. K-Miner generates a (small-size) partition of kernel code which can be reached from a given system call, and (unsoundly) applies SVF for that partition. For comparison, we took K-Miner's implementation from the github [6] and added the logic to count the number of resolved indirect callsites and the average number of targets per callsite. As K-Miner was originally built on LLVM/Clang-3.8.1, we recompiled the same kernel v4.18.5 using `wllvm` with the same kernel configurations.

Table 3 summaries evaluation results of KIRIN, comparing it to the type-based approach and K-Miner approach in terms of the percentage of indirect callsite (ICS) resolved, the average number of targets per ICS, and the total analysis time.

7.2.1 Resolution Rate

For K-Miner, we observe somewhat surprising results: it resolves only 1% of all indirect callsites. After further inves-

tigation, we noticed that SVF runs on each partition whose code base is smaller than the whole kernel, its analysis scope is significantly limited and unable to resolve function pointers in other partitions, leading to the poor resolution rate.

Besides, we found out that K-Miner does not work for `allyesconfig` which contains a much larger code base than `defconfig`. Note that K-Miner evaluated its approach only for `defconfig` in the original paper [22]. The K-Miner approach turns out to be not scalable to handle `allyesconfig` which ends up encountering out of memory error even for analyzing a single system call.

7.2.2 Resolved Average Targets

For KIRIN, the number of average indirect call targets per resolved indirect callsite is much smaller than that of the type-based approach as shown in the second row of Table 3. The reason is that the type-based approach classifies all functions (not only address-taken functions) into different sets based on the function type. This implies that all functions in the set are regarded as possible call targets of that function pointer. For example, as shown in Figure 3a, two functions `ext4_file_read_iter` and `ext4_file_write_iter` share the same type. As a result, the type-based approach incorrectly identifies both functions as possible call targets of the function pointer `f_ops→write_iter`.

7.2.3 Analysis Time

The total analysis times of each ICS resolution approach are shown in the last row of Table 3. As expected, the type-based approach is the fastest, finishing analysis in 1 minute for both configurations. KIRIN runs slower than the type-based approach. However, the analysis time of KIRIN (≈1 minute) is comparable to that of the type-based approach for `defconfig`, while KIRIN takes 6.6 minutes for `allyesconfig`.

For a fair comparison with K-Miner, care must be taken when we collect its indirect call analysis time. For a given system call, we measured K-Miner's running time from the beginning until it produces the SVF point-to result, which does not include the later bug detection time. To obtain the total analysis time of K-Miner, we summed up the running times of all system calls. The result shows that SVF based K-Miner takes about 9,869 minutes to finish analyzing all system calls of `defconfig`, which is much slower than KIRIN's.

7.3 PeX Result

Table 4 summarizes PeX's intermediate program analyses. As `allyesconfig` subsumes `defconfig` in static analysis, we focus on discussing `allyesconfig` results here. Overall, PeX finishes all analyses within a few hours and reports about two thousand groups of warnings, which are classified by privileged functions. One may implement a multi-threaded version of PeX to further reduce the analysis time.

Given the small number of input DAC, CAP, and LSM permission checks (3, 3, and 190 each), PeX's permission check

Table 4: PeX Results.

	defconfig			**allyesconfig**		
	DAC	**CAP**	**LSM**	**DAC**	**CAP**	**LSM**
# of input checks	3	3	190	3	3	190
# of detected wrappers	11	13	34	19	16	53
# of priv func detected	174	869	2030	631	3770	10915
# of priv func after filter	116	582	1635	537	3245	10260
# of warnings grouped by priv func	72	210	853	221	850	1017
total time (min)	6	8	11	83	247	169

Table 5: Comparison of PeX warnings when used with different indirect call analyses.

	defconfig				**allyesconfig**			
	DAC	**CAP**	**LSM**	Bugs	**DAC**	**CAP**	**LSM**	Bugs
KIRIN	72	210	853	21	221	850	1017	36
TYPE	218	348	1319	21	164	964	4364	19 (PeX Timeout)
KM	54	196	241	6	na	na	na	na (SVF Timeout)

detection (§6.2) was able to identify 19, 16 and 53 permission check wrappers. For example, PeX detects wrappers such as `nfs_permission` and `may_open` for DAC; `sk_net_capable` and `netlink_capable` for Capabilities; and `key_task_permission` and `__ptrace_may_access` for LSM.

Table 4 also shows the number of potentially privileged functions detected by PeX (§6.3) and the number of remaining privileged functions after kernel library filtering (§6.4). We found that there are typically 1-to-1 or 2-to-1 mapping between permission checks and privileged functions. Overall, PeX reports 221, 850, and 1017 warnings (grouped by privileged functions) for DAC, CAP, and LSM, respectively.

Table 6 shows the list of 36 bugs we reported, 14 of which have been confirmed by Linux kernel developers. Kernel developers ignored some bugs and decided not to make changes because they believe that the bugs are not exploitable. We discuss them in detail in §7.5.

Comparison. To highlight the effectiveness of KIRIN, we repeated the end-to-end PeX analysis using type-based (PeX+TYPE) and K-Miner-style (PeX+KM) indirect call analyses. Table 5 shows the resulting number of warnings and detected bugs when the 36 bugs— shown in Table 6—are used as an oracle for false negative comparison.

For `allyesconfig`, PeX+TYPE and PeX+KM could not complete the analysis within the 12-hour experiment limit. PeX+TYPE generated too many (false) edges in ICFG and suffered from path explosion during the last phase of PeX analysis; only 19 bugs were reported before the timeout. In the mean time, PeX+KM timed out on an earlier pointer analysis phase, thereby failing to report any bug.

When `defconfig` is used for comparison, PeX+TYPE and PeX+KM were able to complete the analysis. In this setting, PeX+KIRIN (original) and PeX+TYPE both report 21 bugs (a subset of 36 bugs detected with `allyesconfig`). Though PeX+TYPE can capture them all (as type-based analysis is

sound yet imprecise), it generates up to 3x more warnings, placing a high burden on the users side for their manual review. On the other hand, as an unsound solution, PeX+KM produces a limited number of warnings, resulting in the detection of only 6 bugs missing the rest.

7.4 Manual Review of Warnings

The manual review process of reported warnings is to determine whether a privileged function identified by PeX (§6.3) is a *true* privileged function or not. As long as one can confirm that a function is indeed privileged, reported warnings regarding its missing, inconsistent, and redundant permission checks should be *true positives* from PeX's point of view.

Though kernel developers with domain knowledge may be able to discern them with no complication, we (as a third-party) try to understand whether a given function can be used to access critical resources (e.g., device, file system, etc.). As a result, we conservatively reported 36 bug warnings to the community; we suspect that there could be more true warnings missed due to our lack of domain knowledge. We plan to release PeX and the list of potential privileged functions, hoping kernel developers will contribute to identify privileged functions and fix more true permission errors.

Certain static paths reported by PeX may not be feasible dynamically during program execution, resulting in false positives. One may devise a solution solving path constraints as in symbolic execution engines [16] to address this problem, PeX currently does not do so.

7.5 Discussion of Security Bug Findings

7.5.1 Missing Check

Figure 2b is one of the confirmed missing LSM checks (LSM-21). We discuss two more confirmed cases.

The CAP-4 missing check in kernel `random` device driver is particularly critical and triggered active discussion in the kernel developer community (including Torvalds). Random number generator serves as the foundation of many cryptography libraries including OpenSSL, and thus the quality of the random number is very critical. This security bug allows attackers to manipulate entropy pool, which can potentially corrupt many applications using cryptography libraries. Specifically, a problematic path starts from `evdev_write` and reaches the privileged function `credit_entropy_bits`, which can control the entropy in the entropy pool, while bypassing the required `CAP_SYS_ADMIN` permission check.

The LSM-21 missing check in `xfs_file_ioctl` led to another interesting discussion among kernel developers [9]. With this interface, a userspace program may perform low-level file system operations, but `security_inode_read_link` LSM hook was missing. An adversary could exploit this interface and gain access to the whole file system that is not allowed by LSM policy. Interestingly, however, the privileged function performed `CAP_SYS_ADMIN` Capability permission check. This created disagreement between kernel developers: one group argues that the LSM hook is necessary, while another thinks that `CAP_SYS_ADMIN` is sufficient. We agree with the former because LSM is designed to limit the damage of a compromised process to the system, even the ones of root user [40]. We believe that LSM permission checks should still be enforced as always for better security even when the current user is root.

Kernel developers decided not to fix 9 of our reports because they believe these bugs are not exploitable. As discussed earlier, PeX in the current form neither validates if a suspicious static path is dynamically reachable, nor generates a concrete exploit to demonstrate the security issue; we believe both are good future works. Nonetheless, we have one complaint to share.

For the LSM-19 and LSM-20 cases, PeX found that the LSM hooks `security_kernel_read_file` and `security_kernel_post_read_file` were used to protect the privileged functions `kernel_read_file` and `kernel_post_read_file` in some program paths. We reported missing LSM hooks in `load_elf_binary` and `load_elf_library` for these privileged functions. However, the kernel developers responded that those hooks are used to monitor loading firmware/kernel modules only (not other files), and thus no patch is required. Here, the implication we found is three-fold. First, the permission check names are ambiguous and misleading. Second, we were not able to find any documentation of such LSM specification regarding the protection of firmware/kernel modules. Last, PeX actually found a counter-example in `IMA` where the same checks are indeed used for loading other files (neither firmware nor kernel modules). Consequently, we suggest changing the function name and documenting the clear intention to avoid any confusion and to prevent system administrators from creating an LSM policy that does not work.

7.5.2 Inconsistent Check

The CAP-13 inconsistent check has been discussed in Figure 1. One program path in Figures 1a and 1c has two `CAP_SYS_RAWIO` checks and one `CAP_SYS_ADMIN` check, while another path in Figures 1b and 1c has only one `CAP_SYS_ADMIN` check. PeX detects this bug as an inconsistent check, but from the viewpoint of correction, which requires adding `CAP_SYS_RAWIO`, this may also be viewed as a missing check. There is a separate redundant check error in `CAP_SYS_RAWIO`.

Upon further investigation, we were interested in learning the practices in using multiple capabilities together. `scsi_ioctl` in Figure 1a checks both `CAP_SYS_ADMIN` **and** `CAP_SYS_RAWIO`. However, in a different network subsystem (not shown), we found that `too_many_unix_fds` performs a *weaker* permission check with the `CAP_SYS_ADMIN` **or** `CAP_SYS_RAWIO` condition. We believe this OR-based weaker check is not a good practice because this in effect makes `CAP_SYS_ADMIN` too powerful (like root), diminishing the ben-

Table 6: Bugs Reported By PeX. **C**onfirmed or **I**gnored.

Type-#	File	Function	Description	Status
DAC-1	fs/btrfs/send.c	`btrfs_send`	missing DAC check when traversing a snapshot	C
DAC-2	fs/ecryptfs/inode.c	`ecryptfs_removexattr(),_setxattr()`	missing `xattr_permission()`	C
DAC-3	fs/ecryptfs/inode.c	`ecryptfs_listxattr()`	missing `xattr_permission()`	C
CAP-4	drivers/char/random.c	`write_pool(), credit_entropy_bits()`	missing `CAP_SYS_ADMIN`	C
CAP-5	drivers/scsi/sg.c	`sg_scsi_ioctl()`	missing `CAP_SYS_ADMIN` or `CAP_RAW_IO`	I
CAP-6	drivers/block/pktcdvd.c	`add_store(), remove_store()`	missing `CAP_SYS_ADMIN`	I
CAP-7	drivers/char/nvram.c	`nvram_write()`	missing `CAP_SYS_ADMIN`	I
CAP-8	drivers/firmware/efi/efivars.c	`efivar_entry_set()`	missing `CAP_SYS_ADMIN`	C
CAP-9	net/rfkill/core.c	`rfkill_set_block(), rfkill_fop_write()`	missing `CAP_NET_ADMIN`	C
CAP-10	block/scsi_ioctl.c	`mmc_rpmb_ioctl()`	missing verify_command or `CAP_SYS_ADMIN`	I
CAP-11	drivers/platform/x86/thinkpad_acpi.c	`acpi_evalf()`	missing `CAP_SYS_ADMIN`	I
CAP-12	drivers/md/dm.c	`dm_blk_ioctl()`	missing `CAP_RAW_IO`	I
CAP-13	block/bsg.c	`bsg_ioctl`	inconsistent/missing `CAP_SYS_ADMIN`	C
CAP-14	kernel/sys.c	`prctl_set_mm_exe_file`	inconsistent capability check	I
CAP-15	kernel/sys.c	`prctl_set_mm_exe_file`	inconsistent capability and namespace check	I
CAP-16	block/scsi_ioctl.c	`blk_verify_command`	redundant check `CAP_SYS_RAWIO`	I
LSM-17	fs/ecryptfs/inode.c	`ecryptfs_removexattr(), _setxattr()`	missing `security_inode_removexattr()`	C
LSM-18	mm/mmap.c	`remap_file_pages`	missing `security_mmap_file()`	I
LSM-19	fs/binfmt_elf.c	`load_elf_binary()`	missing `security_kernel_read_file`	I
LSM-20	fs/binfmt_elf.c	`load_elf_library()`	missing `security_kernel_read_file`	I
LSM-21	fs/xfs/xfs_ioctl.c	`xfs_file_ioctl()`	missing `security_inode_readlink()`	C
LSM-22	kernel/workqueue.c	`wq_nice_store()`	missing `security_task_setnice()`	C
LSM-23	fs/ecryptfs/inode.c	`ecryptfs_listxattr()`	missing `security_inode_listxattr`	C
LSM-24	include/linux/sched.h	`comm_write()`	missing `security_task_prctl()`	C
LSM-25	fs/binfmt_misc.c	`load_elf_binary()`	missing `security_bprm_set_creds()`	I
LSM-26	drivers/android/binder.c	`binder_set_nice`	missing `security_task_setnice()`	I
LSM-27	fs/ocfs2/cluster/tcp.c	`o2net_start_listening()`	missing `security_socket_bind`	I
LSM-28	fs/ocfs2/cluster/tcp.c	`o2net_start_listening()`	missing `security_socket_listen`	I
LSM-29	fs/dlm/lowcomms.c	`tcp_create_listen_sock`	missing `security_socket_bind`	I
LSM-30	fs/dlm/lowcomms.c	`tcp_create_listen_sock`	missing `security_socket_listen`	I
LSM-31	fs/dlm/lowcomms.c	`sctp_listen_for_all`	missing `security_socket_listen`	I
LSM-32	net/socket.c	`kernel_bind`	missing `security_socket_bind`	I
LSM-33	net/socket.c	`kernel_listen`	missing `security_socket_listen`	I
LSM-34	net/socket.c	`kernel_connect`	missing `security_socket_connect`	I
LSM-35	fs/ocfs2/cluster/tcp.c	`o2net_start_listening()`	redundant `security_socket_create`	C
LSM-36	fs/ocfs2/cluster/tcp.c	`o2net_open_listening_sock()`	redundant `security_socket_create`	C

efit of fine-grained capability-based access control.

The CAP-14 and CAP-15 inconsistent error reports were acknowledged but ignored by the kernel developers for the following reason. For the same privileged function `prctl_set_mm_exe_file`, which is used to set an executable file, PeX discovered one case requiring `CAP_SYS_RESOURCE` in `user namespace`, and another case checking `CAP_SYS_ADMIN` in `init namespace`. Kernel developers responded that each case is fine by design for that specific context. PeX does not consider the precise context in which `prctl_set_mm_exe_file` is used (similar to aforementioned `security_kernel_read_file` used for loading kernel modules), leading to an imprecise report, but we believe that both CAP-14 and CAP-15 are worthwhile for further investigation.

7.5.3 Redundant Check

A redundant check occurs in two forms. **First**, for user-reachable functions, it happens when a privileged function is covered by the same permission checks multiple times. We reported three cases. The CAP-16 case was discussed in Figures 1a and 1c with two `CAP_SYS_RAWIO` checks, which was ignored by kernel developers. On the other hand, for the LSM-35 and LSM-36 cases found in the `ocfs2` file system, the other kernel developer group confirmed and promised to fix the bugs. **Second**, any permission check in kernel-initialization functions is marked as redundant because the boot thread is executed under root. PeX detected tens of such cases, but we did not report them as they are less critical.

8 Related Work

8.1 Hook Verification and Placement

There is a series of studies on checking kernel LSM hooks. Automated LSM hook verification work [56] verifies the complete mediation of LSM hooks relying manually specified security rules. While [20] automates LSM hook placements utilizing manually written specification of security sensitive operations. However, required manual processes are error-prone when applied to huge Linux code base. Edwards et al. [18] proposed to use dynamic analysis to detect LSM hook inconsistencies. While PeX is using static analysis, can achieve better code coverage.

AutoISES [44] is the most closely related work to PeX. AutoISES regards data structures, such as the structure fields and global variables, as privileged, applies static analysis to extract security check usage patterns, and validates the protections to these data structures. The difference between AutoISES and PeX is three-fold. First, PeX is privileged function oriented while AutoISES is more like data structure oriented.

Second, AutoISES is designed for LSM only, whose permission checks (hooks) are clearly defined, and therefore it is not applicable to DAC and Capabilities due to their various permission check wrappers. In contrast, PeX works for all three types of permission checks. Third, AutoISES uses type-based pointer analysis to resolve indirect calls, while PeX uses KIRIN to resolve indirect calls in a more precise manner.

There are also works [21, 32, 33] that extend authorization hook analysis to user space programs, including X server and postgresql. However, their approaches canot be applied to the huge kernel scale. Moreover, all of above works either do not analyze indirect calls at all, or apply over approximate indirect call analysis techniques, such as type-based approach or field insensitive approach. To the contrary, PeX uses KIRIN, a precise and scalable indirect call analysis technique, which can also benefit these works by finding more accurate indirect call targets.

8.2 Kernel Static Analysis Tools

Coccinelle [34] is a tool that detects a bug of pre-defined pattern based on text pattern matching on the symbolic representation of bug cases. This is basically intra-procedural analysis. Building upon Coccinelle, Wang et al. proposed another pattern matching based static tool which detects potential double-fetch vulnerabilities in the Linux kernel [48].

Sparse [11] is designed to detect the problematic use of pointers belonging to different address space (kernel space or userspace). Later, Sparse was used to build a static analysis framework called **Smatch** [10] for detecting different sorts of kernel bugs. However, Smatch is also based on intra-procedural analysis, thus it can only find shallow bugs.

Double-Fetch [52], **Check-it-again** [49] focus on detecting time of check to time of use (TOCTTOU) bugs. **Dr. Checker** [29] is designed for analyzing Linux kernel drivers. It adopts the modular design, allowing new bug detectors to be plug-in easily. **KINT** [50] applies taint analysis to detect integer errors in Linux kernel while **UniSan** [28] leverages the same analysis to detect uninitialized kernel memory leakages to the userspace. **Chucky** [53] also uses a taint analysis to analyze missing checks in different sources in userspace programs and Linux kernel. However, Chucky can handle only kernel file system code due to the lack of pointer analysis. Note that to resolve indirect call targets, all these works leverage a type-based approach, which is not as accurate as KIRIN, thus suffering from false positives.

MECA [54] is an annotation based static analysis framework, and it can detect security rule violations in Linux. **APISan** [55] aims at finding API misuse. It figures out the right API usage through the analysis of existing code base and performs intra-procedural analysis to find bugs. To achieve the former, APISan relies on relaxed symbolic execution which is complementary to the techniques used in PeX.

8.3 Permission Check Analysis Tools

Engler et al. propose to use programmer beliefs to automatically extract checking information from the source code. They apply the checking information to detect missing checks [19]. **RoleCast** [42] leverages software engineering patterns to detect missing security checks in web applications. **TESLA** [14] implements temporal assertions based on LLVM instrument, in which the FreeBSD hooks are checked by inserted assertions dynamically. Different from TESLA, PeX uses KIRIN to analyze jump targets of all kernel function pointers statically, achieving better resolution rate and code coverage. **JIGSAW** [47] is a system that can automatically derive programmer expectations on resources access and enforce it on the deployment. It is designed for analyzing userspace programs, cannot be applied to kernel directly.

JUXTA [30] is a tool designed for detecting semantic bugs in filesystem while **PScout** [15] is a static analysis tool for validating Android permission checking mechanisms. **Kratos** [39] is a static security check framework designed for the Android framework. It builds a call graph using LLVM and tries to discover inconsistent check paths in the framework. However, Android has well-documented permission check specifications [2], i.e., privileged functions and the permission required for them are both clearly defined. In contrast, the Linux kernel has no such documentation, which makes it impossible to apply PScout and Kratos to Linux kernel permission checks.

9 Conclusion

This paper presents PeX, a static permission check analysis framework for Linux kernel, which can automatically infer mappings between permission checks and privileged functions as well as detect missing, inconsistent, and redundant permission checks for any privileged functions. PeX relies on KIRIN, our novel call graph analysis based on kernel interfaces, to resolve indirect calls precisely and efficiently.

We evaluated both KIRIN and PeX for the latest stable Linux kernel v4.18.5. The experiments show that KIRIN can resolve 86%-92% of all indirect callsites in the kernel within 7 minutes. In particular, PeX reported 36 permission check bugs of DAC, Capabilities, and LSM, 14 of which have already been confirmed by the kernel developers. PeX source code is available at https://github.com/lzto/pex, along with the identified mapping between permission checks and privileged functions. We believe that such a mapping allows kernel developers to validate their code with PeX and encourages them to contribute to PeX by refining the mapping with their domain knowledge.

Acknowledgments

The authors would like to thank anonymous reviewers for their insightful comments. This research is partially supported by the NSF under Grant No. CSR-1814430 and CSR-1750503.

References

[1] alse boundaries and arbitrary code execution. https://forums.grsecurity.net/viewtopic.php?f=7&t=2522.

[2] Android Permission Overview. https://developer.android.com/guide/topics/permissions/overview.

[3] Apparmor. https://gitlab.com/apparmor/apparmor/wikis/home/.

[4] capabilities - overview of linux capabilities. http://man7.org/linux/man-pages/man7/capabilities.7.html.

[5] CAP_SYS_ADMIN: the new root. https://lwn.net/Articles/486306/.

[6] K-miner: Data-flow analysis for the linux kernel. https://github.com/ssl-tud/k-miner.

[7] Locationmanager. https://developer.android.com/reference/android/location/LocationManager#getLastKnownLocation(java.lang.String).

[8] Mandatory access control. https://en.wikipedia.org/wiki/Mandatory_access_control.

[9] Re: Leaking path in xfs's ioctl interface(missing lsm check) by stephen smalley. https://lkml.org/lkml/2018/9/26/668.

[10] Smatch: pluggable static analysis for c. https://lwn.net/Articles/691882/.

[11] Sparse. https://www.kernel.org/doc/html/v4.14/dev-tools/sparse.html.

[12] Virtual file system. https://en.wikipedia.org/wiki/Virtual_file_system.

[13] Whole Program LLVM: a wrapper script to build whole-program llvm bitcode files. https://github.com/travitch/whole-program-llvm.

[14] Jonathan Anderson, Robert NM Watson, David Chisnall, Khilan Gudka, Ilias Marinos, and Brooks Davis. Tesla: temporally enhanced system logic assertions. In *Proceedings of the Ninth European Conference on Computer Systems*, page 19. ACM, 2014.

[15] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.

[16] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[17] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Demand-driven computation of interprocedural data flow. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 37–48. ACM, 1995.

[18] Antony Edwards, Trent Jaeger, and Xiaolan Zhang. Runtime verification of authorization hook placement for the linux security modules framework. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 225–234. ACM, 2002.

[19] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 57–72. ACM, 2001.

[20] Vinod Ganapathy, Trent Jaeger, and Somesh Jha. Automatic placement of authorization hooks in the linux security modules framework. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 330–339. ACM, 2005.

[21] Vinod Ganapathy, Trent Jaeger, and Somesh Jha. Towards automated authorization policy enforcement. In *Proceedings of Second Annual Security Enhanced Linux Symposium*. Citeseer, 2006.

[22] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. K-miner: Uncovering memory corruption in linux. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2018.

[23] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *ACM SIGPLAN Notices*, volume 42, pages 290–299. ACM, 2007.

[24] Ben Hardekopf and Calvin Lin. Exploiting pointer and location equivalence to optimize pointer analysis. pages 265–280, 2007.

[25] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 289–298. IEEE Computer Society, 2011.

[26] Bogdan Korel and Juergen Rilling. Program slicing in understanding of large programs. In *Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on*, pages 145–152. IEEE, 1998.

[27] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, 2004.

[28] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 920–932. ACM, 2016.

[29] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. Dr. checker: A soundy analysis for linux kernel drivers. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1007–1024. USENIX Association, 2017.

[30] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 361–377. ACM, 2015.

[31] S.S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.

[32] Divya Muthukumaran, Trent Jaeger, and Vinod Ganapathy. Leveraging choice to automate authorization hook placement. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 145–156. ACM, 2012.

[33] Divya Muthukumaran, Nirupama Talele, Trent Jaeger, and Gang Tan. Producing hook placements to enforce expected access control policies. In *International Symposium on Engineering Secure Software and Systems*, pages 178–195. Springer, 2015.

[34] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. In *Acm sigops operating systems review*, volume 42, pages 247–260. ACM, 2008.

[35] Fernando Magno Quintao Pereira and Daniel Berlin. Wave propagation and deep propagation for pointer analysis. In *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*, pages 126–135. IEEE, 2009.

[36] Lili Qiu, Yin Zhang, Feng Wang, Mi Kyung, and Han Ratul Mahajan. Trusted computer system evaluation criteria. In *National Computer Security Center*. Citeseer, 1985.

[37] Sanjay Rawat, Laurent Mounier, and Marie-Laure Potet. Listt: An investigation into unsound-incomplete yet practical result yielding static taintflow analysis. In *Availability, Reliability and Security (ARES), 2014 Ninth International Conference on*, pages 498–505. IEEE, 2014.

[38] Ravi S Sandhu and Pierangela Samarati. Access control: principle and practice. *IEEE communications magazine*, 32(9):40–48, 1994.

[39] Yuru Shao, Qi Alfred Chen, Zhuoqing Morley Mao, Jason Ott, and Zhiyun Qian. Kratos: Discovering inconsistent security policy enforcement in the android framework. In *NDSS*, 2016.

[40] Stephen Smalley and Robert Craig. Security enhanced (se) android: Bringing flexible mac to android. In *NDSS*, volume 310, pages 20–38, 2013.

[41] Stephen Smalley, Chris Vance, and Wayne Salamon. Implementing selinux as a linux security module. *NAI Labs Report*, 1(43):139, 2001.

[42] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. Rolecast: finding missing security checks when you do not know what checks are. In *ACM SIGPLAN Notices*, volume 46, pages 1069–1084. ACM, 2011.

[43] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 265–266. ACM, 2016.

[44] Lin Tan, Xiaolan Zhang, Xiao Ma, Weiwei Xiong, and Yuanyuan Zhou. Autoises: Automatically inferring security specification and detecting violations. In *USENIX Security Symposium*, pages 379–394, 2008.

[45] Frank Tip. *A survey of program slicing techniques*. Centrum voor Wiskunde en Informatica, 1994.

[46] National Computer Security Center (US). *A guide to understanding discretionary access control in trusted systems*, volume 3. National Computer Security Center, 1987.

[47] Hayawardh Vijayakumar, Xinyang Ge, Mathias Payer, and Trent Jaeger. Jigsaw: Protecting resource access by inferring programmer expectations. In *USENIX Security Symposium*, pages 973–988, 2014.

[48] Pengfei Wang, Jens Krinke, Kai Lu, Gen Li, and Steve Dodier-Lazaro. How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel. In *USENIX Security Symposium*, 2017.

[49] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. Check it again: Detecting lacking-recheck bugs in os kernels. In *Proceedings of ACM conference on Computer and communications security*. ACM, 2018.

[50] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. Improving integer security for systems with kint. In *OSDI*, volume 12, pages 163–177, 2012.

[51] Chris Wright, Crispin Cowan, James Morris, Stephen Smalley, and Greg Kroah-Hartman. Linux security module framework. In *Ottawa Linux Symposium*, volume 8032, pages 6–16, 2002.

[52] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and scalable detection of double-fetch bugs in os kernels. 2018.

[53] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 499–510. ACM, 2013.

[54] Junfeng Yang, Ted Kremenek, Yichen Xie, and Dawson Engler. Meca: an extensible, expressive system and language for statically checking security properties. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 321–334. ACM, 2003.

[55] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. Apisan: Sanitizing api usages through semantic cross-checking. In *USENIX Security Symposium*, pages 363–378, 2016.

[56] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using cqual for static analysis of authorization hook placement. In *USENIX Security Symposium*, pages 33–48, 2002.

ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK)

Anjo Vahldiek-Oberwagner Eslam Elnikety Nuno O. Duarte
Michael Sammler Peter Druschel Deepak Garg
Max Planck Institute for Software Systems (MPI-SWS), Saarland Informatics Campus

Abstract

Isolating sensitive state and data can increase the security and robustness of many applications. Examples include protecting cryptographic keys against exploits like OpenSSL's Heartbleed bug or protecting a language runtime from native libraries written in unsafe languages. When runtime references across isolation boundaries occur relatively infrequently, then conventional page-based hardware isolation can be used, because the cost of kernel- or hypervisor-mediated domain switching is tolerable. However, some applications, such as the isolation of cryptographic session keys in network-facing services, require very frequent domain switching. In such applications, the overhead of kernel- or hypervisor-mediated domain switching is prohibitive.

In this paper, we present ERIM, a novel technique that provides hardware-enforced isolation with low overhead on x86 CPUs, even at high switching rates (ERIM's measured overhead is less than 1% for 100,000 switches per second). The key idea is to combine protection keys (MPKs), a feature recently added to x86 that allows protection domain switches in userspace, with binary inspection to prevent circumvention. We show that ERIM can be applied with little effort to new and existing applications, doesn't require compiler changes, can run on a stock Linux kernel, and has low runtime overhead even at high domain switching rates.

1 Introduction

It is good software security practice to partition sensitive data and code into isolated components, thereby limiting the effects of bugs and vulnerabilities in a component to the confidentiality and integrity of that component's data. For instance, isolating cryptographic keys in a network-facing service can thwart vulnerabilities like the OpenSSL Heartbleed bug [37]; isolating a managed language's runtime can protect its security invariants from bugs and vulnerabilities in co-linked native libraries; and, isolating jump tables can prevent attacks on an application's control flow.

Isolation prevents an untrusted component from directly accessing the private memory of other components. Broadly speaking, isolation can be enforced using one of two approaches. First, in software fault isolation (SFI) [47], one instruments the code of untrusted components with bounds checks on indirect memory accesses, to prevent access to other components' memory. The bounds checks can be added by the compiler or through binary rewriting. Bounds checks impose overhead on the execution of all untrusted components; additional overhead is required to prevent control-flow hijacks [30], which could circumvent the bounds checks. On x86-64, pointer masking-based SFI techniques like Native Client [42] incur overheads of up to 42% on the execution of untrusted code [30]. Even with hardware-supported bounds checks, like those supported by the Intel MPX ISA extension [26], the overhead is up to 30%, as shown in by Koning *et al.* [30] and later in Section 6.5.

Another approach is to use hardware page protection for memory isolation [9, 10, 13, 32, 33, 34]. Here, access checks are performed in hardware as part of the address translation with no additional overhead on execution *within* a component. However, transferring control *between* components requires a switch to kernel or hypervisor mode in order to change the (extended) page table base. Recent work such as Wedge, Shreds, SeCage, SMVs, and light-weight contexts (lwCs) [10, 13, 24, 33, 34] have reduced the overhead of such switching, but the cost is still substantial. For instance, Litton *et al.* [33] report a switching cost of about 1us per switch for lwCs, which use kernel-managed page tables for in-process isolation. This amounts to an overhead of nearly 10% for an application that switches 100,000 times a second and, in our experiments, an overhead of up to 65% on the throughput of the web server NGINX when lwCs are used to isolate session keys (Section 6.5). Techniques based on Intel VT-x extended page tables with VMFUNC [34] have less overhead, but the overhead is still high—up to 14.4% on NGINX's throughput in our experiments (Section 6.5).

In this paper, we present ERIM, the first isolation technique for x86 that combines near-zero overhead on in-component execution with very low cost switching among components. ERIM relies on a recent x86 ISA extension called protection keys (MPK) [28]. With MPK, each virtual page can be tagged with a 4-bit domain id, thus partitioning a process's address space into up to 16 disjoint domains. A special register, PKRU, that is local to each logical core determines which domains the core can read or write. Switching domain permissions requires writing the PKRU register in userspace, which takes only 11–260 cycles on current Intel CPUs, corresponding to an overhead of 0.07% to 1.0% per 100,000 switches/s on a 2.6 GHz CPU. This amounts to an overhead of at most 4.8% on the throughput of NGINX when isolating all session keys, which is up to 6.3x, 13.5x and 3x lower than the overhead of similar protection using SFI (with Intel MPX), lwCs and Intel VT-x, respectively.

However, MPK by itself does not provide strong security because a compromised or malicious component can sim-

ply write to the PKRU register and grant itself permission to access any component. ERIM relies on *binary inspection* to ensure that all occurrences of instructions that update the PKRU in the binary are *safe*, i.e., they cannot be exploited to gain unauthorized access. With this, ERIM provides isolation without requiring control-flow integrity in untrusted code, and therefore avoids the runtime overhead of ensuring control-flow integrity in unsafe languages.

While ERIM's binary inspection enforces the safety of its MPK-based isolation, it creates a potential usability issue: What to do if a binary has *unintentional* occurrences of PKRU-updating instructions? Since x86 does not require instruction alignment, such occurrences could arise within a longer instruction, or spanning the bytes of two or more adjacent instructions. Any such sequence could be exploited by a control-flow hijack attack and must be rejected by the binary inspection mechanism. To handle such cases, we describe a novel procedure to *rewrite* any instruction sequence containing an unaligned PKRU-updating instruction to a functionally equivalent sequence without the instruction. This rewriting procedure can be integrated with a compiler or our binary inspection.

ERIM is the first technique that enables efficient isolation in applications that require very high domain switching rates (~10^5/s or more) and also spend significant time executing inside untrusted components. We evaluate our ERIM prototype on three such applications: 1) Isolating the frequently accessed session keys in a web server (NGINX), 2) isolating a managed language runtime from native libraries written in unsafe languages, and 3) efficiently isolating the safe region in code-pointer integrity [31]. In all cases, we observe switching rates of order 10^5 or more per second per core. ERIM provides strong, hardware-based isolation in all these cases, with overheads that are considerably lower than those of existing techniques. Moreover, ERIM does not require compiler support and can run on stock Linux.

In summary, this paper makes the following contributions. 1) We present ERIM, an efficient memory isolation technique that relies on a combination of Intel's MPK ISA extension and binary inspection, but does not require or assume control-flow integrity. 2) We describe a complete rewriting procedure to ensure binaries cannot be exploited to circumvent ERIM. 3) We show that ERIM can protect applications with high inter-component switching rates with low overhead, unlike techniques based on hardware (extended) page tables and SFI (even with hardware support).

2 Background and related work

In this section, we survey background and related work. Enforcing relevant security or correctness invariants while trusting only a small portion of an application's code generally requires *data encapsulation*. Encapsulation itself requires *isolating* sensitive data so it cannot be accessed by untrusted code, and facilitating *switches* to trusted code that has access to the isolated state. We survey techniques for isolation and switching provided by operating systems, hypervisors, compilers, language runtimes, and binary rewriting, as well as other work that uses MPK for memory isolation.

OS-based techniques Isolation can be easily achieved by placing application components in separate OS processes. However, this method has high overhead even with a moderate rate of cross-component invocation. Novel kernel abstractions like light-weight contexts (lwCs) [33], secure memory views (SMVs) [24] and nested kernels [14], combined with additional compiler support as in Shreds [13] or runtime analysis tools as in Wedge [10], have reduced the cost of such data encapsulation to the point where isolating *long-term* signing keys in a web server is feasible with little overhead [33]. Settings that require more frequent switches like isolating *session keys* or the safe region in CPI [31], however, remain beyond the reach of OS-based techniques.

Mimosa [20] relies on the Intel TSX hardware transactional memory support to protect private cryptographic keys from software vulnerabilities and cold-boot attacks. Mimosa restricts cleartext keys to exist only within uncommitted transactions, and TSX ensures that an uncommitted transaction's data is never written to the DRAM or other cores. Unlike ERIM, which is a general-purpose isolation technique, Mimosa specifically targets cryptographic keys, and is constrained by hardware capacity limits of TSX.

Virtualization-based techniques In-process data encapsulation can be provided by a hypervisor. Dune [9] enables user-level processes to implement isolated compartments by leveraging the Intel VT-x x86 virtualization ISA extensions [28]. Koning et al. [30] sketch how to use the VT-x VMFUNC instruction to switch extended page tables in order to achieve in-process data isolation. SeCage [34] similarly relies on VMFUNC to switch between isolated compartments. SeCage also provides static and dynamic program analysis based techniques to automatically partition monolithic software into compartments, which is orthogonal to our work. TrustVisor [36] uses a thin hypervisor and nested page tables to support isolation and additionally supports code attestation. SIM [44] relies on VT-x to isolate a security monitor within an untrusted guest VM, where it can access guest memory with native speed. In addition to the overhead of the VMFUNC calls during switching, these techniques incur overheads on TLB misses and syscalls due to the use of extended page tables and hypercalls, respectively. Overall, the overheads of virtualization-based encapsulation are much higher than those of ERIM.

Nexen [45] decomposes the Xen hypervisor into isolated components and a security monitor, using page-based protection within the hypervisor's privilege ring 0. Control of the MMU is restricted to the monitor; compartments are de-privileged by scanning and removing exploitable MMU-modifying instructions. The goal of Nexen is quite different

from ERIM's: Nexen aims to isolate co-hosted VMs and the hypervisor's components from each other, while ERIM isolates components of a user process. Like ERIM Nexen scans for and removes exploitable instructions.

Language and runtime techniques Memory isolation can be provided as part of a memory-safe programming language. This encapsulation is efficient if most of the checks can be done statically. However, such isolation is language-specific, relies on the compiler and runtime, and can be undermined by co-linked libraries written in unsafe languages.

Software fault isolation (SFI) [47] provides memory isolation in unsafe languages using runtime memory access checks inserted by the compiler or by rewriting binaries. SFI imposes a continuous overhead on the execution of untrusted code. Additionally, SFI by itself does not protect against attacks that hijack control flow (to possibly bypass the memory access checks). To get strong security, SFI must be coupled with an additional technique for control-flow integrity (CFI) [6]. However, existing CFI solutions have nontrivial overhead. For example, code-pointer integrity (CPI), one of the cheapest reasonably strong CFI defenses, has a runtime overhead of at least 15% on the throughput of a moderately performant web server (Apache) [31, Section 5.3]. In contrast, ERIM does not rely on CFI for data encapsulation and has much lower overhead. Concretely, we show in Section 6 that ERIM's overhead on the throughput of a much more performant web server (NGINX) is no more than 5%.

The Intel MPX ISA extension [28] provides architectural support for bounds checking needed by SFI. A compiler can use up to four bounds registers, and each register can store a pair of 64-bit starting and ending addresses. Specialized instructions check a given address and raise an exception if the bounds are violated. However, even with MPX support, the overhead of bounds checks is of the order of tens of percent points in many applications (Section 6.5 and [12, 30, 40]).

Hardware-based trusted execution environments Intel's SGX [27] and ARM's TrustZone [8] ISA extensions allow (components of) applications to execute with hardware-enforced isolation. JITGuard [17], for instance, uses SGX to protect the internal data structures of a just-in-time compiler from untrusted code, thus preventing code-injection attacks. While SGX and TrustZone can isolate data even from the operating system, switching overheads are similar to other hardware-based isolation mechanisms [30].

IMIX [18] and MicroStach [38] propose minimal extensions to the x86 ISA, adding load and store instructions to access secrets in a safe region. The extended ISA can provide data encapsulation. Both systems provide compilers that automatically partition secrets. However, for data encapsulation in the face of control-flow hijack attacks, both systems require CFI. As mentioned, CFI techniques have nontrivial overhead. ERIM, on the other hand, provides strong isolation without relying on CFI and has lower overhead.

ASLR Address space layout randomization (ASLR) is widely used to mitigate code-reuse exploits such as those based on buffer overflow attacks [43, 23]. ASLR has also been used for data encapsulation by randomizing data layout. For example, as one of the isolation techniques used in CPI [31, 46], a region of sensitive data is allocated at a random address within the 48-bit x86-64 address space and its base address is stored in a segment descriptor. All pointers stored in memory are offsets into the region and do not reveal its actual address. However, all forms of ASLR are vulnerable to attacks like thread spraying [43, 25, 16, 19, 39]. Consequently, ASLR is not viable for strong memory isolation, despite proposals such as [35] to harden it.

ARM memory domains ARM memory domains [7] are similar to Intel MPK, the x86 feature that ERIM relies on. However, unlike in MPK, changing domains is a kernel operation in ARM. Therefore, unlike MPK, ARM's memory domains do not support low-cost user-mode switching.

MPK-based techniques Koning et al. [30] present MemSentry, a general framework for data encapsulation, implemented as a pass in the LLVM compiler toolchain. They instantiate the framework with several different memory isolation techniques, including many described above and one based on MPK domains. However, MemSentry's MPK instance is secure only with a separate defense against control-flow hijack/code-reuse attacks to prevent adversarial misuse of PKRU-updating instructions in the binary. Such defenses have significant overhead of their own. As a result, the overall overhead of MemSentry's MPK instance is significantly higher than that of ERIM, which does not rely on a defense against control-flow hijacks.

In concurrent work [22], Hedayati *et al.* describe how to isolate userspace libraries using VMFUNC or Intel MPK. The MPK-based method is similar to ERIM, but does not address the challenge of ensuring that there are no exploitable occurrences of PKRU-modifying instructions. Rewriting binaries in this manner is a key contribution of our work (Section 4). Finally, Hedayati *et al.* rely on kernel changes while ERIM can run safely on a stock Linux kernel.

libmpk [41] virtualizes MPK memory domains beyond the 16 supported in hardware. It also addresses potential security issues in the API of Linux's MPK support. libmpk addresses concerns orthogonal to ERIM because neither limitation is relevant to ERIM's use of MPK. libmpk could be combined with ERIM in applications that require more than 16 components, but the integration remains as future work.

In recent work, Burow *et al.* [11] survey implementation techniques for shadow stacks. In particular, they examine the use of MPK for protecting the integrity of shadow stacks. Burow *et al.*'s measurements of MPK overheads (Fig. 10 in [11]) are consistent with ours. Their use of MPK could be a specific use-case for ERIM, which is a more general framework for memory isolation.

3 Design

Goals ERIM enables efficient data isolation within a user-space process. Like prior work, it enables a (trusted) application component to isolate its sensitive data from untrusted components. Unlike prior work, ERIM supports such isolation with *low overhead* even at *high switching rates* between components *without requiring control-flow integrity*. In the following, we focus on the case of two components that are isolated from each other within a single-threaded process. Later, we describe generalizations to multi-threaded processes, more than two components per process, and read-only sharing among components.

We use the letter T to denote a trusted component and U to denote the remaining, untrusted application component. ERIM's key primitive is memory isolation: it reserves a region of the address space and makes it accessible exclusively from the trusted component T. This reserved region is denoted M_T and can be used by T to store sensitive data. The rest of the address space, denoted M_U, holds the application's regular heap and stack and is accessible from both U and T. ERIM enforces the following invariants:
(1) While control is in U, access to M_T remains disabled.
(2) Access to M_T is enabled atomically with a control transfer to a designated entry point in T and disabled when T transfers control back to U.
The first invariant provides isolation of M_T from U, while the second invariant prevents U from confusing T into accessing M_T improperly by jumping into the middle of M_T's code.

Background: Intel MPK To realize its goals, ERIM uses the recent MPK extension to the x86 ISA [28]. With MPK, each virtual page of a process can be associated with one of 16 protection keys, thus partitioning the address space into up to 16 *domains*. A new register, PKRU, that is local to each logical core, determines the current access permissions (read, write, neither or both) on each domain for the code running on that core. Access checks against the PKRU are implemented in hardware and impose no overhead on program execution.

Changing access privileges requires writing new permissions to the PKRU register with a *user-mode* instruction, WRPKRU. This instruction is relatively fast (11–260 cycles on current Intel CPUs), does not require a syscall, changes to page tables, a TLB flush, or inter-core synchronization.

The PKRU register can also be modified by the XRSTOR instruction by setting a specific bit in the eax register prior to the instruction (XRSTOR is used to restore the CPU's previously-saved extended state during a context switch).

For strong security, ERIM must ensure that untrusted code cannot exploit WRPKRU or XRSTOR instructions in executable pages to elevate privileges. To this end, ERIM combines MPK with binary inspection to ensure that all executable occurrences of WRPKRU or XRSTOR are *safe*, i.e., they cannot be exploited to improperly elevate privilege.

Background: Linux support for MPK As of version 4.6, the mainstream Linux kernel supports MPK. Page-table entries are tagged with MPK domains, there are additional syscall options to associate pages with specific domains, and the PKRU register is saved and restored during context switches. Since hardware PKRU checks are disabled in kernel mode, the kernel checks PKRU permissions explicitly before dereferencing any userspace pointer. To avoid executing a signal handler with inappropriate privileges, the kernel updates the PKRU register to its initial set of privileges (access only to domain 0) before delivering a signal to a process.

3.1 High-level design overview

ERIM can be configured to provide either complete isolation of M_T from U (confidentiality and integrity), or only write protection (only integrity). We describe the design for complete isolation first. Section 3.7 explains a slight design re-configuration that provides only write protection.

ERIM's isolation mechanism is conceptually simple: It maps T's reserved memory, M_T, and the application's general memory, M_U, to two different MPK domains. It manages MPK permissions (the PKRU registers) to ensure that M_U is always accessible, while only M_U is accessible when control is in U. It allows U to securely transfer control to T and back via *call gates*. A call gate enables access to M_T using the WRPKRU instruction and immediately transfers control to a specified entry point of T, which may be an explicit or inlined function. When T is done executing, the call gate disables access to M_T and returns control to U. This enforces ERIM's two invariants (1) and (2) from Section 3. Call gates operate entirely in user-mode (they don't use syscalls) and are described in Section 3.3.

Preventing exploitation A key difficulty in ERIM's design is preventing the untrusted U from exploiting occurrences of the WRPKRU or XRSTOR instruction sequence on executable pages to elevate its privileges. For instance, if the sequence appeared at any byte address on an executable page, it could be exploited using control-flow hijack attacks. To prevent such exploits, ERIM relies on *binary inspection* to enforce the invariant that only *safe* WRPKRU and XRSTOR occurrences appear on executable pages.

A WRPKRU occurrence is safe if it is immediately followed by one of the following: (A) a pre-designated entry point of T, or (B) a specific sequence of instructions that checks that the permissions set by WRPKRU do not include access to M_T and terminates the program otherwise. A safe WRPKRU occurrence cannot be exploited to access M_T inappropriately. If the occurrence satisfies (A), then it does not give control to U at all; instead, it enters T at a designated entry point. If the occurrence satisfies (B), then it would terminate the program immediately when exploited to enable access to M_T.

A XRSTOR is safe if it is immediately followed by a specific sequence of instructions to check that the eax bit that

causes XRSTOR to load the PKRU register is not set. Such a XRSTOR cannot be used to change privilege and continue execution.[1]

ERIM's call gates use only safe WRPKRU occurrences (and do not use XRSTOR at all). So, they pass the binary inspection. Section 3.4 describes ERIM's binary inspection.

Creating safe binaries An important question is how to construct binaries that do not have unsafe WRPKRUs and XRSTORs. On x86, these instructions may arise inadvertently spanning the bytes of adjacent instructions or as a subsequence in a longer instruction. To eliminate such inadvertent occurrences, we describe a binary rewriting mechanism that rewrites any sequence of instructions containing a WRPKRU or XRSTOR to a functionally equivalent sequence without any WRPKRUs and XRSTORs. The mechanism can be deployed as a compiler pass or integrated with our binary inspection, as explained in Section 4.

3.2 Threat model

ERIM makes no assumptions about the untrusted component (U) of an application. U may behave arbitrarily and may contain memory corruption and control-flow hijack vulnerabilities that may be exploited during its execution.

However, ERIM assumes that the trusted component T's binary does not have such vulnerabilities and does not compromise sensitive data through explicit information leaks, by calling back into U while access to M_T is enabled, or by mapping executable pages with unsafe/exploitable occurrences of the WRPKRU or XRSTOR instruction.

The hardware, the OS kernel, and a small library added by ERIM to each process that uses ERIM are trusted to be secure. We also assume that the kernel enforces standard DEP—an executable page must not be simultaneously mapped with write permissions. ERIM relies on a list of legitimate entry points into T provided either by the programmer or the compiler, and this list is assumed to be correct (see Section 3.4). The OS's dynamic program loader/linker is trusted to invoke ERIM's initialization function before any other code in a new process.

Side-channel and rowhammer attacks, and microachitectural leaks, although important, are beyond the scope of this work. However, ERIM is compatible with existing defenses. Our current *prototype* of ERIM is incompatible with applications that simultaneously use MPK for other purposes, but this is not fundamental to ERIM's design. Such incompatibilities can be resolved as long as the application does not re-use the MPK domain that ERIM reserves for T.

3.3 Call gates

A call gate transfers control from U to T by enabling access to M_T and executing from a designated entry point of T, and later returns control to U after disabling access to M_T. This requires two WRPKRUs. The primary challenge in designing the call gate is ensuring that both these WRPKRUs are safe in the sense explained in Section 3.1.

```
  xor ecx, ecx                                  1
  xor edx, edx                                  2
  mov PKRU_ALLOW_TRUSTED, eax                   3
  WRPKRU // copies eax to PKRU                  4

// Execute trusted component's code           6

  xor ecx, ecx                                  8
  xor edx, edx                                  9
  mov PKRU_DISALLOW_TRUSTED, eax                10
  WRPKRU // copies eax to PKRU                  11
  cmp PKRU_DISALLOW_TRUSTED, eax                12
  je continue                                   13
  syscall exit // terminate program             14
continue:                                       15
// control returns to the untrusted            16
    application here
```

Listing 1: Call gate in assembly. The code of the trusted component's entry point may be inlined by the compiler on line 6, or there may be an explicit direct call to it.

Listing 1 shows the assembly code of a call gate. WRPKRU expects the new PKRU value in the eax register and requires ecx and edx to be 0. The call gate works as follows. First, it sets PKRU to enable access to M_T (lines 1–4). The macro PKRU_ALLOW_TRUSTED is a constant that allows access to M_T and M_U.[2] Next, the call gate transfers control to the designated entry point of T (line 6). T's code may be invoked either by a direct call, or it may be inlined.

After T has finished, the call gate sets PKRU to disable access to M_T (lines 8–11). The macro PKRU_DISALLOW_TRUSTED is a constant that allows access to M_U but not M_T. Next, the call gate checks that the PKRU was actually loaded with PKRU_DISALLOW_TRUSTED (line 12). If this is not the case, it terminates the program (line 14), else it returns control to U (lines 15–16). The check on line 12 may seem redundant since eax is set to PKRU_DISALLOW_TRUSTED on line 10. However, the check prevents *exploitation* of the WRPKRU on line 11 by a control-flow hijack attack (explained next).

Safety Both occurrences of WRPKRU in the call gate are safe. Neither can be exploited by a control flow hijack to get unauthorized access to M_T. The first occurrence of WRPKRU (line 4) is immediately followed by (a direct control transfer to) a designated entry point of T. This instance can-

[1] We know of only one user-mode Linux application – the dynamic linker, `ld`, that legitimately uses XRSTOR. However, `ld` categorically does *not* restore PKRU through XRSTOR, so this safe check can be added to it.

[2] To grant read (resp. write) access to domain i, bit $2i$ (resp. $2i+1$) must be set in the PKRU. PKRU_ALLOW_TRUSTED sets the 4 least significant bits to grant read and write access to domains 0 (M_U) and 1 (M_T).

not be exploited to transfer control to anywhere else. The second occurrence of WRPKRU (line 11) is followed by a check that terminates the program if the new permissions include access to M_T. If, as part of an attack, the execution jumped directly to line 11 with any value other than PKRU_DISALLOW_TRUSTED in eax, the program would be terminated on line 14.

Efficiency A call gate's overhead on a roundtrip from U to T is two WRPKRUs, a few very fast, standard register operations and one conditional branch instruction. This overhead is very low compared to other hardware isolation techniques that rely on pages tables and syscalls or hypervisor trampolines to change privileges (see also Section 6.5).

Use considerations ERIM's call gate omits features that readers may expect. These features have been omitted to avoid having to pay their overhead when they are not needed. First, the call gate does not include support to pass parameters from U to T or to pass a result from T to U. These can be passed via a designated shared buffer in M_U (both U and T have access to M_U). Second, the call gate does not scrub registers when switching from T to U. So, if T uses confidential data, it should scrub any secrets from registers before returning to U. Further, because T and U share the call stack, T must also scrub secrets from the stack prior to returning. Alternatively, T can allocate a private stack for itself in M_T, and T's entry point can switch to that stack immediately upon entry. This prevents T's secrets from being written to U's stack in the first place. (A private stack is also necessary for multi-threaded applications; see Section 3.7).

3.4 Binary inspection

Next, we describe ERIM's binary inspection. The inspection prevents U from mapping any executable pages with unsafe WRPKRU and XRSTOR occurrences and consists of two parts: (i) an inspection function that verifies that a sequence of pages does not contain unsafe occurrences; and, (ii) an interception mechanism that prevents U from mapping executable pages without inspection.

Inspection function The inspection function *scans* a sequence of pages for instances of WRPKRU and XRSTOR. It also inspects any adjacent executable pages in the address space for instances that cross a page boundary.

For every WRPKRU, it checks that the WRPKRU is safe, i.e., either condition (A) or (B) from Section 3.1 holds. To check for condition (A), ERIM needs a list of designated entry points of T. The source of this list depends on the nature of T and is trusted. If T consists of library functions, then the programmer marks these functions, e.g., by including a unique character sequence in their names. If the functions are not inlined by the compiler, their names will appear in the symbol table. If T's functions are subject to inlining or if they are generated by a compiler pass, then the compiler must be directed to add their entry locations to the symbol table with the unique character sequence. In all cases, ERIM can identify designated entry points by looking at the symbol table and make them available to the inspection function.

Condition (B) is checked easily by verifying that the WRPKRU is immediately followed by *exactly* the instructions on lines 12–15 of Listing 1. These instructions ensure that the WRPKRU cannot be used to enable access to M_T and continue execution.

For every XRSTOR, the inspection function checks that the XRSTOR is followed immediately by the following instructions, which check that the eax bit that causes XRSTOR to load PKRU (bit 9) is not set: `bt eax, 0x9; jnc .safe; EXIT; .safe:...`. Here, `EXIT` is a macro that exits the program. Trivially, such a XRSTOR cannot be used to enable access to M_T and continue execution.

Interception On recent ($\geq$4.6) versions of Linux, interception can be implemented *without kernel changes*. We install a seccomp-bpf filter [29] that catches mmap, mprotect, and pkey_mprotect syscalls which attempt to map a region of memory as executable (mode argument `PROT_EXEC`). Since the bpf filtering language currently has no provisions for reading the PKRU register, we rely on seccomp-bpf's `SECCOMP_RET_TRACE` option to notify a ptrace()-based tracer process. The tracer inspects the tracee and allows the syscall if it was invoked from T and denies it otherwise. The tracer process is configured so that it traces any child of the tracee process as well. While ptrace() interception is expensive, note that it is required only when a program maps pages as executable, which is normally an infrequent operation.

If programs map executable pages frequently, a more efficient interception can be implemented with a simple Linux Security Module (LSM) [50], which allows mmap, mprotect and pkey_mprotect system calls only from T. (Whether such a call is made by U or T is easily determined by examining the PKRU register value at the time of the syscall.) Our prototype uses this implementation of interception. Another approach is to implement a small (8 LoC) change to seccomp-bpf in the Linux kernel, which allows a bpf filter to inspect the value of the PKRU register. With this change in place, we can install a bpf filter that allows certain syscalls only from T, similar to the LSM module.

With either interception approach in place, U must go through T to map executable pages. T maps the pages only after they have passed the inspection function. Regardless of the interception method, pages can be inspected upfront when T attempts to map them as executable, or on demand when they are executed for the first time.

On-demand inspection is preferable when a program maps a large executable segment but eventually executes only a small number of pages. With on-demand inspection, when the process maps a region as executable, T instead maps the region read-only but records that the pages are pending inspection. When control transfers to such a page, a fault occurs. The fault traps to a dedicated signal handler, which

ERIM installs when it initializes (the LSM or the tracer prevents U from overriding this signal handler). This signal handler calls a T function that checks whether the faulting page is pending inspection and, if so, inspects the page. If the inspection passes, then the handler remaps the page with the execute permission and resumes execution of the faulting instruction, which will now succeed. If not, the program is terminated.

The interception and binary inspection has very low overhead in practice because it scans an executable page at most once. It is also fully transparent to U's code if all WRPKRUs and XRSTORs in the binary are already safe.

Security We briefly summarize how ERIM attains security. The binary inspection mechanism prevents U from mapping any executable page with an unsafe WRPKRU or XRSTOR. T does not contain any executable unsafe WRPKRU or XRSTOR by assumption. Consequently, only safe WRPKRUs and XRSTORs are executable in the entire address space at any point. Safe WRPKRUs and XRSTORs preserve ERIM's two security invariants (1) and (2) by design. Thus M_T is accessible only while T executes starting from legitimate T entry points.

3.5 Lifecycle of an ERIM process

As part of a process's initialization, before control is transferred to main(), ERIM creates a second MPK memory domain for M_T in addition to the process's default MPK domain, which is used for M_U. ERIM maps a memory pool for a dynamic memory allocator to be used in M_T and hooks dynamic memory allocation functions so that invocations are transparently redirected to the appropriate pool based on the value of the PKRU register. This redirection provides programmer convenience but is not required for security. If U were to call T's allocator, it would be unable to access M_T's memory pool and generate a page fault. Next, ERIM scans M_U's executable memory for unsafe WRPKRUs and XRSTORs, and installs one of the interception mechanisms described in Section 3.4. Finally, depending on whether main() is in U or T, ERIM initializes the PKRU register appropriately and transfers control to main(). After main() has control, the program executes as usual. It can map, unmap and access data memory in M_U freely. However, to access M_T, it must invoke a call gate.

3.6 Developing ERIM applications

We describe here three methods of developing applications or modifying existing applications to use ERIM.

The *binary-only* approach requires that either U or T consist of a set of functions in a dynamic link library. In this case, the library and the remaining program can be used in unmodified binary form. An additional ERIM dynamic wrapper library is added using LD_PRELOAD, which wraps the entry points with stub functions that implement the call gates and have names that indicate to the ERIM runtime the valid entry points. We have used this approach to isolate SQLite within the Node.js runtime (Section 5).

```
typedef struct secret {                          1
  int number; } secret;                          2
secret* initSecret() {                           3
  ERIM_SWITCH_T;                                 4
  secret * s = malloc(sizeof(secret));           5
  s->number = random();                          6
  ERIM_SWITCH_U;                                 7
  return s;                                      8
}                                                9
int compute(secret* s, int m) {                  10
  int ret = 0;                                   11
  ERIM_SWITCH_T;                                 12
  ret = f(s->number, m);                         13
  ERIM_SWITCH_U;                                 14
  return ret;                                    15
}                                                16
```

Listing 2: C component isolated with ERIM

The *source* approach requires that either U or T consist of a set of functions that are not necessarily in a separate compilation unit or library. In this case, the source code is modified to wrap these functions with stubs that implement the call gates, and choose names that indicate valid entry points. We used this approach to isolate the crypto functions and session keys in OpenSSL (Section 5).

The *compiler* approach requires modifications to the compiler to insert call gates at appropriate points in the executable and generate appropriate symbols that indicate valid entry points. This approach is the most flexible because it allows arbitrary inlining of U and T code. We used this approach to isolate the metadata in CPI (Section 5).

Next, we give a simple example describing the process of developing a new C application using the *source* approach. ERIM provides a C library and header files to insert call gates, initialize ERIM, and support dynamic memory allocation. Listing 2 demonstrates an example C program that isolates a data structure called `secret` (lines 1–2). The structure contains an integer value. Two functions, `initSecret` and `compute`, access secrets and bracket their respective accesses with call gates using the macros `ERIM_SWITCH_T` and `ERIM_SWITCH_U`. ERIM isolates `secret` such that only code that appears between `ERIM_SWITCH_T` and `ERIM_SWITCH_U`, i.e., code in T, may access `secret`. `initSecret` allocates an instance of `secret` while executing inside T by first allocating memory in M_T and then initializing the secret value. `compute` computes a function `f` of the secret inside T.

3.7 Extensions

Next, we discuss extensions to ERIM's basic design.

Multi-threaded processes ERIM's basic design works as-is with multi-threaded applications. Threads are created as usual, e.g. using libpthread. The PKRU register is saved and restored by the kernel during context switches. However, multi-threading imposes an additional requirement on T (not on ERIM): In a multi-threaded application, it is essential that T allocate a private stack in M_T (not M_U) for each thread and execute its code on these stacks. This is easy to implement by switching stacks at T's entry points. Not doing so and executing T on standard stacks in M_U runs the risk that, while a thread is executing in T, another thread executing in U may corrupt or read the first thread's stack frames. This can potentially destroy T's integrity, leak its secrets and hijack control while access to M_T is enabled. By executing T's code on stacks in M_T, such attacks are prevented.

More than two components per process Our description of ERIM so far has been limited to two components (T and U) per process. However, ERIM generalizes easily to support as many components as the number of domains Linux's MPK support can provide (this could be less than 16 because the kernel may reserve a few domains for specific purposes). Components can have arbitrary pairwise trust relations with each other, as long as the trust relations are transitive. A simple setting could have a default domain that trusts all other domains (analogous to U) and any number of additional domains that do not trust any others. ERIM's initialization code creates a private heap for each component, and ERIM's custom allocator allocates from the heap of the currently executing component. Each component can also (in its own code) allocate a per-thread stack, to protect stack-allocated sensitive data when calling into other untrusted domains. Stacks can be mandatorily switched by ERIM's call gates.

ERIM for integrity only Some applications care only about the integrity of protected data, but not its confidentiality. Examples include CPI, which needs to protect only the integrity of code pointers. In such applications, efficiency can be improved by allowing U to *read* M_T directly, thus avoiding the need to invoke a call gate for reading M_T. The ERIM design we have described so far can be easily modified to support this case. Only the definition of the constant PKRU_DISALLOW_TRUSTED in Listing 1 has to change to also allow read-only access to M_T. With this change, read access to M_T is always enabled.

Just-in-time (jit) compilers with ERIM ERIM works with jit compilers that follow standard DEP and do not allow code pages that are writable and executable at the same time. Such jit compilers write new executable code into newly allocated, non-executable pages and change these pages' permissions to non-writable and executable once the compilation finishes. ERIM's mprotect interception defers enabling execute permissions until after a binary inspection, as described in Section 3.4. When a newly compiled page is executed for the first time, ERIM handles the page execute permission fault, scans the new page for unsafe WRPKRUs/XRSTORs and enables the execute permission if no unsafe occurrences exist. This mechanism is safe, but may lead to program crashes if the jit compiler accidentally emits an unsafe WRPKRU or XRSTOR. ERIM-aware jit compilers can emit WRPKRU- and XRSTOR-free binary code by relying on the rewrite strategy described in Section 4, and inserting call gates when necessary.

OS privilege separation The design described so far provides memory isolation. Some applications, however, require privilege separation between T and U with respect to OS resources. For instance, an application might need to restrict the filesystem name space accessible to U or restrict the system calls available to U.

ERIM can be easily extended to support privilege separation with respect to OS resources, using one of the techniques described in Section 3.4 for intercepting systems calls that map executable pages. In fact, intercepting and disallowing these system calls when invoked from U is just a special case of privilege separation. During process initialization, ERIM can instruct the kernel to restrict U's access rights. After this, the kernel refuses to grant access to restricted resources whenever the value of the PKRU is not PKRU_ALLOW_TRUSTED, indicating that the syscall does not originate from T. To access restricted resources, U must invoke T, which can filter syscalls.

4 Rewriting program binaries

The binary inspection described in Section 3.4 guarantees that executable pages do not contain unsafe instances of the WRPKRU and XRSTOR instructions. This is *sufficient* for ERIM's safety. In this section, we show how to generate or modify program binaries to not contain unsafe WRPKRUs and XRSTORs, so that they pass the binary inspection.

Intentional occurrences of WRPKRU that are not immediately followed by a transfer to T and all occurrences of XRSTOR, whether they are generated by a compiler or written manually in assembly, can be made safe by inserting the checks described in Section 3.4 after the instances. Inadvertent occurrences—those that arise unintentionally as part of a longer x86 instruction and operand, or spanning two consecutive x86 instructions/operands—are more interesting. We describe a rewrite strategy to eliminate such occurrences and how the strategy can be applied by a compiler or a binary rewriting tool. The strategy can rewrite any sequence of x86 instructions and operands containing an inadvertent WRPKRU or XRSTOR to a functionally equivalent sequence without either. In the following we describe the strategy, briefly argue why it is complete, and summarize an empirical evaluation of its effectiveness.

Rewrite strategy WRPKRU is a 3 byte instruction, 0x0F01EF. XRSTOR is also always a 3-byte instruction, but it has more variants, fully described by the regular expres-

Overlap with	Cases	Rewrite strategy	ID	Example
Opcode	Opcode = WRPKRU/ XRSTOR	Insert safety check after instruction	1	
Mod R/M	Mod R/M = 0x0F	Change to unused register + move command	2	add ecx, [ebx + 0x01EF0000] → mov eax, ebx; add ecx, [eax + 0x01EF0000];
		Push/Pop used register + move command	3	add ecx, [ebx + 0x01EF0000] → push eax; mov eax, ebx; add ecx, [eax + 0x01EF0000]; pop eax;
Displacement	Full/Partial sequence	Change mode to use register	4	add eax, 0x0F01EF00 → (push ebx;) mov ebx, 0x0F010000; add ebx, 0x0000EA00; add eax, ebx; (pop ebx;)
	Jump-like instruction	Move code segment to alter constant used in address	5	call [rip + 0x0F01EF00] → call [rip + 0x0FA0EEFF]
Immediate	Full/Partial sequence	Change mode to use register	6	add eax, 0x0F01EF → (push ebx;) mov ebx, 0x0F01EE00; add ebx, 0x00000100; add eax, ebx; (pop ebx;)
	Associative opcode	Apply instruction twice with different immediates to get equivalent effect	7	add ebx, 0x0F01EF00 → add ebx, 0x0E01EF00; add ebx, 0x01000000

Table 1: Rewrite strategy for intra-instruction occurrences of WRPKRU and XRSTOR

sion 0x0FAE[2|6|A][8-F]. There are two cases to consider. First, a WRPKRU or XRSTOR sequence can span two or more x86 instructions. Such sequences can be "broken" by inserting a 1-byte nop like 0x90 between the two consecutive instructions. 0x90 does not coincide with any individual byte of WRPKRU or XRSTOR, so this insertion cannot generate a new occurrence.

Second, a WRPKRU or XRSTOR may appear entirely within a longer instruction including any immediate operand. Such cases can be rewritten by replacing them with a semantically equivalent instruction or sequence of instructions. Doing so systematically requires an understanding of x86 instruction coding. An x86 instruction contains: (i) an opcode field possibly with prefix, (ii) a MOD R/M field that determines the addressing mode and includes a register operand, (iii) an optional SIB field that specifies registers for indirect memory addressing, and (iv) optional displacement and/or immediate fields that specify constant offsets for memory operations and other constant operands.

The strategy for rewriting an instruction depends on the fields with which the WRPKRU or XRSTOR subsequence overlaps. Table 1 shows the complete strategy.

An opcode field is at most 3-bytes long. If the WRPKRU (XRSTOR) starts at the first byte, the instruction *is* WRPKRU (XRSTOR). In this case, we make the instruction safe by inserting the corresponding check from Section 3.4 after it. If the WRPKRU or XRSTOR starts after the first byte of the opcode, it must also overlap with a later field. In this case, we rewrite according to the rule for that field below.

If the sequence overlaps with the MOD R/M field, we change the register in the MOD R/M field. This requires a free register. If one does not exist, we rewrite to push an existing register to the stack, use it in the instruction, and pop it back. (See lines 2 and 3 in Table 1.)

If the sequence overlaps with the displacement or the immediate field, we change the mode of the instruction to use a register instead of a constant. The constant is computed in the register before the instruction (lines 4 and 6). If a free register is unavailable, we push and pop one. Two instruction-specific optimizations are possible. First, for jump-like instructions, the jump target can be relocated in the binary; this changes the displacement in the instruction, obviating the need a free register (line 5). Second, associative operations like addition can be performed in two increments without an extra register (line 7). Rewriting the SIB field is never required because any WRPKRU or XRSTOR must overlap with at least one non-SIB field (the SIB field is 1 byte long while these instructions are 3 bytes long).

Compilers and well-written assembly programs normally do not mix data like constants, jump tables, etc. with the instruction stream and instead place such data in a non-executable data segment. If so, WRPKRU or XRSTOR sequences that occur in such data can be ignored.

Compiler support For binaries that can be recompiled from source, rewriting can be added to the codegen phase of the compiler, which converts the intermediate representation (IR) to machine instructions. Whenever codegen outputs an inadvertent WRPKRU or XRSTOR, the surrounding instructions in the IR can be replaced with equivalent instructions as described above, and codegen can be run again.

Runtime binary rewriting For binaries that cannot be recompiled, binary rewriting can be integrated with the interception and inspection mechanism (Section 3.4). When the inspection discovers an unsafe WRPKRU or XRSTOR on an executable page during its scan, it overwrites the page with

1-byte traps, makes it executable, and stores the original page in reserve without enabling it for execution. Later, if there is a jump into the executable page, a trap occurs and the trap handler discovers an entry point into the page.

The rewriter then disassembles the *reserved page* from that entry point on, rewriting any discovered WRPKRU or XRSTOR occurrences, and copies the rewritten instruction sequences back to the executable page. To prevent other threads from executing partially overwritten instruction sequences, we actually rewrite a fresh copy of the executable page with the new sequences, and then swap this rewritten copy for the executable page. This technique is transparent to the application, has an overhead proportional to the number of entry points in offending pages (it disassembles from every entry point only once) and maintains the invariant that only safe pages are executable.

A rewritten instruction sequence is typically longer than the original sequence and therefore cannot be rewritten in-place. In this case, binary rewriting tools place the rewritten sequence on a new page, replace the first instruction in the original sequence with a direct jump to the rewritten sequence, and insert a direct jump back to the instruction following the original sequence after the rewritten sequence. Both pages are then enabled for execution.

Implementation and testing The rewrite strategy is arguably complete. We have implemented the strategy as a library, which can be used either with the inspection mechanism as explained above or with a *static* binary rewrite tool, as described here. To gain confidence in our implementation, we examined all binaries of five large Linux distributions (a total of 204,370 binaries). Across all binaries, we found a total of 1213 WRPKRU/XRSTOR occurrences in code segments. We then used a standard tool, Dyninst [15], to try to disassemble and rewrite these occurrences. Dyninst was able to disassemble 1023 occurrences and, as expected, our rewriter rewrote all instances successfully. Next, we wanted to run these 1023 rewritten instances. However, this was infeasible since we did not know what inputs to the binaries would cause control to reach the rewritten instances. Hence, we constructed two hand-crafted binaries with WRPKRUs/XRSTORs similar to the 1023 occurrences, rewrote those WRPKRUs/XRSTORs with Dyninst and checked that those rewritten instances ran correctly. Based on these experiments, we are confident that our implementation of WRPKRU/XRSTOR rewriting is robust.

5 Use Cases

ERIM goes beyond prior work by providing efficient isolation with very high component switch rates of the order of 10^5 or 10^6 times a second. We describe three such use cases here, and report ERIM's overhead on them in Section 6.

Isolating cryptographic keys in web servers Isolating *long-term* SSL keys to protect from web server vulnerabilities such as the Heartbleed bug [37] is well-studied [33, 34]. However, long-term keys are accessed relatively infrequently, typically only a few times per user session. *Session keys*, on the other hand, are accessed far more frequently—over 10^6 times a second per core in a high throughput web server like NGINX. Isolating sessions keys is relevant because these keys protect the confidentiality of individual users. With its low-cost switching, ERIM can be used to isolate session keys efficiently. To verify this, we partitioned OpenSSL's low-level crypto library (libcrypto) to isolate the session keys and basic crypto routines, which run as T, from the rest of the web server, which runs as U.

Native libraries in managed runtimes Managed runtimes such as a Java or JavaScript VM often rely on third-party native libraries written in unsafe languages for performance. ERIM can isolate the runtime from bugs and vulnerabilities in a native library by mapping the managed runtime to T and the native libraries to U. This use case leverages the "integrity only" version of ERIM (Section 3.7). We isolated Node.js from a native SQLite plugin. Node.js is a state-of-the-art managed runtime for JavaScript and SQLite is a state-of-the-art database library written in C [1, 2]. The approach generalizes to isolating several mutually distrusting libraries from each other by leveraging ERIM's multi-component extension from Section 3.7.

CPI/CPS Code-pointer integrity (CPI) [31] prevents control-flow hijacks by isolating sensitive objects—code pointers and objects that can lead to code pointers—in a *safe region* that cannot be written without bounds checks. CPS is a lighter, less-secure variant of CPI that isolates only code pointers. A key challenge is to isolate the safe region efficiently, as CPI can require switching rates on the order of 10^6 or more switches/s on standard benchmarks. We show that ERIM can provide strong isolation for the safe region at low cost. To do this, we override the CPI/CPS-enabled compiler's intrinsic function for writing the sensitive region to use a call gate around an inlined sequence of T code that performs a bounds check before the write. (MemSentry [30] also proposes using MPK for isolating the safe region, but does not actually implement it.)

6 Evaluation

We have implemented two versions of an ERIM prototype for Linux.[3] One version relies on a 77 line Linux Security Module (LSM) that intercepts all mmap and mprotect calls to prevent U from mapping pages in executable mode, and prevents U from overriding the binary inspection handler. We additionally added 26 LoC for kernel hooks to Linux v4.9.110, which were needed by the LSM. We also implemented ERIM on an *unmodified* Linux kernel using the ptrace-based technique described in Section 3.4. In the

[3] Available online at `https://gitlab.mpi-sws.org/vahldiek/erim`.

following, we show results obtained with the modified kernel. The performance of ERIM on the stock Linux kernel is similar, except that the costs of `mmap`, `mprotect`, and `pkey_mprotect` syscalls that enable execute permissions are about 10x higher. Since the evaluated applications use these operations infrequently, the impact on their overall performance is negligible.

Our implementation also includes the ERIM runtime library, which provides a memory allocator over M_T, call gates, the ERIM initialization code, and binary inspection. These comprise 569 LoC. Separately, we have implemented the rewriting logic to eliminate inadvertent WRPKRU occurrences (about 2250 LoC). While we have not yet integrated the logic into either a compiler or our inspection handler, the binaries used in our performance evaluation experiments do not have any unsafe WRPKRU occurrences and do not load any libraries at runtime. However, the binaries did have two legitimate occurrences of XRSTOR (in the dynamic linker library `ld.so`), which we made safe as described in Section 3.4. Two other inadvertent XRSTOR occurred in data-only pages of executable segments in `libm`, which is used by the SPEC benchmarks. We made these safe by re-mapping the pages read-only. Hence, the results we report are on completely safe binaries.

We evaluate the ERIM prototype on microbenchmarks and on the three applications mentioned in Section 5. Unless otherwise mentioned, we perform our experiments on Dell PowerEdge R640 machines with 16-core MPK-enabled Intel Xeon Gold 6142 2.6GHz CPUs (with the latest firmware; Turbo Boost and SpeedStep were disabled), 384GB memory, 10Gbps Ethernet links, running Debian 8, Linux kernel v4.9.60. For the OpenSSL/webserver experiments in Sections 6.2 and 6.5, we use NGINX v1.12.1, OpenSSL v1.1.1 and the ECDHE-RSA-AES128-GCM-SHA256 cipher. For the managed language runtime experiment (Section 6.3), we use Node.js v9.11.1 and SQLite v3.22.0. For the CPI experiment (Section 6.4), we use the Levee prototype v0.2 available from `http://dslab.epfl.ch/proj/cpi/` and Clang v3.3.1 including its CPI compile pass, runtime library extensions and link-time optimization.

6.1 Microbenchmarks

Switch cost We performed a microbenchmark to measure the overhead of invoking a function with and without a switch to a trusted component. The function adds a constant to an integer argument and returns the result. Table 2 shows the cost of invoking the function, in cycles, as an inlined function (I), as a directly called function (DC), and as a function called via a function pointer (FP). For reference, the table also includes the cost of a simple syscall (getpid), the cost of a switch on lwCs, a recent isolation mechanism based on kernel page table protections [33], and the cost of a VMFUNC (Intel VT-x)-based extended page table switch.

In our microbenchmark, calls with an ERIM switch are between 55 and 80 cycles more expensive than their no-switch counterparts. The most expensive indirect call costs less than the simplest system call (getpid). ERIM switches are up to 3-5x faster than VMFUNC switches and up to 100x faster than lwC switches.

Call type	Cost (cycles)
Inlined call (no switch)	5
Direct call (no switch)	8
Indirect call (no switch)	19
Inlined call + switch	60
Direct call + switch	69
Indirect call + switch	99
getpid system call	152
Call + VMFUNC EPT switch	332
lwC switch [33] (Skylake CPU)	6050

Table 2: Cycle counts for basic call and return

Because the CPU must not reorder loads and stores with respect to a WRPKRU instruction, the overhead of an ERIM switch depends on the CPU pipeline state at the time the WRPKRUs are executed. In experiments described later in this section, we observed average overheads ranging from 11 to 260 cycles per switch. At a clock rate of 2.6GHz, this corresponds to overheads between 0.04% and 1.0% for 100,000 switches per second, which is significantly lower than the overhead of any kernel- or hypervisor-based isolation.

Binary inspection To determine the cost of ERIM's binary inspection, we measured the cost of scanning the binaries of all 18 applications in the CINT/FLOAT SPEC 2006 CPU benchmark. These range in size from 9 to 3918 4KB pages, contain between 35 and 63765 intentional WRPKRU instructions when compiled with CPI (see Section 6.4), no unintended WRPKRU and no XRSTOR instructions. The overhead is largely independent of the number of WRPKRU instructions and ranges between 3.5 and 6.2 microseconds per page. Even for the largest binary, the scan takes only 17.7ms, a tiny fraction of a typical process' runtime.

6.2 Protecting session keys in NGINX

Next, we use ERIM to isolate SSL session keys in a high performance web server, NGINX. We configured NGINX to use only the ECDHE-RSA-AES128-GCM-SHA256 cipher and AES encryption for sessions. We modified OpenSSL's libcrypto to isolate all session keys and the functions for AES key allocation and encryption/decryption into ERIM's T, and use ERIM call gates to invoke these functions.

To measure ERIM's overhead on the peak throughput, we configure a single NGINX worker pinned to a CPU core, and connect to it remotely over HTTPS with keep-alive from 4 concurrent ApacheBench (`ab`) [3] instances each simulating 75 concurrent clients. The clients all request the same file, whose size we vary from 0 to 128KB across experi-

File size (KB)	1 worker		3 workers		5 workers		10 workers	
	Native (req/s)	ERIM rel. (%)	Native (req/s)	ERIM rel. (%)	Native (req/s)	ERIM rel. (%)	Native (req/s)	ERIM rel. (%)
0	95,761	95.8	276,736	96.1	466,419	95.7	823,471	96.4
1	87,022	95.2	250,565	94.5	421,656	96.1	746,278	95.5
2	82,137	95.4	235,820	95.1	388,926	96.6	497,778	100.0
4	76,562	95.3	217,602	94.9	263,719	100.0		
8	67,855	96.0	142,680	100.0				

Table 3: Nginx throughput with multiple workers. The standard deviation is below 1.5% in all cases.

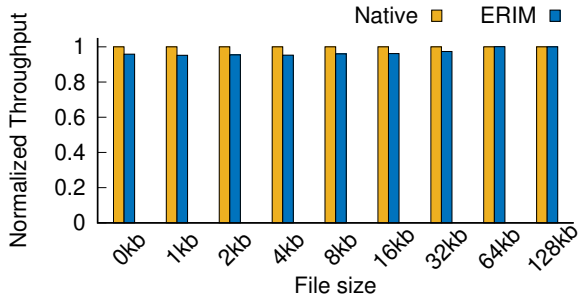


Figure 1: Throughput of NGINX with one worker, normalized to native (no protection), with varying request sizes. Standard deviations were all below 1.1%.

File size (KB)	Throughput		Switches/s	CPU load native (%)
	Native (req/s)	ERIM rel. (%)		
0	95,761	95.8	1,342,605	100.0
1	87,022	95.2	1,220,266	100.0
2	82,137	95.4	1,151,877	100.0
4	76,562	95.3	1,073,843	100.0
8	67,855	96.0	974,780	100.0
16	45,483	97.1	820,534	100.0
32	32,381	97.3	779,141	100.0
64	17,827	100.0	679,371	96.7
128	8,937	100.0	556,152	86.4

Table 4: Nginx throughput with a single worker. The standard deviation is below 1.1% in all cases.

ments.[4] Figure 1 shows the average throughput of 10 runs of an ERIM-protected NGINX relative to native NGINX without any protection for different file sizes, measured after an initial warm-up period.

ERIM-protected NGINX provides a throughput within 95.18% of the unprotected server for all request sizes. To explain the overhead further, we list the number of ERIM switches per second in the NGINX worker and the worker's CPU utilization in Table 4 for request sizes up to 128KB. The overhead shows a general trend up to requests of size 32 KB: The worker's core remains saturated but as the request size increases, the number of ERIM switches per second decrease, and so does ERIM's relative overhead. The observations are consistent with an overhead of about 0.31%–0.44% for 100,000 switches per second. For request sizes 64KB and higher, the 10Gbps network saturates and the worker does not utilize its CPU core completely in the baseline. The free CPU cycles absorb ERIM's CPU overhead, so ERIM's throughput matches that of the baseline.

Note that this is an extreme test case, as the web server does almost nothing and serves the same cached file repeatedly. To get a more realistic assessment, we set up NGINX to serve from main memory static HTML pages from a 571 MB (15,520 pages) Wikipedia snapshot of 2006 [48]. File sizes vary from 417 bytes to 522 KB (average size 37.7 KB). 75 keep-alive clients request random pages (selected based on pageviews on Wikipedia [49]). The average throughput with a single NGINX worker was 22,415 requests/s in the baseline and 21,802 requests/s with ERIM (std. dev. below 0.6% in both cases). On average, there were 615,000 switches a second. This corresponds to a total overhead of 2.7%, or about 0.43% for 100,000 switches a second.

Scaling with multiple workers To verify that ERIM scales with core parallelism, we re-ran the first experiment above with 3, 5 and 10 NGINX workers pinned to separate cores, and sufficient numbers of concurrent clients to saturate all the workers. Table 3 shows the relative overheads with different number of workers. (For requests larger than those shown in the table, the network saturates, and the spare CPU cycles absorb ERIM's overhead completely.) The overheads were independent of the number of workers (cores), indicating that ERIM adds no additional synchronization and scales perfectly with core parallelism. This result is expected as updates to the per-core PKRU do not affect other cores.

6.3 Isolating managed runtimes

Next, we use ERIM to isolate a managed language runtime from an untrusted native library. Specifically, we link the widely-used C database library, SQLite, to Node.js, a state-of-the-art JavaScript runtime and map Node.js's runtime to T and SQLite to U. We modified SQLite's entry points to invoke call gates. To isolate Node.js's stack from SQLite, we run Node.js on a separate stack in M_T, and switch to the

[4] Since NGINX only serves static files in this experiment, its support for Lua and JavaScript is not used. As a result, this experiment does not rely on any support for Jit, which we have not yet implemented.

Test #	Switches/s	ERIM overhead (%)
100	11,183,281	12.73%
110	8,329,914	12.18%
400	8,161,584	15.42%
120	7,190,766	13.81%
142	7,074,553	9.41%
500	6,419,008	12.13%
510	5,868,395	5.60%
410	5,091,212	3.64%
240	2,358,524	3.74%
280	2,303,516	3.22%
170	1,264,366	4.22%
310	1,133,364	2.92%
161	1,019,138	2.81%
160	1,014,829	2.73%
230	670,196	2.04%
270	560,257	2.28%

Table 5: Overhead relative to native execution for SQLite speedtest1 tests with more than 100,000 switches/s. Standard deviations were below 5.6%.

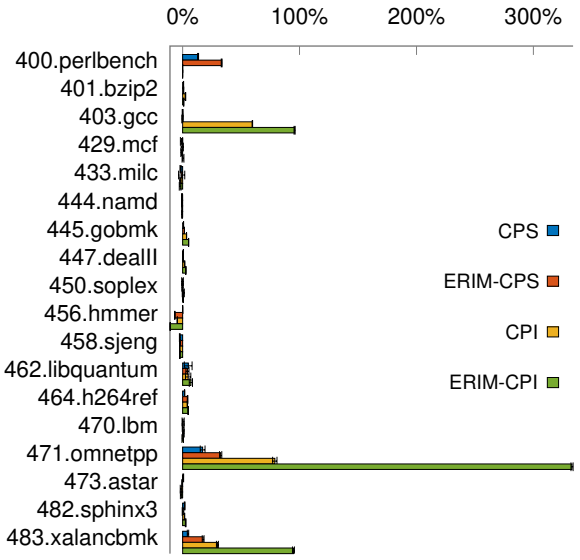


Figure 2: Percentage overhead relative to no protection.

standard stack (in M_U) prior to calling a SQLite function. Finally, SQLite uses the libc function `memmove`, which accesses libc constants that are in M_T, so we implemented a separate `memmove` for SQLite. In total, we added 437 LoC.

We measure overheads on the speedtest1 benchmark that comes with SQLite and emulates a typical database workload [4]. The benchmark performs 32 short tests that stress different database functions like selects, joins, inserts and deletes. We increased the iterations in each test by a factor of four to make the tests longer. Our baseline for comparison is native SQLite linked to Node.js without any protection. We configure the benchmark to store the database in memory and report averages of 20 runs.

The geometric mean of ERIM's runtime overhead across all tests is 4.3%. The overhead is below 6.7% on all tests except those with more than 10^6 switches per second. This suggests that ERIM can be used for isolating native libraries from managed language runtimes with low overheads up to a switching cost of the order of 10^6 per second. Beyond that the overhead is noticeable. Table 5 shows the relative overheads for tests with switching rates of at least 100,000/s. The numbers are consistent with an average overhead between 0.07% and 0.41% for 100,000 switches/s. The actual switch cost measured from direct CPU cycle counts varies from 73 to 260 cycles across all tests. It exceeds 100 cycles only when the switch rate is less than 2,000 times/s. We verified that these are due to i-cache misses—at low switch rates, the call gate instructions are evicted between switches.

6.4 Protecting sensitive data in CPI/CPS

Next, we use ERIM to isolate the safe region of CPI and CPS [31] in a separate domain. We modified CPI/CPS's LLVM compiler pass to emit additional ERIM switches, which bracket any code that modifies the safe region. The switch code, as well as the instructions modifying the safe region, are inlined with the application code. In addition, we implemented simple optimizations to safely reduce the frequency of ERIM domain switches. For instance, the original implementation sets sensitive code pointers to zero during initialization. Rather than generate a domain switch for each pointer initialization, we generate loops of pointer set operations that are bracketed by a single pair of ERIM domain switches. This is safe because the loop relies on direct jumps and the code to set a pointer is inlined in the loop's body. In all, we modified 300 LoC in LLVM's CPI/CPS pass.

Like the original CPI/CPS paper [31], we compare the overhead of the original and our ERIM-protected CPI/CPS system on the SPEC CPU 2006 CINT/FLOAT benchmarks, relative to a baseline compiled with Clang without any protection. The original CPI/CPS system is configured to use ASLR for isolation, the default technique used on x86-64 in the original paper. ASLR imposes almost no switching overhead, but also provides no security [43, 25, 16, 19, 39].

Figure 2 shows the average runtime overhead of 10 runs of the original CPI/CPS (lines "CPI/CPS") and CPI/CPS over ERIM (lines "ERIM-CPI/CPS"). All overheads are normalized to the unprotected SPEC benchmark. We could not obtain results for 400.perlbench for CPI and 453.povray for both CPS and CPI. 400.perlbench does not halt when compiled with CPI and SPEC's result verification for 453.povray fails due to unexpected output. These problems exist in the code generated by the Levee CPI/CPS prototype with CPI/CPS enabled (-fcps/-fcpi), not our modifications.

Benchmark	Switches/sec	ERIM-CPI overhead relative to orig. CPI in %
403.gcc	16,454,595	22.30%
445.gobmk	1,074,716	1.77%
447.dealII	1,277,645	0.56%
450.soplex	410,649	0.60%
464.h264ref	1,705,131	1.22%
471.omnetpp	89,260,024	144.02%
482.sphinx3	1,158,495	0.84%
483.xalancbmk	32,650,497	52.22%

Table 6: Domain switch rates of selected SPEC CPU benchmarks and overheads for ERIM-CPI without binary inspection, *relative to the original CPI with ASLR*.

CPI: The geometric means of the overheads (relative to no protection) of the original CPI and ERIM-CPI across all benchmarks are 4.7% and 5.3%, respectively. The relative overheads of ERIM-CPI are low on all individual benchmarks except gcc, omnetpp, and xalancbmk.

To understand this better, we examined switching rates across benchmarks. Table 6 shows the switching rates for benchmarks that require more than 100,000 switches/s. From the table, we see that the high overheads on gcc, omnetpp and xalancbmk are due to extremely high switching rates on these three benchmarks (between 1.6×10^7 and 8.9×10^7 per second). Further profiling indicated that the reason for the high switch rate is tight loops with pointer updates (each pointer update incurs a switch). An optimization pass could hoist the domain switches out of the loops safely using only direct control flow instructions and enforcing store instructions to be bound to the application memory, but we have not implemented it yet.

Table 6 also shows the overhead of ERIM-CPI excluding binary inspection, relative to the original CPI over ASLR (not relative to an unprotected baseline as in Figure 2). This relative overhead is exactly the cost of ERIM's switching. Depending on the benchmark, it varies from 0.03% to 0.16% for 100,000 switches per second or, equivalently, 7.8 to 41.6 cycles per switch. These results again indicate that ERIM can support inlined reference monitors with switching rates of up to 10^6 times a second with low overhead. Beyond this rate, the overhead becomes noticeable.

CPS: The results for CPS are similar to those for CPI, but the overheads are generally lower. Relative to the baseline without protection, the geometric means of the overheads of the original CPS and ERIM-CPS are 1.1% and 2.4%, respectively. ERIM-CPS's overhead relative to the original CPS is within 2.5% on all benchmarks, except except perlbench, omnetpp and xalancbmk, where it ranges up to 17.9%.

6.5 Comparison to existing techniques

In this section, we compare ERIM to isolation using SFI (with Intel MPX), extended page tables (with Intel VT-x/VMFUNC), kernel page tables (with lwCs), and instrumentation of untrusted code for full memory safety (with WebAssembly). In each case, our primary goal is a *quantitative* comparison of the technique's overhead to that of ERIM. As we show below, ERIM's overheads are substantially lower than those of the other techniques. But before presenting these results, we provide a brief *qualitative* comparison of the techniques in terms of their threat models.

Qualitative comparison of techniques Isolation using standard kernel page tables affords a threat model similar to ERIM's. In particular, like ERIM, the OS kernel must be trusted. In principle, isolation using a hypervisor's extended page tables (VMFUNC) can afford a stronger threat model, in which the OS kernel need not be trusted [34].

Isolation using SFI, with or without Intel MPX, affords a threat model weaker than ERIM's since one must additionally trust the transform that adds bounds checks to the untrusted code. For full protection, a control-flow integrity (CFI) mechanism is also needed to prevent circumvention of bounds checks. This further increases both the trusted computing base (TCB) and the overheads. In the experiments below, we omit the CFI defense, thus underestimating SFI overheads for protection comparable to ERIM's.

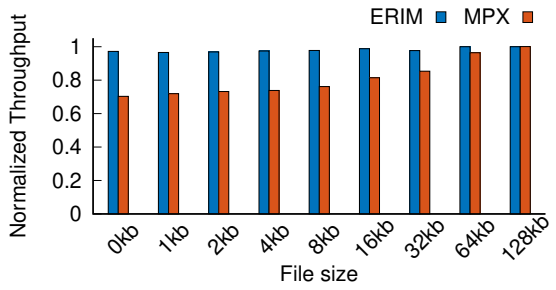
Instrumenting untrusted code for full memory safety, i.e., bounds-checking at the granularity of individual memory allocations, implicitly affords the protection that SFI provides. Additionally, such instrumentation also protects the untrusted code's data from other outside threats, a use case that the other techniques here (including ERIM) do not handle. However, as for SFI, the mechanism used to instrument the untrusted code must be trusted. In our experiments below, we enforce memory safety by compiling untrusted code to WebAssembly, and this compiler must be trusted.

Next, we quantitatively compare the overheads of these techniques to those of ERIM.

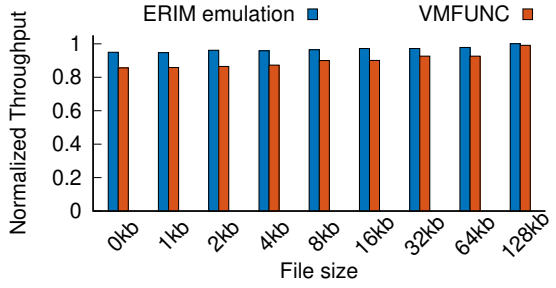
SFI using MPX We start by comparing the cost of ERIM's isolation to that of isolation based on SFI using MPX. For this, we follow the NGINX experiment of Section 6.2. We place OpenSSL (trusted) in a designated memory region, and use MemSentry [30] to compile all of NGINX (untrusted) with MPX-based memory-bounds checks that prevent it from accessing the OpenSSL region directly.[5] To get comparable measurements on the (no protection) baseline and ERIM, we recompile NGINX with Clang version 3.8, which is the version that MemSentry supports. We then re-run the single worker experiments of Section 6.2.

Figure 3a shows the overheads of MPX and ERIM on NGINX's throughput, relative to a no-protection baseline. The MPX-based instrumentation reduces the throughput of NGINX by 15-30% until the experiment is no longer CPU-

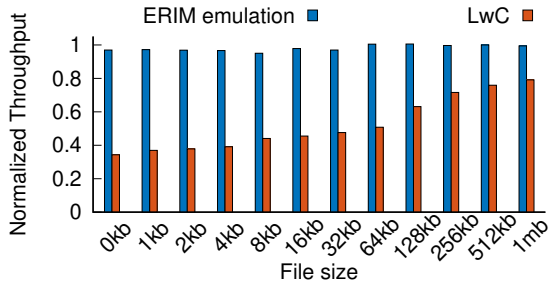
[5] This setup *reduces* the overheads of MPX as compared to the setup of Section 6.2, which isolates only small parts of OpenSSL. It is also less secure. Hence, the MPX overheads reported here are conservative.



(a) ERIM vs. SFI using MPX (averages of 3 runs, std. devs. below 1.9%)



(b) Emulated ERIM vs. VMFUNC (averages of 3 runs, std. devs. below 0.9%)



(c) Emulated ERIM vs. LwC (averages of 5 runs, std. devs. below 1.1%)

Figure 3: Comparison of NGINX throughput with ERIM and alternative isolation techniques

bound (file sizes $\geq$ 64kb). In contrast, ERIM reduces overheads by no more than 3.5%. Across all file sizes, MPX overheads are 4.2-8.5x those of ERIM.

MPX (more generally, SFI) and ERIM impose overhead in different ways. MPX imposes an overhead during the execution of NGINX (the untrusted component), while ERIM imposes an overhead on component switches. Consequently, one could argue that, as the switch rate increases, ERIM *must* eventually become more expensive than MPX. While this is theoretically true, in this experiment, we already observe extremely high switch rates of 1.2M/s (for file size 0kb) and, even then, MPX's overhead is 8.4x that of ERIM's overhead.

Further, as explained earlier, for strong security, SFI must be supported by control-flow integrity, which would induce additional overheads that are not included here.

Extended page tables (VMFUNC) Next, we compare ERIM to isolation based on extended page tables (EPTs) using Intel VT-x and VMFUNC. To get access to EPTs, we use Dune [9] and a patch from MemSentry. We create two page tables—one maps the trusted region that contains session keys, and the other maps the untrusted region that contains all the remaining state of NGINX and OpenSSL. Access to the first table is efficiently switched on or off using the VMFUNC EPT switch call provided by the MemSentry patch. This call is faster than an OS process switch since it does not switch the process context or registers. Since we use Dune, the OS kernel runs in hypervisor mode. It has the switch overheads of hypervisor-based isolation using VMFUNC but includes the OS kernel in the TCB.

Unfortunately, MemSentry's patch works only on old Linux kernels which do *not* have the page table support needed for MPKs and, hence, cannot support ERIM. Consequently, for this comparison, we rely on an emulation of ERIM's switch overhead using standard x86 instructions. This emulation is described later in this section, and we validate that it is accurate to within 2% of ERIM's actual overheads on a variety of programs. So we believe that the comparative results presented here are quite accurate.

Figure 3b shows the throughput of NGINX protected with VMFUNC and emulated ERIM, relative to a baseline with no protection for different file sizes (we use Linux kernel v3.16). Briefly, VMFUNC induces an overhead of 7-15%, while the corresponding overhead of emulated ERIM is 2.1-5.3%. Because both VMFUNC and ERIM incur overhead on switches, overheads of both reduce as the switching rate reduces, which happens as the file size increases. (The use of Dune and extended page tables also induces an overhead on all syscalls and page walks in the VMFUNC isolation.)

To directly compare VMFUNC's overheads to *actual* ERIM's, we calculated VMFUNC's overhead as a function of switch rate. Across different file sizes, this varies from 1.4%-1.87% for 100,000 switches/s. In contrast, actual ERIM's overhead in the similar experiment of Section 6.2 never exceeds 0.44% for 100,000 switches/s. This difference is consistent with the microbenchmark results in Table 2.

Kernel page tables (lwCs) Next, we compare ERIM's overhead to that of lwCs [33], a recent system for in-process isolation based on kernel page-table protections. LwCs map each isolated component to a separate address space in the same process. A switch between components requires kernel mediation to change page tables, but does not require a process context switch. To measure lwC overheads, we rerun the NGINX experiment of Section 6.2, using two lwC contexts, one for the session keys and encryption/decryption functions and the other for NGINX and the rest of OpenSSL. Unfortunately, lwCs were prototyped in FreeBSD, which does not support MPK, so we again use our emulation of

ERIM's switch overhead to compare. All experiments reported here were run on Dell OptiPlex 7040 machines with 4-core Intel Skylake i5-6500 CPUs clocked at 3.2 GHz, 16 GB memory, 10 Gbps Ethernet cards, and FreeBSD 11.

Figure 3c shows the throughput of NGINX running with lwCs and emulated ERIM, relative to a baseline without any protection. With lwCs, the throughput is never above 80% of the baseline, and for small files, where the switch rate is high, the throughput is below 50%. In contrast, the throughput with emulated ERIM is within 95% of the baseline for all file sizes. In terms of switch rates, lwCs incur a cost of 10.5-18.3% for 100,000 switches/s across different file sizes. *Actual* ERIM's switch overhead during the similar experiment of Section 6.2 is no more than 0.44% across all file sizes, which is two orders of magnitude lower than that of lwCs.

Memory safety (WebAssembly) Finally, we compare ERIM's overheads to those of full memory safety on untrusted code. Specifically, we compare to compilation of untrusted code through WebAssembly [21], a memory-safe, low-level language that is now supported natively by all major web browsers and expected to replace existing SFI techniques like Native Client in the Chrome web browser. We compare to ERIM using the experiment of Section 6.3. We re-compile the (untrusted) SQLite library to WebAssembly via emscripten v1.37.37's WebAssembly backend [5], and run the WebAssembly within Node.js, which supports the language. Accross tests of Table 5, the overhead of using WebAssembly varies from 81% to 193%, which is one to two orders of magnitude higher than ERIM's overhead.

Emulating ERIM's switch cost We describe how we emulate ERIM's switch cost when comparing to VMFUNC and lwCs above. Specifically, we need to emulate the cost of a WRPKRU instruction, which isn't natively supported in the environments of those experiments. We do this using xor instructions to consume the appropriate number of CPU cycles, followed by RDTSCP, which causes a pipeline stall and prevents instruction re-ordering. Specifically, we execute a loop five times, with `xor eax,ecx; xor ecx,eax; xor eax,ecx`, followed by a single RDTSCP after the loop.

To validate the emulation we re-ran the SPEC CPU 2006 benchmark with CPI/CPS (Section 6.4) after swapping actual WRPKRU instructions with the emulation sequence shown above and compared the resulting overheads. In each *individual* test, the difference in overhead between actual ERIM and the emulation is below 2%. We note that a perfectly precise emulation is impossible since emulation cannot exactly reproduce the effects of WRPKRU on the execution pipeline. (WRPKRU must prevent the reordering of loads and stores with respect to itself.) Depending on the specific benchmark, our emulation slightly over- or underestimates the actual performance impact of WRPKRU. We also observed that emulations of WRPKRU using LFENCE or MFENCE (the latter was suggested by [30]) in place of RDTSCP incur too little or too much overhead, respectively.

7 Conclusion

Relying on the recent Intel MPK ISA extension and simple binary inspection, ERIM provides hardware-enforced isolation with an overhead of less than 1% for every 100,000 switches/s between components on current Intel CPUs, and almost no overhead on execution within a component. ERIM's switch cost is up to two orders of magnitude lower than that of kernel page-table based isolation, and up to 3-5x lower than that of VMFUNC-based isolation. For VMFUNC, virtualization can cause additional overhead on syscalls and page table walks. ERIM's overall overhead is lower than that of isolation based on memory-bounds checks (with Intel MPX), even at switch rates of the order of 10^6/s. Additionally, such techniques require control-flow integrity to provide strong security, which has its own overhead. ERIM's comparative advantage prominently stands out on applications that switch very rapidly and spend a non-trivial fraction of time in untrusted code.

Acknowledgements We thank the anonymous reviewers, our shepherd Tom Ritter, Bobby Bhattacharjee, and Mathias Payer for their feedback, which helped improve this paper. This work was supported in part by the European Research Council (ERC Synergy imPACT 610150) and the German Science Foundation (DFG CRC 1223).

References

[1] `https://www.sqlite.org`.

[2] `https://nodejs.org`.

[3] `https://httpd.apache.org/docs/2.4/programs/ab.html`.

[4] `https://www.sqlite.org/testing.html`.

[5] `https://github.com/kripken/emscripten`.

[6] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2005.

[7] ARM Limited. Developer guide: ARM memory domains. `http://infocenter.arm.com/help/`, 2001.

[8] ARM Limited. ARM Security Technology. `http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf`, 2009.

[9] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mezières, and Christos Kozyrakis. Dune:

Safe user-level access to privileged CPU features. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[10] Andrea Bittau and Petr Marchenko. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of Networked System Design and Implementation (NSDI)*, 2008.

[11] Nathan Burow, Xinping Zhang, and Mathias Payer. SoK: Shining Light On Shadow Stacks. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2019.

[12] Scott A. Carr and Mathias Payer. Datashield: Configurable data confidentiality and integrity. In *Proceedings of ACM ASIA Conference on Computer and Communications Security (AsiaCCS)*, 2017.

[13] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-Grained Execution Units with Private Memory. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2016.

[14] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[15] Dyninst. Dyninst: An application program interface (API) for runtime code generation. `http://www.dyninst.org`.

[16] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the point(er): On the effectiveness of code pointer integrity. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[17] Tommaso Frassetto, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. JITGuard: Hardening just-in-time compilers with SGX. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[18] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. IMIX: In-process memory isolation extension. In *Proceedings of USENIX Security Symposium*, 2018.

[19] Enes Göktas, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. Undermining Information Hiding (and What to Do about It). In *Proceedings of USENIX Security Symposium*, 2016.

[20] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[21] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[22] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2019.

[23] Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. librando: Transparent Code Randomization for Just-in-Time Compilers. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013.

[24] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. Enforcing least privilege memory views for multithreaded applications. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[25] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2013.

[26] Intel Corporation. Memory Protection Extensions (Intel MPX). `https://software.intel.com/en-us/isa-extensions/intel-mpx`.

[27] Intel Corporation. Software Guard Extensions Programming Reference. `https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf`, 2014.

[28] Intel Corporation. Intel(R) 64 and IA-32 Architectures Software Developer's Manual, 2016. `https://software.intel.com/en-us/articles/intel-sdm`.

[29] Kernel.org. SECure COMPuting with filters. `https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt`, 2017.

[30] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In

Proceedings of ACM European Conference on Computer Systems (EuroSys), 2017.

[31] Volodymyr Kuznetsov, László Szekeres, and Mathias Payer. Code-pointer integrity. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[32] Hojoon Lee, Chihyun Song, and Brent Byunghoon Kang. Lord of the x86 rings: A portable user mode privilege separation architecture on x86. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[33] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[34] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[35] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[36] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient TCB reduction and attestation. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2010.

[37] MITRE. CVE-2014-0160. `https://nvd.nist.gov/vuln/detail/CVE-2014-0160`, 2014.

[38] Lucian Mogosanu, Ashay Rane, and Nathan Dautenhahn. MicroStache: A Lightweight Execution Context for In-Process Safe Region Isolation. In *Proceedings of International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2018.

[39] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. Poking Holes in Information Hiding. In *Proceedings of USENIX Security Symposium*, 2016.

[40] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Analysis of Computing Systems (ACM Sigmetrics)*, 2018.

[41] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software abstraction for Intel Memory Protection Keys (Intel MPK). In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2019.

[42] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary CPU architectures. In *Proceedings of USENIX Security Symposium*, 2010.

[43] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2004.

[44] Monirul I. Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure in-VM monitoring using hardware virtualization. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2009.

[45] Lei Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, Haibing Guan, and Jinming Li. Deconstructing Xen. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2017.

[46] The Clang Team. Clang 5 documentation: Safestack. `http://clang.llvm.org/docs/SafeStack.html`, 2017.

[47] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 1993.

[48] Wikimedia Foundation. Static HTML dump. `http://dumps.wikimedia.org/`, 2008.

[49] Wikimedia Foundation. Page view statistics April 2012. `http://dumps.wikimedia.org/other/pagecounts-raw/2012/2012-04/`, 2012.

[50] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Proceedings of USENIX Security Symposium*, 2002.

SafeHidden: An Efficient and Secure Information Hiding Technique Using Re-randomization

Zhe Wang[1,2], Chenggang Wu[1,2*], Yinqian Zhang[3], Bowen Tang[1,2], Pen-Chung Yew[4], Mengyao Xie[1,2], Yuanming Lai[1,2], Yan Kang[1,2], Yueqiang Cheng[5], and Zhiping Shi[6]

[1]*State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences,*
[2]*University of Chinese Academy of Sciences,* [3]*The Ohio State University,*
[4]*University of Minnesota at Twin-Cities,* [5]*Baidu USA,* [6]*The Capital Normal University*

Abstract

Information hiding (IH) is an important building block for many defenses against code reuse attacks, such as code-pointer integrity (CPI), control-flow integrity (CFI) and fine-grained code (re-)randomization, because of its effectiveness and performance. It employs randomization to probabilistically "hide" sensitive memory areas, called safe areas, from attackers and ensures their addresses are not leaked by any pointers directly. These defenses used safe areas to protect their critical data, such as jump targets and randomization secrets. However, recent works have shown that IH is vulnerable to various attacks.

In this paper, we propose a new IH technique called SafeHidden. It continuously re-randomizes the locations of safe areas and thus prevents the attackers from probing and inferring the memory layout to find its location. A new thread-private memory mechanism is proposed to isolate the thread-local safe areas and prevent adversaries from reducing the randomization entropy. It also randomizes the safe areas after the TLB misses to prevent attackers from inferring the address of safe areas using cache side-channels. Existing IH-based defenses can utilize SafeHidden directly without any change. Our experiments show that SafeHidden not only prevents existing attacks effectively but also incurs low performance overhead.

1 Introduction

Information hiding (IH) is a software-based security technique, which hides a memory block (called "safe area") by randomly placing it into a very large virtual address space, so that memory hijacking attacks relying on the data inside the safe area cannot be performed. As all memory pointers pointing to this area are ensured to be concealed, attackers could not reuse existing pointers to access the safe area. Moreover, because the virtual address space is huge and mostly inaccessible by attackers, the high randomization entropy makes brute-force probing attacks [45, 47] very difficult to succeed without crashing the program. Due to its effectiveness and efficiency, IH technique has become an important building block for many defenses against code reuse attacks. Many prominent defense methods, such as code-pointer integrity (CPI), control-flow integrity (CFI) and fine-grained code (re-)randomization, rely on IH to protect their critical data. For example, O-CFI [40] uses IH to protect all targets of indirect control transfer instructions; CPI [30] uses IH to protect all sensitive pointers; RERANZ [57], Shuffler [59], Oxymoron [4], Isomeron [15] and ALSR-Guard [36] use IH to protect the randomization secrets.

For a long time, IH was considered very effective. However, recent advances of software attacks [20, 35, 19, 43, 22] have made it vulnerable again. Some of these attacks use special system features to avoid system crashes when scanning the memory space [19, 35]; some propose new techniques to gauge the unmapped regions and infer the location of a safe area [43]; some exploit the thread-local implementation of safe areas, and propose to duplicate safe areas by using a thread spraying technique to increase the probability of successful probes [20]; others suggest that cache-based side-channel attacks can be used to infer the location of safe areas [22]. These attacks have fundamentally questioned the security promises offered by IH, and severely threatened the security defenses that rely on IH techniques.

To counter these attacks, this paper proposes a new information hiding technique, which we call SafeHidden. Our key observation is as follows: The security of IH techniques relies on (1) a high entropy of the location of the safe areas, and (2) the assumption that no attacks can reduce the entropy without being detected. Prior IH techniques have failed because they solely rely on the program crashes to detect attacks, but recent attacks have devised novel methods to reduce entropy without crashing the programs.

SafeHidden avoids these design pitfalls. It mediates all types of probes that may leak the locations of the safe areas, triggers a re-randomization of the safe areas upon detecting

*To whom correspondence should be addressed.

legal but suspicious probes, isolates the *thread-local* safe areas to maintain the high entropy, and raises security alarms when illegal probes are detected. To differentiate accidental accesses to unmapped memory areas and illegal probing of safe areas, SafeHidden converts safe areas into *trap areas* after each re-randomization, creating a number of trap areas after a sequence of re-randomization operations. Accesses to any of these trap areas are captured and flagged by SafeHidden. SafeHidden is secure because it guarantees that *any attempt to reduce the entropy of the safe areas' locations either lead to a re-randomization (restoring the randomness) or a security alarm (detecting the attack).*

SafeHidden is designed as a loadable kernel module, which is self-contained and can be transparently integrated with existing software defense methods (e.g., CPI and CFI). The design and implementation of SafeHidden entail several unconventional techniques: First, to mediate all system events that may potentially lead to the disclosure of safe area locations, SafeHidden needs to intercept all system call interfaces, memory access instructions, and TLB miss events that may be exploited by attackers to learn the virtual addresses of the safe areas. Particularly interesting is how SafeHidden traps TLB miss events: It sets the reserved bits of the page table entries (PTE) of the safe area so that all relevant TLB miss events are trapped into the page fault handler. However, because randomizing safe areas also invalidates the corresponding TLB entries, subsequent benign safe area accesses will incur TLB misses, which may trigger another randomization. To address this challenge, after re-randomizing the safe areas, SafeHidden utilizes hardware transactional memory (i.e., Intel TSX [2]) to determine which TLB entries were loaded before re-randomization and preload these entries to avoid future TLB misses.

Detecting TLB misses is further complicated by a new kernel feature called kernel page table isolation (KPTI) [1]. Because KPTI separates kernel page tables from user-space page tables, TLB entries preloaded in the kernel cannot be used by the user-space code. To address this challenge, SafeHidden proposes a novel method to temporarily use user-mode PCIDs in the kernel mode. To prevent the Meltdown attack (the reason that KPTI is used), it also flushes all kernel mappings of newly introduced pages from TLBs.

Second, SafeHidden proposes to isolate the *thread-local* safe area (by placing it in the *thread-private* memory) to prevent the attackers from reducing its randomization entropy. Unlike conventional approaches to achieve *thread-private* memory, SafeHidden leverages hardware-assisted *extended page table* (EPT) [2]. It assigns an EPT to each thread; the physical pages in other threads' *thread-local* safe area are configured not accessible in current thread's EPT. Compared to existing methods, this method does not need any modification of kernel source code, thus facilitating adoption.

To summarize, this paper makes the following contributions to software security:

- It proposes the re-randomization based IH technique to protect the safe areas against all known attacks.
- It introduces the use of *thread-private* memory to isolate *thread-local* safe areas. The construction of *thread-private* memory using hardware-assisted *extended page tables* is also proposed for the first time.
- It devises a new technique to detect TLB misses, which is the key trait of cache side-channel attacks against the locations of the safe areas.
- It develops a novel technique to integrate SafeHidden with KPTI, which may be of independent interest to system researchers.
- It implements and evaluates a prototype of SafeHidden, and demonstrates its effectiveness and efficiency through extensive experiments.

The rest of the paper is organized as follows. Section 2 reviews information hiding techniques and existing attacks. Section 3 explains the threat model. Section 4 presents the core design of SafeHidden. Section 5 details the implementation of SafeHidden. Section 6 provides the security and the performance evaluation. Discussion, related work, and conclusion are provided in Section 7, 8 and 9.

2 Background and Motivation

2.1 Information Hiding

Information hiding (IH) technique is a simple and efficient isolation defense to protect the data stored in a safe area. It places the safe area at a random location in a very large virtual address space. It makes sure that no pointer pointing to the safe area exists in the regular memory space, hence, making it unlikely for attackers to find the locations of the safe areas through pointers. Instead, normal accesses to the safe area are all done through an offset from a dedicated register.

Table 1 lists some of the defenses using the IH technique. The column "TL" shows whether the safe area is used only by its own thread or by all threads. The column "AF" shows how frequent the code accesses the safe area. Because most accesses to the safe area are through indirect/direct control transfer instructions, their frequencies are usually quite high. The column "Content in protected objects" shows the critical data tried to protect in safe areas. The column "Reg" shows the designated register used to store the (original) base address of the safe area. Some of them use the x86 segmentation register `%fs/%gs`. Others use the stack pointer register on X86_64, `%rsp`, that originally points to the top of the stack. They access a safe area via an offset from those registers. For the `%gs` register, they often use the following formats: *%gs:0x10*, *%gs:(%rax)*, *%gs:0x10(%rax)*, etc. For the `%rsp` register, they often use the following formats: *0x10(%rsp)*, *(%rsp, %rax, 0x8)*, *pushq %rax* [1], etc.

[1] It still conforms to the access model. The designated register is %rsp,

Defense	Protected Objects	Reg	Content in Protected Objects	AF	TL
O-CFI [40]	Bounds Lookup Table	%gs	The address boundaries of basic blocks targeted by an indirect branch instruction.	High	No
RERANZ [57]	Real Return Address Table	%gs	The table that contains the return addresses pushed by call instructions.	High	Yes
Isomeron [15]	Execution diversifier data	%gs	The mapping from the randomized code to the original code.	Hign	No
ASLR-Guard[36]	AG-Stack	*%rsp*	Dynamic code locators stored on the stack, such as return addresses.	High	Yes
	Safe-stack	%gs	ELF section remapping information and the key of code locator encryption.	High	No
Oxymoron [4]	Randomization-agnostic translation table	%fs	The translation table that contains the assigned indexes that are used to replace all references to code and data.	High	No
Shuffler [59]	Code pointer table	%gs	The table that contains all indexes that are transformed from all function pointers at their initialization points.	High	No
CFCI [61]	Protected Memory	%gs	File name and descriptors, and the mapping between file names and file descriptors.	Low	No
CPI [30]	Safe Stack	*%rsp*	Return address, spilled register, and objects accessed within the function through the stack pointer register with a constant offset.	High	Yes
	Safe Pointer Store	%gs	Sensitive pointers and the bounds of target objects pointed by these pointers.	High	No

Table 1: The list of defenses using information hiding (IH) techniques. **AF** is short for *Access Frequency*. **TL** is short for *Thread Local*.

A safe area is usually designed to be very small. For example, the size of a safe area shown in Table 1 is usually limited to be within 8 MB in practice. On today's mainstream X86_64 CPUs, the randomization entropy of an 8 MB safe area is 2^{24}. Such a high randomization entropy makes brute force probing attacks [45, 47] hard to guess its location successfully. A failed guess will result in a crash and detected by administrators.

2.2 Attacks against Information Hiding

Recent researches have shown that the IH technique is vulnerable to attacks. To locate a safe area, attackers may either improve the memory scanning technique to avoid crashes, or trigger the defense's legal access to the safe area and infer its virtual address using side-channels.

2.2.1 Memory Scanning

The attackers could avoid crashes during their brute-force probing. For example, some adversaries have discovered that some daemon web servers have such features. The daemon servers can fork worker processes that inherit the memory layout. If a worker process crashes, a new worker process will be forked. This enables the so-called *clone-probing* attacks where an adversary repeatedly probes different clones in order to scan the target memory regions [35]. CROP [19] chooses to use the exception handling mechanism to avoid crashes. During the probing, an access violation will occur when an inadmissible address is accessed. But, it can be captured by an exception handler instead of crashing the system.

Attackers could also use memory management APIs to infer the memory allocation information, and then locate the safe area. In [43], it leverages the *allocation oracles* to obtain the location of a safe area. In a user's memory space, there are many unmapped areas that are separated by code and data areas. To gauge the size of the largest unmapped area, it uses a binary search method to find the exact size by allocating and freeing a memory region repeatedly. After getting the exact size, it will allocate the memory in this area through the *persistent allocation oracle*. It then uses the same method to gauge the second largest unmapped area. Because a safe area is mostly placed in an unmapped area, an attacker can probe its surrounding areas to find its location without causing exceptions or crashes.

All probing attacks need to use such covert techniques to probe the memory many times without causing crashes because the size of a user's memory space is very large. In [20], it finds the safe area in many defenses is thread local (see Table 1). So, it proposes to leverage the thread "spraying" technique to "spray" a large number of safe areas to reduce the number of probings. After spraying, the attackers only need very few probes to locate the safe area.

2.2.2 Cache-based Side-Channel Attacks

To translate a virtual address to a physical address, the MMU initiates a page table (PT) walk that visits each level of the page table sequentially in the memory. To reduce the latency, most-recently accessed page table entries are stored in a special hardware cache, called *translation lookaside buffer* (TLB). Because of the large virtual address space in 64-bit architectures, a hierarchy of cache memories has been used to support different levels of page-table lookup. They are called the page table caches, or *paging-structure caches* by Intel [2]. In addition, the accessed PT entries are also fetched into the last level cache (LLC) during the page-table walk.

It has been demonstrated that cache-based side-channels can break coarse-grained address space layout randomization [22]. The location of the safe area can be determined through the following attack method: First, the attacker triggers the defense system's access to the safe area. To ensure this memory access invokes a PT walk, the attacker cleanses the corresponding TLB entries for the safe area's virtual address beforehand. Second, the attacker conducts a `Prime+Probe` or `Evict+Time` cache side-channel

and the offset is equal to 0. The only difference is that it will change the value of the designated register.

attack [44] to monitor which cache sets are used during the PT walk. As only certain virtual addresses map to a specific cache set, the virtual address of the safe area can be inferred using cache side-channel analysis.

However, it is worth mentioning that *to successfully determine the virtual address of one memory area, hundreds of such Prime+Probe or Evict+Time tests are needed. It is also imperative that the addresses of the PTEs corresponding to this memory area are not changed during these tests. That is, the cache entries mapped by these PTEs are not changed.* Our defense effectively invalidates such assumptions.

3 Threat Model

We consider an IH-based defense that protects a vulnerable application against code reuse attacks. This application either stands as a server that accepts and handles remote requests (e.g., through a web interface), or executes a sandboxed scripting code such as JavaScript as done in a modern web browsers. Accordingly, we assume the attacker has the permission to send malicious remote requests to the web servers or lure the web browsers to visit attacker-controlled websites and download malicious JavaScript code.

This IH-based defense has a safe area hidden in the victim process's memory space. We assume the *design* of the defense is not flawed: That is, before launching code reuse attacks, the attacker must circumvent the defense by revealing the locations of the safe areas (e.g., using one of many available techniques discussed in Section 2.2). We also assume the implementation of defense system itself is not vulnerable, and it uses IH correctly. We assume the underlying operating system is trusted and secured.

We assume the existence of some vulnerabilities in the application that allows the attacker to (a) read and write arbitrary memory locations; (b) allocate or free arbitrary memory areas (e.g., by interacting with the application's web interface or executing script directly); (c) create any number of threads (e.g., as a JavaScript program). These capabilities already represent the strongest possible adversary given in the application scenarios. Given these capabilities, all known attacks against IH can be performed.

3.1 Attack Vectors

Particularly, we consider the following attack vectors. All known attacks employ one of the four vectors listed below to disclose the locations of the safe areas.

- **Vector-1:** Gathering memory layout information to help to locate safe areas, by probing memory regions to infer if they are mapped (or allocated);
- **Vector-2:** Creating opportunities to probe safe areas without crashing the system, e.g., by leveraging resumable exceptions;

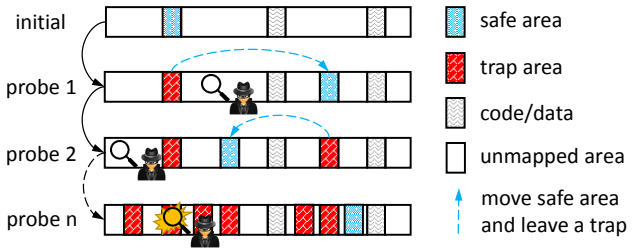


Figure 1: The high-level overview of the proposed re-randomization with the dispersed trap areas.

- **Vector-3:** Reducing the entropy of the randomized safe area locations to increase the success probability of probes, by decreasing the size of unmapped areas or increasing the size of safe areas;
- **Vector-4:** Monitoring page-table access patterns using cache side-channels to infer the addresses of safe areas, while triggering legal accesses to safe areas.

4 SafeHidden Design

We proposed SafeHidden, an IH technique that leverages re-randomization to prevent the attackers from locating the safe areas. It protects safe areas in both single-threaded programs and multi-threaded programs. It is designed primarily for Linux/X86_64 platform, as most of the defenses leveraging IH are developed on this platform.

At runtime, SafeHidden detects all potential memory probes. To avoid overly frequent re-randomization, it migrates the safe area to a new randomized location only after the detection of a suspicious probing. It then leaves a trap area of the same size behind. Figure 1 illustrates the high-level overview of the re-randomization method. In the figure, the memory layout is changed as the location of the safe area is being moved continuously, and the unmapped memory space becomes more fragmented by trap areas. The ever-changing memory layout could block **Vector-1**.

As the attackers continue to probe, new trap areas will be created. Gradually, it becomes more likely for probes to stumble into trap areas. If the attacker touches a trap area through any type of accesses, SafeHidden will trigger a security alarm and capture the attack. The design of trap areas mitigates the attacks from **Vector-2**, and significantly limits the attackers' ability to probe the memory persistently. While the attackers are still able to locate a safe area before accessing the trap areas, the probability is proven to be very small (see Section 4.4).

To block **Vector-3**, SafeHidden prevents unlimited shrink of unmapped areas and unrestricted growth of safe areas: (1) *Unmapped areas.* Because IH assumes that safe areas are hidden in a very large unmapped area, SafeHidden must prevent extremely large mapped areas. In our design, the

Events	Interception Points	Responses in SafeHidden			
		SA	UA	TA	OA
memory management syscalls	*mmap*, *munmap*, *mremap*, *mprotect*, *brk*, ...	Alarm	Rand	Alarm	–
syscalls that could return EFAULT	*read*, *write*, *access*, *send*, ...	Alarm	Rand	Alarm	–
cloning memory space	*clone*, *fork*, *vfork*	Rand	Rand	Rand	Rand
memory access instructions	*page fault exception*	–	Rand	Alarm	–

Table 2: Summary of potential stealthy probings and SafeHidden's responses. "**SA**": safe areas, "**UA**": unmapped areas, "**TA**": trap areas, "**OA**": other areas. "Alarm": triggering a security alarm. "Rand": triggering re-randomization. "–": do nothing.

maximum size of the mapped area allowed by SafeHidden is 64 TB, which is half of the entire virtual address space in the user space. Rarely do applications consume terabytes of memory; even big data applications only use gigabytes of virtual memory space; (2) *Safe areas.* Although safe areas in IH techniques are typically small and do not expand at runtime, attackers could create a large number of threads to increase the total size of the *thread-local* safe areas. To defeat such attacks, SafeHidden uses *thread-private* memory space to store *thread-local* safe areas. It maintains strict isolation among threads. When the *thread-local* safe area is protected using such a scheme, the entropy will not be reduced by thread spraying because any thread sprayed by an attacker can only access its own local safe area.

To mitigate **Vector-4**, SafeHidden also monitors legal accesses to the safe area that may be triggered by the attacker. Once such a legal access is detected, SafeHidden randomizes the location of the safe area. As the virtual address of the safe area is changed during re-randomization, the corresponding PTEs and their cache entries that are used by the attacker to make inferences no longer reflects the real virtual address of the safe area. Thus, **Vector-4** is blocked. It is worth noting that unlike the cases of detecting illegal accesses to the safe area, no trap area is created after the re-randomization.

In the following subsections, we will detail how SafeHidden recognizes and responds to the stealthy memory probes (see Section 4.1), how SafeHidden achieves the *thread-private* memory (see Section 4.2) and how SafeHidden defeats cache-based side-channel analysis (see Section 4.3).

4.1 Stealthy Memory Probes

In order to detect potential stealthy memory probes, we list all memory operations in the user space that can potentially be used as probings from the attackers (see Table 2).

The first row of Table 2 lists system calls that are related to memory management. The attackers could directly use them to gauge the layout of the memory space by allocating/deallocating/moving the memory or changing the permission to detect whether the target memory area is mapped or not. The second row lists the system calls that could return an EFAULT (bad address) error, such as "`ssize_t write (int fd, void * buf, size_t count)`". These system calls have a parameter pointing to a memory address. If the target memory is not mapped, the system call will fail without causing a crash, and the error code will be set to EFAULT. These system calls can be used to probe the memory layout without resulting in a crash. The third row lists the system calls that can clone a memory space. The attackers could use them to reason about the memory layout of the parent process from a child process. The fourth row lists memory access instructions that can trigger a *page fault exception* when the access permission is violated. The attackers could register or reuse the signal handler to avoid a crash when probing an invalid address.

Four types of memory regions are considered separately: safe areas, unmapped areas, trap areas, and other areas. Unmapped areas are areas in the address space that are not mapped; trap areas are areas that were once safe areas; other areas store process code and data. As shown in Table 2, SafeHidden intercepts different types of memory accesses to these areas and applies different security policy accordingly:

- If the event is an access to an unmapped area, SafeHidden will randomize the location of all safe areas. The original location of a safe area become a trap area.
- If the event is a memory cloning, it will perform randomization in the parent process after creating a child process, in order to make the locations of their safe areas different.
- If the event is an access to safe areas through memory management system calls or system calls with EFAULT return value, SafeHidden will trigger a security alarm.
- If the event is an access to trap areas through memory access instructions, memory management system calls, or system calls with EFAULT return value, it will trigger a security alarm.
- SafeHidden does not react to memory accesses to other areas. Since they do not have pointers pointing to the safe areas, probing other areas do not leak the locations.

To avoid excessive use of the virtual memory space, SafeHidden sets an upper limit on the total size of all trap areas (the default is 1 TB). Once the size of trap areas reaches the upper limit, SafeHidden will remove some randomly chosen trap area in each randomization round.

The design of such a security policy is worth further discussion here. Trap areas are previous locations of safe areas, which should be protected from illegal accesses in the same way as safe areas. As normal application behaviors never access safe areas and trap areas in an illegal way, accesses to them should raise alarms. For accesses to unmapped areas, an immediate alarm may cause false positives because the application may also issue memory management system calls, system calls with an EFAULT return value, or a memory access that touches unmapped memory areas. Therefore, accesses to unmapped areas only trigger re-randomization of the safe area to restore the randomness (that could invalidate the knowledge of previous probes), but no alarm will be raised. An alternative design would be counting the number of accesses to unmapped areas and raising a security alarm when the count exceeds a threshold. However, setting a proper threshold is very difficult because different probing algorithms could have different probing times. Therefore, monitoring critical subsets of the unmapped areas—the safe areas and trap areas—appears a better design choice.

4.2 Thread-private Memory

Thread-private memory technique was usually used in multi-threaded record-and-replay techniques [25, 7, 31]. We propose to use *thread-private* memory to protect safe areas. Conventional methods to implement *thread-private* memory is to make use of *thread-private* page tables in the OS kernel. As a separate page table is maintained for each thread, a reference page table for the entire process is required to keep track of the state of each page. The modification of the kernel is too complex, which cannot be implemented as a loadable kernel module: For example, to be compatible with `kswapd`, the reference page table must be synchronized with the private page tables of each thread, which requires tracking of CPU accesses of each PTE (especially the setting of the accessed and dirty bits[2] by CPU). The need for kernel source code modification and recompilation restricts the practical deployment of this approach.

To address this limitation, we propose a new approach to implement *thread-private* memory using the hardware virtualization support. Currently, a memory access in a guest VM needs to go through two levels of address translation: a *guest virtual address* is first translated into a *guest physical address* through the guest page table (GPT), which is then translated to its *host physical address* through a hypervisor maintained table, e.g., the *extended page table* (EPT) [38] in Intel processors, or the *nested page table* (NPT) [56] in AMD processors. Using Intel's EPT as an example, multiple virtual CPUs (VCPU) within a guest VM will share the same EPT. For instance, when the two VCPUs of a guest VM run

[2]These flags are provided for use by memory-management software to manage the transfer of pages into and out of physical memory. CPU is responsible for setting these bits via the physical address directly.

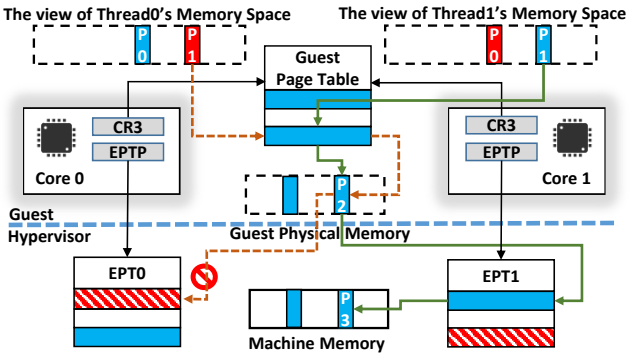


Figure 2: An example of the *thread-private* memory mechanism. P0 and P1 is *thread-private* memory page of Thread0 and Thread1, respectively.

two threads of the same program, both the virtual CR3 registers point to the page table of the program, and both EPT pointers (EPTPs) of VCPUs are pointing to a shared EPT.

To implement a *thread-private* memory, we can instead make each EPTP to point to a *separate* EPT to maintain its own *thread-private* memory. In such a scheme, each thread will have its own private EPT. The physical pages mapped in a thread's private memory in other threads' private EPTs will be made inaccessible. Figure 2 depicts an example of our *thread-private* memory scheme. When Thread1 tries to access its *thread-private* memory page P1, the hardware will walk both GPT and EPT1 to get the P3 successfully. But when Thread0 tries to access P1, it will trigger an EPT violation exception when the hardware walking EPT0 and be captured by the hypervisor.

In such a scheme, when a thread is scheduled on a VCPU, the hypervisor will set EPTP to point to its own EPT. In addition, SafeHidden synchronizes the EPTs by tracking the updates of the entries for the *thread-local* safe areas. For example, when mapping a guest physical page, SafeHidden needs to add the protection of all threads' EPTs for this page.

The *thread-private* memory defeats **Vector-3** completely. When *thread-local* safe areas are stored in such *thread-private* memory, spraying *thread-local* safe areas is no longer useful for the attackers because it will spray many prohibited areas that are similar to trap areas, called *shielded areas* (e.g., P1 is Thread0's shielded area in Figure 2), and be captured more easily.

4.3 Thwarting Cache Side-Channel Attacks

As discussed in Section 2.2.2, a key step in the cache side-channel attack by Gras et al. [22] is to force a PT walk when an access to the safe area is triggered. Therefore, a necessary condition for such an attack is to allow the attacker to induce TLB misses in a safe area. SafeHidden mitigates such attacks by intercepting TLB misses when accessing safe areas.

To only intercept the TLB miss occurred in safe areas,

SafeHidden leverages a reserved bit in a PTE on X86_64 processors. When the reserved bit is set, a *page fault exception* with a specific error code will be triggered when the PTE is missing in TLB. Using this mechanism, a TLB miss can be intercepted and handled by the page fault handler. SafeHidden sets the reserved bit in all of the PTEs for the safe areas. Thus, when a TLB miss occurs, it is trapped into the page fault handler and triggers the following actions: (1) It performs one round of randomization for the safe area; (2) It clears the reserved bit in the PTE of the faulting page; (3) It loads the PTE (after re-randomization) of the faulting page into the TLB; (4) It then sets the reserved bit of the PTE again. It is worth noting that loading the TLB entry of the faulting page is a key step. Without this step, the program's subsequent accesses to the safe area will cause TLB misses again, which will trigger another randomization.

The re-randomization upon TLB miss effectively defeats cache-based side-channel analysis. As mentioned in Section 2.2.2, a successful side-channel attack requires hundreds of `Prime+Probe` or `Evict+Time` tests. However, as each test triggers a TLB miss, the safe area is re-randomized after every test. The PTEs used to translate the safe areas in each PT levels are re-randomized. Thus, the cache entries mapped by these PTEs are also re-randomized that completely defeating cache-based side-channels [22].

Nevertheless, two issues may arise: First, the PTEs of a safe area could be updated by OS (e.g., during a page migration or a reclamation), and thus clearing the reserved bits. To avoid these unintended changes to the safe areas' PTEs, SafeHidden traps all updates to the corresponding PTEs to maintain the correct values of the reserved bits. Second, as the location of a safe area is changed after a randomization, it will cause many TLB misses when the safe area is accessed at the new location, which may trigger many false alarms and re-randomizations. To address this problem, SafeHidden reloads the safe area's PTEs that were already loaded in the TLB back to the TLB after re-randomization. This, however, requires SafeHidden to know which PTEs were loaded in the TLB before the re-randomization. To do so, SafeHidden exploits an additional feature in Intel transactional synchronization extensions (TSX), which is Intel's implementation of hardware transactional memory [2]. During a re-randomization, SafeHidden touches each page in the safe area from inside of a TSX transaction. If there is a TLB miss, a *page fault exception* will occur because the reserved bit of its PTE is set. But this exception will be suppressed by a TSX transaction and handled by its abort handler. Therefore, SafeHidden can quickly find out all loaded PTEs before the re-randomization and reload them for the new location in the TLB without triggering any *page fault exception*.

Integrating SafeHidden with kernel page table isolation (KPTI) [1] introduces additional challenges. KPTI is a default feature used in the most recent Linux kernels. It separates the kernel page tables from user-space page tables, which renders the pre-loaded TLB entries of the safe areas in kernel unusable by the user-space application. We will detail our solution in section 5.

4.4 Security Analysis

SafeHidden by design completely blocks attacks through **Vector-1**, **Vector-3**, and **Vector-4**. However, it only probabilistically prevents attacks through **Vector-2**. As such, in this section, we outline an analysis of SafeHidden's security guarantee. Specifically, we consider a defense system with only one safe area hidden in the unmapped memory space. We abstract the attackers' behavior as a sequence of memory probes, each of which triggers one re-randomization of the safe area and creates a new trap area.

$$P_{c_ith} = \begin{cases} (i \cdot P_t) \cdot \prod_{j=1}^{i-1} (1 - P_h - j \cdot P_t) & if\ i \le M \\ (M \cdot P_t) \cdot (\prod_{j=1}^{M} (1 - P_h - j \cdot P_t)) \cdot (1 - P_h - M \cdot P_t)^{i-1-M} & \\ & if\ i > M \end{cases} \quad (1)$$

The probability of detecting probes. Let the probability of detecting the attacks within N probes be P_{c_n}. Then the cumulative probability $P_{c_n} = \sum_{i=1}^{n} P_{c_ith}$, where P_{c_ith} represents the probability that an attacker *escapes* all $i-1$ probes, but is captured in the ith probe when it hits a trap area. An *escape* means that the attacker's probe is unsuccessful but remains undetected. P_{c_ith} is calculated in Equation (1), where i denotes the number of probes, j denotes the number of existing trap areas, P_h denotes the probability that the attacker hits the safe area in a probe, P_t represents the probability that the attacker hits one of the trap areas in a probe, M denotes the maximum number of trap areas. As an *escape* results in one re-randomization and the creation of a trap area, we approximate the number of existing trap areas with the number of escapes. But the number only increases up to M. So we consider if i reaches M separately. In the equation, $(i \cdot P_t)$ or $(M \cdot P_t)$ represents the probability that the probes are detected in the ith probe and $(1 - P_h - j \cdot P_t)$ or $(1 - P_h - M \cdot P_t)$ represents the probability of *escaping* the ith probe.

The attacker's success probability. We denote the probability of the attacker's successfully locating the safe area within N probes as P_{s_n}. $P_{s_n} = \sum_{i=1}^{n} P_{s_ith}$, where P_{s_ith} represents the probability that the attacker escapes in the first $i-1$ probes, but succeed in the ith probe. P_{s_ith} is provided in Equation (2).

$$P_{s_ith} = \begin{cases} P_h \cdot \prod_{j=1}^{i-1} (1 - P_h - j \cdot P_t) & if\ i \le M \\ P_h \cdot (\prod_{j=1}^{M} (1 - P_h - j \cdot P_t)) \cdot (1 - P_h - M \cdot P_t)^{i-1-M} & \\ & if\ i > M \end{cases} \quad (2)$$

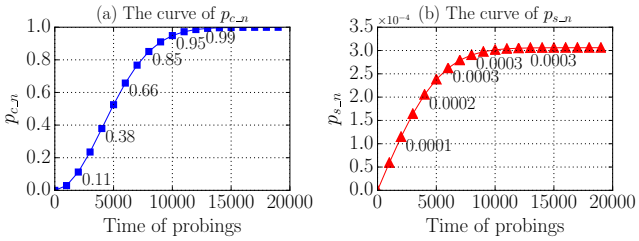


Figure 3: The probability of being captured by SafeHidden within N probes (a) and the probability of locating the safe areas within N probes successfully (b).

Discussion. When the size of the safe area is set to 8 MB, and the maximum size of all trap areas is set to 1 TB, as shown in Figure 3(a), P_{c_n} increases as the number of probes grows. When the number of probes reaches 15K, SafeHidden detects the attack with a probability of 99.9%; P_{c_n} approaches 100% as the number of probes reaches 20K. Figure 3(b) suggests the value of P_{s_n} increases as the number of probes increases, too. But even if the attacker can escape in 15K probes (which is very unlikely given Figure 3(a)), the probability of successfully locating the safe area is still only 0.03% (shown in Figure 3(b)), which is the maximum that could ever be achieved by the attacker. Notice that our abstract model favors the attackers, for example: (1) no *shielded area*s are considered in the analysis; (2) randomization triggered by applications' normal activities and TLB misses is ignored in the analysis. Obviously, in the real world situation, the attacker's success probability will be even lower, and the attack will be caught much sooner.

5 System Implementation

SafeHidden is designed as a loadable kernel module. Users could deploy SafeHidden by simply loading the kernel module, and specifying, by passing parameters to the module, which application needs to be protected and which registers point to the safe area. *No modification of the existing defenses or re-compiling the OS kernel is needed.*

5.1 Architecture Overview of SafeHidden

As described in Section 4.2, SafeHidden needs the hardware virtualization support. It can be implemented within a Virtual Machine Monitor (VMM), such as Xen or KVM. However, the need for virtualization does not preclude its application in non-virtualized systems. To demonstrate this, we integrated a thin hypervisor into the kernel module for a non-virtualized OS. The thin hypervisor virtualizes the running OS as the guest without rebooting the system. The other components inside the kernel module are collectively called GuestKM, which runs in the guest kernel.

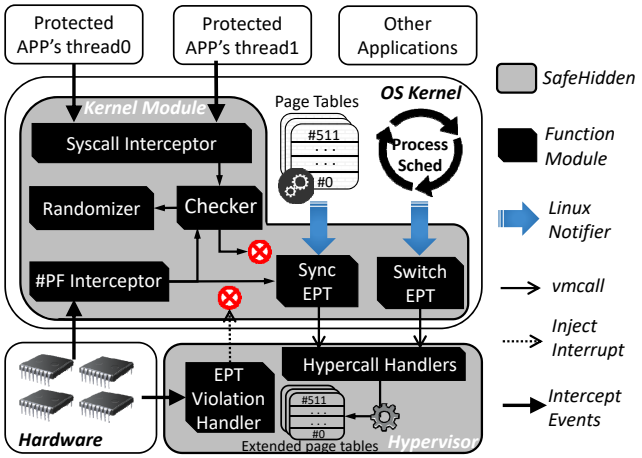


Figure 4: Architecture overview of SafeHidden.

After loading the SafeHidden module, it first starts the hypervisor and then triggers the initialization of GuestKM to install hooks during the *Initialization Phase*. Figure 4 shows an overview of SafeHidden's architecture. We can see that SafeHidden is composed of two parts: the hypervisor and the GuestKM. In the initialization phase, GuestKM installs hooks to intercept three kinds of guest events: *context switching*, *page fault exception*s, and certain *system call*s.

SafeHidden then starts to protect the safe areas by randomizing their locations and isolating the *thread-local* safe areas during the *Runtime Monitoring Phase*. In the GuestKM, the *Syscall Interceptor* and the *#PF Interceptor* modules are used to intercept *system call*s and *page fault exceptions*. When these two types of events are intercepted, they will request the *Checker* module to determine if SafeHidden needs to raise a security alarm, or if it needs to notify the *Randomizer* module to perform randomization. Meanwhile, SafeHidden needs to maintain the *thread-private* EPT to isolate the *thread-local* safe areas. The *sync EPT* module is used to synchronize the protected threads' page tables with their EPTs. The *switch EPT* module will switch EPTs when a protected thread is scheduled. Because both modules need to operate EPTs, they are coordinated by the *Hypercall Handlers* module. The *EPT Violation Handler* module is used to monitor illegal accesses to the *thread-local* safe areas.

5.2 Initialization Phase

Task-1: starting hypervisor. When the kernel module is launched, the hypervisor starts immediately. It configures the EPT paging structures, enables virtualization mode, and places the execution of the non-virtualized OS into the virtualized guest mode (non-root VMX mode). At this time, it only needs to create a default EPT for guest. Because the guest is a mirror of the current running system, the default EPT stores a one-to-one mapping that maps each guest physical address to the same host physical address.

Task-2: installing hooks in guest kernel. When the guest starts to run, GuestKM will be triggered to install hooks to intercept three kinds of events: 1) To intercept the *system calls*, GuestKM modifies the `system_call_table`'s entries and installs an alternative handler for each of them; 2) To intercept the *page fault exception*, GuestKM uses the `ftrace` framework in Linux kernel to hook the `do_page_fault` function; 3) To intercept *context switches*, GuestKM uses the standard preemption notifier in Linux, `preempt_notifier_register`, to install hooks. It can be notified through two callbacks, the `sched_in()` and the `sched_out()`, when a context switch occurs.

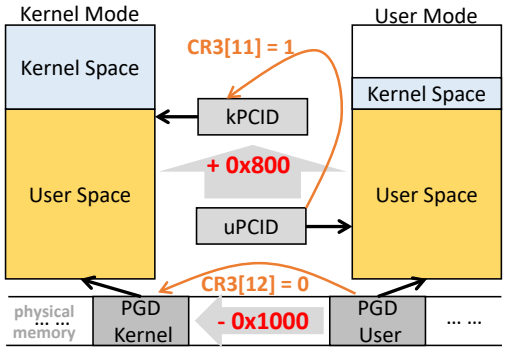


Figure 5: Overview of kernel page-table isolation.

5.3 Runtime Monitoring Phase

Recognizing safe areas. GuestKM intercepts the `execve()` system call to monitor the startup of the protected process. Based on the user-specified dedicated register, GuestKM can monitor the event of setting this register to obtain the value. In Linux kernel, the memory layout of a process is stored in a list structure, called `vm_area_struct`. GuestKM can obtain the safe area by searching the link using this value. According to Table 1, there are two kinds of registers that store the pointer of a safe area: 1) The 64-bit Linux kernel only allows a user process to set the `%gs` or `%fs` segmentation registers through the `arch_prctl()` system call [3]. So, GuestKM intercepts this system call to obtain the values of these registers; 2) All existing methods listed in Table 1 use `%rsp` pointed safe area to protect the stack. So, GuestKM analyzes the execution result of the `execve()` and the `clone()` system calls to obtain the location of the safe area, i.e., the stack, of the created thread or process. Once a safe area is recognized, its PTEs will be set invalid by setting the reserved bits.

To determine whether a safe area is *thread-local* or not, GuestKM monitors the event of setting the dedicated register in child threads. If the register is set to point to a different memory area, it means that the child thread has created its *thread-local* safe area, and the original safe area belongs to the parent. Until the child thread modifies the register to point to a different memory area, it shares the same safe area with its parent.

Randomizing safe areas. As described in Section 4.1 and 4.3, when GuestKM needs to perform randomization, it invokes the customized implementation of `do_mremap()` function in the kernel with a randomly generated address (by masking the output of the `rdrand` instruction with The `0x7fffffff000`) to change the locations of the safe areas. If the generated address has been taken, the process is repeated until a usable address is obtained. It is worth noting that GuestKM only changes the virtual address of the safe area, the physical pages are not changed. After migrating each safe area (not triggered by the TLB miss event), GuestKM will invoke `do_mmap()` with the protection flag `PROT_NONE` to set the original safe area to be a trap area. For multi-threaded programs, when the execution of a thread triggers a randomization (not triggered by the TLB miss event), the safe areas of all threads need to be randomized. To ensure the correctness, GuestKM needs to block all threads before randomizing all (or *thread-shared*) safe areas.

Although all safe areas used in existing defenses in Table 1 are *position-independent*, we do not rule out the possibility that future defenses may store some *position-dependent* data in the safe area. However, as any data related to an absolute address can be converted to the form of a base address with an offset, they can be made position independent. Therefore, after randomizing all safe areas, SafeHidden just needs to modify the values of the dedicated registers to point to the new locations of the safe areas.

Loading TLB entries under KPTI. The kernel page table isolation (KPTI) feature [1] was introduced into the mainstream Linux kernels to mitigate the Meltdown attack [32]. For each procecss, it splits the page table into a user-mode page table and a kernel-mode page table (as shown in Figure 5). The kernel-mode page table includes both kernel-space and user-space addresses, but it is only used when the system is running in the kernel mode. The user-mode page table used in the user mode contains all user-space address mappings and a minimal set of kernel-space mappings for serving system calls, handling interrupts and exceptions. Whenever entering or exiting the kernel mode, the kernel needs to switch between the two page tables by setting the CR3 register. To accelerate the page table switching, the roots of the page tables (i.e., PGD kernel and PGD user in Figure 5) are placed skillfully in the physical memory so that the kernel only needs to set or clear the bit 12 of CR3.

Moreover, to avoid flushing TLB entries when switching page tables, the kernel leverages the Process Context Identifier (PCID) feature [2]. When PCID is enabled, the first 12 bits (bit 0 to bit 11) of the CR3 register represents the PCID of the process which is used by the processor to identify the

[3] Recent CPUs supporting the `WRGSBASE/WRFSBASE` instructions allow setting the `%gs` and `%fs` base directly, but they are restricted by the Linux kernel to use in user mode.

owner of a TLB entry. The kernel assigns different PCIDs to the user and kernel modes (i.e., kPCID and uPCID in the figure). When entering or exiting the kernel mode, the kernel needs to switch between kPCID and uPCID. To accelerate this procedure, kPCID and uPCID of the same process only differ in one bit. Therefore, the kernel only needs to set or clear the bit 11 of CR3.

```
1  // .S file
2  .globl asm_load_pte_irqs_off
3  .align 0x1000
4  asm_load_pte_irqs_off:
5  /* 1. Get CR3 (kernel-mode page table with kPCID) */
6      mov %cr3, %r11
7  /* 2. Switch to kernel-mode page table with uPCID */
8      bts 63, %r11 // set noflush bit
9      bts 11, %r11 // set uPCID bit
10     mov %r11, %cr3 // set CR3
11 /* 3. Access user-mode pages to load pte into TLB */
12     stac // Allow user-mode pages accesses
13     movb (%rdi), %al // Read a byte from user-mode page
14     clac // Disallow user-mode pages accesses
15 /* 4. Get uPCID value */
16     mov %r11, %rax
17     and $0xfff, %rax
18 /* 5. Switch to kernel-mode page table with kPCID */
19     bts 63, %r11 // set noflush bit
20     btc 11, %r11 // clear uPCID bit
21     mov %r11, %cr3 //set CR3
22     retq //return uPCID
23 // .c file
24 void load_pte_into_TLB(unsigned long addr) {
25     unsigned long flags, uPCID;
26     // disable preemption and interrupts
27     get_cpu(); local_irq_save(flags);
28     uPCID = asm_load_pte_irqs_off(addr);
29     // flush the TLB entries for a given pcid and addr
30     invpcid_flush_one(uPCID, asm_load_pte_irqs_off);
31     // enable preemption and interrupts
32     local_irq_restore(flags); put_cpu();
33 }
```

Listing 1: The code snippet to load the TLB entries under KPTI.

As mentioned in Section 4.3, SafeHidden needs to load PTEs of the safe areas into the TLB every time it randomizes the safe areas. However, it is challenging to make SafeHidden compatible with KPTI. This is because SafeHidden only runs in the kernel mode—it uses the kernel-mode page table with kPCID, but the TLB entries of the safe areas must be loaded from the user-mode page table using uPCID.

An intuitive solution is to map SafeHidden into the kernel space portion of the user-mode page tables. Then the PTE loading is performed with uPCID. However, this method introduces more pages into the user-mode page tables and thus increases the attack surface of the Meltdown attack.

We propose the following alternative solution: SafeHidden still runs in the kernel mode using the kernel-mode page table. Before loading the TLB entries of the safe areas, it switches from kPCID to uPCID temporarily. Then without switching to the user-mode page table, it accesses the safe area pages to load the target PTEs into the TLB with uPCID. There is no need to switch to the user-mode page table for two reasons: (1) TLB entries are only tagged with PCIDs and virtual addresses; (2) the user-space addresses are also mapped in the kernel-mode page table. After the PTE loading, SafeHidden switches back to kPCID and then flushes the TLBs of the instruction/data pages related to the loading operation. This is to avoid these TLB entries (tagged with uPCID) to be exploited by the Meltdown attack.

Listing 1 illustrates the details of how to load user PTEs into the TLB from the kernel mode code under KPTI. Line 24 shows the function definition of this loading operation. Line 27 disables interrupts and preemptions to avoid unintended context switches. Line 28 invokes the assembly code for the loading operation. Line 6 reads the current CR3 register which contains the root of the kernel-mode page table and the kPCID. Line 8-10 switch to use uPCID (but keeping the kernel-mode page table unchanged). Line 8 sets the *noflush* bit to avoid flushing the target PCID's TLB entries when setting the CR3 register. Line 12 enables data access to user pages by disabling SMAP temporarily. Line 13-14 load the target PTE into TLB with uPCID by reading a byte from this page. Line 16-21 switch back to kPCID. Because line 12-21 code runs under the kernel-mode page table with uPCID, this code page mapping will be loaded into the TLB that can be accessed by user-mode code later. This page content could be leaked from the malicious process using the Meltdown attack. So line 30 flushes the mapping from the TLB.

Reloading TLB entries after randomization. SafeHidden uses Intel TSX to test which PTEs of the safe areas are loaded in the TLB. The implementation is very similar to the method of loading the user-mode TLB entries. The only difference is that SafeHidden encloses the code of line 13 (Listing 1) into a transaction (between `xbegin` and `xend` instructions). In fact, not all PTEs of the safe area need to be tested. SafeHidden only tests the PTEs that were reloaded in the last re-randomization.

Tracking GPT updates. The GPT entries of safe areas will be updated dynamically. In order to track such updates efficiently, we choose to integrate the Linux MMU notifier `mmu_notifier_register` in GuestKM. The MMU notifier provides a collection of callback functions to notify two kinds of page table updates: invalidation of a physical page and migration of a physical page. But it does not issue a callback when OS maps a physical page to a virtual page. To address this problem, we handle it in a lazy way by intercepting the *page fault exception* to track this update. Once GuestKM is notified about these updates, GuestKM makes the modified entry invalid or valid, and then issues a *hypercall* to notify the hypervisor to synchronize all EPTs.

Creating and destructing thread-private EPT. If a thread has no *thread-local* safe area, it shares its parent's EPT. If it is the main thread, it will be configured to use the default EPT. If a thread has a *thread-local* safe area, GuestKM will

issue a *hypercall* to notify the hypervisor to initialize an EPT for this thread. When initializing an EPT, SafeHidden will configure the entries based on other threads' local safe areas by walking the GPT to find all physical pages in the safe areas. Meanwhile, SafeHidden will also modify the entries of other thread's EPT to make all *thread-local* safe areas isolated from each other. Whenever SafeHidden changes other thread's EPT, it will block the other threads first. GuestKM also intercepts the `exit()` system call to monitor a thread's destruction. Once a thread with a private EPT is killed, GuestKM notifies the hypervisor to recycle its EPT.

Monitoring context switches. When a thread is switched out, GuestKM will be notified through the `sched_out()` and it will switch to the default EPT assigned to the corresponding VCPU. When GuestKM knows a new thread is switched in through the `sched_in()`, it will check whether the thread has a private EPT or not, and switches to its EPT in if it does.

Monitoring illegal accesses. GuestKM intercepts all system calls in Table 2 and checks their access areas by analyzing their arguments. If there is an overlap between their access areas with any of the trap areas, the safe areas, or the shielded areas, GuestKM will trigger a security alarm. Because there is no physical memory allocated to the trap areas, any memory access to those areas will be captured by intercepting the *page fault exception*. With the isolation of the *thread-local* safe area, any memory access to the shielded areas will trigger an *EPT violation exception*, which will be captured by the hypervisor (that notifies GuestKM). GuestKM triggers a security alarm in cases of any of these events.

Handling security alarms. How these security alarms are handled depends on the applications. For example, when SafeHidden is applied in browsers to prevent exploitation using JS code, it could mark the website from which the JS code is downloaded as malicious and prevent the users from visiting the websites. When SafeHidden is used to protect web servers, alarms can be integrated with application firewalls to block the intrusion attempts.

6 Evaluation

We implemented SafeHidden on Ubuntu 18.04 (Kernel 4.20.3 with KPTI enabled by default) that runs on a 3.4GHZ Intel(R) Core(TM) i7-6700 CPU with 4 cores and 16GB RAM. To evaluate the security and performance of SafeHidden, we implemented by ourselves two defenses that use safe areas, *OCFI* and *SS*. *OCFI* is a prototype implementation of O-CFI [40], which uses *thread-shared* safe areas (Table 1). *OCFI* first randomizes the locations of all basic blocks and then instruments all indirect control transfer instructions that access the safe areas, i.e., indirect calls, indirect jumps, and returns. Each indirect control transfer instruction has an entry in the safe areas, which contains the boundaries of possible targets. For each instrumented instruction, *OCFI* obtains its jump target and checks if it is within the legal range.

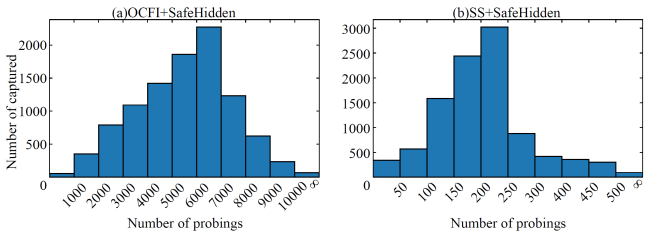


Figure 6: The distribution of probing times before being captured (10,000 probes launched).

SS is our implementation of a shadow stack, which is an example of the *thread-local* safe areas (see Table 1). Shadow stacks are used in Safe Stack [30], ASLR-Guard [36], and RERANZ [57]. *SS* adopts a compact shadow stack scheme [41] (in contrast to a parallel shadow stack scheme). In our implementation, the pointer (i.e., offset) to the stack top is stored at the bottom of the shadow stack. To be compatible with uninstrumented libraries, *SS* instruments function prologues and epilogues to access the shadow stacks (i.e., the safe areas). Listing 2 shows the function prologue for operating shadow stacks. The epilogue is similar but in an inverse order. The epilogue additionally checks if the return address has been modified.

In both cases, the size of the safe area is set to be 8 MB. To use SafeHidden with *SS* and *OCFI*, one only needs to specify in SafeHidden that the `%gs` register points to the safe areas. No other changes are needed.

```
1 mov (%rsp), %rax //get the return address
2 mov %gs:0x0, %r10 //get the shadow stack (ss) pointer
3 mov %rax, %gs:(%r10) //push the return address into ss
4 mov %rsp, %gs:0x8(%r10) //push the stack frame into ss
5 add $0x10, %gs:0x0//increment the shadow stack pointer
```

Listing 2: The shadow stack prologue.

6.1 Security Evaluation

We evaluated SafeHidden in four experiments. Each experiment evaluates its defense against one attack vector.

In the first experiment, we emulated an attack that uses the *allocation oracle*s [43] to probe *Firefox* browsers under *OCFI*'s protection. The prerequisite of this attack is the ability to accurately gauge the size of the unmapped areas around the safe areas. To emulate this attack, we inserted a shared library into *Firefox* to gauge the size of the unmapped areas. When SafeHidden is not deployed, the attack can quickly locate the safe area with only 104 attempts. Then we performed 10,000 trials of this attack on *Firefox* protected by *OCFI* and SafeHidden. The result shows that all the 10,000 trials failed, but in two different scenarios: In the first scenario (9,217 out of 10,000 trials), the attacks failed to gauge the size of the unmapped areas even when the powerful binary search method is used. The prerequisite of a

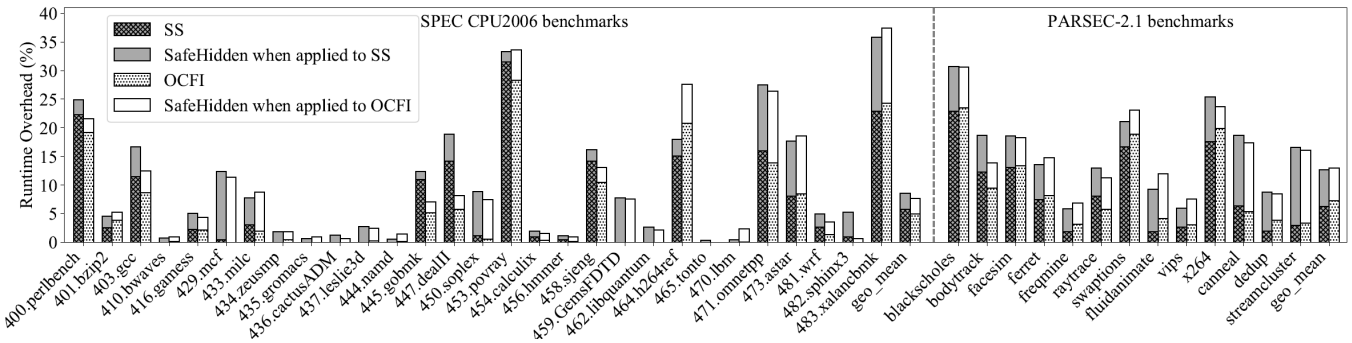


Figure 7: Performance overhead of SPEC and Parsec-2.1 benchmarks brought by SafeHidden when applied to the SS and OCFI defenses.

binary search is that the location of the target object does not change. However, SafeHidden's re-randomization confuses the binary search because the safe area moves continuously. In the second scenario, even though the attacks can gauge the exact size of an unmapped area, they always stumble into one of the trap areas when accessing the surroundings of the unmapped area, which triggers security alarms.

In the second experiment, we launched 10,000 trials of CROP attacks [19] to probe a *Firefox* protected by *OCFI*. The result shows that the attacks always successfully identified the location of the safe area when SafeHidden is not deployed. The time required is less than 17 minutes with no more than 81,472,151 probes. However, the attacks always fail when SafeHidden is deployed. Figure 6 (a) shows the distribution of the number of probes before an attack is detected by hitting a trap area. We can see that the distribution is concentrated in the range between [2000, 9000]. This experiment shows that SafeHidden can prevent the continuous probing attacks effectively.

In the third experiment, we launched 10,000 trials of the CROP attack using thread spraying to probe *Firefox* protected by *SS*. We sprayed 2^{14} (=16,384) threads with more than 16,384 *thread-local* safe areas, and then scanned the *Firefox* process with a CROP attack. The result shows that when SafeHidden is not deployed, the attacks can probe the locations of the safe areas successfully. The time taken is 0.16s, with only 2,310 probes. With SafeHidden deployed, all probes are captured before succeeding. Figure 6 (b) shows the distribution of the number of probes before being captured. The distribution is concentrated in the range between [50, 300], which is much lower than those in the second experiment. There are two reasons for that: 1) The other threads' local safe areas become the current thread's *shielded area*s, which increases the probability of the probes being captured; 2) All safe areas will be randomized after each probe, which increases the number of trap areas quickly.

In the fourth experiment, we emulated a cache side-channel attack against page tables using Revanc [54], which is a tool based on [46]. This tool allocates a memory buffer and then measures the access time of different pages in this buffer repeatedly. It could infer the base address of this buffer. To utilize this attack method against IH, we kept this memory buffer in a safe area by modifying the source code to force any access to this memory buffer through an offset from the `%gs` register. When SafeHidden is not deployed, this attack can obtain the correct base address of this buffer. The attack fails when SafeHidden is deployed.

6.2 Performance Evaluation

We evaluated SafeHidden's impact on the application's performance in terms of CPU computation, network I/O, and disk I/O, respectively. For the experiment of CPU computation, we ran SPEC CPU2006 benchmarks with *ref* input and multi-threaded Parsec-2.1 benchmarks using *native* input with 8 threads; For the experiment of network I/O, We chose the Apache web server httpd-2.4.38 and Nginx-1.14.2 web server. Apache was configured to work in *mpm-worker* mode, running in one worker process with 8 threads. Nginx was configured to work with 4 worker processes; For the experiment of disk I/O, we chose benchmark tool Bonnie++ (version 1.03e). For each benchmark, we prepared two versions of the benchmark: (1) protected by *SS*, and (2) protected by *OCFI*. We evaluated both the performance overhead of protecting these benchmarks using *SS* and *OCFI* defenses and the additional overhead of deploying SafeHidden to enhance the *SS* and *OCFI* defenses.

6.2.1 CPU Intensive Performance Evaluation

Figure 7 shows the performance overhead of the *OCFI* and *SS* defenses, and also the performance overhead of SafeHidden when applied to enhance the *OCFI* and *SS* defenses. For SPEC benchmarks, we can see that the geometric mean performance overhead incurred by *OCFI* and *SS* is 4.94% and 5.79%, respectively. For Parsec benchmarks, the geometric mean performance overhead incurred by *OCFI* and *SS* is 7.23% and 6.24%. The overhead of some applications (e.g., *perlbench*, *povray*, *Xalancbmk* and *blacksholes*) is higher because these applications frequently execute direct function calls and indirect control transfer instructions, which trigger accesses to safe areas. Note these overheads were caused by

Program	#randomization		Details of #randomization				Program	#randomization		Details of #randomization			
	SS	OCFI	#brk()	#mmap()	#tlb_miss SS	#tlb_miss OCFI		SS	OCFI	#brk()	#mmap()	#tlb_miss SS	#tlb_miss OCFI
SPEC CPU2006 benchmark													
bzip2	3,260	4,816	36	100	3,124	4,680	calculix	40,914	37,319	32,095	139	8,680	5,085
gcc	150,550	148,649	6,816	194	143,540	141,639	hmmer	2,851	2,430	13	25	2,813	2,392
bwaves	757	764	701	45	11	18	sjeng	207,559	196,562	3	10	207,546	196,549
gamess	192	311	27	30	135	254	GemsFDTD	184	205	11	160	13	14
mcf	435,637	424,266	3	11	435,623	424,252	libquantum	373,904	201,652	14	39	373,851	201,599
milc	685,788	576,056	2,687	44	683,057	573,325	h264ref	6,650	2,496	545	60	6,045	1,891
zeusmp	108	425	3	10	95	412	tonto	327	335	298	20	9	17
gromacs	287	134	44	36	207	54	lbm	6,110	5,822	3	11	6,096	5,808
cactusADM	11,884	11,826	8,997	66	2,821	2,763	omnetpp	320,474	223,832	1,245	56	319,173	222,531
leslie3d	60	94	5	27	28	62	astar	872,397	667,817	3,928	46	868,423	663,843
namd	474	630	100	31	343	499	wrf	53,018	49,230	419	253	52,346	48,558
gobmk	10,062	64,491	59	594	9,409	63,838	sphinx3	3,572	2,790	144	146	3,282	2,500
dealII	53,113	53,618	40,103	53	12,957	13,462	xalancbmk	921,406	781,973	3,099	94	918,213	778,780
soplex	168,463	186,807	168	49	168,246	186,590	**average**	**151,131**	**129,452**	**3,778**	**85**	**147,268**	**125,588**
Parsec-2.1 benchmark													
blackscholes	156,968	114,375	3	22	156,943	114,350	fluidanimate	168,896	175,816	231	23	168,642	175,562
bodytrack	11,205	10,426	2,486	6,558	2,161	1,382	vips	2,375	1,961	4	115	2,256	1,842
facesim	41,775	22,813	359	69	41,347	22,385	x264	5,768	8,055	42	162	5,564	7,851
ferret	93,815	62,870	222	39,032	54,561	23,616	canneal	244,669	251,238	5,917	24	238,728	245,297
freqmine	3,729	2,386	499	64	3,166	1,823	dedup	58,868	33,631	1,571	715	56,582	31,345
raytrace	27,510	22,859	1,279	57	26,174	21,523	streamcluster	273,684	219,572	7	23	273,654	219,542
swaptions	6,477	5,127	3	22	6,452	5,102	**average**	**84,288**	**71,625**	**971**	**3,607**	**79,710**	**67,048**

Table 3: Statistical data of SafeHidden when applied to the SS and OCFI defenses to protect SPEC CPU2006 and Parsec-2.1 benchmarks.

the adoption of *OCFI* and *SS*, but not SafeHidden.

For SPEC benchmarks, we can see that the geometric mean performance overhead incurred by SafeHidden when protecting *OCFI* and *SS* is 2.75% and 2.76%, respectively. For Parsec benchmarks, the geometric mean performance overhead incurred by SafeHidden is 5.78% and 6.44%, respectively. It shows that SafeHidden is very efficient in protecting safe areas. Based on the experimental results, we can also see that SafeHidden is more efficient in protecting single-threaded applications. This is due to two reasons: (1) All threads need to be blocked when randomizing the *thread-shared* safe areas or the *thread-local* safe areas (when not triggered by a TLB miss); (2) When protecting the *thread-local* safe areas, SafeHidden needs to synchronize the *thread-private* EPTs with the guest page table, which could introduce VM-Exit events.

Table 3 details some statistical data of SafeHidden when applied to the *OCFI* and *SS* defenses to protect SPEC and Parsec benchmarks. The column "#randomization" shows the number of re-randomization to safe areas. On SPEC and Parsec benchmarks, there are three operations that can trigger a re-randomization: (1) Using `brk()` to move the top of the heap; (2) Using `mmap()` to allocate a memory chunk; (3) TLB misses occurred in safe areas. Because *OCFI* and *SS* did not introduce extra invocation of system calls, the numbers of `brk()` and `mmap()` are the same. Combined with Figure 7, we can see that for most of SPEC benchmarks (except *mcf*, *soplex*, *GemsFDTD* and *omnetpp*), the performance overhead is related to the total number of re-randomization. The reason why those four benchmarks had different performance overhead is the virtualization overhead incurred by the hypervisor. For example, the hypervisor introduced 7.18% performance overhead for *GemsFDTD*. Except *x264* using *SS*, *canneal* and *streamcluster*, the overhead of most Parsec benchmarks is also related to the total number of re-randomization. For *canneal* and *streamcluster*, most of performance overhead is introduced by the virtualization. For *x264*, it spawns child threads more frequently than other benchmarks, which causes SafeHidden to frequently create and initialize *thread-private* EPTs.

6.2.2 Network I/O Performance Evaluation

Figure 8 shows the performance degradation of Apache and Nginx servers under the protection of *SS* and *OCFI* with and without SafeHidden. We use *ApacheBench (ab)* to simulate 100 concurrent clients constantly sending 10,000 requests, each request asks the server to transfer a file. We also varied the size of the requested file, i.e., {1K, 5K, 20K, 100K, 200K, 500K}, to represent different configurations. From the figure, we can see that *SS* only incurs 1.60% and 1.98% overhead on average when protecting Apache and Nginx. *OCFI* only incurs 1.45% and 2.13% overhead on average when protecting Apache and Nginx. We can also see that SafeHidden incurs 12.18% and 12.07% on average when applied to *SS* and *OCFI* to protect Apache. But SafeHidden incurs only 5.51% and 5.35% on average when applied to SS and OCFI to protect Nginx. So SafeHidden is more efficient in protecting Nginx than Apache. This is due to two reasons: (1) For each request to Nginx, Nginx will invoke several I/O system calls, such as `recvfrom()`, `write()`, `writev()`, etc., which only access the allocated memory space in the

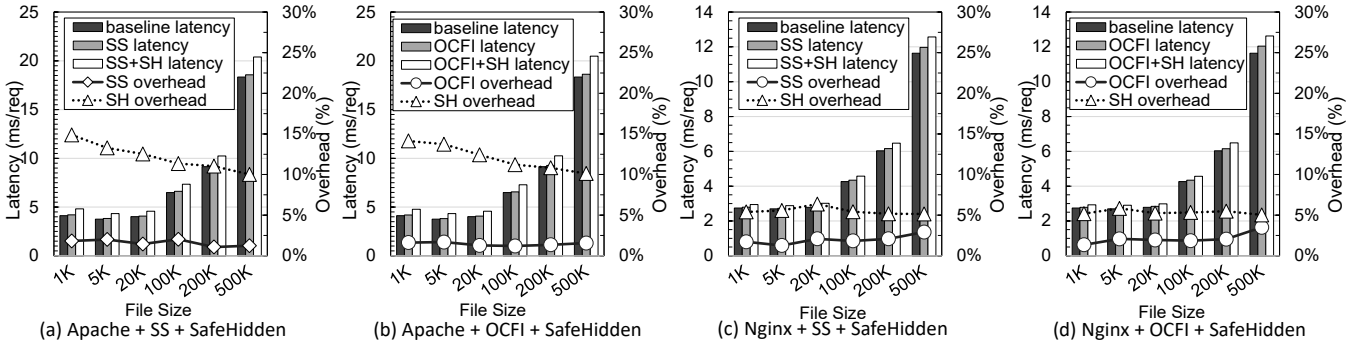


Figure 8: Network I/O Performance overhead brought by SafeHidden (short for SH) when applied to the SS and OCFI defenses.

Nginx process. The system calls in Nginx will not trigger randomization of the safe area. But for each request to Apache, Apache will invoke the `mmap()` system call to map the requested file into the virtual memory space which could trigger the extra randomization of all safe areas compared with Nginx; (2) Apache is a multi-threaded program. SafeHidden needs to block all threads when performing randomization of safe areas triggered by the `mmap()` system call.

6.2.3 Disk I/O Performance Evaluation

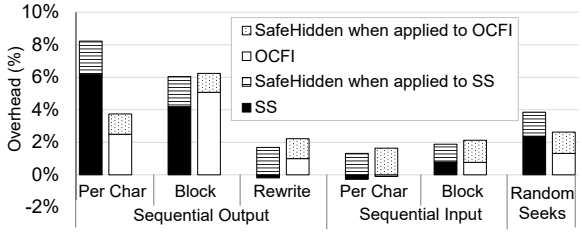


Figure 9: Disk I/O Performance overhead brought by SafeHidden when applied to the SS and OCFI defenses.

The Bonnie++ sequentially reads/writes data from/to a particular file in different ways. The read/write granularity varies from a character to a block (i.e., 8192 Bytes). Furthermore, we also test the time cost of the random seeking. Figure 9 shows the disk I/O measurement results: *SS* and *OCFI* defenses incur low performance overhead, i.e., 2.18% overhead on average for *SS* and 1.76% overhead on average for *OCFI*. SafeHidden brings only 1.58% overhead on average for *SS* and 3.08% overhead on average for *OCFI*. Compared with SPEC and Parsec benchmarks, this tool invokes the *write()* and *read()* system calls to write and read a very large file frequently. But these system calls only access the allocated memory space that does not trigger randomization of safe areas.

7 Discussion

TLBleed attack. TLBleed [21] exploits the shared TLBs between the hyper-threads on the same core to infer victim programs' memory access patterns. Potentially, it could be used to reduce the entropy of ASLR by triggering TLB misses and observing into which TLB set the target object is mapped. When TLBleed is used against SafeHidden, by triggering only L1 DTLB misses without L2 TLB misses, TLBleed may reduce the entropy of the safe area location by 4 bits (in the case of a 16-set L1 DTLB), which leads to roughly 20 bits entropy remaining for 8 MB safe area. However, attempts to further reduce the entropy will trigger re-randomization of safe areas with high probability. So, TLBleed is not able to defeat SafeHidden.

Spectre attack. The Spectre attack [28] leverages speculative execution and side-channels to read the restricted virtual memory space. As the memory protection related exceptions are suppressed in the speculatively executed code, SafeHidden could not detect Spectre attacks.

Resilience to attacks. SafeHidden is resilient to all known attacks against safe areas. Variants of existing attacks would also be prevented: (1) The attacker may try to fill up the address space quickly by using the *persistent allocation oracle* [43] to avoid SafeHidden from creating too many trap areas. But as SafeHidden sets an upper limit for the total mapped memory regions, such attacks are prevented; (2) The attacker could exploit the *paging-structure caches* to conduct the side-channel analysis. However such attacks will also trigger TLB misses, which will be detected by SafeHidden. Although it is difficult to prove SafeHidden has eliminated all potential threats, we believe it has considerably raised the cost of attacks in this arms race.

The impact of NMI on the solution of integrating KPTI. During the execution of the assembly code in listing 1, the interrupts are disabled to avoid unintended context switches. But the non-maskable interrupt (NMI) could break this protection. If a NMI occurs when the code is running, the NMI handler will run on the kernel-mode page table with the uPCID. So the memory pages accessed in the NMI handler could be leaked via the Meltdown attack. To avoid this, the entry of the NMI handler could be instrumented (by rewriting the NMI entry in IDT) to switch back to the kPCID.

8 Related Work

Protecting safe areas. MemSentry [29], IMIX [16], MicroStache [39], and ERIM [52] are the closest to our work. MemSentry adopts a *software-fault isolation* (SFI) approach to protecting frequently accessed safe areas by leveraging Intel's memory protection extensions (MPX) technology. It restricts the addresses of all memory accesses that can not access the safe area. But it is still not practical because it significantly increases the performance overhead [16]. The main disadvantage of MemSentry is the SFI approach is not safe, i.e., un-instrumented instructions can still access the safe region [16]. By modifying the Intel's simulation, IMIX extends the x86 ISA with a new memory-access permission to mark safe areas as security sensitive and allows accesses to safe areas only using a newly introduced instruction. Similarly, MicroStache achieves it by modifying the Gem5 simulator. However, IMIX and MicroStache are not yet supported by commodity hardware. ERIM protects safe areas by turning on access permission only when accesses are requested. To quickly switch the access permission on and off, it adopts the newly released Intel hardware feature memory protection keys (MPK) [2]. But it is still not suitable to protect the frequently accessed safe areas. For example, it incurs >1X performance overhead when protecting the shadow stack [29]. Different from SafeHidden, all these methods require modification of the source code of both the defense and the protected applications. Please note that most defenses listed in Table 1 (except two) work on COTS binaries. In particular, Shuffler [59] mentioned that defeats probing attacks by moving the location of its code pointer table (i.e., the safe area) continuously. But this method only blocks attacks from **Vector-1**. For example, using **Vector-2**, persistent attacks could always succeed. Different from Shuffler, SafeHidden blocks all existing attack vectors against IH.

Protecting CFI metadata. CFI is an important defense against code reuse attacks [3]. A CFI mechanism stores control-flow restrictions in its metadata. Like other types of safe areas, the metadata of CFI mechanisms needs to be protected. However, many CFI metadata only needs write protection without concerning about its secrecy. Therefore, these CFI mechanisms do not need IH. In contrast, some CFI metadata is writable, as it needs to be dynamically updated [8, 41, 42, 37], and others need to be kept as secrets [40, 61, 53, 60]. These CFI mechanisms must protect their metadata either by memory isolation [8, 53, 42, 41, 37] or IH [40, 60, 61]. SafeHidden can be applied to improve the security of IH for these CFI mechanisms.

Intra-process isolation. SFI is commonly used to restrict intra-process memory accesses [55]. However, both software-only and hardware-assisted SFIs incur high performance overhead [20, 43]. SeCage uses double-EPT to protect sensitive data, e.g., the session key and the private key [34]. Shreds [11] utilizes the domain-based isolation support provided by the ARM platform to protect the thread-sensitive data. Intel software guard extension (SGX) [2] protects the sensitive data using a secure enclave inside the application which cannot be accessed by any code outside the enclave. However, none of the approaches mentioned above can be used to protect frequently accessed safe areas because of their high switching overhead.

Tracking TLB misses. Intel performance monitoring units (PMU) [2] can be used to profile the TLB miss, but it is not precise enough. In contrast, setting reserved bits in PTE can help to track the TLB miss precisely. Some works had used this feature for performance optimization [17, 6, 5]. SafeHidden extends this method to detect side-channel attacks against the safe areas, which is the first time to our best knowledge such a feature is used in security.

Trap areas as security defenses. Booby-traps [12] first proposes to defeat code reuse attacks by inserting the trap gadgets in applications. CodeArmor [10] inserts the trap gadgets into the virtual (original loaded) code space. To protect the secret table's content against probing attacks, Readactor++ [14] inserts trap entries into the PLT and vtable, and Shuffler [59] inserts the trap entries into its code pointer table. To defeat the JIT-ROP [49] attacks, Heisenbyte [51] and NEAR [58] propose to trap the code after being read. Different from these works, SafeHidden uses the trap to capture the probing attacks against IH.

TSX for Security. The TSX is proposed to improve the performance of multi-threaded programs, but many studies have utilized TSX to improve system security. For example, Mimosa [23] uses TSX to protect private keys from memory disclosure attacks. TxIntro leverages the strong atomicity to ensure consistent and concurrent virtual machine introspection (VMI) [33]. In addition, TSX has been used to perform or detect the side-channel attacks against the Kernel ASLR [27] or the enclave in SGX [48, 9]. Different from these works, SafeHidden uses the TSX to identify TLB entries.

EPT for Security. The EPT has been used to isolate VMs [26], to protect processes from the malicious OS and/or other processes [18, 50, 24], and to protect sensitive code/-data within a process [34]. The EPT also supports more restrict memory permission check (i.e., the execute-only permission), which has been used to prevent the JIT-ROP [49] attacks [51, 13, 58]. Different from prior works, SafeHidden uses the EPT to achieve the *thread-private* memory.

9 Conclusion

This paper presented a new IH technique, called SafeHidden, which is transparent to existing defenses. It re-randomizes the locations of safe areas at runtime to prevent attackers from persistently probing and inferring the memory layout to

locate the safe areas. A new *thread-private* memory mechanism is proposed to isolate the *thread-local* safe areas and prevent the adversaries from reducing the randomization entropy via thread spraying. It also randomizes the safe areas after the TLB miss event to prevent the cache-based side-channel attacks. The experimental results show that our prototype not only prevents all existing attacks successfully but also incurs low performance overhead.

Acknowledgments

We are grateful to our shepherd Mathias Payer for guiding us in the final version of this paper. We would like to thank the anonymous reviewers for their insightful suggestions and comments. This research is supported by the National High Technology Research and Development Program of China under grant 2016QY07X1406 and the National Natural Science Foundation of China (NSFC) under grant U1736208. Pen-Chung Yew is supported by the National Science Foundation under the grant CNS-1514444. Yinqian Zhang is supported in part by gifts from Intel and DFINITY foundation.

References

[1] Kernel page-table isolation. https://www.kernel.org/doc/html/latest/x86/pti.html.

[2] Intel corporation. intel 64 and ia-32 architectures software developer's manual.

[3] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (2005), CCS '05, ACM.

[4] BACKES, M., AND NÜRNBERGER, S. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*.

[5] BASU, A., GANDHI, J., CHANG, J., HILL, M. D., AND SWIFT, M. M. Efficient virtual memory for big memory servers. *SIGARCH Comput. Archit. News* (2013).

[6] BHATTACHARJEE, A. Large-reach memory management unit caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (2013), MICRO-46.

[7] BITTAU, A., MARCHENKO, P., HANDLEY, M., AND KARP, B. Wedge: Splitting applications into reduced-privilege compartments. In *the 5th USENIX Symposium on Networked Systems Design and Implementation* (2008).

[8] BUROW, N., MCKEE, D., A. CARR, S., AND PAYER, M. Cfixx: Object type integrity for c++. In *NDSSS 2018*.

[9] CHEN, S., ZHANG, X., REITER, M. K., AND ZHANG, Y. Detecting privileged side-channel attacks in shielded execution with déjà vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (2017), ASIA CCS '17.

[10] CHEN, X., BOS, H., AND GIUFFRIDA, C. CodeArmor: Virtualizing the Code Space to Counter Disclosure Attacks. In *2017 IEEE European Symposium on Security and Privacy* (2017).

[11] CHEN, Y., REYMONDJOHNSON, S., SUN, Z., AND LU, L. Shreds: Fine-grained execution units with private memory. In *IEEE Symposium on Security and Privacy* (2016).

[12] CRANE, S., LARSEN, P., BRUNTHALER, S., AND FRANZ, M. Booby trapping software. In *NSPW* (2013), ACM, pp. 95–106.

[13] CRANE, S., LIEBCHEN, C., HOMESCU, A., DAVI, L., LARSEN, P., SADEGHI, A. R., BRUNTHALER, S., AND FRANZ, M. Readactor: Practical code randomization resilient to memory disclosure. In *2015 IEEE Symposium on Security and Privacy* (2015).

[14] CRANE, S. J., VOLCKAERT, S., SCHUSTER, F., LIEBCHEN, C., LARSEN, P., DAVI, L., SADEGHI, A.-R., HOLZ, T., DE SUTTER, B., AND FRANZ, M. It's a trap: Table randomization and protection against function-reuse attacks. In *ACM SIGSAC Conference on Computer and Communications Security* (2015).

[15] DAVI, L., LIEBCHEN, C., SADEGHI, A., SNOW, K. Z., AND MONROSE, F. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *22nd Annual Network and Distributed System Security Symposium, NDSS* (2015).

[16] FRASSETTO, T., JAUERNIG, P., LIEBCHEN, C., AND SADEGHI, A.-R. IMIX: In-process memory isolation extension. In *27th USENIX Security Symposium*.

[17] GANDHI, J., BASU, A., HILL, M. D., AND SWIFT, M. M. Badgertrap: A tool to instrument x86-64 tlb misses. *SIGARCH Comput. Archit. News* (2014).

[18] GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., AND BONEH, D. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (2003), SOSP '03.

[19] GAWLIK, R., KOLLENDA, B., KOPPE, P., GARMANY, B., AND HOLZ, T. Enabling client-side crash-resistance to overcome diversification and information hiding. In *23nd Annual Network and Distributed System Security Symposium, NDSS* (2016).

[20] GÖKTAS, E., GAWLIK, R., KOLLENDA, B., ATHANASOPOULOS, E., PORTOKALIDIS, G., GIUFFRIDA, C., AND BOS, H. Undermining information hiding (and what to do about it). In *25th USENIX Security Symposium*.

[21] GRAS, B., RAZAVI, K., BOS, H., AND GIUFFRIDA, C. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *27th USENIX Security Symposium (USENIX Security 18)*.

[22] GRAS, B., RAZAVI, K., BOSMAN, E., BOS, H., AND GIUFFRIDA, C. Aslr on the line: Practical cache attacks on the mmu. In *NDSS* (2017).

[23] GUAN, L., LIN, J., LUO, B., JING, J., AND WANG, J. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *2015 IEEE Symposium on Security and Privacy* (2015).

[24] HOFMANN, O. S., KIM, S., DUNN, A. M., LEE, M. Z., AND WITCHEL, E. Inktag: Secure applications on an untrusted operating system. In *Conference on Architectural Support for Programming Languages and Operating Systems* (2013).

[25] HSU, T. C.-H., HOFFMAN, K., EUGSTER, P., AND PAYER, M. Enforcing least privilege memory views for multithreaded applications. In *the 2016 ACM Conference on Computer and Communications Security*.

[26] JAMES E. SMITH AND RAVI NAIR. *Virtual machines - versatile platforms for systems and processes*. Elsevier, 2005.

[27] JANG, Y., LEE, S., AND KIM, T. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016).

[28] KOCHER, P., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre attacks: Exploiting speculative execution. *CoRR abs/1801.01203* (2018).

[29] KONING, K., CHEN, X., BOS, H., GIUFFRIDA, C., AND ATHANASOPOULOS, E. No need to hide: Protecting safe regions on commodity hardware. In *the Twelfth European Conference on Computer Systems* (2017).

[30] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., SEKAR, R., AND SONG, D. Code-pointer integrity. In *the 11th USENIX Conference on Operating Systems Design and Implementation* (2014).

[31] LAADAN, O., VIENNOT, N., AND NIEH, J. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (2010), SIGMETRICS '10.

[32] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., FOGH, A., HORN, J., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)* (2018).

[33] LIU, Y., XIA, Y., GUAN, H., ZANG, B., AND CHEN, H. Concurrent and consistent virtual machine introspection with hardware transactional memory. In *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)* (2014).

[34] LIU, Y., ZHOU, T., CHEN, K., CHEN, H., AND XIA, Y. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (2015).

[35] LU, K., LEE, W., NÜRNBERGER, S., AND BACKES, M. How to make ASLR win the clone wars: Runtime re-randomization. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016*.

[36] LU, K., SONG, C., LEE, B., CHUNG, S. P., KIM, T., AND LEE, W. Aslr-guard: Stopping address space leakage for code reuse attacks. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (2015), CCS '15.

[37] MASHTIZADEH, A. J., BITTAU, A., BONEH, D., AND MAZIÈRES, D. CCFI: cryptographically enforced control flow integrity. In *ACM Conference on Computer and Communications Security* (2015), ACM.

[38] MERRIFIELD, T., AND TAHERI, H. R. Performance implications of extended page tables on virtualized x86 processors. In *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (2016), VEE '16.

[39] MOGOSANU, L., RANE, A., AND DAUTENHAHN, N. Microstache: A lightweight execution context for in-process safe region isolation. In *RAID* (2018).

[40] MOHAN, V., LARSEN, P., BRUNTHALER, S., HAMLEN, K. W., AND FRANZ, M. Opaque control-flow integrity. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015*.

[41] NATHAN BUROW, X. Z., AND PAYER, M. Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy* (2019).

[42] NIU, B., AND TAN, G. Per-input control-flow integrity. In *the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (2015).

[43] OIKONOMOPOULOS, A., ATHANASOPOULOS, E., BOS, H., AND GIUFFRIDA, C. Poking holes in information hiding. In *25th USENIX Security Symposium*.

[44] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache attacks and countermeasures: the case of AES. In *6th Cryptographers' Track at the RSA conference on Topics in Cryptology* (2006).

[45] ROGLIA, G. F., MARTIGNONI, L., PALEARI, R., AND BRUSCHI, D. Surgically Returning to Randomized lib(c). In *ACSAC* (2009).

[46] SCHAIK, S. V., RAZAVI, K., GRAS, B., BOS, H., AND GIUFFRIDA, C. Revanc: A framework for reverse engineering hardware page table caches. In *European Workshop on Systems Security* (2017).

[47] SHACHAM, H., PAGE, M., PFAFF, B., GOH, E.-J., MODADUGU, N., AND BONEH, D. On the Effectiveness of Address-space Randomization. In *the 11th ACM Conference on Computer and Communications Security* (2004).

[48] SHIH, M.-W., LEE, S., KIM, T., AND PEINADO, M. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *NDSS* (2017).

[49] SNOW, K. Z., MONROSE, F., DAVI, L., DMITRIENKO, A., LIEBCHEN, C., AND SADEGHI, A. R. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Security and Privacy 2013* (2013).

[50] TA-MIN, R., LITTY, L., AND LIE, D. Splitting interfaces: Making trust between applications and operating systems configurable. In *the 7th Symposium on Operating Systems Design and Implementation* (2006).

[51] TANG, A., SETHUMADHAVAN, S., AND STOLFO, S. J. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015).

[52] VAHLDIEK-OBERWAGNER, A., ELNIKETY, E., DUARTE, N. O., GARG, D., AND DRUSCHEL, P. ERIM: Secure and Efficient In-process Isolation with Memory Protection Keys. *ArXiv e-prints* (2018).

[53] VAN DER VEEN, V., ANDRIESSE, D., GÖKTAŞ, E., GRAS, B., SAMBUC, L., SLOWINSKA, A., BOS, H., AND GIUFFRIDA, C. Practical Context-Sensitive CFI. In *Proceedings of the 22nd Conference on Computer and Communications Security (CCS'15)*.

[54] VUSEC. Reverse engineering page table caches in your processor, 2017. https://github.com/vusec/revanc.

[55] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (1993).

[56] WANG, X., ZANG, J., WANG, Z., LUO, Y., AND LI, X. Selective hardware/software memory virtualization. In *the 7th ACM Conference on Virtual Execution Environments* (2011).

[57] WANG, Z., WU, C., LI, J., LAI, Y., ZHANG, X., HSU, W.-C., AND CHENG, Y. Reranz: A light-weight virtual machine to mitigate memory disclosure attacks. In *the 13th ACM Conference on Virtual Execution Environments* (2017).

[58] WERNER, J., BALTAS, G., DALLARA, R., OTTERNESS, N., SNOW, K. Z., MONROSE, F., AND POLYCHRONAKIS, M. No-execute-after-read: Preventing code disclosure in commodity software. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security* (2016), ASIA CCS '16.

[59] WILLIAMS-KING, D., GOBIESKI, G., WILLIAMS-KING, K., BLAKE, J. P., YUAN, X., COLP, P., ZHENG, M., KEMERLIS, V. P., YANG, J., AND AIELLO, W. Shuffler: Fast and deployable continuous code re-randomization. In *12th USENIX Conference on Operating Systems Design and Implementation*.

[60] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical control flow integrity and randomization for binary executables. In *IEEE Symposium on Security and Privacy* (2013).

[61] ZHANG, M., AND SEKAR, R. Control flow and code integrity for cots binaries: An effective defense against real-world rop attacks. In *the 31st Annual Computer Security Applications Conference* (2015).

Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization

Mengyuan Li
The Ohio State University

Yinqian Zhang
The Ohio State University

Zhiqiang Lin
The Ohio State University

Yan Solihin
University of Central Florida

Abstract

AMD's Secure Encrypted Virtualization (SEV) is an emerging technology to secure virtual machines (VM) even in the presence of malicious hypervisors. However, the lack of trust in the privileged software also introduces an assortment of new attack vectors to SEV-enabled VMs that were mostly unexplored in the literature. This paper studies the insecurity of SEV from the perspective of the unprotected I/O operations in the SEV-enabled VMs. The results are alerting: not only have we discovered attacks that breach the confidentiality and integrity of these I/O operations—which we find very difficult to mitigate by existing approaches—but more significantly we demonstrate the construction of two attack primitives against SEV's memory encryption schemes, namely a memory decryption oracle and a memory encryption oracle, which enables an adversary to decrypt and encrypt arbitrary messages using the memory encryption keys of the VMs. We evaluate the proposed attacks and discuss potential solutions to the underlying problems.

1 Introduction

Secure Encrypted Virtualization (SEV) is an emerging processor feature available in recent AMD processors that encrypts the entire memory of virtual machines (VM) transparently. Memory encryption is performed by a hardware memory encryption engine (MEE) embedded in the memory controller that encrypts memory traffic on the fly, with a key unique to each of the VMs. As the encryption keys are generated from random sources at the time of VM launches and are securely protected inside the secure processor in their lifetime, privileged software, including the hypervisor, is not able to extract the keys and use them to decrypt the VMs' memory content. Therefore, SEV enables a stronger threat model, where the hypervisor is removed from the trusted computing base (TCB). It is explicitly stated in AMD's SEV whitepaper [20] that "SEV technology is built around a threat model where an attacker is assumed to have access to not only execute user level privileged code on the target machine, but can potentially execute malware at the higher privileged hypervisor level as well." Hence, SEV provides a trusted execution environment (TEE) for (mostly) unmodified VMs to perform confidential computation that is shielded from strong adversaries that control the entire privileged software stack.

The lack of trust in the hypervisor, unfortunately, increases considerably the attack surface that a VM has to guard against. As the hypervisor controls the VMs' access to hardware resources including CPU, physical memory, I/O devices, the VM's CPU scheduling, memory management, and I/O operations must be mediated by untrusted software. As a result, new attack vectors have emerged as researchers explore the security properties of this new hardware feature. For instance, Hetzelt and Buhren [17] demonstrated attacks against SEV-enabled VMs by exploiting unencrypted virtual machine control block (VMCB) at the time of `VMExit`. They show that a malicious hypervisor may learn the machine state of the guest VM by reading register values stored in the VMCB and alter these register values before returning to the VM. The lack of integrity of the encrypted memory has been identified by several prior studies [9, 12, 17, 25], which enables a malicious hypervisor to perform a variety of attacks.

This paper studies a previously unexplored problem under SEV's trust model—the unprotected I/O operations of SEV-enabled VMs. While the entire memory of the VMs can be encrypted using keys that are not known to the hypervisor, direct memory access (DMA) from the virtualized I/O devices must operate on *unencrypted memory* or *memory shared with the hypervisor*. As a result, neither the confidentiality nor the integrity of the I/O operations can be guaranteed under SEV's trust models.

More importantly, this paper goes beyond the investigation of I/O insecurity itself. In particular, we further demonstrate that these unprotected I/O operations can be leveraged by the adversary to construct (1) an encryption oracle to encrypt arbitrary memory blocks using the guest VM's memory encryption key, and (2) a decryption oracle to decrypt any memory pages of the guest. We demonstrate in the paper that these

two powerful attack primitives can be constructed in a very stealthy manner—the oracles can be queried by the adversary repeatedly and frequently without crashing the attacked VMs.

In addition, as a by-product of the study, this paper also reveals a severe side-channel vulnerability of SEV: As the adversary is able to manipulate the nested page tables, it could alter the `present` bit or `reserve` bit of the nested page table entries to force guest VM's memory accesses to the corresponding pages to trigger page faults. While this page-fault side channel has been previously studied in the context of Intel SGX [36] and even used in previous attacks against SEV [17], it is also reported that page faults from SEV-enabled guest VMs leak the entire faulting addresses (and error code) to the hypervisor (unlike in SGX where the page offset is masked). This fine-grained page-fault attack enables fine-grained tracing of the encrypted VM's memory access patterns, and particularly in this paper is used to facilitate the construction of the memory decryption oracle.

Contribution. This paper contributes to the study of TEE security in the following aspects:

- The paper studies a previously unexplored security issue of AMD SEV—the unprotected I/O operations of SEV-enabled guest VMs. The root cause of the problem is the incompatibility between AMD-V's I/O virtualization with SEV's memory encryption scheme.
- The paper demonstrates that the unprotected I/O operations could also be exploited to construct powerful attack primitives, enabling the adversary to perform arbitrary memory encryption and decryption.
- The paper reports the lack of page-offset masking of the faulting addresses during SEV's page faults handling, which leads to fine-grained side-channel leakage. The paper also demonstrates the use of both fine-grained and coarse-grained side channels in its I/O attacks.
- The paper empirically evaluates the fidelity of the attacks and discusses both hardware and software approaches to mitigating the I/O security issues.

Responsible disclosure. We have reported our findings to AMD and disclosed the technical details with AMD researchers. While we were confirmed that the presented attacks work on current release of SEV processors, AMD researchers suggested future generations of SEV chipsets are likely to be immune from these attacks. Some of the technical feedback we obtained from AMD has been integrated into the paper.

Roadmap. Section 2 presents the overview of AMD's SEV and explains the root causes of the exploited I/O operations in this paper. Section 3 describes several attacks exploiting the unprotected I/Os. Section 4 presents an evaluation of the fidelity of the attacks. Section 5 discusses potential solutions to securing SEV's I/O operations. Section 6 summarizes related work and Section 7 concludes the paper.

2 Overview of AMD SEV

2.1 Overview

AMD Secure Encrypted Virtualization (SEV) is a security extension for AMD Virtualization (AMD-V) architecture [2]. AMD-V is designed as a virtualization substrate for cloud computing services, which allows one physical server to run multiple isolated guest virtual machines (VM) concurrently. AMD's SEV is designed atop its Secure Memory Encryption (SME) technology.

Secure Memory Encryption. SME [20] is AMD's technology for real-time memory encryption, which aims at providing strong protection against memory snooping and cold boot attacks. An Advanced Encryption Standard (AES) engine is embedded in the on-die memory controller that encrypts/decrypts memory traffic when it is transferred out of or into the processor. A single ephemeral encryption key is generated for the entire machine from a random source every time system resets. The key is managed by a 32-bit ARM Cortex-A5 Secure Processor (AMD-SP). When SME is enabled, physical address bit 47 (also called the *C-bit*) in the page table entry (PTE) is used to mark whether the memory page is encrypted, thus enabling page-granularity encryption. Transparent SME, or TSME, is a mode of operating of SME, which allows encryption of the entire memory regardless of the C-bit. Thus TSME allows unmodified operating system to use the memory encryption technology.

Secure Encrypted Virtualization. AMD's SEV combines the AMD-V architecture and the SME technology to support encrypted virtual machines [20]. SEV aims to protect the security of guest VMs even in the presence of a malicious hypervisor, by using two isolation techniques: First, the data of guest VMs inside the processor is protected by an access control mechanism using Address Space Identifier (ASID). Specifically, the data in the CPU cache is tagged with ASID of each VM; thus it is prevented from being accessed by other VMs or the hypervisor. Second, the guest VM's data outside the processor (*e.g.*, in the DRAM) is protected via memory encryption. Rather than using a single AES key for the whole machine as in the case of SME, SEV allows each VM to use a distinct ephemeral key, thus preventing the hypervisor from reading the encrypted memory of each VM. Because memory encryption keys are managed by the secure co-processor, privileged software layers are not allowed to access or manipulate these keys.

Beside confidentiality, authenticity of the platform and integrity of the guest VMs are also provided by SEV. An identification key embedded in the firmware is signed by both AMD and the owner of the machine to demonstrate that the platform is an authentic AMD platform with SEV capabilities, which is administered by the machine owner. The initial contents of memory, along with a set of metadata of the VM, can be signed by the firmware so that the users of the guest VMs

Table 1: Effects of C-bits in guest page tables (gPT) and nested page tables (nPT). M is the plaintext; $E_k()$ is the encryption function under a memory encryption key k; k_g and k_h represent the guest VM and the hypervisor's memory encryption keys, respectively.

gPT	nPT	
	C-bit=0	C-bit=1
C-bit=0	M	$E_{k_h}(M)$
C-bit=1	$E_{k_g}(M)$	$E_{k_g}(M)$

may verify the identity and the initial states of the launched VMs through remote attestation.

2.2 Memory Encryption

ASID and memory encryption. The encryption keys used for memory encryption are generated from random sources when the VMs are launched. They are securely stored inside the secure processor for their entire life-cycle. Each VM has its own unique memory encryption key K_{vek}, which is indexed by the ASID of the VM. When the VM accesses a memory page that is mapped to its address space with its C-bit set, the memory will be first decrypted using the VM's K_{vek} before loaded into the CPU caches. Data in the caches are stored in plaintext; each cache line, in addition to the regular cache tags, is also tagged by the ASID of the VM. As such, the same physical memory may have multiple copies cached in the hardware caches. AMD does not maintain the consistency of the cache copies with different ASID tags [20].

Encryption with nested paging. AMD-V utilizes nested paging structures [1] to facilitate memory isolation between guest VMs. When the virtual address used by the guest VM (gVA) is to be translated into physical address, it is first translated into a guest physical addressing (gPA) using the guest page table (gPT), and the gPA is then translated into the host physical address (hPA) using the nested page table (nPT). While gPT is located in guest VM's address space, nPT is controlled directly by the host.

With AMD's SME technology, bit 47 of a PTE is called the *C-bit*, which is used to indicate whether or not the corresponding page is encrypted. When the C-bit of a page is set (*i.e.*, 1), the page is encrypted. As both the gPT and the nPT has C-bits, the encryption state of a page is controlled by the combination of the two C-bits in its PTEs in the gPT and nPT. The effect of C-bits in the gPT and nPT is shown in Table 1. To summarize, whenever the C-bit of gPT is set to 1, the memory page is encrypted with the guest VM's encryption key k_g; when the C-bit of gPT is cleared, the C-bit of nPT determines the encryption state of the page: the page is encrypted under the hypervisor's key k_g when C-bit is 1; otherwise the page is not encrypted.

To share memory pages between a guest VM and the hypervisor while preventing physical attacks, it is required to have the memory page's C-bit set to 0 in its gPT and the C-bit set to 1 in its nPT, so that the page is encrypted under the hypervisor's encryption key.

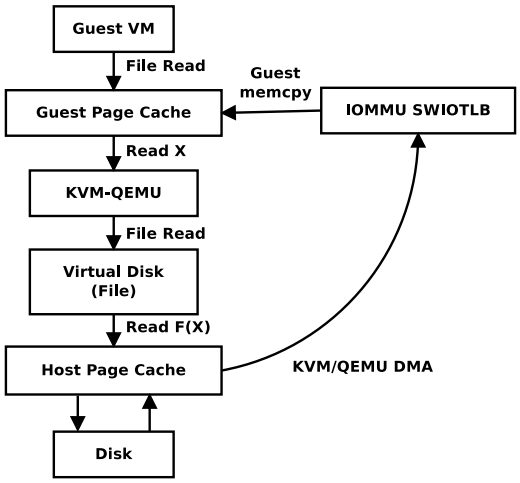


Figure 1: An example of a disk I/O operation by an SEV-enabled VM.

Encryption modes of operation. SEV uses AES as its encryption algorithm. The memory encryption engine encrypts data with a 128-bit key using the Electronic Codebook (ECB) mode of operation [12]. Therefore, each 16-byte aligned memory block is encrypted independently. A physical address-based tweak function $T()$ is utilized to make the ciphertext dependent of not only the plaintext but also its physical address [20]. Specifically, the tweak function is defined as $T(x) = \oplus_{x_i=1} t_i$, where x_i is the ith bit of host physical address x, $\oplus$ is the bitwise exclusive-or (*i.e.*, XOR) and t_i ($1 \leq i \leq 128$) is a 128-bit constant vector. The tweak function takes a physical address as an input and outputs a 128-bit value $T(x)$. Therefore, the ciphertext c of a plaintext m at address P_m is $c = E_K(m \oplus T(P_m))$. The tweak function prevents attacker from inferring plaintext by comparing the ciphertext of two 16-byte memory blocks. However, as the constant vectors t_is remain the same for all VMs (and the hypervisor) on the machine, they can be easily reverse engineered by an adversary.

Known issues with SME memory encryption. One root problem exploited in prior studies on SEV's insecurity is the lack of integrity of its encrypted memory [9, 12, 17, 25].

2.3 Virtualized I/O Operations

Similar to other virtualization technologies, SEV-enabled VMs interact with I/O devices through virtual hardware using Quick Emulator (QEMU). Common methods for VMs to perform I/O operations are programmed I/O, memory-mapped I/O, and direct memory access (DMA). Among these methods, DMA is most frequently used method for SEV-enabled VMs to do I/O accesses.

Direct Memory Access in SEV. With the assistance of DMA chips, programmable peripheral devices can transfer data to and from the main memory without involving the processor. With virtualization, a common way to support DMA is through IOMMU, which is a hardware memory management unit that maps the DMA-capable I/O buses to the main memory. However, unique to SEV is that the memory is encrypted. While the MMU supports memory encryption with multiple ASIDs, IOMMU only supports one ASID (*i.e.*, ASID=0). Therefore, in SEV-enabled VMs, DMA operations are performed on memory pages that are shared between the guest and the hypervisor (encrypted with the hypervisor's K_h). A bounce buffer, called Software I/O Translation Buffer (`SWIOTLB`), is allocated on these memory pages.

To illustrate the DMA operation from the guest, a disk I/O read is shown in Figure 1. When a guest application needs to read data from file, it first checks whether the file is already stored in its page cache. A miss in the guest page cache will trigger read from virtual disks, which is emulated by QEMU-KVM. The data is actually read from the physical disk by QEMU-KVM's DMA operation into `SWIOTLB` and then copied to the disk device driver's I/O buffer by the guest VM itself. The disk write operation is the inverse of this process, in which the data is first copied from the guest into `SWIOTLB` and then processed by QEMU.

3 Security Issues

In this section, we explore the security issues of the lack of protection for SEV's I/O operations. We start of our exploration with the most straightforward consequence of vulnerability—the insecurity of I/O operations itself—and present an attack example that breaches the integrity of I/O operations. To comprehensively study the attack surface, we also enumerate the I/O operations from a guest VM that are vulnerable to such attacks and discuss the challenges of implementing effective countermeasures. Next, we show that I/O insecurity leads to a complete compromise of the memory encryption scheme of SEV, by constructing powerful attack primitives that leverage the unprotected I/O operations to enable the adversary to encrypt or decrypt arbitrary messages with the guest VM's memory encryption key, k_{vek}.

3.1 Threat Model

We consider a scenario in which the VMs' memory are encrypted and protected by AMD SEV technology. The hypervisor run on a machine controlled by a third-party service provider. Under the threat model we consider, the third-party service is not trusted to respect the integrity or confidentiality of the computation inside the VMs. This could happen when the service provider is dishonest or when the hypervisor has been compromised.

The goal of the attacks is either to compromise the I/O operations themselves or the memory encryption of SEV. Out of scope in this paper are denial-of-service (DoS) attacks, in which the service provider simply refuses to run the VM. SEV is not designed to prevent DoS attacks.

3.2 I/O Security

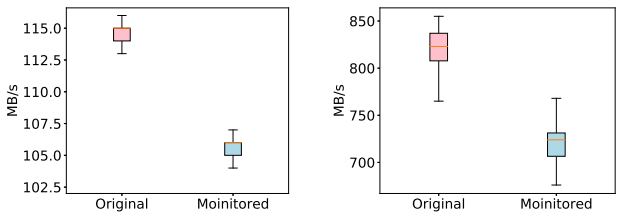
In this section, we explore the direct consequences of unprotected I/O operations from SEV-enabled guests.

3.2.1 Case Study: Integrity Breaches of Disk I/O

We first present a case study to show how SEV's guest VMs' unprotected I/O operations can be exploited to breach I/O security in practice. In this case study, we show that a malicious hypervisor is able to gain control of the guest VMs through an OpenSSH server without passwords by exploiting unprotected disk I/O. Therefore, we assume the disk is *not* encrypted with disk encryption key in this example. However, we note it is recommended by AMD to only use encrypted storage. As such, this case study only serves the purpose of proof-of-concept, rather than a practical attack. We will discuss its security implications in Section 3.2.2.

Specifically, the adversary controls the entire host and launches the SEV-enabled VM using the standard procedure [3]. During the system bootup, the binary code of `sshd` that performs user authentication is loaded into the memory. To monitor the disk I/O streams, whenever the QEMU performs a DMA operation for the guest, the adversary checks the memory buffer used for this DMA operation (*i.e.*, `SWIOTLB`) and search for the binary code of `sshd`. In our implementation, we used a 32-byte memory content (*i.e.*, 0*x*ff85 0*x*c041 0*x*89c4 0*x*8905 0*x*4e05 0*x*2900 0*x*0f85 0*x*1b01 0*x*0000 0*x*488b 0*x*3d49 0*x*0529 0*x*0089 0*x*eee8 0*x*c2bf 0*x*fdff) as the signature of the `sshd` binary and no false detection was observed. Once the DMA operation for `sshd` is identified, the adversary modifies the binary code inside `SWIOTLB`, before the QEMU commits the DMA operation. In particular, this is done by replacing the crucial code used in authentication that corresponds to `callq pam_authenticate`, which is a five-byte binary string 0*x*e8 0*x*c2 0*x*bf 0*x*fd 0*x*ff, to `mov $0 %eax` (a binary string of 0*x*b8 0*x*00 0*x*00 0*x*00 0*x*00). `pam_authenticate()` is used to perform user authentication; only when it returns 0 will the authentication succeed. Therefore, by moving 0 to the register `%eax` (the register used to store return value of a function call) directly, the adversary can successfully bypass the user authentication without knowing the password. To validate the attack, we empirically conducted the attack three times and all were successful.

Performance degradation due to I/O monitoring. We also conducted experiments to measure the performance degradation due to the hypervisor's monitoring of disk I/O streams. We used the `dd` command to write 1GB of data to the local



(a) I/O write (b) I/O read

Figure 2: Read/write performance overhead due to I/O monitoring.

disk to measure the I/O write speed. The `dsync` flag of set to make sure the data is written to the disk directly, bypassing the page caches. To measure the read speed, we cleaned the page caches in the memory by setting `vm.drop_caches=3` before reading 1GB of data from local disk. In both the read and write experiments, we measured the performance with and without I/O stream monitoring and repeated the measurements 200 times. The results show the performance degradation of I/O read and write is 11.8% and 7.9% respectively (see Figure 2).

3.2.2 Estimating The Attack Surface

As shown in the above example, I/O operations that are not encrypted by the software can be intercepted by the malicious hypervisor and manipulated to compromise the SEV-enabled guests. This vulnerability exists in all emulated I/O devices that are commonly used in cloud VMs, such as disk I/O, network I/O, and display I/O, *etc*. While a straightforward solution is to encrypt I/O streams by software, however, this simple method has many practical limitations in practice:

Network I/O. Network traffic can only be partially encrypted, as headers of IP or TCP cannot be encrypted. The adversary is still able to modify the network traffic to forge the IP addresses, port numbers, and encrypted metadata of the network packets. This is true for both TLS traffic and VPN traffic. As we will show in Section 3.3, encrypted traffic like SSH can still be exploited to construct memory decryption oracles.

Display I/O. Encrypting I/O traffic cannot be applied when the I/O devices cannot decrypt the I/O stream by themselves. Display I/O is one such example. For instance, Virtual Network Computing (VNC) is a graphical desktop sharing protocol that allows VMs to be remotely controlled. In KVM, the QEMU redirects the VGA display from the guest to the VNC protocol, which is not encrypted. Therefore, if the user of the guest VM uses VNC to control the VM, keystroke and mouse clicking will be learned and manipulated by the adversary. To protect display I/O operations, the guest VM must be modified to encrypt all display I/O traffic and the remote user interface must be modified accordingly to decrypt the traffic.

Disk I/O. For disk I/O operations, the method recommended by SEV [4] is for each SEV-enabled VMs to use encrypted disk filesystems. To use encrypted disks, however, the owner needs to first provision the disk encryption key into the protected VMs by using the `Launch_Secret` [3] command. This command first decrypts a packet sent by the VM owner (that contains the disk encryption key) encrypted using K_{tek} (Transport Encryption Key), atomically re-encrypts it using the memory encryption K_{vek}, and then injects it into the guest physical address specified by `GUEST_PADDR` (a parameter of the `Launch_Secret` command). As the address of the disk encryption key is known, if memory confidentiality is compromised (using methods to be described in Section 3.3), the disk encryption key can be learned and used to decrypt the entire image. Therefore, disk I/O is not secure, either.

3.3 Decryption Oracles

In this section, we show that the DMA operations under SEV's memory encryption technology can be exploited to construct a decryption oracle, which allows the adversary to decrypt any memory block encrypted with the guest VMs' memory encryption key K_{vek}. The oracle can be frequently and repeatedly queried and thus can be exploited as an attack primitive for more advanced attacks against SEV-enabled guests.

As mentioned in Section 2.3, the DMA operation from the SEV-enabled VM is conducted with the help of memory pages shared with the hypervisor. When DMA operates in the `DMA_TO_DEVICE` mode, data is transferred by the IOMMU hardware to the shared memory, and then copied by CPU in the SEV-enabled VM to its private memory; when DMA operates in the `DMA_BIDIRECTIONAL` mode, the SEV-enabled VM first copies the data from encrypted memory to the shared memory, and then the DMA reads or writes are performed on the shared memory.

Both these modes of operations provide the adversary an opportunity to observe the transfer of data blocks from memory pages encrypted by K_{vek} to memory pages that is not encrypted (from the hypervisor's perspective). Therefore, if the adversary alters the ciphertext of the data blocks in the encrypted memory page before they are copied by the guest VM, after the memory copy, the corresponding plaintext can be learned from the shared memory directly.

The construction of such a decryption oracle is shown in Figure 3. The decryption oracle can be constructed in three steps: *pattern matching*, *ciphertext replacement*, and *packets recovery*. We use network I/O as an example. The adversary exploits the network traffic in Secure Shell (SSH) to construct the decryption oracle. But we stress that any I/O traffic can be exploited in similar manners. In the following experiments, we configured the guest VM to use `OpenSSH_7.6p1` with `OpenSSL` 1.0.2n, which is default on Ubuntu 18.04.

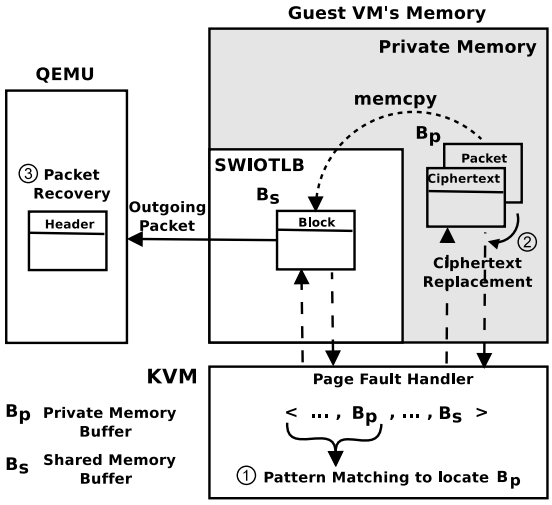


Figure 3: A decryption oracle. Step ①, the hypervisor conducts pattern matching using page-fault side channels to determine the address of B_p. Step ②, the hypervisor replaces a ciphertext block in B_p with the target memory block, which will be decrypted when copied to B_s. Step ③, QEMU recovers the network packet headers.

3.3.1 SSH and Network Stacks

To control the SEV-enabled guest remotely, the owner of the VM typically uses SSH protocol to remotely login into the VM and controls its activities. To copy data to and from the VM, protocols like SCP, which is built on top of SSH, is commonly used. Particularly, we consider the SSH traffic after the remote owner has already authenticated with `sshd` and a secure communication channel has been established. Because the SSH handshake protocol is performed in plaintext, the adversary who controls the hypervisor and QEMU can act as a man-in-the-middle attacker and recognize the established the secure channel by its IP addresses and TCP port number. Once the secure channel is established, SSH command and output data will be transferred using encrypted SSH packets that are transmitted in interactive mode [31].

In the interactive mode, each individual keystroke guest owner types will generate a packet that is sent to the SEV-enabled VM, which will be transferred by DMA to a memory buffer shared between the guest and the hypervisor. The packet is then copied by the guest to a private memory page encrypted using K_{vek}. Then the data is handled by the network stack in the guest OS kernel. The headers of the packet are then removed and the payload data is forwarded to the user-space application. Then the SSH server processes the keystroke and responds with an acknowledgement packet. The acknowledgement packet is copied back to the kernel space, wrapped by the corresponding header information, and then copied to the shared memory buffer. The last memory copying also decrypts the memory using the guest VM's K_{vek}. Therefore, our attack primitives target this process. As a result, every network packet generated by the guest VM can be exploited as a decryption oracle that helps the adversary decrypt one or multiple memory blocks.

3.3.2 Pattern Matching Using Fine-grained Page-fault Side Channels

Let us denote the private memory buffer as B_p, whose gPA is P_{priv}, and the shared memory buffer as B_s, whose gPA is P_{share}. The primary challenge in this attack is to identify the P_{priv}. As this address is never directly leaked, the adversary needs to perform a page-fault side-channel analysis.

Fine-grained page-fault side channels in SEV. The page fault side channel was first studied by Xu *et al.* in the context of Intel SGX [36]. As an SGX attacker controls the entire operating system, he or she can manipulate the page table entries (PTE) and set the `present` bit of the PTEs of pages that are mapped to the targeted enclave. By doing so, once the enclave program accesses the corresponding memory pages, the control flow will be trapped into the OS kernel through a page fault exception. On x86 processors, the faulting address will be stored in a control register, `CR2` so that the page-fault handler could learn the entire faulting address. To provide secrecy, SGX masks the page offset of the faulting address and leaves only the virtual page number in `CR2`.

Similarly, on the AMD platform, the adversary that compromises the hypervisor could also exploit the page-fault side channels to track the execution of the SEV-enabled VMs. Although the mapping between the guest VM's guest virtual address (gVA) to gPA is maintained by the guest VM's page table and is encrypted by K_{vek}, the hypervisor could manipulate the nested page tables (NPT) to trap the translation from gPAs to host physical addresss (hPA). Unlike SGX, SEV does not mask the page offset, providing more fine-grained observation to the adversary.

Moreover, the page-fault error code returned in the `EXITINTINFO` field of VMCB can also be exploited in the SEV page-fault side-channel analysis. Specifically, the page-fault error code is a 5-bit value, revealing the information of the page fault. For example, when bit 0 is cleared, the page fault is caused by non-present pages; when bit 1 is set, the page fault is caused by a memory write; when bit 2 is cleared, the page fault takes place in the kernel mode; when bit 3 is set, the fault is generated form a `reserved` bit; when bit 4 is set, the fault is generated by an instruction fetch. The error code provides detailed information regarding the reasons of the page fault, which can be leveraged in side-channel analysis.

Pattern matching. With such a fine-grained side channel, the adversary could monitor the memory access pattern of the guest when it receives an SSH packet. Particularly, after delivering an SSH packet to the SEV-enabled VM, the adversary immediately initiates the monitoring process and marks all of

the guest VM's memory pages inaccessible by clearing the `present` bit of the PTEs. Every time a memory page is accessed by the guest, a page fault takes place and the adversary is able to learn the entire faulting address P_i. Note here the faulting address in the guest VM refers to the guest physical address as the guest virtual address is not observable by the hypervisor. After the page fault, the adversary resets the `present` bit in the PTE to allow future accesses to the page. Therefore, with the fine-grained page fault side channel, one only needs to collect information regarding the first access to a memory page. The monitoring procedure stops when the acknowledgement packet is copied into B_s. At this point, the adversary has collected a sequence of faulting addresses $<P_1, P_2, \cdots, P_m>$.

Internally in the guest VM, when *sshd* is sending a packet, the encrypted data is first copied to the buffer of the transport layer, then the buffer of the network layer, and then the buffer of the data link layer. In each layer, new packet headers are added. Eventually, the entire network packet is stored in a data structure called `sk_buff`. Finally, the kernel will call `dev->hard_start_xmit` to transfer the data in `sk_buff` to the device driver, where B_p is located.

Both P_{priv} and the address of `sk_buff`, P_{sk}, should be found in the faulting addresses sequence $<P_1, P_2, \cdots, P_m>$. It is because the memory pages that store the private memory buffer B_p and `sk_buff` are not otherwise used during the process of sending network packets. The adversary could combine page offsets, page frame numbers, the page-fault error code, and the number of page faults between the two page faults of B_p and `sk_buff` to create a signature, which can be used to find P_{priv}. For example, the page-fault error code of B_p is $0b110$ and the page-fault error code of `sk_buff` is $0b100$; the page offset of P_{priv} is usually $0x0$fa or $0x8$fa and the offset of `sk_buff` usually ends with $0x$e8 or $0x00$; and the number of page faults between B_p and `sk_buff` is roughly 20. With these signatures, the adversary can identify P_{priv} from the sequence of faulting addresses. Of course, the signature may change from one OS version to another, or change with different OS kernel. However, because the adversary controls the hypervisor, such information can be re-trained offline, before performing the attacks.

It was indicated by AMD researchers (during an offline discussion) that SEV-ES should mask the page offset information when there is a VMEXIT. However, we were not able to find related public documentation. Moreover, as the KVM patch for SEV-ES support is not yet available at the time of writing, we were not able to validate the claim or estimate the remaining leakage (*e.g.*, error code, page offset) after the patch. However, regardless of the hardware changes, a coarse-grained page-fault side channel in which the page frame number of the faulting address is leaked must remain. To show that the demonstrated attack still works, we conducted experiments to perform pattern matching without page fault offsets and error code information. Specifically, we performed pattern matching using only the faulting page numbers, with the guest VM running different Ubuntu versions (*e.g.*, 18.04, 18.04.1 and 19.04) and different kernel versions (4.15.0-20-generic, 4.15.0-48-generic and 5.0.0-13-generic). The results show that after training in one virtual machine, the pattern matching rules can work well even in different virtual machines with the same Ubuntu version and kernel version—the attacker is still able to successfully identify the page frame number of P_{priv}. To determine the complete address of P_{priv}, the attacker could determine the offset by scanning the entire memory page and looking for content changes (*e.g.*, in a 90-byte buffer).

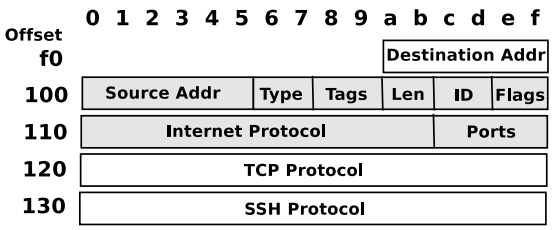


Figure 4: Format of an SSH packet.

3.3.3 Replacing Ciphertext

After determining P_{priv}, the adversary replaces aligned SSH header in B_p with the ciphertext he or she chooses to decrypt. As shown in Figure 4, the packet headers include a 6-byte destination address, a 6-byte source address, a 2-byte IP type (*e.g.*, IPv4 or IPv6), 1-byte IP version and IP header length, 1-byte of differentiated services field, 2-byte packet length, 2-byte identification, 2-byte of IP flags, 1-byte time-to-live, 1-byte protocol type, 2-byte checksum, and 4-byte source IP address and 4-byte destination address, and 20-bytes TCP headers (start with 2-byte source port and 2-byte destination port).

As shown in Figure 4, P_{priv} has the offset address ending with $0x$fa. Because SEV encrypts data in 16-byte aligned blocks, only part of the TCP/IP header (i.e., header in gray blocks in Figure 4) can be used to decrypt ciphertext. Additional constraints apply if the packet needs to be recovered later. Before replacing the packet header with the chosen ciphertext, the adversary performs a `WBINVD` instruction to flush the guest VM's cached copy of B_p back to memory. It is because cache coherence is not maintained by the hardware between cache lines with different ASIDs. To make sure the guest VM's copy does not overwrite our changes to the memory, `WBINVD` instruction needs to be called first.

The ciphertext replacement takes place before `memcpy`, after B_p is accessed and before B_s is accessed. B_s is located inside the `SWIOTLB` pool, which is the next available address within `SWIOTLB` that can be used by the guest. After replacing a few blocks in B_p, another `WBINVD` instruction is performed to ensure the guest VM reads and decrypts up-to-date cipher-

text in memory. All replacement operation is achieved by `IOremap` instead of `Kmap`, since `Kmap` decrypts data with the hypervisor's key first and `IOremap` directly operates data in the memory without decryption.

We use the following example to illustrate the attack. Let the ciphertext c be a 16-byte aligned memory block with the gPA of P_c. The function which can translate gPA to hPA is called $hPA()$. The goal of the attack is to decrypt c. The adversary replaces a 16-byte data in the SSH header that begins with address $(P_{priv}+16)/16*16$ with c. After the data in B_p is copied to B_s, the adversary could read the decrypted SSH packet and extract the plaintext of decrypted memory block, d, from the corresponding location of the packet. However, d is not the plaintext of c yet, as SEV's memory encryption involves a tweak function $T()$. That is, $c = E_{K_{vek}}(m \oplus T(hPA(P_c)))$ but $d = D_{K_{vek}}(c) \oplus T(hPA((P_{priv}+16))/16*16)$. Therefore, the plaintext message m of ciphertext c can be calculated by $m = d \oplus T(hPA((P_{priv}+16))/16*16) \oplus T(hPA(P_c))$.

3.3.4 Packets Recovery

To make the attack stealthy, the adversary needs to recover the network packet with decrypted data before those packets are passed to the physical NIC device. As shown in Figure 4, the SSH header also contains metadata of the packet. When the malicious hypervisor injects chosen ciphertext into the memory block with offset = $0x100$, the adversary only needs to be concerned about a portion of the source IP address, IP protocol type, IP tags, TCP header length, and the identification of the packet. Majority of the fields are determined. The identification of the IP packet increases by 1 every time SSH server replies a packet. So when hypervisor tries to recovery the (plaintext) packet from the QEMU side, it only need to correct the packet length, increase identification by 1 and copy the remaining portion from previous packet such as source address, header length, time to live and protocol number.

3.4 Encryption Oracle

We next show the construction of a memory encryption oracle using unprotected I/O operations. The encryption oracle stealthily encrypts a chosen plaintext message using a guest VM's memory encryption key K_{vek}. Similar to the construction of the decryption oracle, during the DMA operation of the guest that transfers data from the device to the encrypted memory, the adversary changes the message m in the shared memory buffer B_s, waits until it is copied to the private buffer B_p in the encrypted page, and then extracts the corresponding ciphertext $E_{k_g}(m)$ from B_p.

To determine the gPA address of B_p and retrieve the ciphertext of the plaintext message at address P_t, the steps shown in Figure 5 are taken. Again, we leverage the fine-grained page-fault side channel we used in the previous section. Specifically,

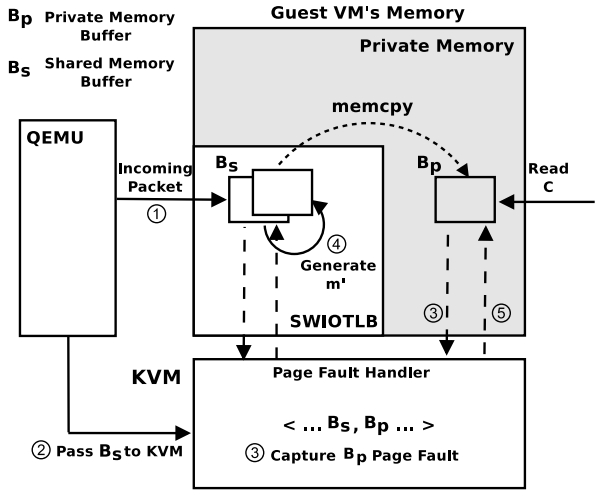


Figure 5: An encryption oracle. Step ①, QEMU forwards an incoming packet to the guest. Step ②, QEMU passes the address of B_s to the hypervisor. Step ③, a page fault immediately after the fault at B_s is captured by the page fault handler. Step ④, message m' is placed in B_p. Step ⑤, page fault handler returns the control to the guest.

we modified all memory pages' PTEs right after the QEMU finishes writing the packet into `SWIOTLB` and before the QEMU notifying guest VM about the DMA write. Then, when the guest VM performs a `memcpy` operation to copy the data, the adversary will observe a sequence of page faults: <...P_{share}, P_{priv}...>, where P_{share} is the address of B_s and P_{priv} is the address of B_p. The page fault at P_{priv} will take place right after the page fault at P_{share}. When the hypervisor handles the page fault at P_{priv}, it replaces the 16-byte aligned data block with the message m', where $m' = m \oplus T(hPA(P_{priv})) \oplus T(hPA(P_t))$, where P_t is the gPA of the target address to which the adversary wishes to copy m. The corresponding ciphertext will be $c = E_{k_g}(m \oplus T(hPA(P_t)))$, which can be used to replace the ciphertext at address P_t.

The encryption oracle can be typically exploited to inject code or data into the SEV-enabled VM's encrypted memory, or it can be used to make guesses of the memory content by providing a probable plaintext. We note that to use the encryption oracle, the adversary may simply generate meaningless packets and send them to the guest VM, which will be discarded. But the oracle can still be constructed and used. The only downside of this approach is that the guest VM will observe large volume of meaningless network traffic and may become suspicious of attacks.

4 Evaluation

We implemented our attacks on a blade server with an 8-Core AMD EPYC 7251 Processor, which has SEV enabled on the chipset. The host OS runs Ubuntu 64-bit 18.04 with

Linux kernel v4.17 (KVM hardware-assisted virtualization supported since v4.16) and the guest OS also runs Ubuntu 64-bit 18.04 with Linux kernel v4.15 (SEV supported since v4.15). The QEMU version used was QEMU 2.12. The SEV-enabled guest VMs were configured with 1 virtual CPU, 30GB disk storage, and 2GB DRAM. The OpenSSH server was installed from the default package archives.

4.1 Pattern Matching

We first evaluate the pattern matching algorithm's accuracy of determining P_{priv}. To obtain the ground truth, we modified the guest kernel to log the gPA address of `sk_buff`, the source gPA and destination gPA of `memcpy`, as well as the size of each DMA read or write. All the data was recorded in the kernel debug information, which can be retrieved using a Linux command `dmesg`.

The experiments were conducted as follows: We ran a software program *AnJian* [13] (an automated keystroke generation tool) on a remote machine, which opened a terminal that was remotely connected to the SEV-enable VM through an SSH communication channel. *AnJian* automatically typed on the SSH terminal two Linux commands `cat security.txt |grep sev` and `dmesg` at the rate of 10 keystrokes per second. This was used to simulate the remote owner controlling the SEV-enabled VM through SSH. The adversary would make use of the generated SSH packets to perform memory decryption. The `dmesg` command also retrieved the kernel debug message that recorded the ground truth.

At the same time, the pattern matching was performed by the adversary on the hypervisor side. The page-fault side-channel analysis was conducted upon receiving every incoming SSH packet to guess the address P_{priv}. There were three outcomes of the guesses: a correct guess, an incorrect guess, and unable to make a guess. Because there were 33 keystrokes generated by *AnJian*, the adversary was allowed to guess P_{priv} for 33 times in each experiment. The experiments were conducted 20 times.

Figure 6 shows the precision and recall of these 20 rounds of experiments. Precision is defined as the ratio of the number of correct guesses and the number of times that a guess can be made. Recall is defined as the ratio of the number of correct guesses and the number of total SSH packets. The average precision is 0.956, the average recall is 0.847 and the average F_1 Score is 0.897.

4.2 Persistent B_p

According to our experiments, the B_p will remain unchanged and reused for multiple network packets. This greatly helps the adversary, either by performing pattern matching once and reusing the same B_p directly in subsequent packets, or by improving the accuracy of the guesses.

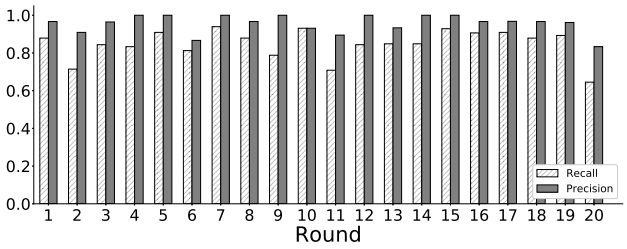


Figure 6: The precision and recall of determining P_{priv} in 20 rounds or experiments.

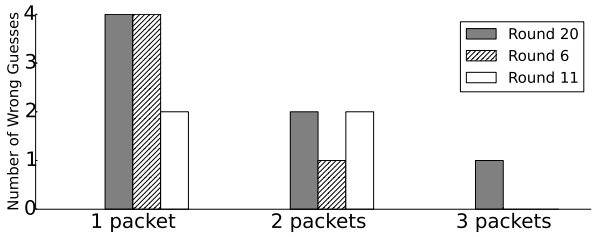


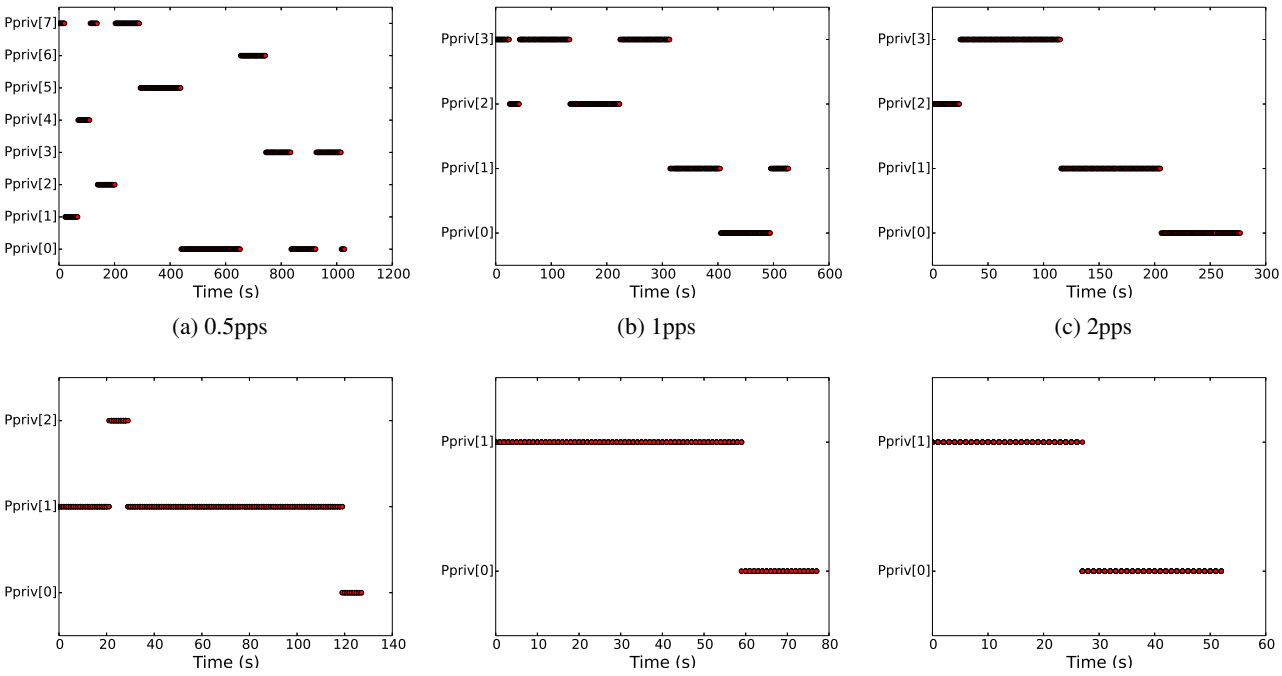
Figure 7: Reduction of incorrect guesses using the N-Streak strategy.

Improving attack fidelity using persistent B_p. The persistent B_p can be used to reduce the number of incorrect guesses. During a real-world attack, when P_{priv} is incorrectly guessed, the ciphertext replacement may crash the guest VM (although we have not experienced any crashes in our experiments). As such, a safer strategy of when to perform ciphertext replacement is only after correctly guessing P_{priv} N times in a streak, which we call the *N-streak strategy*. We then applied this strategy to Round 20, 6 and 11, which have the highest FPR (*i.e.*, 0.167, 0.133, 0.103, respectively). As shown in Figure 7, when by increasing N (*i.e.*, 1, 2, 3), the number of incorrectly performed ciphertext replacement is reduced.

Packet rate vs. B_p persistence. We further evaluated the effect of B_p persistence when the rate of SSH packets varies. Again, on the remote machine, we used *AnJian* to generate keystrokes at a fixed rate, ranging from 0.5 keystrokes per second, to 20 keystrokes per second. The rate of SSH acknowledgement packets is close to the keystroke rate. For each keystroke rate, 500 keystrokes were generated and the number of different B_ps were reported in Figure 8. We can see that as the packet rate increases, fewer number of B_ps will be used to send SSH packets. We repeated this experiment and collected over 200 different B_ps after generating 5000 keystrokes with rates ranging from 0.5 to 20 per second. The statistics of the repeated use of B_ps are shown in Figure 9.

4.3 I/O Performance Degradation

Conducting page-fault based side-channel analysis to guess P_{priv} and performing ciphertext replacement will slow down



(a) 0.5pps (b) 1pps (c) 2pps

(d) 5pps (e) 10pps (f) 20pps

Figure 8: The number of different B_ps used with various rates of packets (pps).

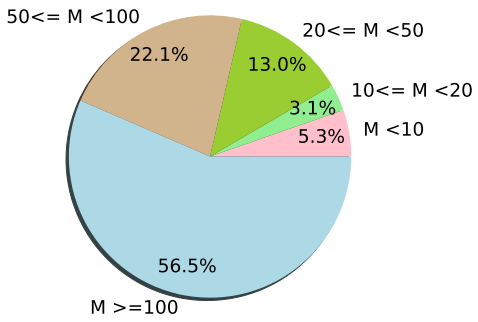


Figure 9: Statistics of repeated B_ps.

the I/O operations of the guest VM. To evaluate the degree of performance degradation, we evaluate the SSH response time on the server side during the attacks. The SSH response time measures the time interval between the QEMU receives an incoming SSH packet to the time that an SSH response packet is sent to QEMU. Note the measurements do not include network latency.

Figure 10 shows the SSH response time under three conditions: Original (not under attack), B_p Persistent (assuming B_p does not change) , and Guess Every Time (assuming B_p changes and making guesses every time). The keystroke rate used in the experiments were 10 keystrokes per second, and in total 1,000 keystrokes were generated during the tests. We can see from the figure, the average SSH response latency without attack is 2.5ms and the median is 0.99ms. The average latency for SSH connection under a B_p-persistent strategy is 6.81ms and the median is 2.4ms. The average latency for SSH connection under a guess-every-time strategy is 8.0ms and the median is 8.7ms. Because the typical network latency of cloud servers are 40-60ms within US and more than 100ms worldwide [5], it is very difficult for the VM owners to detect the latency caused by the attacks.

4.4 An End-to-End Attack

We conducted an end-to-end attack in which the adversary decrypts a 4KB memory page that is encrypted with the guest VM's K_{vek}. The attack assumes a network traffic with the rate of 10 pps, which is simulated using the same method used in the previous sections. Table 2 shows the number of packets and time used to complete the attack, when one or two 16-byte aligned blocks were exploited for the data decryption. We can see that in the four trials we conducted, roughly 300 packets are needed to decrypt the 4KB page, which takes about 40 seconds. The speed of the attack doubles if the first two blocks of the packets were used to decrypt data.

Table 2: End-to-end attack performance.

Round	1 Block		2 Blocks	
	Packets used	Time(s)	Packets used	Time(s)
1	292	43.56	148	21.29
2	329	40.78	177	20.04
3	326	39.21	154	18.99
4	299	33.58	154	16.95

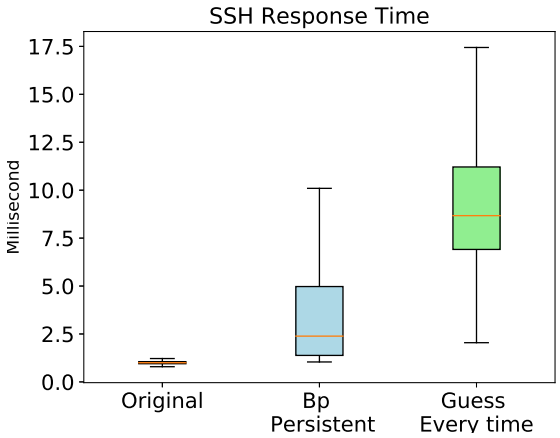


Figure 10: I/O performance degradation evaluated using SSH response time (network latency excluded).

5 A Path Towards I/O Security in SEV

The root cause of the problem is the incompatibility between AMD-V's I/O virtualization with SEV's memory encryption scheme. Specifically, the primary reason of the attacks described in Section 3 is that existing IOMMU hardware only supports memory encryption with ASID = 0 and the operated memory is encrypted with the hypervisor's memory encryption key. Therefore, every I/O operation from the guest VM must go through a shared memory page with the hypervisor. To address this limitation, IOMMU must allow DMA operations to be performed under the ASIDs of other contexts. Meanwhile, it must prevent the privileged software from abusing such IOMMU operations. This design, however, will be very challenging to implement in practice. According to our discussion with AMD researchers, future releases of SEV CPUs are *unlikely* to address this issue. Therefore, alternative solutions must be identified.

In addition to this fundamental issue, the decryption oracle is also enabled by two other vulnerabilities of SEV: (1) no integrity protection of the encrypted memory, and (2) knowledge of the tweak function $T()$. AMD researchers suggested that future SEV CPUs will disable the encryption oracle by providing memory integrity and altering the implementation of the tweak function $T()$. While authenticated memory encryption disables all known attacks against SEV, details of its implementation are yet to be disclosed. We discuss some of

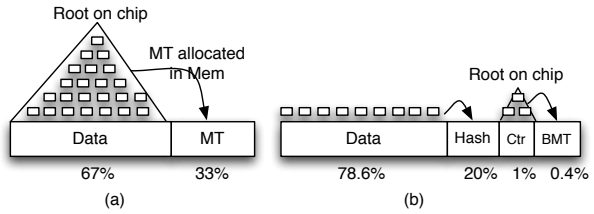


Figure 11: Merkle Tree (a) and Bonsai Merkle Tree used in conjunction with split counter mode encryption (b).

the potential considerations in Section 5.1. Future versions of the tweak function will be implemented as $T(k,a)$, where a is the physical address and k is a random input that changes after every system reboot. We leave the investigation of these vulnerabilities to future work when the technical details are published. In Section 5.2, we present a temporary software fix that works on existing AMD processors (Section 5.2).

It is worth noting that AMD researchers suggested that SEV-ES masks the page offset during page fault. However, we could not find relevant documentation or validate the claim on our testbed. Nevertheless, our analysis (see Section 3.3.2) suggests that the attack is still effective when the page offset information is unavailable. Specifically, we empirically evaluated the attack method that does not rely on page offsets by repeating the experiments in Section 4.1: the mean precision is 0.900, the mean recall is 0.730 and the mean F_1 score is 0.800, which is only slightly lower.

5.1 Authenticated Encryption

Authenticated encryption must be adopted to prevent replay attacks and replacement attacks of the encrypted ciphertext. Merkle Tree (MT) [14] has been proposed for detecting replay and replacement attacks for protecting memory integrity. MT can be built and maintained over any region of memory, and hence it can be used to protect the entire memory or only memory allocated to a VM, or any portion of it. There are two types of MT that can be used, depending on the encryption mode. For direct encryption mode, the MT covers data. For counter-mode encryption, it was shown that replay was only possible if the attacker replays data, its hash, and its counter simultaneously. Hence, protecting counter freshness is sufficient to protect against replay [26]. MT over counters is referred to as Bonsai Merkle Tree (BMT), a variant of which was chosen for implementation in Intel SGX MEE [18].

A fundamental trade off exits between the choice of encryption mode and the overheads of MT. When 128-bit hash is used for MT, MT (and hashes) over data incur memory capacity overhead of 33% (i.e. data-to-MT nodes ratio of 2:1), as illustrated in Figure 11. On the other hand, BMT incurs an overhead of 20% for hashes, plus 0.4% for BMT nodes. Hashes are needed for both encryption modes to pro-

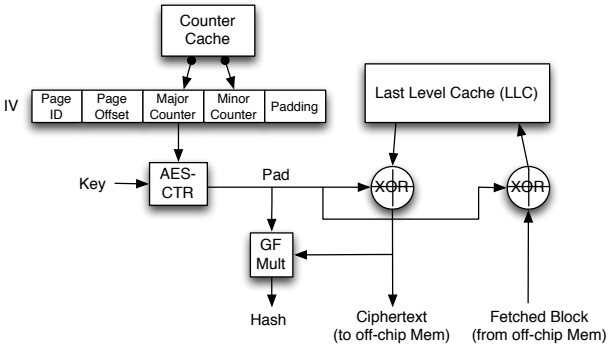


Figure 12: Counter mode encryption with split counters using Galois Counter Mode authentication.

tect against non-replay tampering of memory data. In addition, counter-mode encryption requires additional storage for counters, which depends on the type of counters. 64-bit monolithic counters take up 9% overheads, but split-counters [37] take up only 1%. Taken together, protection against replay incurs 13% memory capacity with direct encryption but only 1.4% with counter mode encryption (Figure 11). For a 1 TB memory, the difference amounts to a substantial 116 GB.

Counter mode encryption is illustrated in Figure 12. Counters are cached on chip, either in regular caches, or in a special "counter cach". With split counters, each page (4KB) of data has its own major counter, and each block (64B) in a page has its own minor counter. When there is a last level cache (LLC) miss, the counter values (major and minor) are concatenated with the page ID and block address of the page (i.e. page offset of the block) to produce a spatially and temporally unique initial vector (IV) [37]. The IV is then encrypted to produce a pad, which will be XOR-ed with ciphertext of data fetched from memory to yield data plaintext. With Galois Counter Mode, the hash of data is obtained a few clock cycles later. Counter mode encryption is more secure than direct mode encryption due to spatial and temporal uniqueness of ciphertexts even for a single plaintext value. Furthermore, as illustrated in the figure, decryption latency is largely overlapped with LLC miss latency; the only exposed latency is 1-cycle XOR of pad and data ciphertext. In contrast, direct memory mode fully exposes decryption latency in the critical path of memory fetch. Therefore, in terms of security protection against replay, memory capacity costs, and performance, AMD SEV can benefit from counter mode encryption and BMT.

A MT/BMT protects the memory from replay or tampering at all time. It is also possible to selectively protect memory region integrity only at times in which they are "expected" to be vulnerable. For example, the time window in which IOMMU buffer is vulnerable is between the time it is written by the DMA until it is read/consumed by the VM. One could take a hash of the memory region at DMA write and verify it when the VM reads the region. Any tampering or replay attempts will be detected. Selective integrity protection may obviate the need for full MT/BMT for attacks that occur within the vulnerable window, but leaves the memory integrity unprotected at other time.

5.2 A Temporary Software Solution

In this section, we present a software solution that can temporarily solve the I/O insecurity issues discussed in this paper. The key idea is to make sure the hypervisor never observe any unencrypted I/O data to/from the SEV-enabled VM. This can be achieved using SEV's platform management APIs [3] and the transport encryption key of the VM K_{tek}.

K_{tek} is a shared Diffie-Hellman (DH) key between the VM owner and the SEV firmware. Particularly, to launch an SEV-enabled VM on an SEV platform, the owner of the VM first requests the Diffie-Hellman (DH) certificate from the platform, which contains the platform's DH public key. The corresponding private key is kept inside the SEV firmware, which cannot be extracted by the system administrator or the hypervisor. The VM owner then sends her DH public key to SEV platform, so that she establishes a shared transport encryption key K_{tek} with the SEV firmware. K_{tek} is only known by the VM owner and the SEV firmware, but not known to the VM itself or hypervisor. `SEND_UPDATE_DATA` and `RECEIVE_UPDATE_DATA` are two commands (among many others) implemented by SEV to assist the hypervisor to launch and manage SEV-enabled VMs [3]. After the VM is launched, the hypervisor may use SEV's `SEND_UPDATE_DATA` command to *atomically* decrypt a piece of memory with K_{vek} and re-encrypt with K_{tek} or use `RECEIVE_UPDATE_DATA` command to decrypt the memory with K_{tek} and re-encrypt with K_{vek}.

Our proposed solution retrofits these APIs and K_{tek} to protect I/O operations. Particularly, the guest VM kernel and the QEMU can be modified so that the guest VM never copies data between the encrypted memory and the unencrypted memory. Instead, to perform any I/O operation to the SEV-enabled VM, the hypervisor issues the `SEND_UPDATE_DATA` and `RECEIVE_UPDATE_DATA` commands to atomically decrypt and re-encrypt data using the two keys K_{vek} and K_{tek}. As both keys are protected inside the SEV firmware, the hypervisor is not able to learn the plaintext during the I/O operations. The SEV firmware serves as a trusted relay of the I/O paths.

However, this solution is only a temporary fix of the issue. This is because the I/O traffic is encrypted with K_{tek}, which is only known to the owner of the VM. Therefore, all I/O operations, including network I/O, disk I/O, and display I/O must be forwarded to a trusted server that is controlled by the VM owner (as shown in Figure 13). Acting as an I/O proxy, the trusted server may limit the application scenarios of SEV and greatly reduce the I/O performance.

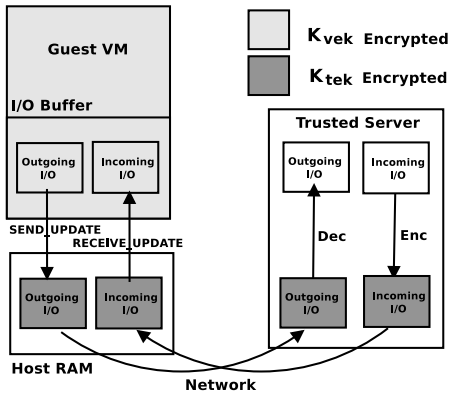


Figure 13: An illustration of the temporary software solution.

6 Related Work

6.1 Existing Security Studies on SEV

The security issues of AMD's SEV have been placed under the spotlight since its debut. Demonstrated security attacks mainly targets SEV's *unencrypted VMCB* [17] and SME's *unauthenticated memory encryption* [9, 12, 17, 25]. The former issue has been fixed using SEV-ES [19] and the latter could be addressed with integrity protection of the encrypted memory. An implementation bug in the firmware of AMD secure processors have also been reported [11]. But since the issue was not related to a design failure, we leave it as out of scope of the paper. We detail these related work as follows:

Unencrypted VMCB. Hetzelt and Buhren analyzed the security of SEV from the perspective of unencrypted virtual machine control block (VMCB) [17]. VMCB is a data structure in memory shared by the hypervisor and the guest VM, which stores the values of guest's general purpose registers and control bits for handling virtual interrupts. At the time of `VMExit`, a malicious hypervisor may learn the machine state of the guest VM by reading register values stored in the VMCB and subsequently alter their values before `VMRun` to control the registers of the guest VM. Hetzelt and Buhren [17] exploit unencrypted VMCB using code gadgets in the guest memory (similar to return-oriented programming (ROP) [29]) to arbitrarily read and write encrypted memory in the guest VM. The security issue caused by unencrypted VMCB, however, has been mitigated by SEV-ES [19], which adds another indirection layer during `VMExit` that allows the guest VM to be notified before Non-Automatic Exits (NAE)—exits requiring hypervisor emulation—and prepares a new data structure called Guest Hypervisor Communication Block—a subset of VMCB—to communicate with the hypervisor. The machine states stored in the VMCB are instead encrypted with authentication, such that they are inaccessible from the hypervisor.

Unauthenticated memory encryption. Because SME does not use authenticated encryption schemes, the integrity of the encrypted memory is not protected. As such, malicious hypervisors may alter the ciphertext of the encrypted memory without triggering errors in the decryption process of the guest VMs. Prior studies have demonstrated a variety of approaches to exploit such unauthenticated memory encryption:

- *Chosen plaintext attacks.* Du *et al.* discovered that SME uses Electronic Codebook (ECB) mode of operation in its memory encryption [12], which implies that the same plaintext always leads to the same ciphertext after encryption. As the only security measure is a physical address based tweak function *XOR*ed with the plaintext before encryption, knowledge of the tweak function will enable the adversary to deduce the relationship between the plaintext of two memory blocks (*i.e.*, of 16 bytes) if their ciphertext are the same. Du *et al.* exploit this weakness by constructing a chosen plaintext attack (via an HTTP server installed on the guest VM) and then replace the ciphertext of an `sshd` program with the ciphertext of instructions specified by the adversary (after applying the tweak functions).
- *Fault injection attacks.* Buhren *et al.* studied fault injection attacks on a simulated SME implementation [9]. Their work considers a different threat model, which assumes that the adversary is able to conduct physical DMA attacks [6] and also run an unprivileged process on the target OS. The unprivileged process performs Prime+Probe side-channel attacks to trace the execution of the SME protected application and, at the proper moment of a cryptographic operation, utilizes DMA attacks to inject memory faults to infer secret keys (or key components). We believe Buhren *et al.*'s attack against SME can be migrated to SEV as well, which is even easier to conduct as the hypervisor can be assumed to be malicious.
- *Page table manipulation.* Remapping guest pages in the nested paging structures to replay previously captured memory pages was first studied by Hetzelt and Buhren [17]. A similar idea was later demonstrated by Morbitzer *et al.* in SEVered, an attack that by manipulating the nested page table alters the virtual memory of the guest VMs to breach the confidentiality of the memory encryption [25]. More specifically, SEVered is carried out in the following steps: First, the malicious hypervisor sends network requests to the guest's network-facing application, *e.g.*, an HTTP server, which allows the attacker to download files larger than one memory page. Second, using a coarse-grained page-level side channel, the attacker determines which of the encrypted guest VM's memory pages are used to store the response data. Third, after locating these pages, the malicious hypervisor changes the page mappings in the nested page table so that these virtual pages used by the guest are mapped to different physical pages. As a result, memory content of these pages can be leaked through the responses of the network applications. The same authors further extend SEVered to perform more

realistic attacks [24], by extracting secret keys in real-world protocols and applications such as TLS, SSH, full disk encryption (FDE). Their attack makes use of the same side channels to identify the set of memory pages that are likely to contain those secrets and scans those pages (roughly 100 pages) until the secrets are found. Both these works only present decryption oracles but not encryption oracles.

While the security issues of SEV's I/O operations are orthogonal to the problems of unauthenticated memory encryption, the decryption oracle presented in this paper does rely on the lack of integrity protection for the ciphertext blocks. However, compared to previous memory decryption attacks against SEV [24, 25], our work differs primarily in three aspects. First, Morbitzer *et al.* [24, 25] manipulate unprotected nested page tables to decouple the mapping between the gVAs and the memory contents, while our decryption oracle directly replaces memory blocks used in the I/O buffer. The hardware mechanisms to defend against these two attacks may differ. Our attack highlights the necessity of mitigating both threats. Second, instead of exploiting a network-facing application executed in the guest VM to accept attacker-controlled data, our attack could make use of any I/O traffic, which is more general. Our paper suggests that application-specific defenses, such pruning secrets after use [24], may not work. Third, the attack in Morbitzer *et al.* requires the attacker to actively generate network traffic to the guest VM, which makes it easily detectable. In contrast, our decryption oracle can make use of existing I/O traffic, which can be very stealthy. Moreover, while the memory integrity issues are expected to be addressed in the next release of SEV CPUs, the fundamental I/O security problem studied in this paper will remain. The encryption oracle will not be mitigated unless the tweak function is completely secured.

Other studies. Mofrad *et al.* [23] compare Intel SGX and AMD SEV, in terms of their functionality, use scenarios, security, and performance implications. The study suggests SEV is more vulnerable than SGX as it lacks memory integrity and has a bloated trust computing base (TCB). Moreover, the performance comparison suggests AMD SEV technology performs better than Intel SGX. Wu *et al.* proposes Fidelius [35], a system that leverages a sibing-based protection mechanism to partition an untrusted hypervisor into two components, one for resource management and the other for security protection. The security of guest VMs is enhanced by the "trusted" security protection component, which, while interesting and effective, unfortunately contradicts with SEV's original intention of eliminating the hypervisor from the TCB. Fidelius mentioned a method to protect disk I/O that is similar to our temporary fix (see Section 5.2) but implies that the disk image is shipped to the SEV platform. Thus it requires using the same K_{tek} every time the disk image is used. Our proxy-style solutions in Section 5.2 is a generalization of their approach.

6.2 Security Threats of Intel TEEs

Intel TME and MKTME. Intel's counterparts of AMD's SME and SEV are Total Memory Encryption (TME) and Multi-Key Total Memory Encryption (MKTME) [18]. The concept of TME is almost the same as AMD SME: an AES-XTS encryption engine sits between a direct data path and external memory buses to encrypt data when leaving the processor and decrypt it when entering the processor. TME supports a single ephemeral encryption key for the entire processor. In contrast, MKTME supports multiple keys; it labels each page table entry with a KeyID to select one of the ephemeral AES keys generated in the encryption engine. Different from AMD SEV, guest VMs in MKTME may have more than one AES key. KeyID0 is used for guest VM to share pages with hyperviosr. KeyID*N* is assigned to guest the *N*th VM by hypervisor for guest's private page. However, the guest VM is able to obtain other KeyIDs to share memory with another guest VMs. As we were not able to purchase a machine with TME and MKTME on the market at the time of writing, we leave the analysis of these Intel's technologies to future work.

Intel SGX. Intel Software Guard eXtension (SGX) is an instruction set architecture extension that supports isolation of memory regions of userspace processes. Through a microcode-extended memory management unit, memory accesses to the protected memory regions, dubbed *enclaves*, are mediated so that only instructions belonging to the same enclave are permitted. Software attacks from all privileged software layers, including operating systems, hypervisors, system management software, are prevented by SGX. A hardware Memory Encryption Engine sits between the processor and the memory to encryption memory traffic on the fly, so that confidentiality of the enclave memory is guaranteed even with physical attackers. Remote attestation is supported in SGX to guard the *integrity* of the enclave code.

Similar to AMD's SEV, SGX constructs TEE on Intel processors. However, it differs from SEV as it only isolates portions of the user processes' memory space, whereas SEV encrypts the memory of the entire virtual machine. Developers of SEV do not need to rewrite the software when using AMD's TEE; but SGX developers have to manually partition applications into trusted and untrusted components, and recompile the source code with the SDKs provided by Intel. SGX machines have been available on market since late 2015. So far, two major types of attacks have been demonstrated to SGX applications.

- *Side-channel attacks.* Prior studies have demonstrated that enclave secrets in SGX can be exfiltrated through side channels on the CPU caches [8, 15, 16, 27], branch target buffers [22], DRAM's row buffer contention [34], page-table entries [33, 34], and page-fault exception handlers [30, 36]. More recently, side-channel attacks exploiting speculative and out-of-order execution have been

shown on SGX as well [10, 32]. Similar to SGX, SEV is not designed to thwart side-channel attacks. Therefore, we expect similar attacks can be carried out on AMD's SEV as well. Because the attacks demonstrated in this paper already completely breaks the confidentiality of SEV-protected VMs, there is no need to rely on side channels to extract secrets. However, in some of the attacks we demonstrate, side channels do facilitate the attacks. We leave the discussion on side-channel surface of SEV to future work.

- *Memory hijacking attacks.* SGX does not guard memory safety inside the enclaves. Studies [7, 21] have shown that attackers could exploit vulnerabilities in enclave programs and perform return-oriented programming (ROP) attacks [29]. Randomization-based security defenses have been proposed to mitigate ROP attacks [28]. However, as pointed out by Biondo *et al.* [7], SGX runtimes inherently contains memory regions that are hard to randomize, and thus completely eliminating the threats of memory hijacking attacks requires eradicating vulnerabilities from the enclave code. As neither SGX nor SEV is designed to provide memory safety, memory hijacking attacks are feasible on SEV as well. We will not further discuss these attacks on SEV in this paper.

AMD SEV is also vulnerable to these attacks. In this paper, we have explored a fine-grained page-fault side channel to locate the memory buffers used in the I/O operations. We leave a comprehensive study of SEV side-channel and memory-hijacking attacks to future work.

7 Conclusion

In this paper, we have reported our study of the insecurity of SEV from the perspective of the unprotected I/O operations in SEV-enabled VMs. The results of our study are two fold: First, I/O operations from SEV guests are not secure; second, I/O operations can be used by the adversary to construct memory encryption and decryption oracles. The concrete attacks have been demonstrated in the paper, along with discussion of potential solutions to the underlying problems.

Acknowledgments. We would like to thank our shepherd Dave Tan and also the anonymous reviewers for the helpful comments. The work was supported in part by the NSF grants 1750809, 1718084, 1834213, and 1834216, and research gifts from Intel and DFINITY foundation to Yinqian Zhang. Yan Solihin is supported in part by UCF.

References

[1] AMD. AMD-V nested paging. http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf, 2008.

[2] AMD. Amd64 architecture programmer's manual volume 2: System programming, 2017.

[3] AMD. Secure encrypted virtualization api version 0.17, 2018.

[4] AMD. Solving the cloud trust problem with WinMagic and AMD EPYC hardware memory encryption. *White paper*, 2018.

[5] Amazon AWS. Optimizing latency and bandwidth for AWS traffic, 2016.

[6] Michael Becher, Maximillian Dornseif, and Christian N. Klein. FireWire: all your memory are belong to us. In *CanSecWest*, 2005.

[7] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The guard's dilemma: Efficient code-reuse attacks against intel SGX. In *27th USENIX Security Symposium*, pages 1213–1227. USENIX Association, 2018.

[8] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies*, 2017.

[9] Robert Buhren, Shay Gueron, Jan Nordholz, Jean-Pierre Seifert, and Julian Vetter. Fault attacks on encrypted general purpose compute platforms. In *7th ACM on Conference on Data and Application Security and Privacy*. ACM, 2017.

[10] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. Sgxpectre attacks: Stealing intel secrets from sgx enclaves via speculative execution. In *4th IEEE European Symposium on Security and Privacy*. IEEE, 2019.

[11] CTS. Severe security advisory on AMD processors. https://safefirmware.com/amdflaws_whitepaper.pdf, 2017.

[12] Zhao-Hui Du, Zhiwei Ying, Zhenke Ma, Yufei Mai, Phoebe Wang, Jesse Liu, and Jesse Fang. Secure encrypted virtualization is unsecure. *arXiv preprint arXiv:1712.05090*, 2017.

[13] Fujian Chuang YI Jia He Digital Inc. Anjian v1.1.0. www.anjian.com, 2019.

[14] Blaise Gassend, G. Edward Suh, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Caches and hash trees for efficient memory integrity verification. In *9th International Symposium on High-Performance Computer Architecture*, 2003.

[15] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *EUROSEC*, 2017.

[16] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *USENIX Annual Technical Conference*, 2017.

[17] Felicitas Hetzelt and Robert Buhren. Security analysis of encrypted virtual machines. In *ACM SIGPLAN Notices*. ACM, 2017.

[18] Intel. Intel architecture: Memory encryption technologies specification, 2017.

[19] David Kaplan. Protecting VM register state with SEV-ES. *White paper*, 2017.

[20] David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption. *White paper*, 2016.

[21] Jae-Hyuk Lee, Jin Soo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *26th USENIX Security Symposium*, 2017.

[22] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium*, 2017.

[23] Saeid Mofrad, Fengwei Zhang, Shiyong Lu, and Weidong Shi. A comparison study of intel SGX and AMD memory encryption technology. In *7th International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2018.

[24] Mathias Morbitzer, Manuel Huber, and Julian Horsch. Extracting secrets from encrypted virtual machines. In *9th ACM Conference on Data and Application Security and Privacy*. ACM, 2019.

[25] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. SEVered: Subverting AMD's virtual machine encryption. In *11th European Workshop on Systems Security*. ACM, 2018.

[26] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. Using address independent seed encryption and bonsai Merkle trees to make secure processors os- and performance-friendly. In *40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[27] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. *Malware Guard Extension: Using SGX to Conceal Cache Attacks*. Springer International Publishing, 2017.

[28] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. Sgx-shield: Enabling address space layout randomization for SGX programs. In *24th Annual Network and Distributed System Security Symposium*, 2017.

[29] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *14th ACM Conference on Computer and Communications Security*. ACM, 2007.

[30] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *11th ACM on Asia Conference on Computer and Communications Security*, 2016.

[31] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on ssh. In *USENIX Security Symposium*, 2001.

[32] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium*, 2018.

[33] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *26th USENIX Security Symposium*. USENIX Association, 2017.

[34] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[35] Yuming Wu, Yutao Liu, Ruifeng Liu, Haibo Chen, Binyu Zang, and Haibing Guan. Comprehensive VM protection against untrusted hypervisor through retrofitted AMD memory encryption. In *International Symposium on High Performance Computer Architecture*, 2018.

[36] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy*. IEEE, 2015.

[37] Chenyu Yan, B. Rogers, D. Englender, D. Solihin, and M. Prvulovic. Improving cost, performance, and security of memory encryption and authentication. In *33rd International Symposium on Computer Architecture*, 2006.

Detecting and Characterizing Lateral Phishing at Scale

Grant Ho[†○] *Asaf Cidon*[○Ψ] *Lior Gavish*[○] *Marco Schweighauser*[○]
Vern Paxson[†ϕ] *Stefan Savage*[⋆] *Geoffrey M. Voelker*[⋆] *David Wagner*[†]

[○]*Barracuda Networks* [†]*UC Berkeley* [⋆]*UC San Diego* [Ψ]*Columbia University* [ϕ]*International Computer Science Institute*

Abstract

We present the first large-scale characterization of lateral phishing attacks, based on a dataset of 113 million employee-sent emails from 92 enterprise organizations. In a lateral phishing attack, adversaries leverage a compromised enterprise account to send phishing emails to other users, benefitting from both the implicit trust and the information in the hijacked user's account. We develop a classifier that finds hundreds of real-world lateral phishing emails, while generating under four false positives per every one-million employee-sent emails. Drawing on the attacks we detect, as well as a corpus of user-reported incidents, we quantify the scale of lateral phishing, identify several thematic content and recipient targeting strategies that attackers follow, illuminate two types of sophisticated behaviors that attackers exhibit, and estimate the success rate of these attacks. Collectively, these results expand our mental models of the 'enterprise attacker' and shed light on the current state of enterprise phishing attacks.

1 Introduction

For over a decade, the security community has explored a myriad of defenses against phishing attacks. Yet despite this long line of work, modern-day attackers routinely and successfully use phishing attacks to compromise government systems, political figures, and companies spanning every economic sector. Growing in prominence each year, this genre of attacks has risen to the level of government attention, with the FBI estimating $12.5 billion in financial losses worldwide from 78,617 reported incidents between October 2013 to May 2018 [12], and the US Secretary of Homeland Security declaring that phishing represents "the most devastating attacks by the most sophisticated attackers" [39].

By and large, the high-profile coverage around targeted spearphishing attacks against major entities, such as Google, RSA, and the Democratic National Committee, has captured and shaped our mental models of enterprise phishing attacks [35, 43, 46]. In these newsworthy instances, as well as many of the targeted spearphishing incidents discussed in the academic literature [25, 26, 28], the attacks come from external accounts, created by nation-state adversaries who cleverly craft or spoof the phishing account's name and email address to resemble a known and legitimate user. However, in recent years, work from both industry [7, 24, 36] and academia [6, 18, 32, 41] has pointed to the emergence and growth of *lateral phishing* attacks: a new form of phishing that targets a diverse range of organizations and has already incurred billions of dollars in financial harm [12]. In a lateral phishing attack, an adversary uses a compromised enterprise account to send phishing emails to a new set of recipients. This attack proves particularly insidious because the attacker automatically benefits from the implicit trust in the hijacked account: trust from both human recipients and conventional email protection systems.

Although recent work [10, 15, 18, 19, 41] presents several ideas for detecting lateral phishing, these prior methods either require that organizations possess sophisticated network monitoring infrastructure, or they produce too many false positives for practical usage. Moreover, no prior work has characterized this attack at a large, generalizable scale. For example, one of the most comprehensive related work uses a multi-year dataset from one organization, which only contains two lateral phishing attacks [18]. This state of affairs leaves many important questions unanswered: How should we think about this class of phishing with respect to its scale, sophistication, and success? Do attackers follow thematic strategies, and can these common behaviors fuel new or improved defenses? How are attackers capitalizing on the information within the hijacked accounts, and what does their behavior say about the state and trajectory of enterprise phishing attacks?

In this joint work between academia and Barracuda Networks we take a first step towards answering these open questions and understanding *lateral phishing* at scale. This paper seeks to both explore avenues for practical defenses against this burgeoning threat and develop accurate mental models for the state of these phishing attacks in the wild.

First, we present a new classifier for detecting URL-based lateral phishing emails and evaluate our approach on a dataset of 113 million emails, spanning 92 enterprise organizations. While the dynamic churn and dissimilarity in content across phishing emails proves challenging, our approach can detect 87.3% of attacks in our dataset, while generating less than 4 false positives per every 1,000,000 employee-sent emails.

Second, combining the attacks we detect with a corpus of user-reported lateral phishing attacks, we conduct the first large-scale characterization of lateral phishing in real-world organizations. Our analysis shows that this attack is potent and widespread: dozens of organizations, ranging from ones with fewer than 100 employees to ones with over 1,000 employees, experience lateral phishing attacks within the span

of several months; in total, 14% of a set of randomly sampled organizations experienced at least one lateral phishing incident within a seven-month timespan. Furthermore, we estimate that over 11% of attackers successfully compromise at least one additional employee. Even though our ground truth sources and detector face limitations that restrict their ability to uncover stealthy or narrowly targeted attacks, our results nonetheless illuminate a prominent threat that currently affects many real-world organizations.

Examining the behavior of lateral phishers, we explore and quantify the popularity of four recipient (victim) selection strategies. Although our dataset's attackers target dozens to hundreds of recipients, these recipients often include a subset of users with some relationship to the hijacked account (e.g., fellow employees or recent contacts). Additionally, we develop a categorization for the different levels of content tailoring displayed by our dataset's phishing messages. Our categorization shows that while 7% of attacks deploy targeted messages, most attacks opt for generic content that a phisher could easily reuse across multiple organizations. In particular, we observe that lateral phishers rely predominantly on two common lures: a pretext of a shared document and a fake warning message about a problem with the recipient's account. Despite the popularity of non-targeted content, nearly one-third of our dataset's attackers invest additional time and effort to make their attacks more convincing and/or to evade detection; and, over 80% of attacks occur during the normal working hours of the hijacked account.

Ultimately, this work yields two contributions that expand our understanding of enterprise phishing and potential defenses against it. First, we present a novel detector that achieves an order-of-magnitude better performance than prior work, while operating on a minimal data requirement (only leveraging historical emails). Second, through the first large-scale characterization of lateral phishing, we uncover the scale and success of this emerging class of attacks and shed light on common strategies that lateral phishers employ. Our analysis illuminates a prevalent class of enterprise attackers whose behavior does not fully match the tactics of targeted nation-state attacks or industrial espionage. Nonetheless, these lateral phishers still achieve success in the absence of new defenses, and many of our dataset's attackers do exhibit some signs of sophistication and focused effort.

2 Background

In a *lateral phishing* attack, attackers use a compromised, but legitimate, email account to send a phishing email to their victim(s). The attacker's goals and choice of malicious payload can take a number of different forms, from a malware-infected attachment, to a phishing URL, to a fake payment request. Our work focuses on lateral phishing attacks that employ a malicious URL embedded in the email, which is the most common exploit method identified in our dataset.

Listing 1: An anonymized example of a lateral phishing message that uses the lure of a fake contract document.

```
From: "Alice" <alice@company.com>
To: "Bob" <bob@company.com>
Subject: Company X (New Contract)

New Contract

View Document [this text linked to a phishing website]

Regards,
Alice [signature]
```

Listing 1 shows an anonymized example of a lateral phishing attack from our study. In this attack, the phisher tried to lure the recipient into clicking on a link under the false pretense of a new contract. Additionally, the attacker also tried to make the deception more credible by responding to recipients who inquired about the email's authenticity; and they also actively hid their presence in the compromised user's mailbox by deleting all traces of their phishing email.

Lateral phishing represents a dangerous but understudied attack at the intersection of phishing and account hijacking. Phishing attacks, broadly construed, involve an attacker crafting a deceptive email from any account (compromised or spoofed) to trick their victim into performing some action. Account hijacking, also known as account takeover (ATO) in industry parlance, involves the use of a compromised account for any kind of malicious means (e.g., including spam). While prior work primarily examines each of these attacks at a smaller scale and with respect to personal accounts, our work studies the intersection of both of these at a large scale and from the perspective of enterprise organizations. In doing so, we expand our understanding of important enterprise threats, avenues for defending against them, and strategies used by the attackers who perpetrate them.

2.1 Related Work

Detection: An extensive body of prior literature proposes numerous techniques for detecting traditional phishing attacks [1,3,13,14,44], as well as more sophisticated spearphishing attacks [8, 10, 23, 41, 47]. Hu et al. studied how to use social graph metrics to detect malicious emails sent from compromised accounts [19]. Their approach detects hijacked accounts with false positive rates between 20–40%. Unfortunately, in practice, many organizations handle tens of thousands of employee-sent emails per day, so a false positive rate of 20% would lead to thousands of false alerts each day. IdentityMailer, proposed by Stringhini et al. [41], detects lateral phishing attacks by training behavior models based on timing patterns, metadata, and stylometry for each user. If a new email deviates from an employee's behavioral model,

their system flags it as an attack. While promising, their approach produces false positive rates in the range of 1–10%, which is untenable in practice given the high volume of benign emails and low base rate of phishing. Additionally, their system requires training a behavioral model for each employee, incurring expensive technical debt to operate at scale.

Ho et al. developed methods for detecting lateral spearphishing by applying a novel anomaly detection algorithm on a set of features derived from historical user login data and enterprise network traffic logs [18]. Their approach detects both known and newly discovered attacks, with a false positive rate of 0.004%. However, organizations with less technical expertise often lack the infrastructure to comprehensively capture the enterprise's network traffic, which this prior approach requires. This technical prerequisite begs the question, can we practically detect lateral phishing attacks with a more minimalist dataset: only the enterprise's historical emails? Furthermore, their dataset reflects a single enterprise that experienced only two lateral phishing attacks across a 3.5-year timespan, which prevents them from characterizing the nature of lateral phishing at a general scale.

Characterization: While prior work shows that attackers frequently use phishing to compromise accounts, and that attackers occasionally conduct (lateral) phishing from these hijacked accounts, few efforts have studied the nature of lateral phishing in depth and at scale. Examining a sample of phishing emails, webpages, and compromised accounts from Google data sources, one prior study of account hijacking discovered that attackers often use these accounts to send phishing emails to the account's contacts [6]. However, they concluded that automatically detecting such attacks proves challenging. Onaolapo et al. studied what attackers do with hijacked accounts [32], but they did not examine lateral phishing. Separate from email accounts, a study of compromised Twitter accounts found that infections appear to spread laterally through the social network. However their dataset did not allow direct observation of the lateral attack vector itself [42], nor did it provide insights into the domain of compromised enterprise accounts (given the nature of social media).

Open Questions and Challenges: Prior work makes clear that account compromise poses a significant and widespread problem. This literature also presents promising defenses for enterprises that have sophisticated monitoring in place. Yet despite these advances, several key questions remain unresolved. Do organizations without comprehensive monitoring and technical expertise have a practical way to defend against lateral phishing attacks? What common strategies and tradecraft do lateral phishers employ? How are lateral phishers capitalizing on their control of legitimate accounts, and what does their tactical sophistication say about the state of enterprise phishing? This paper takes a step towards answering these open questions by presenting a new detection strategy and a large-scale characterization of lateral phishing attacks.

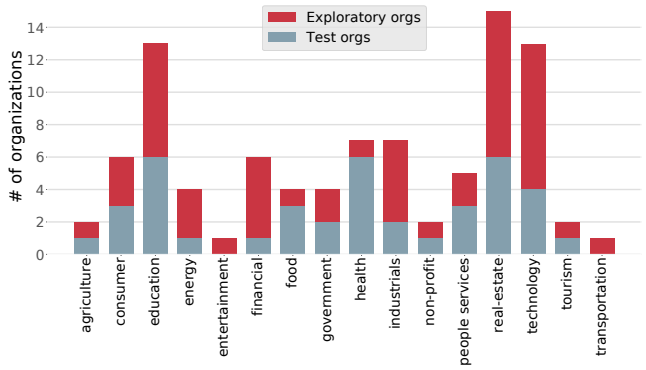


Figure 1: Breakdown of the economic sectors across our dataset's 52 exploratory organizations versus the 40 test organizations.

2.2 Ethics

In this work, our team, consisting of researchers from academia and a large security company, developed detection techniques using a dataset of historical emails and reported incidents from 92 organizations who are active customers of Barracuda Networks. These organizations granted Barracuda permission to access their Office 365 employee mailboxes for the purpose of researching and developing defenses against lateral phishing. Per Barracuda's policies, all fetched emails are stored encrypted, and customers have the option of revoking access to their data at any time.

Due to the sensitivity of the data, only authorized employees at Barracuda were allowed to access the data (under standard, strict access control policies). No personally identifying information or sensitive data was shared with any non-employee of Barracuda. Our project also received legal approval from Barracuda, who had permission from their customers to analyze and operate on the data.

Once Barracuda deployed a set of lateral phishing detectors to production, any detected attacks were reported to customers in real time to prevent financial loss and harm.

3 Data

Our dataset consists of employee-sent emails from 92 English-language organizations; 23 organizations came from randomly sampling enterprises that had reports of lateral phishing, and 69 were randomly sampled from all organizations. Across these enterprises, 25 organizations have 100 or fewer user accounts, 34 have between 101–1000 accounts, and 33 have over 1000 accounts. Real-estate, technology, and education constitute the three most common industries in our dataset, with 15, 13, and 13 enterprises respectively; Figures 1 and 2 show the distribution of the economic sectors and sizes of our dataset's organizations, broken down by *exploratory organizations* versus *test organizations* (§ 3.2).

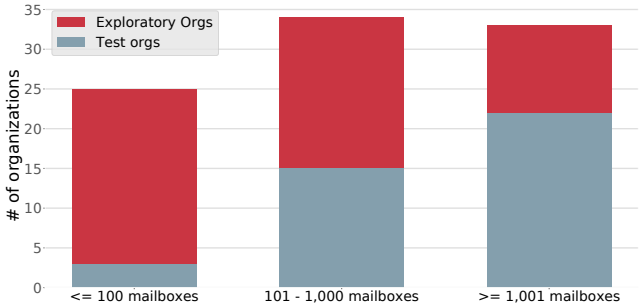


Figure 2: Breakdown of the organization sizes across our dataset's 52 exploratory organizations versus the 40 test organizations.

3.1 Schema

The organizations in our dataset use Office 365 as their email provider. At a high level, each email object contains: a unique Office 365 identifier; the email's metadata (SMTP header information), which describes properties such as the email's sent timestamp, recipients, purported sender, and subject; and the email's *body*, the contents of the email message in full HTML formatting. Office 365's documentation describes the full schema of each email object [29]. Additionally, for each organization, we have a set of *verified domains*: domains which the organization has declared that it owns.

3.2 Dataset Size

Our dataset consists of 113,083,695 unique, employee-sent emails. To ensure our detection techniques generalized (Section 5.1), we split our data into a training dataset of emails from 52 'exploratory organizations' during April–June 2018, and a test dataset covering July–October 2018 from 92 organizations. Our test dataset consists of emails from the 52 exploratory organizations (but from a later, disjoint time period than our training dataset), plus data from an additional, held-out set of 40 'test organizations'. We selected the 40 test organizations via a random sample that we performed prior to analyzing any data. Our training dataset has 25,670,264 emails, and our test dataset has 87,413,431 emails. Both sets of organizations cover a diverse range of industries and sizes as shown in Figures 1 and 2. The exploratory organizations span a total of 89,267 user mailboxes that sent or received email, and the test organizations have 138,752 mailboxes (based on the data from October 2018).[1]

3.3 Ground truth

Our set of lateral phishing emails comes from two sources: (1) attack emails reported to Barracuda by an organization's security administrators, as well as attacks reported by users to their organization or directly to Barracuda, and (2) emails

[1]The number of mailboxes is an upper bound on the number of employees due to the use of mailing lists and aliases.

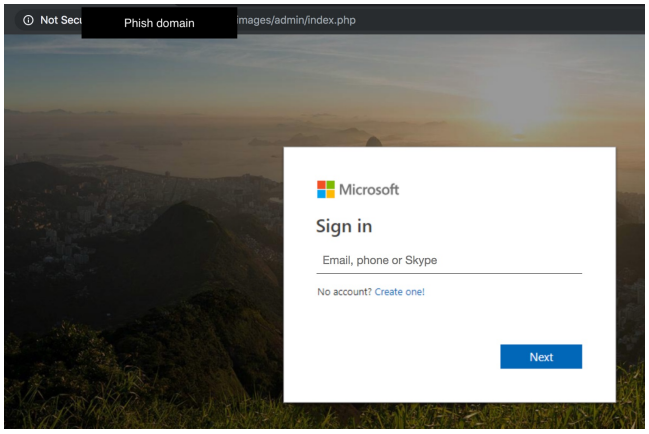


Figure 3: An anonymized screenshot of the web page that a phishing URL in a lateral phishing email led to.

flagged by our detector (§4), which we manually reviewed and labeled before including.

At a high-level, to manually label an email as phishing, or not, we examined its message content, Office 365 metadata, and Internet Message Headers [33] to determine whether the email contained phishing content, and whether the email came from a compromised account (versus an external account, which we do not treat as lateral phishing). For example, if the Office 365 metadata showed that a copy of the email resided in the employee's *Sent Items* folder, or if its headers showed that the email passed the corresponding SPF or DKIM [9] checks, then we considered the email to be lateral phishing. Appendix §A.1 describes our labeling procedure in detail.

Additionally, for a small sample of URLs in these lateral phishing emails, employees at Barracuda accessed the phishing URL in a VM-contained browser to better understand the end goals of the attack. To minimize potential harm and side effects, these employees only visited phishing URLs which contained no unique identifiers (i.e., no random strings or user/organization information in the URL path). To handle any phishing URLs that resided on URL-shortening domains, we used one of Barracuda's URL-expansion APIs that their production services already apply to email URLs, and only visited suspected phishing links that expanded to a non-side-effect URL. Most phishing URLs we explored led to a SafeBrowsing interstitial webpage, likely reflecting our use of historical emails, rather than what users would have encountered contemporaneously. However, more recent malicious URLs consistently led to credential phishing websites designed to look like a legitimate Office 365 login page (the email service provider used by our study's organizations); Figure 3 shows an anonymized example of one phishing website.

In total, our dataset contains 1,902 lateral phishing emails (unique by subject, sender, and sent-time), sent by 154 hijacked employee accounts from 33 organizations. 1,694 of these emails were reported by users, with the remainder found solely by our detector (§ 4); our detector also finds many of the

user-reported attacks as well (§ 5). Among the user-reported attacks, 40 emails (from 12 hijacked accounts) contained a fake wire transfer or malicious attachment, while the remaining 1,862 emails used a malicious URL.

We focus our detection strategy on URL-based phishing, given the prevalence of this attack vector. This focus means that our analysis and detection techniques do not reflect the full space of lateral phishing attacks. Despite this limitation, our dataset's attacks span dozens of organizations, enabling us to study a prevalent class of enterprise phishing that poses an important threat in its own right.

4 Detecting Lateral Phishing

Adopting the *lateral attacker* threat model defined by Ho et al. [18], we focus on phishing emails sent by a compromised employee account, where the attack embeds a malicious URL as the exploit (e.g., leading the user to a phishing webpage).

We explored three strategies for detecting lateral phishing attacks, but ultimately found that one of the strategies detected nearly all of the attacks identified by all three approaches. At a high level, the two less fruitful strategies detected attacks by looking for emails that contained (1) a rare URL and (2) a message whose text seemed likely to be used for phishing (e.g., similar text to a known phishing attack). Because our primary detection strategy detected all-but-two of the attacks found by the other strategies, while finding over ten times as many attacks, we defer discussion of the two less successful approaches to our extended technical report [17]; below, we focus on exploring the more effective strategy in detail. In our evaluation, we include the two additional attacks found by the alternative approaches as false negatives for our detector.

Overview: We examined the user-reported lateral phishing incidents in our training dataset (April–June 2018) to identify widespread themes and behaviors that we could leverage in our detector. Grouping this set of attacks by the hijacked account (*ATO*) that sent them, we found that 95% of these ATOs sent phishing emails to 25 or more distinct recipients.[2] This prevalent behavior, along with additional feature ideas inspired by the lure-exploit detection framework [18], provide the basis for our detection strategy. In the remainder of this section, we describe the features our detector uses, the intuition behind these features, and our detector's machine learning procedure for classifying emails.

Our techniques provide neither an all-encompassing approach to finding every attack, nor guaranteed robustness against motivated adversaries trying to evade detection. However, we show in Section 5 that our approach finds hundreds of lateral phishing emails across dozens of real-world organizations, while incurring a low volume of false positives.

[2]To assess the generalizability of our approach, our evaluation uses a withheld dataset, from a later timeframe and with new organizations (§ 5).

Features: Our detector extracts three sets of features. The first set consists of two features that target the popular behavior we observed earlier: contacting many recipients. Given an email, we first extract the number of unique recipients across the email's `To`, `CC`, and `BCC` headers. Additionally, we compute the Jaccard similarity of this email's recipient set to the closest set of historical recipients seen in any employee-sent email from the preceding month. We refer to this latter (similarity) feature as the email's recipient likelihood score.

The next two sets of features draw upon the lure-exploit phishing framework proposed by Ho et al. [18]. This framework posits that phishing emails contain two necessary components: a 'lure', which convinces the victim to believe the phishing email and perform some action; and an 'exploit': the malicious action the victim should execute. Their work finds that using features that target both of these two components significantly improves a detector's performance.

To characterize whether a new email contains a potential phishing lure, our detector extracts a single, lightweight boolean feature based on the email's text. Specifically, Barracuda provided us with a set of roughly 150 keywords and phrases that frequently occur in phishing attacks. They developed this set of 'phishy' keywords by extracting the link text from several hundred real-world phishing emails (both external and lateral phishing) and selecting the (normalized) text that occurred most frequently among these attacks. Thematically, these suspicious keywords convey a call to action that entices the recipient to click a link. For our 'lure' feature, we extract a boolean value that indicates whether an email contains any of these phishy keywords.

Finally, we complete our detector's feature set by extracting two features that capture whether an email might contain an exploit. Since our work focuses on URL-based attacks, this set of features reflects whether the email contains a potentially dangerous URL.

First, for each email, we extract a *global URL reputation* feature that quantifies the rarest URL an email contains. Given an email, we extract all URLs from the email's body and ignore URLs if they fall under two categories: we exclude all URLs whose domain is listed on the organization's *verified domain* list (§ 3.1), and we also exclude all URLs whose displayed, hyperlinked text exactly matches the URL of the hyperlink's underlying destination. For example, in Listing 1's attack, the displayed text of the phishing hyperlink was "Click Here", which does not match the hyperlink's destination (the phishing site), so our procedure would keep this URL. In contrast, Alice's signature from Listing 1 might contain a link to her personal website, e.g., `www.alice.com`; our procedure would ignore this URL, since the displayed text of `www.alice.com` matches the hyperlink's destination.

This latter filtering criteria makes the assumption that a phishing URL will attempt to obfuscate itself, and will not display the true underlying destination directly to the user. After these filtering steps, we extract a numerical feature by

mapping each remaining URL to its registered domain, and then looking up each domain's ranking on the Cisco Umbrella Top 1 Million sites [20];[3] for any unlisted domain, we assign it a default ranking of 10 million. We treat two special cases differently. For URLs on shortener domains, our detector attempts to recursively resolve the shortlink to its final destination. If this resolution succeeds, we use the global ranking of the final URL's domain; otherwise, we treat the URL as coming from an unranked domain (10 million). For URLs on content hosting sites (e.g., Google Drive or Sharepoint), we have no good way to determine its suspiciousness without fetching the content and analyzing it (an action that has several practical hurdles). As a result, we treat all URLs on content hosting sites as if they reside on unranked domains.

After ranking each URL's domain, we set the email's *global URL reputation* feature to be the worst (highest) domain ranking among its URLs. Intuitively, we expect that phishers will rarely host phishing pages on popular sites, so a higher *global URL reputation* indicates a more suspicious email. In principle a motivated adversary could evade this feature; e.g., if an adversary can compromise one of the organization's verified domains, they can host their phishing URL from this compromised site and avoid an accurate ranking. However, we found no such instances in our set of user-reported lateral phishing. Additionally, since the goal of this paper is to begin exploring practical detection techniques, and develop a large set of lateral phishing incidents for our analysis, this feature suffices for our needs.

In addition to this global reputation metric, we extract a local metric that characterizes the rareness of a URL with respect to the domains of URLs that an organization's employees typically send. Given a set of URLs embedded within an email, we map each URL to its fully-qualified domain name (FQDN) and count the number of days from the preceding month where at least one employee-sent email included a URL on the FQDN. We then take the minimum value across all of an email's URLs; we call this minimum value the *local URL reputation* feature. Intuitively, suspicious URLs will have both a low global reputation and a low local reputation. However, our evaluation (§ 5.2) finds that this *local URL reputation* feature adds little value: URLs with a low *local URL reputation* value almost always have a low *global URL reputation* value, and vice versa.

Classification: To label an email as phishing or not, we trained a Random Forest classifier [45] with the aforementioned features. To train our classifier, we take all user-reported lateral phishing emails in our training dataset, and combine them with a set of likely-benign emails. We generate this set of "benign" emails by randomly sampling a subset of the training window's emails that have not been reported as phishing; we sample 200 of these benign emails for each attack email to form our set of benign emails for training. Following standard machine learning practices, we selected both the hyperparameters for our classifier and the exact downsampling ratio (200:1) using cross-validation on this training data. Appendix A.2 describes our training procedure in more detail.

Once we have a trained classifier, given a new email, our detector extracts its features, feeds the features into this classifier, and outputs the classifier's decision.

[3] We use a list fetched in early March 2018 for our feature extraction, but in practice, one could use a continuously updated list.

5 Evaluation

In this section we evaluate our lateral phishing detector. We first describe our testing methodology, and then show how well the detector performs on millions of emails from over 90 organizations. Overall, our detector has a high detection rate, generates few false positives, and detects many new attacks.

5.1 Methodology

Establishing Generalizability: As described earlier in Section 3.2, we split our dataset into two disjoint segments: a *training dataset* consisting of emails from the 52 exploratory organizations during April–June 2018 and a *test dataset* from 92 enterprises during July–October 2018; in § 5.2, we show that our detector's performance remains the same if our test dataset contains only the emails from the 40 withheld test organizations. Given these two datasets, we first trained our classifier and tuned its hyperparameters via cross validation on our training dataset (Appendix A.2). Next, to compute our evaluation results, we ran our detector on each month of the held-out test dataset. To simulate a classifier in production, we followed standard machine learning practices and used a continuous learning procedure to update our detector each month [38]. Namely, at the end of each month, we aggregated the user-reported and detector-discovered phishing emails from all previous months into a new set of phishing 'training' data; and, we aggregated our original set of randomly sampled benign emails with our detector's false positives from all previous months to form a new benign 'training' dataset. We then trained a new model on this aggregated training dataset and used this updated model to classify the subsequent month's data. However, to ensure that any tuning or knowledge we derived from the training dataset did not bias or overfit our classifier, we did not alter any of the model's hyperparameters or features during our evaluation on the test dataset.

Our evaluation's temporal-split between the training and test datasets, along with the introduction of new data from randomly withheld organizations into the test dataset, follows best practices that recommend this approach over a randomized cross-validation evaluation [2, 31, 34]. A completely randomized evaluation (e.g., cross-validation) risks training on data from the future and testing on the past, which might lead us to overestimate the detector's effectiveness. In contrast,

Metric	Training April – June 2018	Testing July – October 2018
Organizations	52 Exploratory	52 Exploratory + 40 Test
Detected Known Attacks	34	47
Detected New Attacks	28	49
Missed Attacks (FN)	8	14
Detection Rate	88.6%	87.3%
Total Emails	25,670,264	87,413,431
False Positives (FP)	136	316
False Positive Rate	0.00053%	0.00036%
Precision	31.3%	23.3%

Table 1: Evaluation results of our detector. 'Detected Known Attacks' shows the number of incidents that our detector identified, and were also reported by an employee at an organization. 'Detected New Attacks' shows the number of incidents that our detector identified, but were not reported by anyone. 'Missed Attacks (FN)' shows all incidents either reported by a user or found by any of our detection strategies, but our detector marked it as benign (false negative). Of the 22 incidents our detector misses, 12 are attachment-based attacks, a threat model which our detector explicitly does not target but which we include in our FN and Detection Rate results for completeness.

our methodology evaluates our detector with fresh data from a "future" time period and introduces 40 new organizations, neither of which our detector saw during training time; this also reflects how a detector operates in practice.

Alert Metric (Incidents): We have several choices for modeling our detector's alert generation process (i.e., how we count *distinct* attacks). For example, we could evaluate our detector's performance in terms of how many unique emails it correctly labels. Or, we could measure our detector's performance in terms of how many distinct employee accounts it marks as compromised (modeling a detector that generates one alert per account and suppresses the rest). Ultimately, we select a notion commonly used in practice, that of an *incident*, which corresponds to a unique (subject, sender email address) pair. At this granularity, our detector's alert generation model produces a single alert per unique (subject, sender) pair. This metric avoids biased evaluation numbers that overemphasize compromise incidents that generate many identical emails during a single attack. For example, if there are two incidents, one which generates one hundred emails to one recipient each, and another which generates one email to 100 recipients, a detector's performance on the hundred-email incident will dominate the result if we count attacks at the email level.

In total, our training dataset contains 40 lateral phishing incidents from our user-reported ground truth sources, and our test dataset contains 61 user-reported incidents. Our detector finds an additional 77 unreported incidents (row 2 of Table 1).

5.2 Detection Results

Table 1 summarizes the performance metrics for our detector. We use the term *Detection Rate* to refer to the percentage of lateral phishing incidents that our detector finds, divided by all known attack incidents in our dataset (i.e., any user-reported incident and any incident found by any detection technique we tried). For completeness, we include the 12 attachment-based incidents in our False Negative and Detection Rate computations, which our detector obviously misses since we designed it to catch URL-based lateral phishing. Additionally, we also include, as false negatives, 2 training incidents that our less successful detectors identified [17]; these two alternative strategies did not find any new attacks in the test dataset. Thus, the *Detection Rate* reflects a best-effort assessment that potentially overestimates the true positive rate of our detector, since we have an imperfect ground truth that cannot account for narrowly targeted attacks that go unreported by users. *Precision* equals the percent of attack alerts (incidents) produced by our detector divided by the total number of alerts our detector generated (attacks plus false positives).

Training and Tuning: On the training dataset, our detector correctly identified 62 out of 70 lateral phishing incidents (88.6%), while generating a total of 62 false positives (on 25.7 million employee-sent emails).

Our PySpark Random Forest classifier exposes a built-in estimate of each feature's relative importance [40], where each feature receives a score between 0.0–1.0 and the sum of all the scores adds up to 1.0. Based on these feature weights, our model places the most emphasis on the *global URL reputation* feature, giving it a weight of 0.42, and the email's 'number of recipients' feature (0.34). In contrast, our model essentially ignores our *local URL reputation*, assigning it a score of 0.01, likely because most globally rare domains tend to also be locally rare. Of the remaining features, the recipient likelihood feature has a weight of 0.17 and the 'phishy' keyword feature has a weight of 0.06.

Test Dataset: Our detector correctly identified 96 lateral phishing incidents out of the 110 test incidents (87.3%) across our ground truth dataset. Additionally, our detector discovered 49 incidents that, according to our ground truth, were not reported by a user as phishing. With respect to its cost, our detector generated 312 total false positives across the entire test dataset (a false positive rate of less than 0.00035%, assuming that emails not identified as an attack by our ground truth are benign). Across our test dataset, 82 out of the 92 organizations accumulated 10 or fewer false positives across the entire four month window, with 44 organizations encountering zero false positives across this timespan. In contrast, only three organizations had more than 40 total false positives across all four months (encountering 44, 66, and 83 false positives, respectively). Our detector achieves similar results if we evaluate on just the data from our 40 withheld test organizations, with a Detection Rate of 91.0%, a precision of 23.1%, and a false positive rate of 0.00038%.

Bias and Evasion: We base our evaluation numbers on the best ground truth we have: a combination of all user-reported

lateral phishing incidents (including some attacks outside our threat model), and all incidents discovered by any detection technique we tried (which includes two approaches orthogonal to our detector's strategy). This ground truth suffers from a bias towards phishing emails that contact many potential victims, and attacks that users can more easily recognize. Additionally, since our detector focuses on URL-based exploits, our dataset of attacks likely underestimates the prevalence of non-URL-based phishing attacks, which come solely from user-reported instances in our dataset. As a result, our work does not capture the full space of lateral phishing attacks, such as ones where the attacker targets a narrow, select set of victims with stealthily deceptive content. Rather, given that our detector identifies many known and unreported attacks, while generating only a few false positives per month, we provide a starting point for practical detection that future work can extend. Moreover, even if our detector does not capture every possible attack, the fact that the attacks in our dataset span dozens of different organizations, across a multi-month timeframe, allows us to illuminate a class of understudied attacks that many enterprises currently face.

Aside from obtaining more comprehensive ground truth, more work is needed to explore defenses against potential evasion attacks. Attackers could attempt to evade our detector by targeting different features we draw upon, such as the composition or number of recipients they target. Against many of these evasion attacks, future work could leverage additional features and data, such as the actions a user takes within an email account (e.g., reconnaissance actions, such as unusual searches, that indicate an attacker mining the account for targeted recipients to attack) or information from the user's account log-on (e.g., the detector proposed by Ho et al. used an account's login IP address [18] to detect lateral phishing). At the same time, future work should study which evasion attacks remain economically feasible for attackers to conduct. For example, an attacker could choose to only target a small number of users in the hopes of evading our detector; but even if this evasion succeeded, the conversion rate of fooling a recipient might be so low that the attack ultimately fails to compromise an economically viable number of victims. Indeed, as we explore in the following section (§ 6), the attackers captured in our dataset already engage in a range of different behaviors, including a few forms of sophisticated, manual effort to increase the success of their attacks.

6 Characterizing Lateral Phishing

In this section, we conduct an analysis of real-world lateral phishing using all known attacks across our entire dataset (both training and test). During the seven month timespan, a total of 33 organizations experienced lateral phishing attacks, with the majority of these compromised organizations experiencing multiple incidents. Examining the thematic message content and recipient targeting strategies of the attacks, our analysis suggests that most lateral phishers in our dataset do not actively mine a hijacked account's emails to craft personalized spearphishing attacks. Rather, these attackers operate in an opportunistic fashion and rely on commonplace phishing content. This finding suggests that the space of enterprise phishing has expanded beyond its historical association with sophisticated APTs and nation-state adversaries.

Scale and Success	
# distinct phishing emails	1,902
# incidents	180
# ATOs	154
# organizations w/ 1+ incident	33
# phishing recipients	101,276
% successful ATOs	11%
# employee recip (average) for compromise	542

Table 2: Summary of the scale and success of the lateral phishing attacks in our dataset (§ 6.1).

At the same time, these attacks nonetheless succeed, and a significant fraction of attackers do exhibit some signs of sophistication and attention to detail. As an estimate of the success of lateral phishing attacks, at least 11% of our dataset's attackers successfully compromise at least one other employee account. In terms of more refined tactics, 31% of lateral phishers invest some manual effort in evading detection or increasing their attack's success rate. Additionally, over 80% of the attacks in our dataset occur during the normal working hours of the hijacked account. Taken together, our results suggest that lateral phishing attacks pose a prevalent enterprise threat that still has room to grow in sophistication.

In addition to exploring attacks at the incident granularity (as done in § 5), this section also explores attacks at the granularity of a lateral phisher (hijacked account) when studying different attacker behaviors. As described in Section 2, industry practitioners often refer to such hijacked accounts as ATOs, and throughout this section, we use the terms *hijacked account*, *lateral phisher*, and *ATO* synonymously.

6.1 Scale and Success of Lateral Phishing

Scale: Our dataset contains 1,902 distinct lateral phishing emails sent by 154 hijacked accounts.[4] A total of 33 organizations in our dataset experience at least one lateral phishing incident: 23 of these organizations came from sampling the set of enterprises with known lateral phishing incidents (§ 3), while the remaining 10 came from the 69 organizations we sampled from the general population. Assuming our random sample reflects the broader population of enterprises, over 14% of organizations experience at least one lateral phishing incident within a 7 month timespan. Furthermore, based

[4]Distinct emails are defined by having a fully unique tuple of (sender, subject, timestamp, and recipients).

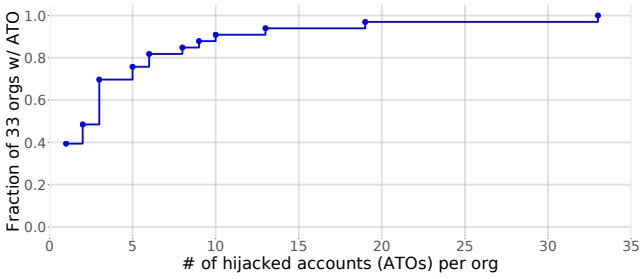


Figure 4: Fraction of organizations with x hijacked accounts that sent at least one lateral phishing email. 13 organizations had only 1 ATO; the remaining 20 saw lateral phishing from 2+ ATOs (§ 6.1).

on Figure 4, over 60% of the compromised organizations in our dataset experienced lateral phishing attacks from at least two hijacked employee accounts. Given that our set of attacks likely contains false negatives (thus underestimating the prevalence of attacks), these numbers illustrate that lateral phishing attacks are widespread across enterprise organizations.

Successful Attacks: Given our dataset, we do not definitively know whether an attack succeeded. However, we conservatively (under)estimate the success rate of lateral phishing using the methodology below. Based on this procedure, we estimate that at least 11% of lateral phishers successfully compromise at least one new enterprise account.

Let Alice and Bob represent two different ATOs at the same organization, where P_A and P_B represent one of Alice's and Bob's phishing emails respectively, and $Reply_B$ represents a reply from Bob to a lateral phishing email he received from Alice. Intuitively, our methodology concludes that Alice successfully compromised Bob if (1) Bob received a phishing email from Alice, (2) shortly after receiving Alice's phish, Bob then subsequently sent his own phish, and (3) we have strong evidence that the two employees' phishing emails are related (reflected in criteria 3 and 4 below).

Formally, we say that P_A succeeded in compromising Bob's account if *all* of the following conditions are true:

1. Bob was a recipient of P_A
2. After receiving P_A, Bob subsequently sent his own lateral phishing emails (P_B)
3. Either of the following two conditions are met:
   (a) P_B and P_A used similar phishing content: if the two attacks used identical subjects or if both of the phishing URLs they used belonged to the same fully-qualified domain
   (b) Bob sent a reply ($Reply_B$) to P_A, where his reply suggests he fell for Alice's attack and where Bob sent $Reply_B$ prior to his own attack (P_B)
4. Either of the following two conditions are met:
   (a) P_B was sent within two days after Bob received P_A
   (b) P_B and P_A used identical phishing messages or their phishing URLs' paths followed nearly identical structures (e.g., '`http://X.com/z/office365/index.html`' vs. '`http://Y.com/z/office365/index.html`')

Unpacking the final criteria (#4), in the first case (4.a), we settled on a two-day interarrival threshold based on prior literature [21, 22], which suggests that 50% of users respond to an email within 2 days and roughly 75% of users who click on a spam email do so within 2 days. Assuming that phishing follows similar time constants for how long it takes a recipient to take action, 2 days represented a conservative threshold to establish a link between P_A and P_B. At the same time, both prior works show there exists a long tail of users who take weeks to read and act on an email. The second part (4.b) attempts to address this long tail by raising the similarity requirements between Alice and Bob's attacks before concluding that former caused the latter. For successful attackers labeled by heuristic 4.b, the longest observed time gap between P_A and P_B is 17 days, which falls within a plausible timescale based on the aforementioned literature.

From this methodology, we conclude that 17 ATOs successfully compromised at least 23 future ATOs. While our procedure might erroneously identify cases where an attacker has concurrently compromised both Alice and Bob (rather than compromising Bob's account via Alice's), the first two criteria (requiring Bob to be a recent recipient of Alice's phishing email) help reduce this error. Our procedure likely underestimates the general success rate of lateral phishing attacks, since it does not identify successful attacks where the attacker does not subsequently use Bob's account to send phishing emails, nor does it account for false negatives in our dataset or attacks outside of our visibility (e.g., compromise of recipients at external organizations).

6.2 Recipient Targeting

In this section, we estimate the conversion rate of our dataset's lateral phishing attacks, and discuss four recipient targeting strategies that reflect the behavior of most attackers in our dataset.

Recipient Volume and Estimated Conversation Rate: Cumulatively, the lateral phishers in our dataset contact 101,276 unique recipients, where 41,740 belong to the same organization as the ATO. As shown in Figure 5, more than 94% of the attackers send their phishing emails to over 100 recipients; with respect to the general population of all lateral phishers, this percentage likely overestimates the prevalence of high "recipient-volume" attackers, since our detector draws on recipient-related features.

Targeting hundreds of people gives attackers a larger pool of potential victims, but it also incurs a risk that a recipient will detect and flag the attack either to their security team or

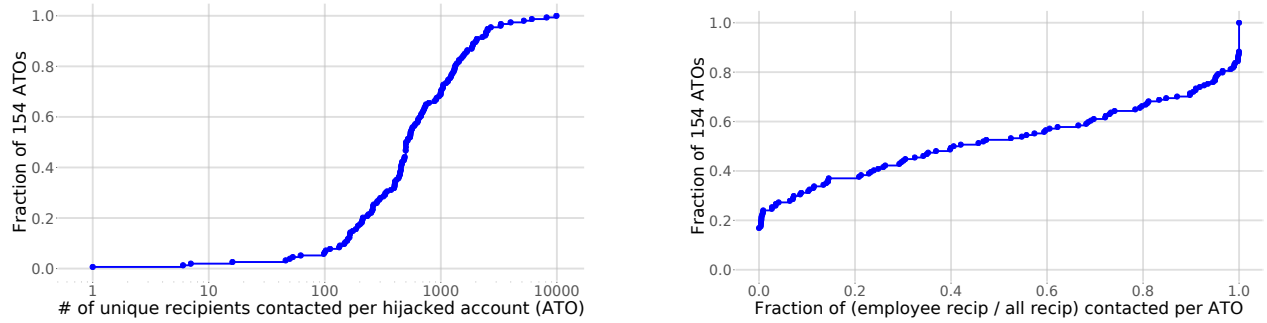


Figure 5: The left CDF shows the distribution of the total number of phishing recipients per ATO. The right CDF shows the fraction of ATOs where $x\%$ of their total recipient set consists of fellow employees.

their fellow recipients (e.g., via Reply-All). To isolate their victims and minimize the ability for fellow recipients to warn each other, we found that attackers frequently contact their recipients via a mass `BCC` or through many individual emails. Aside from this containment strategy, we also estimate that our dataset's lateral phishing attacks have a difficult time fooling an individual employee, and thus might require targeting many recipients to hijack a new account. Earlier in Section 6.1, we found that 17 ATOs successfully compromised 23 new accounts. Looking at the number of accounts they successfully hijacked divided by the number of fellow employees they targeted, the median conversation rate for our attackers was one newly hijacked account per 542 fellow employees; the attacker with the best conversation rate contacted an average of 26 employees per successful compromise. We caution that our method for determining whether an attack succeeded (§ 6.1) does not cover all cases, so our conversation rate might also underestimate the success of these attacks in practice. But if our estimated conversion rate accurately approximates the true rate, it would explain why these attackers contact so many recipients, despite the increased risk of detection.

Recipient Targeting Strategies: Anecdotally, we know that some lateral phishers select their set of victims by leveraging information in the hijacked account to target familiar users; for example, sending their attack to a subset of the account's "Contact Book". Unfortunately our dataset does not include information about any reconnaissance actions that an attacker performed to select their phishing recipients (e.g., explicitly searching through a user's contact book or recent recipients).

Instead, we empirically explore the recipient sets across our dataset's attackers to identify plausible strategies for how these attackers might have chosen their set of victims. Four recipient targeting strategies, summarized in Table 3 (explained below), reflect the behavior of all but six attackers in our dataset. To help assess whether a recipient and the ATO share a meaningful relationship, we compute each ATO's *recent contacts*: the set of all email addresses whom the ATO sent at least one email to in the 30 days preceding the ATO's phishing emails. While some attackers (28.6%) specifically target many of an account's recent contacts, the majority of lateral phishers appear more interested in either contacting many arbitrary recipients or sending phishing emails to a large fraction of the hijacked account's organization.

Recipient Targeting Strategy	# ATOs
Account-agnostic	63
Organization-wide	39
Lateral-organization	2
Targeted-recipient	44
Inconclusive	6

Table 3: Summary of recipient targeting strategies per ATO (§ 6.2).

Account-agnostic Attackers: Starting with the least-targeted behavior, 63 ATOs in our dataset sent their attacks to a wide range of recipients, most of whom do not appear closely related to the hijacked account. We call this group *Account-agnostic attackers*, and identify them using two heuristics.

First, we categorize an attacker as Account-agnostic if less than 1% of the recipients belong to the same organization as the ATO, and further exploration of their recipients does not reveal a strong connection with the account. Examining the right-hand graph in Figure 5, 37 ATOs target recipient sets where less than 1% of the recipients belong to the same organization as the ATO. To rule out the possibility that these attackers' recipients are nonetheless related to the account, we computed the fraction of recipients who appeared in each ATO's recent contacts; for all of the 37 possible Account-agnostic ATOs, less than 17% of their attack's total recipients appeared in their recent contacts. Among these 37 candidate Account-agnostic ATOs, 33 of them contact recipients at 10 or more organizations (unique recipient email domains), 2 of them exclusively target either Gmail or Hotmail accounts, and the remaining 2 ATOs are best described as Lateral-organization attackers (below).[5] Excluding the 2 Lateral-organization attackers, the 35 ATOs identified by this

[5] Our extended technical report provides the distribution of recipient domains contacted by all ATOs [17].

first criteria sent their attacks to predominantly external recipients, belonging to either many different organizations or exclusively to personal email hosting services (e.g., Gmail and Hotmail), and only a small percentage of these recipients appeared in the ATO's recent contacts; as such, we label these 35 attackers as Account-agnostic.

Second, we expand our search for Account-agnostic attackers by searching for attackers where less than 50% of the ATO's total recipients also belong to the ATO's organization, and where the ATO contacts recipients at many different organizations; specifically, where the ATO's phishing recipients belonged to over twice as many unique domains as all of the email addresses in ATO's recent contacts. This search identified 63 ATOs. To filter out attackers in this set who may have drawn on the hijacked account's recent contacts, we exclude any ATO where over 17% of their attack's total recipients also appeared in the ATO's recent contacts (17% was the maximum percentage among ATOs from the first Account-agnostic heuristic). After applying this last condition, our second heuristic identifies 54 Account-agnostic attackers.

Combining and deduplicating the ATOs from both criteria results in a total of 63 Account-agnostic attackers (40.9%): lateral phishers who predominantly target recipients without close relationships to the hijacked account or its organization.

Lateral-organization Attackers: During our exploration of potential Account-agnostic ATOs, we uncovered 2 attackers whom we label under a different category: *Lateral-organization attackers*. In both these cases, less than 1% of the attacker's recipients belonged to the same organization as the ATO, but each attacker's recipients did belong to organizations within the same industry as the ATO's organization. This thematic characteristic among the recipients suggests a deliberate strategy to spread across organizations within the targeted industries, so accordingly, we categorize them as Lateral-organization attackers.

Organization-wide Attackers: Office 365 provides a "Groups" feature that lists the different groups that an account belongs to [30]. For some enterprises, this feature enumerates most, if not all, employees at the organization. Thus, lateral phishers who wish to cast a wide phishing net might adopt a simple strategy of sending their attack to everyone at the organization. We call these ATOs *Organization-wide attackers* and identify them through two ways.

First, we search for any attackers where at least half of their phishing recipients belong to the ATO's organization, and where at least 50% of the organization's employees received the phishing email (i.e., the majority of a phisher's victims were employees and the attacker targeted a majority of the enterprise); this search yielded a total of 16 ATOs. We estimate the list of an organization's employees by building a set of all employee email addresses who sent or received email from

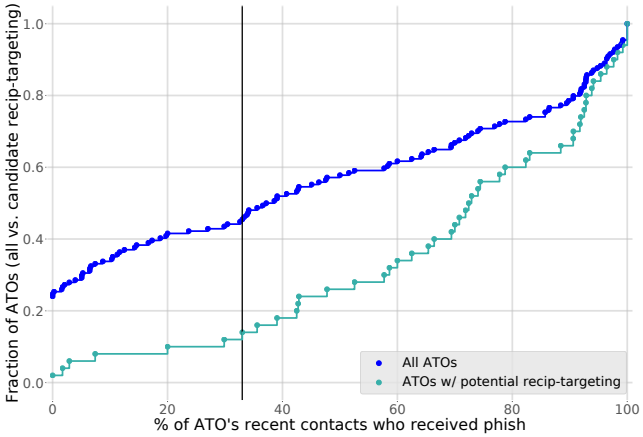


Figure 6: CDF: the x-axis displays what % of the ATO's recent contacts received a lateral phishing email (§ 6.2). The bottom teal graph filters the ATOs to *exclude* any ATO identified as Account-agnostic, Lateral-organization, and Organization-wide attackers; at the vertical black line, 88% of these filtered ATOs send phishing emails to at least $x = 33\%$ of addresses from their recent contacts.

anyone during the entire month of the phishing incident.[6] For all of these 16 ATOs, less than 11% of the recipients they target also appear in their recent contacts. Coupled with the fact that each of these ATOs contacts over 1,300 recipients, their behavior suggests that their initial goal focuses on phishing as many of the enterprise's recipients as possible, rather than targeting users particularly close to the hijacked account. Accordingly, we categorize them as Organization-wide attackers.

Our second heuristic looks for attackers whose recipient set consists nearly entirely of fellow employees, but where the majority of the organization does not necessarily receive a phishing email. Revisiting Figure 5, 36 candidate Organization-wide ATOs sent over 95% of their phishing emails to fellow employee recipients. However, we again need to exclude and account for ATOs who leverage their hijacked account's recent contacts. From the first Organization-wide heuristic discussed previously, we saw that less than 11% of the recipients of that heuristic's Organization-wide attackers came from the ATO's recent contacts. Using this value as a final threshold for this second candidate set of Organization-wide attackers, we identify 29 Organization-wide attackers where over 95% of their recipients belong to the ATO's organization but less than 11% of the recipients came from the ATO's recent contacts; a combination that suggests the attacker seeks primarily to compromise other employees, but who do not necessarily have a personal connection with the hijacked account.

Aggregating and deduplicating the two sets of lateral phishers from above produces a total of 39 Organization-wide attackers (25.3%), who take advantage of the information in a hijacked account to target many fellow employees.

[6] This collection likely overestimates the actual set of employees because of service addresses, mailing list aliases, and personnel churn.

Targeted-recipient Attackers: For the remaining, uncategorized 50 ATOs, we cannot conclusively determine the attackers' recipient targeting strategies because our dataset does not provide us with the full set of information and actions available to the attacker. Nonetheless, Figure 6 presents some evidence that 44 of these remaining attackers do draw upon the hijacked account's prior relationships. Specifically, 44 attackers sent their attacks to at least 33% of the addresses in the ATO's recent contacts.[7] Since these ATOs sent attacks to at least 1 out of every 3 of the ATO's recently contacted recipients, these attackers appear interested in targeting a substantial fraction of users with known ties to the hijacked account. As such, we label these 44 ATOs as Targeted-recipient attackers.

6.3 Message Content: Tailoring and Themes

Since lateral phishers control a legitimate employee account, these attackers could easily mine recent emails to craft personalized spearphishing messages. To understand how much attackers do leverage their privileged access in their phishing attacks, this section characterizes the level of tailoring we see among lateral phishing messages. Overall, only 7% of our dataset's incidents contain targeted content within their messages. Across the phishing emails that used non-targeted content, the attackers in our dataset relied on two predominant narratives (deceptive pretexts) to lure their victim into performing a malicious action. The combination of these two results suggests that, for the present moment, these attackers (across dozens of organizations) see more value in opportunistically phishing as many recipients as possible, rather than investing time to mine the hijacked accounts for personalized spearphishing fodder.

Content Tailoring: When analyzing the phishing messages in our dataset, we found that two dimensions aptly characterized the different levels of content tailoring and customization. The first dimension, "Topic tailoring", describes how personalized the topic or main idea of the email is to the victim or organization. The second dimension, "Name tailoring", describes how specifically the attacker addresses the victim (e.g., "Dear user" vs. "Dear Bob"). For each of these two dimensions, we enumerate three different levels of tailoring and provide an anonymized message snippet below; we use Bob to refer to one of the attack's recipients and FooCorp for the company that Bob works at.

1. Topic tailoring: the uniqueness and relevancy of the message's topic to the victim or organization:
   (a) Generic phishing topic: an unspecific message that could be sent to any user ("You have a new shared document available.")
   (b) Broadly enterprise related topic: a message that appears targeted to enterprise environments, but one that would also make sense if the attacker used it at many other organizations ("Updated work schedule. Please distribute to your teams.")
   (c) Targeted topic: a message where the topic clearly relies on specific details about the recipient or organization ("Please see the attached announcement about FooCorp's 25th year anniversary.", where FooCorp has existed for exactly 25 years.)
2. Name tailoring: whether the phishing message specifically uses the recipient or organization's name:
   (a) Non-personalized naming: the attack does not mention the organization or recipient by name ("Dear user, we have detected an error in your mailbox settings...")
   (b) Organization specifically named: the attack mentions just the organization, but not the recipient ("New secure email message from FooCorp...")
   (c) Recipient specifically named: the attack specifically uses the victim's name in the email ("Bob, please review the attached purchase order...")

	Generic	Enterprise	Targeted
No naming	90	35	9
Organization named	23	16	4
Recipient named	0	3	0

Table 4: Distribution of the number of *incidents* per message tailoring category (§ 6.3). The columns correspond to how unique and specific the message's topic pertains to the victim or organization. The rows correspond to whether the phishing email explicitly names the recipient or organization.

Taken together, this taxonomy divides phishing content into nine different classes of tailoring; Table 4 shows how many of our dataset's 180 incidents fall into each category. From this categorization, two interesting observations emerge. First, only 3 incidents (1.7%) actually address their recipients by name. Since most ATOs (94%) in our dataset email at least 100 recipients, attackers would need to leverage some form of automation to both send hundreds of individual emails and customize the naming in each one. Based on our results, it appears these attackers did not view that as a worthwhile investment. For example, they might fear that sending many individual emails might trigger an anti-spam or anti-phishing mechanism, which we observed in the case of one ATO who attempted to send hundreds of individual emails. Second,

[7] When examining and applying thresholds for the Account-agnostic and Organization-wide Attackers, we used a slightly different fraction: how many of the ATO's phishing recipients also appeared in their recent contacts? Here, we seek to capture attackers who make a specific effort to target a considerable number of familiar recipients. Accordingly, we look at the fraction of the ATO's recent contacts that received phishing emails, where the denominator reflects the number of users in the ATO's recent contacts, rather than the ATO's total number of phishing recipients.

Word	# Incidents	Word	# Incidents
document	89	sent	44
view	76	review	43
attach	56	share	37
click	55	account	36
sign	50	access	34

Table 5: Top 10 most common words across all 180 lateral phishing incidents.

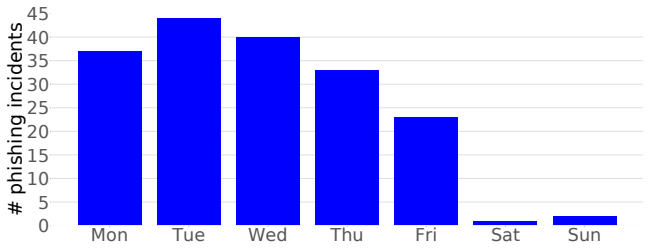


Figure 7: Number of lateral phishing incidents per day of week.

looking at the last column of Table 4, only 13 incidents (7%) use targeted content in their messages. The overwhelming majority (92.7%) of incidents opt for more generic messages that an attacker could deploy at a large number of organizations with minimal changes (e.g., by only changing the name of the victim organization).

While our attack dataset captures a limited view of all lateral phishing attacks, it nonetheless reflects all known lateral phishing incidents across 33 organizations over a 7-month timeframe. Thus, despite the data's limitations, our results show that a substantial fraction of lateral phishers do not fully draw upon their compromised account's resources (i.e., historical emails) to craft personalized spearphishing messages. This finding suggests these attackers act more like an opportunistic cybercriminal, rather than an indomitable APT or nation-state. However, given the arms-race and evolutionary nature of security, these lateral phishers could in the future increase the sophistication and potency of their attacks by drawing upon the account's prior emails to craft more targeted content.

Thematic Content (Lures): When labeling each phishing incident with a level of tailoring, we noticed that the phishing messages in our dataset overwhelmingly relied on one of two deceptive pretexts (lures): (1) an alarming message that asserts some problem with the recipient's account (and urges them to follow a link to remediate the issue); and (2) a message that notifies the recipient of a new / updated / shared document. For the latter 'document' lure, the nature and specificity of the document varied with the level of content tailoring. For example, whereas an attack with generic topic tailoring will just mention a vague document, attacks that use enterprise-related tailoring will switch the terminology to an invoice, purchase order, or some other generic but work-related document.

To characterize this behavior further, we computed the most frequently occurring words across our dataset's phishing messages. First, we selected one phishing email per incident, to prevent incidents with many identical emails from biasing (inflating) the popularity of their lures. Next, we normalized the text of each email: we removed auto-generated text (e.g., user signatures), lowercased all words, removed punctuation, and discarded all non-common English words; all of these can be done with open source libraries such as Talon [27] and NLTK [5]. Finally, we built a set of all words that occurred in any phishing email across our incidents and counted how many incidents each word appeared in.

Interestingly, our dataset's phishing messages draw on a relatively small pool of words: there are just 444 distinct, common English words across the texts of every phishing message in our dataset (i.e., every phishing email's text consists of an arrangement from this set of 444 words). In contrast, a random sample of 1,000 emails from our dataset contained a total of 2,516 distinct words, and only 176 of these emails consisted entirely of words from the phishing term set.

Beyond this small set of total words across lateral phishing emails, all but one incident contained at least one of the top 20 words, illustrating the reliance on the two major lures we identified. Our extended technical report shows the occurrence distribution of each word [17]. Focusing on just the top ten words and the number incidents that use them (Table 5), the dominance of these two thematic lures becomes apparent. Words indicative of the "shared document" lure, such as 'document', 'view', 'attach', and 'review', each occur in over 23% of incidents, with the most popular (document) occurring in nearly half of all incidents. Similarly, we also see many words from the account-related lure in the top ten: 'access', 'sign' (from 'sign on'), and 'account'.

Overall, while our dataset contains several instances of targeted phishing messages, the majority of the lateral phishing emails we observe rely on more mundane lures that an attacker can reuse across multiple organizations with little effort. The fact that we see this behavior recur across dozens of different organizations suggests either the emergence of a new, yet accessible, form of enterprise phishing, or an evolution in the way "ordinary" cybercriminals execute phishing attacks (moving from external accounts that use clever spoofing to compromised, yet legitimate accounts).

6.4 Temporal Aspects of Lateral Phishing

Because attackers might not live or operate in the same geographic region as the hijacked account, prior work has suggested using features that capture unusual timing properties inherent in phishing emails [11, 15, 41]. Contrary to this intuition, in our dataset most lateral phishing attacks occur at "normal" times of the day and week. First, for 98% of lateral phishing incidents, the attacker sent the phishing email dur-

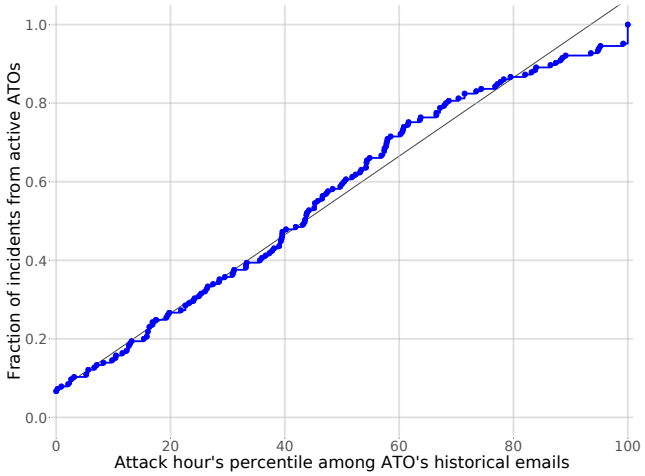


Figure 8: CDF of the fraction of incidents from active ATOs where the time (hour) of day fell within the x'th percentile of the hours at which the ATO's benign emails in the preceding 30 days were sent. Active ATOs are hijacked accounts that sent at least 1 non-phishing email within the 30 days preceding their lateral phishing email.

ing a weekday. Additionally, the majority of attackers in our dataset send their phishing emails during the true account's normal working hours.

Day of the Week: From Figure 7, all but three lateral phishing incidents occurred during a work day (Monday–Friday). This pattern suggests that attackers send their phishing emails on the same days when employees typically send their benign emails, and that the day of the week will provide an ineffective or weak detection signal. Moreover, 67% of incidents occur in the first half of the week (Mon–Wed), indicating that the lateral phishers in our dataset do not follow the folklore strategy where attackers favor launching their attacks on Friday (hoping to capitalize on reduced security team operations over the coming weekend) [37].

Time (Hour) of Day: In addition to operating during the usual work week, most attackers tend to send their lateral phishing emails during the typical working hours of their hijacked accounts. To assess the (ab)normality of an attack's sent-time, for each ATO, we gathered all of the emails that the account sent in the 30 days prior to their first lateral phishing email. We then mapped the sent-time of each of these historical (and presumably benign) emails to the hour-of-day on a 24 hour scale, thus forming a distribution of the typical hour-of-day in which each hijacked account usually sent their emails. Finally, for each lateral phishing incident, we computed the percentile for the phishing email's hour-of-day relative to the hour-of-day distribution for the ATO's historical emails. For example, phishing incidents with a percentile of 0 or 100 were sent at an earlier or later hour-of-day than any email that the true account's owner sent in the preceding 30 days.

Across all lateral phishing incidents sent by an active ATO, Figure 8 shows what hour-of-day percentile the phishing incident's first email occurred at, relative to the hijacked account's historical emails. Out of the 180 incidents, 15 incidents were sent by an "inactive" (quiescent) ATO that sent zero emails across all 30 days preceding their lateral phishing emails; Figure 8 excludes these incidents. Of the remaining 165 incidents sent by an active ATO, 18 incidents fall completely outside of the hijacked account's historical operating hours, which suggests that a feature looking for emails sent at atypical times for a user could help detect these attacks. However, for the remaining 147 incidents, the phishing emails' hour-of-day evenly cover the full percentile range. As shown in Figure 8, the percentile distribution of phishing hours closely resembles the CDF of a uniformly random distribution (a straight $y = x$ line); i.e., the phishing email's hour-of-day appears to be randomly drawn from the true account's historical hour-of-day distribution. This result indicates that for the majority of incidents in our dataset (147 out of 180), the time of day when the ATO sent the attack will not provide a significant signal, since their sent-times mirror the timing distribution of the true user's historical email activity.

Thus, based on the attacks in our dataset, we find that two weak timing-related features exist: searching for quiescent accounts that suddenly begin to send suspicious emails (15 incidents), and searching for suspicious emails sent completely outside of an account's historically active time window (18 incidents). Beyond these two features and the small fraction of phishing attacks they reflect, neither the day of the week nor the time of day provide significant signals for detection.

6.5 Attacker Sophistication

Since most of our dataset's lateral phishers do not mine the hijacked account's mailbox to craft targeted messages, one might naturally conclude that these attackers are lazy or unsophisticated. However, in this subsection, we identify two kinds of sophisticated behavior that required some investment of additional time and manual effort: attackers who continually engage with their attack's recipients in an effort to increase the attack's success rate, and attackers who actively "clean up" traces of their phishing activity in an attempt to hide their presence from the account's legitimate owner. In contrast to the small number of attackers who invested time in crafting tailored phishing messages to a personalized set of recipients, nearly one-third (31%) of attackers engage in at least one of these two sophisticated behaviors.

Interaction with potential victims: Upon receiving a phishing message, some recipients naturally question the email's validity and send a reply asking for more information or assurances. While a lazy attacker might ignore these recipients' replies, 27 ATOs across 15 organizations actively engaged with these potential victims by sending follow-up messages assuring the victim of the phishing email's legitimacy. For example, at one organization, an attacker consistently sent brief follow-up messages such as "Yes I sent it to you" or "Yes,

have you checked it yet?". In other cases, attackers replied with significantly more elaborate ruses: e.g., "Hi [Bob], its a document about [X]. It's safe to open. You can view it by logging in with your email address and password."

To find instances where a phisher actively followed-up with their attack's potential victims, we gathered all of the messages in every lateral phishing email thread and checked to see if the attacker ever received and responded to a recipient's reply (inquiry).[8] In total, we found that 107 ATOs received at least one reply from a recipient. Of these reply-receiving attackers, 27 ATOs (25%) sent a deceptive follow-up response to one or more of their recipients' inquiries.

Stealthiness: Separate from interacting with their potential victims, attackers might expend manual effort to hide their presence from the account's true owner by removing any traces of their phishing emails, particularly since lateral phishers appear to operate during the hijacked account's normal working hours (§ 6.4). To estimate the number of these ATOs, we searched for whether any of the following emails ended up in the hijacked account's Trash folder, and were deleted within 30 seconds of being sent or received: any phishing emails, replies to phishing emails, or follow-up emails sent by the attacker. The 30 second threshold distinguishes stealthy behavior from deletion resulting from remediation of the compromised account. In total, 30 attackers across 16 organizations engage in this kind of evasive clean-up behavior.

Of the 27 ATOs who interactively responded to inquiries about their attack, only 9 also exhibited this stealthy clean-up behavior. Thus, counting the number of attackers across both sets, 48 ATOs engaged in at least one of these behaviors.

The sizeable fraction of attackers who engage in a sophisticated behavior creates a more complex picture of the attacks in our dataset. Given that these attackers do invest dedicated and (often) manual effort in enhancing the success of their attacks, why do so many of them (over 90% in our dataset) use non-targeted phishing content and target dozens to hundreds of recipients? One plausible reason for this generic behavior is that the simple methods they currently use work well enough under their economic model: investing additional time to develop more tailored phishing emails just does not provide enough economic value. Another reason might be that growth of lateral phishing attacks reflects an evolution in the space of phishing, where previously "simple" external phishers have moved to sending their attacks via lateral phishing because attacks from (spoofed) external accounts have become too difficult, due to user awareness and/or better technical mitigations against external phishing. Ultimately, based on our work's dataset, we cannot soundly answer why so many lateral phishers employ simple attacks, and leave it as an interesting question for future work to explore.

[8] Office 365 includes a *ConversationID* field, and all emails in the same thread (the original email and all replies) get assigned the same ConversationID value.

7 Summary

In this work we presented the first large-scale characterization of lateral phishing attacks across more than 100 million employee-sent emails from 92 enterprise organizations. We also developed and evaluated a new detector that found many known lateral phishing attacks, as well as dozens of unreported attacks, while generating a low volume of false positives. Through a detailed analysis of the attacks in our dataset, we uncovered a number of important findings that inform our mental models of the threats enterprises face, and illuminate directions for future defenses. Our work showed that 14% of our randomly sampled organizations, ranging from small to large, experienced lateral phishing attacks within a seven-month time period, and that attackers succeeded in compromising new accounts at least 11% of the time. We uncovered and quantified several thematic recipient targeting strategies and deceptive content narratives; while some attackers engage in targeted attacks, most follow strategies that employ non-personalized phishing attacks that can be readily used across different organizations. Despite this apparent lack of sophistication in tailoring and targeting their attacks, 31% of our dataset's lateral phishers engaged in some form of sophisticated behavior designed to increase their success rate or mask their presence from the hijacked account's true owner. Additionally, over 80% of attacks occurred during a typical working day and hour, relative to the legitimate account's historical emailing behavior; this suggests that these attackers either reside within a similar timezone as the accounts they hijack or make a concerted effort to operate during their victim's normal hours. Ultimately, our work provides the first large-scale insights into an emerging, widespread form of enterprise phishing attacks, and illuminates techniques and future ideas for defending against this potent threat.

Acknowledgements

We thank Itay Bleier, the anonymous reviewers, and our shepherd Gianluca Stringhini for their valuable feedback. This work was supported in part by the Hewlett Foundation through the Center for Long-Term Cybersecurity, NSF grants CNS-1237265 and CNS-1705050, an NSF GRFP Fellowship, the Irwin Mark and Joan Klein Jacobs Chair in Information and Computer Science (UCSD), by generous gifts from Google and Facebook, a Facebook Fellowship, and operational support from the UCSD Center for Networked Systems.

References

[1] Saeed Abu-Nimeh, Dario Nappa, Xinlei Wang, and Suku Nair. A Comparison of Machine Learning Techniques for Phishing Detection. In *Proc. of 2nd ACM eCrime*, 2007.

[2] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Are Your Training Datasets Yet Relevant? In *Proc. of 7th Springer ESSoS*, 2015.

[3] Andre Bergholz, Jeong Ho Chang, Gerhard Paaß, Frank Reichartz, and Siehyun Strobel. Improved Phishing Detection using Model-Based Features. In *Proc. of 5th CEAS*, 2008.

[4] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *JMLR*, 13(Feb), 2012.

[5] Steven Bird, Edward Loper, and Ewan Klein. Natural Language Toolkit. https://www.nltk.org/, 2019.

[6] Elie Bursztein, Borbala Benko, Daniel Margolis, Tadek Pietraszek, Andy Archer, Allan Aquino, Andreas Pitsillidis, and Stefan Savage. Handcrafted Fraud and Extortion: Manual Account Hijacking in the Wild. In *Proc. of 14th ACM IMC*, 2014.

[7] Asaf Cidon. Threat Spotlight: Office 365 Account Takeover — the New "Insider Threat". https://blog.barracuda.com/2017/08/30/threat-spotlight-office-365-account-compromise-the-new-insider-threat/, Aug 2017.

[8] Asaf Cidon, Lior Gavish, Itay Bleier, Nadia Korshun, Marco Schweighauser, and Alexey Tsitkin. High Precision Detection of Business Email Compromise. In *Proc. of 28th Usenix Security*, 2019.

[9] DomainKeys Identified Mail. https://en.wikipedia.org/wiki/DomainKeys_Identified_Mail. Accessed: 2018-11-01.

[10] Sevtap Duman, Kubra Kalkan-Cakmakci, Manuel Egele, William Robertson, and Engin Kirda. EmailProfiler: Spearphishing Filtering with Header and Stylometric Features of Emails. In *Proc. of 40th IEEE COMPSAC*, 2016.

[11] Manuel Egele, Gianluca Stringhini, Christopher Kruegel, and Giovanni Vigna. COMPA: Detecting Compromised Accounts on Social Networks. In *Proc. of 20th ISOC NDSS*, 2013.

[12] FBI. BUSINESS E-MAIL COMPROMISE THE 12 BILLION DOLLAR SCAM, Jul 2018. https://www.ic3.gov/media/2018/180712.aspx.

[13] Ian Fette, Norman Sadeh, and Anthony Tomasic. Learning to Detect Phishing Emails. In *Proc. of 16th ACM WWW*, 2007.

[14] Sujata Garera, Niels Provos, Monica Chew, and Aviel D Rubin. A Framework for Detection and Measurement of Phishing Attacks. In *Proc. of 5th ACM WORM*, 2007.

[15] Hugo Gascon, Steffen Ullrich, Benjamin Stritter, and Konrad Rieck. Reading Between the Lines: Content-Agnostic Detection of Spear-Phishing Emails. In *Proc. of 21st Springer RAID*, 2018.

[16] Google. Classification: ROC and AUC. https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc, 2019.

[17] Grant Ho, Asaf Cidon, Lior Gavish, Marco Schweighauser, Vern Paxson, Stefan Savage, Geoffrey M. Voelker, and David Wagner. Detecting and Characterizing Lateral Phishing at Scale (Extended Report). In *arxiv*, 2019.

[18] Grant Ho, Aashish Sharma, Mobin Javed, Vern Paxson, and David Wagner. Detecting Credential Spearphishing Attacks in Enterprise Settings. In *Proc. of 26th USENIX Security*, 2017.

[19] Xuan Hu, Banghuai Li, Yang Zhang, Changling Zhou, and Hao Ma. Detecting Compromised Email Accounts from the Perspective of Graph Topology. In *Proc. of 11th ACM CFI*, 2016.

[20] Dan Hubbard. Cisco Umbrella 1 Million. https://umbrella.cisco.com/blog/2016/12/14/cisco-umbrella-1-million/, Dec 2016.

[21] Chris Kanich, Christian Kreibich, Kirill Levchenko, Brandon Enright, Geoffrey M Voelker, Vern Paxson, and Stefan Savage. Spamalytics: An Empirical Analysis of Spam Marketing Conversion. In *Proc. of 15th ACM CCS*, 2008.

[22] Thomas Karagiannis and Milan Vojnovic. Email information flow in large-scale enterprises. Technical report, Microsoft Research, 2008.

[23] Mahmoud Khonji, Youssef Iraqi, and Andrew Jones. Mitigation of spear phishing attacks: A content-based authorship identification framework. In *Proc. of 6th IEEE ICITST*, 2011.

[24] FT Labs. A sobering day. https://labs.ft.com/2013/05/a-sobering-day/?mhq5j=e6, May 2013.

[25] Stevens Le Blond, Cédric Gilbert, Utkarsh Upadhyay, Manuel Gomez Rodriguez, and David Choffnes. A Broad View of the Ecosystem of Socially Engineered Exploit Documents. In *Proc. of 24th ISOC NDSS*, 2017.

[26] Stevens Le Blond, Adina Uritesc, Cédric Gilbert, Zheng Leong Chua, Prateek Saxena, and Engin Kirda. A Look at Targeted Attacks Through the Lense of an NGO. In *Proc. of 23rd USENIX Security*, 2014.

[27] Mailgun Team. Talon. https://github.com/mailgun/talon, 2018.

[28] William R Marczak, John Scott-Railton, Morgan Marquis-Boire, and Vern Paxson. When Governments Hack Opponents: A Look at Actors and Technology. In *Proc. of 23rd USENIX Security*, 2014.

[29] Microsoft Graph: message resource type. https://developer.microsoft.com/en-us/graph/docs/api-reference/v1.0/resources/message. Accessed: 2018-11-01.

[30] Microsoft. People overview - Outlook Web App. https://support.office.com/en-us/article/people-overview-outlook-web-app-5fe173cf-e620-4f62-9bf6-da5041f651bf. Accessed: 2018-11-01.

[31] Brad Miller, Alex Kantchelian, Michael Carl Tschantz, Sadia Afroz, Rekha Bachwani, Riyaz Faizullabhoy, Ling Huang, Vaishaal Shankar, Tony Wu, George Yiu, et al. Reviewer Integration and Performance Measurement for Malware Detection. In *Proc. of 13th Springer DIMVA*, 2016.

[32] Jeremiah Onaolapo, Enrico Mariconti, and Gianluca Stringhini. What Happens After You Are Pwnd: Understanding the Use of Leaked Webmail Credentials in the Wild. In *Proc. of 16th ACM IMC*, 2016.

[33] J. Palme. Common Internet Message Headers. https://tools.ietf.org/html/rfc2076.

[34] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. Tesseract: Eliminating experimental bias in malware classification across space and time. In *Proc. of 28th Usenix Security*, 2019.

[35] Kevin Poulsen. Google disrupts chinese spear-phishing attack on senior u.s. officials. https://www.wired.com/2011/06/gmail-hack/, Jul 2011.

[36] Steve Ragan. Office 365 phishing attacks create a sustained insider nightmare for it. https://www.csoonline.com/article/3225469/office-365-phishing-attacks-create-a-sustained-insider-nightmare-for-it.html, Sep 2017.

[37] Fahmida Y. Rashid. Don't like Mondays? Neither do attackers. https://www.csoonline.com/article/3199997/don-t-like-mondays-neither-do-attackers.html, Aug 2017.

[38] Retraining models on new data. https://docs.aws.amazon.com/machine-learning/latest/dg/retraining-models-on-new-data.html, 2019.

[39] Jeff John Roberts. Homeland Security Chief Cites Phishing as Top Hacking Threat. http://fortune.com/2016/11/20/jeh-johnson-phishing/, Nov 2016.

[40] Apache Spark. PySpark DecisionTreeClassificationModel v2.1.0. http://spark.apache.org/docs/2.1.0/api/python/pyspark.ml.html?highlight=featureimportance#pyspark.ml.classification.DecisionTreeClassificationModel.featureImportances.

[41] Gianluca Stringhini and Olivier Thonnard. That Ain't You: Blocking Spearphishing Through Behavioral Modelling. In *Proc. of 12th Springer DIMVA*, 2015.

[42] Kurt Thomas, Frank Li, Chris Grier, and Vern Paxson. Consequences of Connectivity: Characterizing Account Hijacking on Twitter. In *Proc. of 21st ACM CCS*, 2014.

[43] Lisa Vaas. How hackers broke into John Podesta, DNC Gmail accounts. https://nakedsecurity.sophos.com/2016/10/25/how-hackers-broke-into-john-podesta-dnc-gmail-accounts/, Oct 2016.

[44] Colin Whittaker, Brian Ryner, and Marria Nazif. Large-Scale Automatic Classification of Phishing Pages. In *Proc. of 17th ISOC NDSS*, 2010.

[45] Wikipedia. Random forest. https://en.wikipedia.org/wiki/Random_forest, 2019.

[46] Kim Zetter. Researchers uncover rsa phishing attack, hiding in plain sight. https://www.wired.com/2011/08/how-rsa-got-hacked/, Aug 2011.

[47] Mengchen Zhao, Bo An, and Christopher Kiekintveld. Optimizing Personalized Email Filtering Thresholds to Mitigate Sequential Spear Phishing Attacks. In *Proc. of 13th AAAI*, 2016.

A Detector Implementation and Evaluation Details

A.1 Labeling Phishing Emails

Labeling an email as phishing or benign: When manually labeling an email, we started by examining five pieces of information whether the email was a reported phishing incident, the message content, the suspicious URL flagged and if its domain made sense in context, the email's recipients, and the sender. With the exception of a few incidents, we could easily identify a phishing email from the above steps. For example: an email about a "shared Office 365 document" sent to hundreds of unrelated recipients and whose document link pointed to a bit.ly shortened [non-Microsoft] domain; or an

email describing an "account security problem" sent by a non-IT employee, where the "account reset" URL pointed to an unrelated domain. For the more difficult cases, we analyzed all replies and forwards in the email chain, and labeled the email as phishing if it either received multiple replies / forwards that expressed alarm or suspicious, or if the hijacked account eventually sent a reply saying that they did not send the phishing email. Finally, as described in Section 3.3 we visited the non-side-effect, suspicious URLs from a sample of the labeled phishing emails; All of the URLs we visited led to either an interstitial warning page (e.g.,, Google SafeBrowsing), or a spoofed log-on page. For the emails flagged by our detector, but which appeared benign based on examining all the above information, we conservatively labeled them as false positives. In many cases, false positives were readily apparent; e.g., emails where the "suspicious URL" flagged by our detector occurred in the sender's signature and linked to their personal website.

Training exercises vs. actual phishing emails: In addition to distinguishing between a false positive and an attack, we checked to ensure that our lateral phishing incidents represented actual attacks, and not training exercises. First, based on the lateral phishing emails' headers, we verified that all of the sending accounts were legitimate enterprise accounts. Second, all but five of the attack accounts sent one or more unrelated-to-phishing emails in the preceding month. These two points gave us confidence that the phishing emails came from existing, legitimate accounts, and thus represented actual attacks; i.e., training exercises will not hijack an existing account, due to the potential reputational harm this could incur (and enterprise security teams we've previously engaged with do not do this). Furthermore, none of our dataset's lateral phishing incidents are training exercises known to Barracuda, and none of the lateral phishing URLs used domains of known security companies.

A.2 Model Tuning and Hyperparameters

Most machine learning models, including Random Forest, require the user to set various (hyper)parameters that govern the model's training process. To determine the optimal set of hyperparameters for our classifier, we followed machine learning best practices by conducting a three-fold cross-validation grid search over all combinations of the hyperparameters listed below [4].

1. Number of trees: 50–500, in steps of 50 (i.e., 50, 100, 150, . . . , 450, 500)
2. Maximum tree depth: 10–100, in steps of 10
3. Minimum leaf size: 1, 2, 4, 8
4. Downsampling ratio of (benign / attack) emails: 10, 50, 100, 200

Because our training dataset contained only a few dozen incidents, we used three folds to ensure that each fold in the cross-validation contained several attack instances. Our experiments used a Random Forest model with 64 trees, a maximum depth of 8, a minimum leaf size of 4 elements, and a downsampling of 200 benign emails per 1 attack email, since this configuration produced the the highest AUC score [16]. But we note that many of the hyperparameter combinations yielded similar results.

High Precision Detection of Business Email Compromise

Asaf Cidon[1,2] and Lior Gavish, Itay Bleier, Nadia Korshun, Marco Schweighauser and Alexey Tsitkin[1]

[1]Barracuda Networks, [2]Columbia University

Abstract

Business email compromise (BEC) and employee impersonation have become one of the most costly cyber-security threats, causing over $12 billion in reported losses. Impersonation emails take several forms: for example, some ask for a wire transfer to the attacker's account, while others lead the recipient to following a link, which compromises their credentials. Email security systems are not effective in detecting these attacks, because the attacks do not contain a clearly malicious payload, and are personalized to the recipient.

We present BEC-Guard, a detector used at Barracuda Networks that prevents business email compromise attacks in real-time using supervised learning. BEC-Guard has been in production since July 2017, and is part of the Barracuda Sentinel email security product. BEC-Guard detects attacks by relying on statistics about the historical email patterns that can be accessed via cloud email provider APIs. The two main challenges when designing BEC-Guard are the need to label millions of emails to train its classifiers, and to properly train the classifiers when the occurrence of employee impersonation emails is very rare, which can bias the classification. Our key insight is to split the classification problem into two parts, one analyzing the header of the email, and the second applying natural language processing to detect phrases associated with BEC or suspicious links in the email body. BEC-Guard utilizes the public APIs of cloud email providers both to automatically learn the historical communication patterns of each organization, and to quarantine emails in real-time. We evaluated BEC-Guard on a commercial dataset containing more than 4,000 attacks, and show it achieves a precision of 98.2% and a false positive rate of less than one in five million emails.

1 Introduction

In recent years, email-borne employee impersonation, termed by the FBI "Business Email Compromise" (BEC), has become a major security threat. According to the FBI, US organizations have lost $2.7 billion in 2018 and cumulatively $12 billion since 2013 [13]. Numerous well-known enterprises have fallen prey to such attacks, including Facebook, Google [41], and Ubiquiti [44]. Studies have shown that BEC is the cause of much higher direct financial loss than other common cyberattacks, such as ransomware [11, 13]. BEC attacks have also ensnared operators of critical government infrastructure [39]. Even consumers have become the targets of employee impersonation. For example, attackers have impersonated employees of real-estate firms to trick home buyers to wire down payments to the wrong bank account [1, 7, 17].

BEC takes several forms: some emails ask the recipient to wire transfer money to the attacker's account, others ask for W-2 forms that contain social security numbers, and some lead the recipient to follow a phishing link, in order to steal their credentials. The common theme is the impersonation of a manager or colleague of the target [12]. In this work, we focus on attacks where the attacker is external to the organization, and is trying to impersonate an employee. In §6 we discuss other scenarios, such as where the attacker uses a compromised internal email account to impersonate employees [18, 19].

Most email security systems are not effective in detecting BEC. When analyzing an incoming email, email security systems broadly look for two types of attributes: *malicious* and *volumetric*. Examples of malicious attributes are an attachment that contains malware, a link pointing to a compromised website, or an email that is sent from a domain with a low reputation. There are various well-known techniques to detect malicious attributes, including sandboxing [49], and domain reputation [2, 48]. Volumetric attributes are detected when the same email format is sent to hundreds of recipients or more. Examples include the same text or sender email (e.g., spam), and the same URL (e.g., mass phishing campaigns). However, employee impersonation emails do not contain malicious or volumetric attributes: they typically do not contain malware, are not sent from well-known malicious IPs, often do not contain a link, and are sent to a small number of recipients (with the explicit intent of evading volumetric filters). When employee impersonation attacks do contain a link, it is typically

a link to a fake sign up page on a legitimate website that was compromised, which does not appear on any IP black lists. In addition, the text of the attacks is tailored to the recipient, and is typically not caught by volume-based filters.

Our design goal is to detect and quarantine BEC attacks in real-time, at a low false positive rate (1 in a million emails) and high precision (95%). We make the observations that popular cloud email systems, such as Office 365 and Gmail, provide APIs that enable account administrators to allow external applications to access historical emails. Therefore, we design a system that detects BEC by relying on historical emails available through these APIs.

Prior work on detecting impersonation has been conducted either on very small datasets [10, 14, 20, 45]), or focused on stopping a subset of BEC attacks (domain spoofing [14] or emails with links [20]). In addition, most prior work suffers from very low precision (only 1 in 500 alerts is an attack [20]) or very high false positive rates [10, 45]), which makes prior work unsuitable for detecting BEC in real-time.

The main challenge in designing a system that can detect BEC at a low false positive rate is that *BEC emails are very rare* as a percentage of all emails. In fact, in our dataset, less than one out of 50,000 emails is a BEC attack. Therefore, in order to achieve low false positives, we design a system using supervised learning, which relies on a large training set of BEC emails. However, bootstrapping a supervised learning systems presents two practical challenges. First, it is difficult to label a sufficiently large training dataset that includes millions of emails. Second, it is challenging to train a classifier on an *imbalanced dataset*, in which the training dataset contains almost five orders of magnitude fewer positive samples (i.e., BEC attacks) than negative samples (i.e., innocent emails).

In this paper, we present how we initially trained BEC-Guard, a security system that automatically detects and quarantines BEC attacks in real-time using historical emails. BEC-Guard is part of a commercial product, Barracuda Sentinel, used by thousands of corporate customers of Barracuda Networks to prevent BEC, account takeover, spear phishing and other targeted attacks. BEC-Guard does not require an analyst to review the detected emails, but rather relies on offline and infrequent re-training of classifiers. The key insight of BEC-Guard is to split the training and classification into two parts: header and body.

Instead of directly classifying BEC attacks, the *impersonation classifier* detects impersonation attempts, by determining if an attacker is impersonating an employee in the company by inspecting the header of the email. It utilizes features that include information about which email addresses employees typically utilize, how popular their name is, and characteristics of the sender domain. The *content classifiers* are only run on emails that were categorized as impersonation attempts, and inspects the body of the email for BEC. For emails that do not contain links, we use a k-nearest neighbors [43] (KNN) classifier that weighs words using term frequency-inverse document frequency [28, 42] (TFIDF). For emails with links, we train a random forest classifier that relies on the popularity as well as the position of the link in the text. Both of the content classifiers can be retrained frequently using customer feedback.

To create the initial classifiers, we individually label and train each type of classifier: the labels of the impersonation classifier are generated using scripts we ran on the training dataset, while the content classifiers are trained over a manually labeled training dataset. Since we run the content classification only on emails that were detected as impersonation attempts, we need to manually label a much smaller subset of the training dataset. In addition, to ensure the impersonation classifier is trained successfully over the imbalanced dataset, we develop an under-sampling technique for legitimate emails using Gaussian Mixture Models, an unsupervised clustering algorithm. The classifiers are typically re-trained every few weeks. The dataset available for initial training consists of a year worth of historical emails from 1500 customers, with an aggregate dataset of 2 million mailboxes and 2.5 billion emails. Since training the initial classifiers, our dataset has been expanded to include tens of millions of mailboxes.

BEC-Guard uses the APIs of cloud-based email systems (e.g., Office 365 and Gmail), both to automatically learn the historical communication patterns of each organization within hours, and to quarantine emails in real-time. BEC-Guard subscribes to API calls, which automatically alert BEC-Guard whenever a new email enters the organization's mailbox. Once notified by the API call, BEC-Guard classifies the email for BEC. If the email is determined to be BEC, BEC-Guard uses the APIs to move the email from the inbox folder to a dedicated quarantine folder on the end-user's account.

To evaluate the effectiveness of our approach, we measured BEC-Guard's performance on a dataset of emails taken from several hundred organizations. Within this labeled dataset, BEC-Guard achieves a a precision of 98.2%, a false positive rate of only one in 5.3 million. To summarize, we make the following contributions:

- First real-time system for preventing BEC that achieves high precision and low false positive rates.
- BEC-Guard's novel design relies on cloud email provider APIs both to learn the historical communication patterns of each organization, and to detect attacks in real-time.
- To cope with labeling millions of emails, we split the detection problem into two sets of classifiers run sequentially.
- We use different types of classifiers for the header and text of the email. The headers are classified using a random forest, while the text classification relies primarily on a KNN model that is not dependent on any hard-coded features, and can be easily re-trained.
- To train the impersonation classifier on an imbalanced dataset, we utilize a sampling technique for the legitimate emails using a clustering algorithm.

BEC Objective	Link?	Percentage
Wire transfer	No	46.9%
Click Link	Yes	40.1%
Establish Rapport	No	12.2%
Steal PII	No	0.8%

Table 1: The objective of BEC attacks as a percentage of 3,000 randomly chosen attacks. 59.9% of attacks do not involve a phishing link.

Role	Recipient %	Impersonated %
CEO	2.2%	42.9%
CFO	16.9%	2.2%
C-level	10.2%	4.5%
Finance/HR	16.9%	2.2%
Other	53.7%	48.1%

Table 2: The roles of recipients and impersonated employees from a sample of BEC attacks chosen from 50 random companies. C-level includes all executives that are not the CEO and CFO, and Finance/HR does not include executives.

2 Background

Business email compromise, also known as employee impersonation, CEO fraud, and whaling,[1] is a class of email attacks where an attacker impersonates an employee of the company (e.g., the CEO, a manager in HR or finance), and crafts a personalized email to a specific employee. The intent of this email is typically to trick the target to wire money, send sensitive information (e.g., HR or medical records), or lead the employee to follow a phishing link in order to steal their credentials or download malware to their endpoint.

BEC has become one of the most damaging email-borne attacks in recent years, equaling or surpassing other types of attacks, such as spam and ransomware. Due to the severity of BEC attacks, the FBI started compiling annual reports based on US-based organizations that have reported their fraudulent wire transfers to the FBI. Based on the FBI data, between 2013 and 2018, $12 billion have been lost [13]. To put this in perspective, a Google study estimates that the total amount of ransomware payments in 2016 was only $25 million [11].

In this section, we review common examples of BEC, and provide intuition on how their unique characteristics can be exploited for supervised learning classification.

2.1 Statistics

To better understand the goals and methodology of BEC attacks, we compiled statistics for 3,000 randomly selected BEC attacks in our dataset (for more information about our dataset, see §4.2). Table 1 summarizes the objectives of the attacks. The results show that the most common BEC in the sampled attacks is try to deceive the recipient to perform a wire transfer to a bank account owned by the attacker, while about 0.8% of the attacks ask the recipient to send the attacker personal identifiable information (PII), typically in the form of W-2 forms that contain social security numbers. About 40% of attacks ask the recipient to click on a link. 12% of attacks try to establish rapport with the target by starting a conversation with the recipient (e.g., the attacker will ask the recipient whether they are available for an urgent task). For the "rapport" emails, in the vast majority of cases, after the initial email is responded to the attacker will ask to perform a wire transfer.

An important observation is that about 60% of BEC attacks do not involve a link: the attack is simply a plain text email that fools the recipient to commit a wire transfer or send sensitive information. These plain text emails are especially difficult for existing email security systems, as well as prior academic work to detect [20], because they are often sent from legitimate email accounts, tailored to each recipient, and do not contain any suspicious links.

We also sampled attacks from 50 random companies in our dataset, and classified the roles of the recipient of the attack, as well as the impersonated sender. Table 2 presents the results. Based on the results, the term "CEO fraud" used to describe BEC is indeed justified: about 43% of the impersonated senders were the CEO or founder. The targets of the attacks are spread much more equally across different roles. However, even for impersonated senders, the majority (about 57%) are not the CEO. Almost half of the impersonated roles and more than half of targets are not of "sensitive" positions, such as executives, finance or HR. Therefore, simply protecting employees in sensitive departments in not sufficient to protect against BEC.

2.2 Common Types of BEC

To guide the discussion, we describe the three most common examples of BEC attacks within our dataset: wire transfer, rapport, and impersonation phishing. In §6 we will discuss other attacks that are not covered by this paper. All three examples we present are real BEC attacks from within our dataset, in which the names, companies, email addresses and links have been anonymized.

Example 1: Wire transfer example

```
From: "Jane Smith" <jsmith@acrne.com>
To: "Joe Barnes" <jbarnes@acme.com>
Subject: Vendor Payment

Hey Joe,

Are you around? I need to send a wire
transfer ASAP to a vendor.

Jane
```

In Example 1, the attacker asks to execute a wire transfer. Other similar requests include asking for W-2 forms, medical information or passwords. In the example the attacker spoofs the name of an employee, but uses an email address that

[1] We refer to this attack throughout the paper as *BEC*.

Example 2: Rapport example

```
From: "Jane Smith" <jsmith@acme.com>
Reply-to: "Jane Smith" <ceo.executive@outlook.com>
To: "Joe Barnes" <jbarnes@acme.com>
Subject: At desk?

Joe, are you available for something urgent?
```

Example 3: Spoofed Name with Phishing Link

```
From: "Jane Smith" <greyowl1234@comcast.net>
To: "Joe Barnes" <jbarnes@acme.com>
Subject: Invoice due number 381202214

I tried to reach you by phone today but I
couldn't get through. Please get back to me
with the status of the invoice below.

Invoice due number 381202214:
[http://firetruck4u.net/past-due-invoice/]
```

does not belong to the organization's domain. Some attackers even use a domain that looks similar to the target organization's domain (e.g., instead of acme.com, the attacker would use acrne.com). Since many email clients do not display the sender email address, some recipients will be deceived even if the attacker uses an unrelated email address.

Example 2 tries to create a sense of urgency. After the recipient responds to the email, the attacker will typically ask for a wire transfer. The email has the from address of the employee, while the reply-to address will relay the response back to the attacker. Email authentication technologies such as DMARC, SPF and DKIM can help stop spoofed emails. However, the vast majority of organizations do not enforce email authentication [25], because it can be difficult to implement correctly and often causes legitimate emails to be blocked.[2] Therefore, our goal is to detect these attacks without relying on DMARC, SPF and DKIM.

Example 3 uses a spoofed name, and tries to get the recipient to follow a phishing link. Such phishing links are typically not detected by existing solutions, because the link is unique to the recipient ("zero-day") and will not appear in any black lists. In addition, attackers often compromise relatively reputable websites (e.g., small business websites) for phishing links, which are often classified as high reputation links by email security systems. The link within the email will typically lead the recipient to a website, where they will be prompted to log in a web service (e.g., an invoicing application) or download malware.

3 Intuition: Exploiting the Unique Attributes of Each Attack

The three examples all contain unique characteristics, which set them apart from innocent email messages. We first describe the unique attributes in the header of each example, and then discuss the attributes of the email body and how they can be used to construct the features of a machine learning classifier. We also discuss legitimate corner cases of these attributes that might fool a classifier and cause false positives.

Header attributes. In Example 1 and 3, the attacker impersonates the name of a person, but uses a different email address than the corporate email address. Therefore, if an email contains a name of an employee, but uses an email address that is not the typical email address of that employee, there is a higher probability that the sender is an imposter.

However, there are legitimate use cases of non-corporate emails by employees. First, an employee might use a personal email address to send or forward information to themselves or other employees in the company. Ideally, a machine learning classifier should be able to learn all the email addresses that belong to a certain individual, including corporate and personal email addresses. Second, if an external sender has the same name as an internal employee, it might seem like an impersonation.

In Example 2, the attacker spoofs the legitimate email address of the sender, but the reply-to email address is different than the sender address, which is unusual (we will also discuss the case where the attacker sends a message from the legitimate address of the sender without changing the reply-to field in §6). However, such a pattern has legitimate corner cases as well. Some web services and IT systems, such as LinkedIn, Salesforce, and other support and HR applications, "legitimately impersonate" employees to send notifications, and change the reply-to field to make sure the response to the message is recorded by their system.

Other header attributes might aid in the detection of BEC attacks. For example, if an email is sent at an abnormal time of day, or from an abnormal IP or from a foreign country. However, many BEC attacks are designed to seem legitimate, and are sent in normal times of day and from seemingly legitimate email addresses.

Body attributes. The body of Example 1 contains two unique semantic attributes. First, it discusses sensitive information (a wire transfer). Second, it is asking for a special, immediate request. Similarly, the text of Example 2 is asking whether the recipient is available for an urgent request. Such an urgent request for sensitive information or availability might be legitimate in certain circumstances (for example, in an urgent communication within the finance team).

The unique attribute in the body of Example 3 is the link itself. The link is pointing to a website that does not have anything to do with the company: it does not belong to a web service the company typically uses, and it is not related to the company's domain.

Finally, all three examples contain certain textual and visual elements that are unique to the identity of the sender. For example, Example 1 contains the signature of the CEO and all of the emails contain a particular grammar and writing

[2] Many organizations have legitimate systems that send emails on their behalf, for example, marketing automation systems, which can be erroneously blocked if email authentication is not setup properly.

style. If any of these elements deviate from the style of a normal email from a particular sender, they can be exploited to detect an impersonation. Since in many BEC emails the attackers take great care in making the email appear legitimate, we cannot overly-depend on detecting stylistic aberrations.

As shown above, each of the examples has unique anomalous attributes that can be used to categorize it as a BEC attack. However, as we will show in §7, none of these attributes *on its own* is sufficient to classify an email with a satisfactory false positive rate.

Leveraging historical emails. Much of prior work in detecting email-borne threats relies on detecting malicious signals in the email, such as sender and link domain reputation [2, 48], malicious attachments [49], as well as relying on link click logs and IP logins [20]. However, as Table 1 and the examples we surveyed demonstrate, most BEC attacks do not contain any obviously malicious attachments or links. Intuitively, access to the historical emails of an organization would enable a supervised learning system to identify the common types of BEC attacks by identifying anomalies in the header and body attributes. We make the observation that popular cloud-based email providers, such as Office 365 and Gmail, enable their customers to allow third party applications to access their account with certain permissions via public APIs. In particular, these APIs can enable third-party applications to access historical emails. This allows us to design a system that uses historical emails to identify BEC attacks.

4 Classifier and Feature Design

In this section, we describe BEC-Guard's design goals, and its training dataset. We then describe the initial set of classifiers we used in BEC-Guard, and present our approach to training and labeling.

4.1 Design Goals

The goal of BEC-Guard is to detect BEC attacks in real-time, without requiring the users of the system to utilize security analysts to manually sift through suspected attacks. To meet this goal, we need to optimize two metrics: the false positive rate, and the precision. The false positive rate is the rate of false positives as a percentage of total received emails. If we assume an average user receives over 100 emails a day, in an organization with 10,000 employees, our goal is that it will be infrequent to encounter a false positive (e.g., once a day for the entire organization). Therefore, our target false positive rate is less than one in a million. The precision is the rate of true positives (correctly detected BEC attacks) as a percentage of attacks detected by the system, while the false positive rate is a percentage of false positives of all emails (not just emails detected by the system). If the precision is not high, users of BEC-Guard will lose confidence in the validity of its predictions. In addition to these two metrics, we need to ensure high coverage, i.e., that the system catches the vast majority of BEC attacks.

4.2 Dataset and Privacy

We developed the initial version of BEC-Guard using a dataset of corporate emails from 1,500 organizations, which are actively paying customers of Barracuda Networks. The organizations in our dataset vary widely in their type and size. The organizations include companies from different industries (healthcare, energy, finance, transportation, media, education, etc.). The size of the organization varies from 10 mailboxes to more than 100,000. Overall, to train BEC-Guard, we labeled over 7,000 examples of BEC attacks, randomly selected from the 1,500 organizations.

To access the data, these organizations granted us permission to access to the APIs of their Office 365 email environments. The APIs provide access to all historical corporate emails. This includes emails sent internally within the organization, and from all folders (inbox, sent, junk, etc.). The API also allows us to determine which domains are owned by each organization, and even whether an email was read.

Ethical and privacy considerations. BEC-Guard is part of a commercial product, and the 1,500 customers that participate in the dataset provided their legal consent to Barracuda Networks to access their historical corporate emails for the purpose identifying BEC. Customers also have the option of revoking access to BEC-Guard at any time.

Due to the sensitivity of the dataset, it was only exposed to the five researchers who developed BEC-Guard, under strict access control policies. The research team only accessed historical emails for the purposes of labeling data to develop BEC-Guard's classifiers. Once the classifiers were developed, we permanently deleted all of the emails that are not actively used for training the classifiers. The emails used for classification are stored encrypted, and access to them is limited to the research team.

4.3 Dividing the Classification into Two Parts

The relative rare occurrence of BEC attacks influenced several of our design choices. Our first design choice was to rule out unsupervised learning. Unsupervised learning typically uses clustering algorithms (e.g., k-means [15]) to group email categories, such as BEC emails. However, a clustering algorithm would typically categorize many common categories (e.g., social emails, marketing emails), but since BEC is so rare, it results in low precision and many false positives. Therefore, supervised learning algorithms are more suitable for detecting BEC at a high precision. However, using supervised learning presents its own set of challenges.

In particular, BEC is an extreme case of *imbalanced data*. When sampled uniformly, in our dataset, "legitimate" emails are 50,000× more likely to appear than the BEC emails. This presents two challenges. First, in order to label a modest number of BEC emails (e.g., 1,000), we need to label a corpus

on the order of 50 million legitimate emails. Second, even with a large number of labeled emails, training a supervised classifier over imbalanced datasets is known to cause various problems, including biasing the classifier to prefer the larger class (i.e., legitimate emails) [24,26,47,51]. To deal with this extreme case of imbalanced data, we divided the classification and labeling problem into two parts. The first classifier looks only at the metadata of the email, while the second classifier only examines the body and subject of the email.

The first classifier looks for *impersonation* emails. We define an impersonation as an email that is sent with the name of a person, but was not actually sent by that person. Impersonation emails include malicious BEC attacks, and they also include emails that legitimately impersonate an employee, such as internal systems that send automated emails on behalf of an employee. The impersonation classifier only analyzes the metadata of the email (i.e., sender, receiver, CC, BCC fields). The impersonation classifier detects both spoofed name (Example 1 and 3) and spoofed emails (Example 2). The second set of classifiers, the *content* classifiers, only classify emails that were detected as impersonation emails, by examining the email's subject and body to look for anomalies. We use two different content classifiers that each look for different types of BEC attacks.[3] The two content classifiers are: the text classifier, which relies on natural language processing to analyze the text of the email, and the link classifier, which classifies any links that might appear in the email.

All of our classifiers are trained globally on the same dataset. However, to compute some of the features (e.g., the number of time the sender name and email address appeared together), we rely on statistics that are unique to each organization.

4.4 Impersonation Classifier

Table 3 includes the main features used by the impersonation classifier. The features describe the number of times specific email addresses and names have appeared before in the sender and reply-to fields, as well as statistics about the sender's identity.

To demonstrate why it is helpful to maintain historical statistics of a particular organization, consider Figure 1. The figure depicts the number of email addresses that were used by each sender in an organization with 44,000 mailboxes over three months. 82% of the users had emails sent from only one address, and the rest had emails that were sent from more than one address. The reason that some of the senders used a large number of email addresses, is that they were repeatedly impersonated in BEC attacks. For instance, the CEO is a common target for impersonation. and is often targeted dozens of times. However, this signal alone is not sufficient to detect impersonation,. For example, some of the senders that have a large number of email addresses represent shared mailboxes (e.g., "IT" or "HR"), and are legitimate.

[3]There is no inherent advantage in using multiple content classifiers in terms of the false positive rate or precision. We decided to use two different content classifiers, because it made it easier for us debug and maintain them separately.

Feature	Description
Sender has corp domain?	Is sender address from corp domain?
Reply-to != sender address?	Reply-to and sender addresses different?
Num times sender and email	Number of times sender name and email address appeared
Num times reply-to address	Number of times reply-to address appeared
Known reply-to service?	Is reply-to from known web service (e.g., LinkedIn)?
Sender name popularity	How popular is sender name

Table 3: Main features used by the impersonation classifier, which looks for impersonation attempts, including spoofed names and emails.

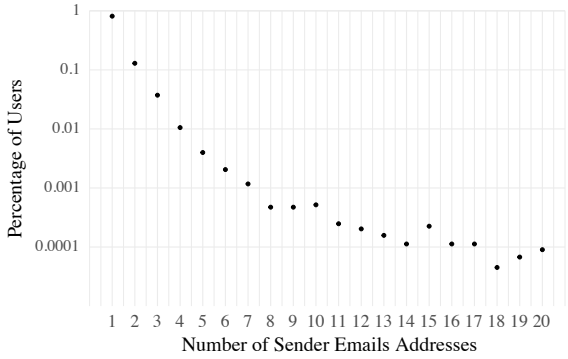


Figure 1: Number of unique emails addresses that were observed for each user in an organization with 44,400 mailboxes. The X axis is the number of unique email addresses that were observed, as a percentage (in the Y axis) of the total number of users of the organization.

Hence, several of the features in the impersonation classifier rely on the historical communication patterns of the organization. This influenced BEC-Guard's architecture. In addition, we maintain a list of known web services that "legitimately" send emails with reply-to addresses that are different than the sender address (e.g., LinkedIn, Salesforce), in order to capture the response. The original list of commonly-used services was populated from a list of the domains of the major web services. We then augmented this list with additional services when we encountered them during the labeling process (in §6 we discuss possible evasion techniques related to this list of legitimate reply-to senders). The sender name popularity score is computed offline by maintaining a list of how frequently names appear across different organizations in our dataset. The more popular a name, the higher the likelihood that a name with an email address the employee typically does not use is another person (a name collision).

Name and nickname matching. In order to detect name spoofing, the impersonation classifier needs to match the

sender name with a name of an employee. However, names can be written in various forms. For example: "Jane Smith" can be written as: "Smith, Jane", "Jane J. Smith" or "Jane Jones Smith". In addition, we need to deal with special characters that might appear in names, such as ì or ä.

To address these problems, BEC-Guard normalizes names. It stores employee name as <first name, last name> tuples, and checks all the variants of the sender name to see if it matches a name of an employee with a corporate email address. These variants include stripping the middle name or initial, reversing order of the first name and surname, and stripping suffixes. Suffixes include examples like "Jr." or when the email address is sent as part of the sender name. In addition, we match the first name against a publicly available list of nicknames [36], to catch cases for example when the attacker sends an email as "Bill Clinton", and the name of the employee is stored as "William Clinton".

Content classifiers. Our system uses two content classifiers: the *text classifier* and *link classifier*. The text classifier catches attacks similar to Example 1 and 2, and the link classifier stops attacks that are similar to Example 3. By design, the content classifiers are meant to be updated more frequently than the impersonation classifier, and should be easily retrained based on false negatives and false positives reported by users.

Text classifier. In BEC attacks similar to Example 1 and 2, the body contains words that are indicative of a sensitive or special request, such as "wire transfer" or "urgent". Therefore, our first iteration of the text classifier was designed to look for specific words that might imply a special request or a financial or HR transaction. The features of the classifiers described the position in the text of a list of sensitive words and phrases. However, over time, we noticed this approach suffered from several problems. First, a classifier that relies on hard-coded keywords can miss attacks when attackers slightly vary a specific word or phrase. Second, to successfully retrain the classifier, we had to modify the lists of keywords that it looks for, which required manually updating the keyword list on a daily basis.

Instead, we developed a text classifier that learns expressions that are indicative of BEC on its own. The first step is to pre-process the text. BEC-Guard removes information from the subject and body of the email that would not be useful for classifying the email. It removes regular expression patterns that include salutations ("Dear", "Hi"), pre-canned headers, as well as footers ("Best,") and signatures. It also removes all English stopwords, as well as any names that may appear in the email.

The second step is to compute the frequency-inverse document frequency [42] (TFIDF) score of each word in the email. TFIDF represents how important each word is in an email, and is defined as:

$$TF(w) = \frac{\text{num times } w \text{ appears in email}}{\text{num words in email}}$$

$$IDF(w) = \frac{log(\text{num emails})}{\text{num emails with } w}$$

Where w is a given word in an email. $TF(w) \cdot IDF(w)$ gives a higher score to a word that appears frequently in a specific email, but which is relatively rare in the whole email corpus. The intuition is that in BEC emails, words for example that denote urgency or a special request would have a high TFIDF score, because they appear frequently in BEC emails but less so in legitimate emails.

When training the text classifier, we compute the TFIDF score of each word in each email of the training set. We also compute the TFIDF for pairs of words (bigrams). We store the global statistics of the IDF as a *dictionary*, which contains number of emails in the training set that contain unique phrases encountered in the training of the text classifier. We limit the dictionary size to 10,000 of the top ranked words (we evaluate how the size of the dictionary impacts classification precision in §7.2).

The feature vector of each email is equal to the the number of words in the dictionary, and each number represents the TFIDF of each one of the words in the dictionary. Words that do not appear in the email, or that do not appear in the dictionary have a TFIDF of zero. The last step is to run a classifier based on these features. Table 4 presents the top 10 phrases (unigram and bigram) in the BEC emails in our dataset. Note that the top phrases all indicate some form of urgency.

Top phrases in BEC emails by TFIDF	
1. got moment	6. need complete
2. response	7. ASAP
3. moment need	8. urgent response
4. moment	9. urgent
5. need	10. complete task

Table 4: The top 10 phrases of BEC emails, sorted by their TFIDF ranking from our evaluation dataset (for more information on evaluation dataset see §7.1). The TFIDF was computed for each word in all of the BEC emails in our evaluation dataset.

Link classifier. The link classifier detects attacks similar to Example 3. In these attacks, the attacker tries to get the recipient to follow a phishing link. As we described earlier, these personalized phishing links are typically not detected by IP blacklists, and are usually unique to the recipient. In this case, since the content classifier only classifies emails that were already classified as impersonation emails, it can mark links as "suspicious", even if they would have a high false positive rate otherwise. For example, a link that points to a

small website, or one that was recently registered, combined with an impersonation attempt would have a high probability of being a BEC email.

Feature	Description
Domain popularity	How popular is the link's least popular domain
URL field length	Length of least popular URL (long URLs are more suspicious)
Domain registration date	Date of domain registration of least popular domain (new domains are suspicious)

Table 5: Main features used by the link request classifier, which stops attacks like in Example 3.

Table 5 describes the main features used by the link request classifier. The domain popularity is calculated by measuring the Alexa score of the domain. In order to deal with link shorteners or link redirections, BEC-Guard expands the URLs before computing their features for the link classifier. In addition, several of the URL characteristics require determining information about the domain (popularity and score). For the domain popularity feature, we cache a list of the top popular domains, and update it offline. To determine the domain registration date, BEC-Guard does a real-time WHOIS lookup. Note that unlike the impersonation classifier, which needs to map the distribution of email address per sender name, none of the features of the text and link classifier are organization-specific. This allows us to easily retrain them based on user reported emails.

4.5 Classifier Algorithm

The impersonation and link classifiers use random forest [5] classification. Random forests are comprised of randomly formed decision trees [40], where each tree contributes a vote, and the decision is determined by the majority of the trees. Our system uses random forests rather than individual decision trees, since we found they provide better precision, but for offline debugging and analysis we often visualize individual decision trees. We decided to use KNN for the text classifier, because it had slightly better coverage than random forests. However, we found that since the text classifier uses a very large number of features (a dictionary of 10,000 phrases), its efficacy was similar across different classifiers. In §7.2 we evaluate the performance of the different classifier algorithms.

In addition, we have explored deep-learning based techniques, such as word2vec [34] and sense2vec [46], which expand each word to a vector that represents its different meanings. We currently do not use such deep-learning techniques, because they are computationally heavy both for training and online classification.

Detecting impersonation of new employees. When a new employee joins the organization, the impersonation classifier will not have sufficient historical information about that employee, since they will not have any historical emails. As that employee receives more emails, BEC-Guard will be start compiling statistics for the employee. A similar problem may also arise in organizations that periodically purge their old emails. In practice, we found that the classifier performs well after only one month of data.

4.6 Labeling

In order to label the initial training set, we made several assumptions about the BEC attack model. First we assumed attackers impersonate employees using their name (under a set of allowed variations, as explained above). Second, we assumed the impersonation does not occur more than 100 times using the same email address. Third, we assumed the attacker uses an email address that is different than the corporate address, either as the from address or the reply-to address. We discuss other types of attacks that do not fit these assumptions, as well as how attackers may evade these assumptions in §6. Under these constraints, we fully covered all of the possible attacks and manually labeled them. In addition, we incorporated missed attacks reported from customers (we discuss this process in §7.3).

The reason we assumed a BEC email does not impersonate an employee using the same email address more than 100 times is that BEC-Guard is designed with the assumption that the organization is already using at a spam filter, which provides protection against volume-based attacks (e.g., the default spam protection of Office 365 or Gmail). Therefore, an attacker that would send an email from an unknown address more than 100 times to the same recipient would likely be blocked by the spam filter. In fact, in our entire dataset, which is only composed of post spam-filtered emails, we have never witnessed an attacker using the email address to impersonate an employee more than 20 times. Note that we only used this assumption for labeling the original training set, and do not use it for ongoing retraining (since retraining is based on customer reported attacks).

Impersonation classifier. In order to label training data for the impersonation classifier, we ran queries on the headers of the raw emails to uncover all emails that might contain BEC attacks under our labeling assumptions (see above). We then labeled the results of all the queried emails as impersonation emails, and all the emails that were not found by the queries as legitimate emails.

Content classifiers. The training dataset for the content classifiers is constructed by running a trained impersonation classifier on a fresh dataset, which is then labeled manually. The initial training set we used for the content classifiers included 300,000 impersonation emails from randomly selected organizations over a year of data. Even within this training data set, we were able significantly further limit the number of emails that needed to be manually labeled. This is due to the fact that the vast majority of these emails were obviously

not BEC attacks, because they were due to a legitimate web services that impersonates a large number of employees (e.g., a helpdesk system sending emails on behalf of the IT staff).

After constructing the initial dataset, training content classifiers is very straightforward, since we continuously collect false negative and false positive emails from users and add them into the training set. Note that we still manually review these samples before retraining as a measure of quality control, to ensure that adversaries do not "poison" our training set, as well as to make sure that users did not label emails erroneously.

Sampling the dataset. Naïvely training a classifier over an imbalanced dataset typically biases the classifier to prefer the majority class. Specifically, it can result in a classifier that will simply always choose to predict the majority class, i.e., legitimate emails, and will thus achieve very high accuracy (i.e., $accuracy = (tp+tn)/(tp+tn+fp+fn)$, where tp is true positives, tn is true negatives, fp is false positives, and fn is false negatives). Since BEC is so rare in our dataset, a classifier that always predicts that an email is legitimate would achieve a high accuracy. This problem is especially acute in the case of our impersonation classifier, which needs to do the initial filtering between legitimate and BEC emails. In the case of content classifiers, we did not have to sample the dataset, because it deals with a much smaller training dataset.

There are various methods of dealing with imbalanced datasets, including over-sampling the minority class and under-sampling the majority class [6,24,27,29,30], as well as assigning higher costs to incorrectly predicting the minority class [9,38].

Our second major design choice was to under-sample the majority class (the legitimate emails). We made this decision for two reasons. First, if we decided to over-sample the BEC attacks, we would need to do so by a large factor. This might overfit our classifier and bias the results based on a relatively small number of positive samples. Second, over-sampling makes training more expensive computationally.

A naïve way to under-sample would be to uniformly sample the legitimate emails. However, this results in a classifier with a low precision, because the different categories of legitimate emails are not well represented. For example, uniformly sampling emails might miss emails from web services that legitimately impersonate employees. The impersonation classifier will flag these emails as BEC attacks, because they are relatively rare in the training dataset.

The main challenge in under-sampling the majority class is how to represent the entire universe of legitimate emails with a relatively small number of samples (i.e., comparable or equal to the number of BEC email samples). To do so, we cluster the legitimate emails using an unsupervised learning algorithm, Gaussian Mixture Models (GMM). The clustering algorithm splits the samples into clusters, each of which is represented by a Normal distribution, projected onto the impersonation classifier feature space. Figure 2 illustrates an example with two features and 14 legitimate email samples. In this example, the samples are split into three clusters. To choose a representative sample of legitimate emails, we randomly pick a certain number of samples from each cluster, proportional to the number of legitimate emails that belong to each cluster. If for example our goal is to use a total of 7 samples, we would choose 4 samples from the first cluster, 2 samples from the second cluster, and 1 sample from the third cluster, because the original number of samples in each cluster is 8, 4, and 2, respectively.

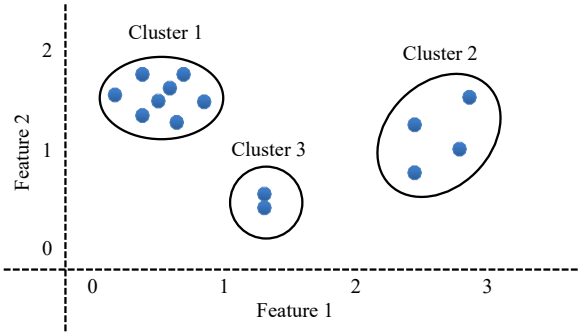


Figure 2: Depiction of running clustering algorithm on a set legitimate emails in a two-dimensional feature space with three clusters. After clustering the legitimate emails, we choose the number of samples from each cluster in proportion to the size of the cluster.

We chose the number of clusters that guarantee a minimal representation for each major "category" of legitimate email. We found that using 85 clusters was sufficient for capturing the legitimate emails in our dataset. When we tried using more than 85 clusters, the clusters beyond the 85th one would be nearly or entirely empty. Even after several iterations of retraining the impersonation classifier, we have have found that 85 clusters are sufficient to represent our dataset.

5 System Design

BEC-Guard consists of two key stages: an online classification stage and an offline training stage. Offline training is conducted periodically (every few days). When a new email arrives, BEC-Guard combines the impersonation and content classifiers to determine whether the email is BEC or not. These classifiers are trained ahead of time in the offline training stage. We describe the key components of our system design in more detail below.

Traditionally, commercial email security solutions have a gateway architecture, or in other words, they sit in the data path of inbound emails and filter malicious emails. As described above, some of BEC-Guard's impersonation classifier features rely on historical statistics of internal communications. The gateway architecture imposes constraints on detecting BEC attacks for two reasons. First, a gateway typically cannot observe internal communications. Second, the gateway usually does not have access to historical communications, so it would require several months or more of observing the communication patterns before the system would be able to detect

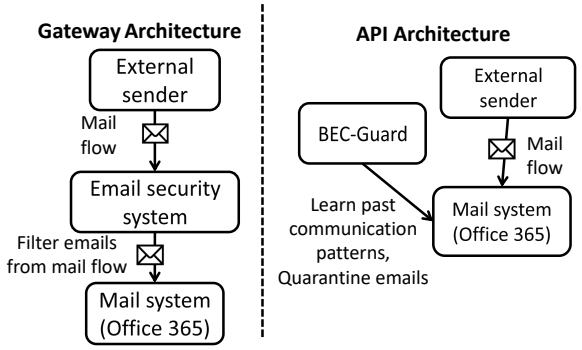


Figure 3: Comparison between the architecture of traditional email security systems, which sit as a gateway that filters emails before they arrive in the mail system, and BEC-Guard's architecture, which relies on APIs for learning the historical communication patterns of each organization, and detecting attacks in real-time.

incoming BEC attacks. Fortunately, cloud-based email services, such as Office 365 and Gmail, provide APIs that enable access to historical communications, as well as to monitor and move emails in real-time. BEC-Guard leverages these APIs both to gain access to historical communication, and also to do near real-time BEC detection. Figure 3 compares the gateway architecture with BEC-Guard's API based architecture. We describe BEC-Guard's design and implementation using the Office 365 APIs.

Warmup phase. We name the process of analyzing each organization's historical communications, the *warmup* phase. In order to start the warmup, the organization enables BEC-Guard to get access to its Office 365 account with an authentication token using OAuth with an Office 365 administrator account. This allows BEC-Guard to access the APIs for all the users associated with the account. Once authenticated, BEC-Guard starts collecting statistics necessary for the impersonation classifier (e.g., number of times a certain user sent an email from a certain email address). The statistics collected by BEC-Guard go back one year. We found that the classifier performs well with as little as one month of historical data.

Online classification. After the warmup phase, BEC-Guard is ready to detect incoming BEC attacks in real-time. To do so, BEC-Guard waits for a webhook API call from any of the users in the organization's Office 365 account. The webhook API calls BEC-Guard anytime there is any new activity for a specific user. When the webhook is triggered, BEC-Guard checks if there is a new received email. If so, BEC-Guard retrieves the email, and classifies it, first using the impersonation classifier, using a database that contains the historical communication statistics unique to each organization. Then, only if it was classified as an impersonation email, BEC-Guard classifies the email using the content classifiers.

If at least one of the content classifiers classifies the email as a BEC attack, BEC-Guard quarantines the email. This is performed by removing the email from the folder where it was received by the user (typically the inbox folder), and moving it into a designated quarantine folder in the end user's mailbox. Since the email is quarantined on the server side, when the user's email clients synchronize the email it will also get quarantined on the user's email clients. In addition, the vast majority of emails get quarantined by BEC-Guard before they are synchronized to the user's email client.

6 Evasion

In this section we discuss attacks that are currently not stopped by BEC-Guard, and evasion techniques that can be used by attackers to bypass BEC-Guard and how they can be addressed.

BEC-Guard is a live service in production, and has evolved rapidly since it was first launched in 2017. We have deployed additional classifiers to augment the ones described in this paper in response to some of the evasion techniques presented below, and the existing classifiers have been retrained multiple times. Another benefit of the API-based architecture is that if we find some attacks were missed by an evasion we can go back in time and find them, and update the system accordingly. The email threat landscape is rapidly changing, and while it is important that the detectors maintain high precision, it is equally important that the security system can be easily adapted and retrained.

6.1 Stopping Other Attacks

BEC-Guard focuses on stopping BEC attacks, in which an external attacker impersonates an employee. However, there are other types of BEC that are not covered by BEC-Guard.

Account takeover. When attackers steal the credentials of an employee, they can login remotely to send BEC emails to other employees. We term this use case "account takeover". There are several approaches to detecting account takeover, including monitoring internal emails for anomalies (e.g., an employee suddenly sending many emails to other employees they typically do not communicate with), monitoring suspicious IP logins, and monitoring suspicious inbox rule changes (e.g., an employee suddenly creates a rule to delete outbound emails) [18–20]. This scenario is not the focus of BEC-Guard, but is covered by our commercial product.

Impersonating both sender name and email without changing reply-to address. It is possible that external attackers could send emails that impersonate both the sender's name and email address, without using a different reply-to address. We have not observed such attacks in our dataset, but they are possible, especially in the case where the attacker asks the recipient to follow a link to steal their credentials. Similar to account takeover, such attacks can be detected by looking for abnormal email patterns. Another possible approach, used by Gascon et al., is to look for anomalies in the actual MIME header [14].

Impersonation of external people. BEC-Guard's impersonation classifier currently relies on having access to the historical inbound email of employees. In order to detect impersonation of external people that frequently communicate

with the organization, BEC-Guard can incorporate emails that are sent from external people to the company.

Text classification in any language. BEC-Guard is currently optimized to catch BEC in languages that appear frequently in our dataset. Both the impersonation classifier and the link classifier are not language-dependent, but the text classifier relies on the TFIDF dictionary is dependent on the language of the labeled dataset. There are a few possible ways to make BEC-Guard's text classifier completely language agnostic. One is to deliberately collect sufficient samples in a variety of languages (either based on user reports or generate them synthetically), and label and train on those emails. Another potentially more scalable approach is to translate the labeled emails (e.g., using Google Translate or a similar tool).

Generic sender names. BEC-Guard explicitly tries to detect impersonations of employee names. However, attackers may impersonate more generic names, such as "HR team" or "IT". This attack is beyond the scope of this paper, but we address it using a similar approach to BEC-Guard in order to detect these attacks: we combine our content classifiers with a new impersonation classifier, which looks for sender names that commonly occur across different organizations, but are sent from a non-corporate email address or have a different reply-to address.

Brand impersonation. Similar to the "generic sender" attack, attackers often impersonate popular online services (e.g., Google Drive or Docusign). These types of attacks are out of the scope for this paper, but we detect them using a similar methodology of combining content classifier, with an impersonation classifier that looks for an anomalous sender (e.g., the sender name has "Docusign", but the sender domain has no relation to Docusign).

6.2 Evading detection

Beyond BEC attacks that BEC-Guard is not designed to detect (as noted above), there are other several ways attackers can try to evade BEC-Guard. We discuss these below and discuss how we have adapted BEC-Guard to address them.

Legitimizing the sender email address. Any system that uses signals based on anomaly detection is vulnerable to attackers that invest extra effort in not appearing "anomalous". For example, when labeling our dataset, we assume that the impersonated employee was not impersonated by the same sender email address more than 100 times. While this threshold is not hard coded into the impersonation classifier, it was a threshold we used to filter emails for the initial training set, and therefore may bias the classifier. Note that we have never observed an attacker impersonating an employee with the same email more than 20 times.

We believe this assumption is valid since BEC-Guard assumes that the organization is already using a volume-based security filter (e.g., the default spam protection of O365 or Gmail or another spam filter), which would pick up a "volumetric" attack. Typically these systems would flag an email that was sent at once from an unknown address to more than 100 employees as spam.

However, a sophisticated attacker may try to bypass these filters by sending a large number of legitimate emails from the impersonated email address to a particular organization, and only after sending hundreds of legitimate emails they would send a BEC using that address. Of course the downside of this approach is that it would require more investment from the attacker, and increase the economic cost of executing a successful BEC campaign. One way to overcome such an attack, is to add artificial samples to the impersonation classifier that have higher thresholds, in order to remove the bias. Of course this may reduce the overall precision of BEC-Guard.

Using infrequent synonyms. Another evasion technique is to send emails that contain text that is different or has a lower TFIDF than the labeled emails used to train our text classifier. For example, the word "bank" has a higher TFIDF, than the word "fund". As mentioned before, one way to overcome these types of attacks is to cover synonyms using a technique, such as word2vec [34].

Manipulating fonts. Attackers have employed various font manipulations to avoid text-based detectors. For example, one technique is to use fonts with a size of zero [35], which are not displayed to the end user, but can be used to obfuscate the impersonation or meaning of the text. Another technique is to use non-Latin letters, such as letters in Cyrillic, which appear similar to the Latin letters to the end user, but are not interpreted as Latin by the text-based detector [16].

In order to deal with these types of techniques, we always normalize any text before feeding it to BEC-Guard's classifiers. For example, we ignore any text with a font size of zero. If we encounter Cyrillic or Greek in conjunction with Latin text, we normalize the non-Latin letters to match the Latin letter that is closest in appearance to it. While these techniques are heuristic based, they have proved effective in stopping the common forms of font-based evasion.

Hiding text in an image. Instead of using text within the email, attackers can hide the text within an embedded image. We have observed this use case very rarely in practice, most likely because these attacks are probably less effective. Many email clients do not display images by default and even when they do, the email may seem odd to the recipient. Therefore, we currently do not address this use case, but a straightforward way to address it would be to use OCR to extract the text within the image.

Using a legitimate reply-to address. As mentioned in §4.4 BEC-Guard relies on a list of legitimate reply-to domains to reduce false positives. This list could potentially be exploited. For example, attackers could craft a LinkedIn or Salesforce profile with the same name of the employee being impersonated and send an impersonation email from that service.

	Precision	FP	Recall
BEC-Guard (Combined)	98.2%	0.000019% (1 in 5,260,000)	96.9%
Impersonation Only	11.7%	0.016% (1 in 6,300)	100%

Table 6: Precision, false positive rate, and recall of BEC-Guard compared to the impersonation classifier alone.

While this is indeed a potential evasion technique, these third party services often have their own anti-fraud mechanisms to stop impersonation. In addition, we believe an impersonation attempt is less likely to succeed if it going through a third-party service, since it would probably seem much less natural than simply sending an email from the email account of the employee. Regardless, we have never seen this evasion technique being used by attackers.

7 Evaluation

In this section, we evaluate the efficacy of BEC-Guard. We first analyze the end-to-end performance of BEC-Guard, using a combination of the impersonation and content classifiers. We then break down the performance of each set of classifiers, and analyze the performance of different classifier algorithms. We also try to estimate the extent of unknown attacks that are not caught by BEC-Guard, by comparing the number of reported missed attacks by customers to the number of true positives.

7.1 End-to-end Evaluation

For the end-to-end evaluation, we randomly sampled emails that were processed by BEC-Guard in June 2018. We manually labeled the emails, and evaluated BEC-Guard's classifiers on the labeled data. We labeled the emails for the evaluation dataset similar to the way we labeled the training data for BEC-Guard's classifiers (see §4.6). We first ran a set of queries that uncover all the BEC attacks that we could find under our labeling assumptions. We then manually labeled the resulting emails, and found 4,221 BEC emails. The entire process took about a week of work for one person. The emails that were not labeled as BEC attacks were assumed to be innocent (In §7.3 we discuss emails that might have been missed by our labeling process).

To evaluate the classifiers, we randomly split the evaluation dataset in half: we used half of the emails for training, and the rest to test the classifiers. The dataset includes 200 million emails from several hundred organizations.

To test the end-to-end efficacy of BEC-Guard, we ran the content classifiers only on the emails that were detected as impersonation emails by the impersonation classifier. Table 6 summarizes the efficacy results. The recall of BEC-Guard is high within the emails we labeled: 96.9% of the BEC emails we labeled were successfully classified by the impersonation classifier as well as one of the content classifiers. The combined false positive rate is only one in 5.3 million emails are falsely detected, which is above our design goal of 1 in a million email. The precision is 98.2%.

Text classifier			
Algorithm	Precision	FP	Recall
Logistic Regression	97.1%	$6.1{\cdot}10^{-5}$%	98.4%
Linear SVM	98.3%	$3.6{\cdot}10^{-5}$%	98.7%
Decision Tree	96.0%	$8.5{\cdot}10^{-5}$%	97.1%
Random Forest	99.2%	$1.7{\cdot}10^{-5}$%	96.4%
KNN	98.9%	$2.3{\cdot}10^{-5}$%	97.5%

Table 7: Text classifier algorithm efficacy using a dictionary of 10,000 words. There is very little difference between the efficacy of the algorithms for the text classifier.

Link classifier			
Algorithm	Precision	FP	Recall
Logistic Regression	33.3%	$85.7{\cdot}10^{-5}$%	96.0%
Linear SVM	92.3%	$3.2{\cdot}10^{-5}$%	90.8%
Decision Tree	94.9%	$2.3{\cdot}10^{-5}$%	96.3%
Random Forest	97.1%	$1.3{\cdot}10^{-5}$%	96.0%
KNN	92.5%	$3.3{\cdot}10^{-5}$%	93.5%

Table 8: Link classifier algorithm efficacy. Random forest provides superior results over the other algorithms.

The false positives of the combined classifiers were due to unlikely incidents where the impersonation classifier detected the email (e.g., due to a personal email address) that also contained anomalous content (e.g., an employee uses a personal email to forward links with low popularity domains to a colleague). Another common false positive occurs when employees leave the organization, and request W-2 forms for tax purposes or other personal information. We plan on addressing such false positives by incorporating features that would indicate whether a sender is no longer an employee of the organization (e.g., if they have stopped sending emails from their corporate address). The false negatives are mostly due to instances where the URL is not deemed suspicious, because it belongs to a domain that got compromised that had a relatively high domain popularity, or because the text of the email is not classified as suspicious. The latter case is typically because the attacker did not use phrases that were similar to any of the BEC attacks that were used to train the text classifier. For example, one of the false negatives asked the recipient for gift card information, which was not a request that was used in any prior attacks.

We also ran the impersonation classifier on the evaluation dataset. Its precision is 11.7%, and its false positive rate is 0.016%. Organizations that are only concerned about recall and have the ability to tolerate a relatively large number of false alerts can run the impersonation classifier on its own. The vast majority of false positives of the impersonation classifier are due to employees using their personal or university (alumni) email addresses.

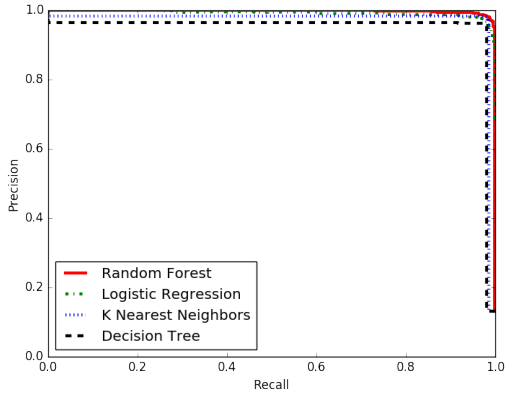


Figure 4: ROC curve of text classifier with different algorithms. All four algorithms perform very similarly, and reach a precision cliff at about 99% recall.

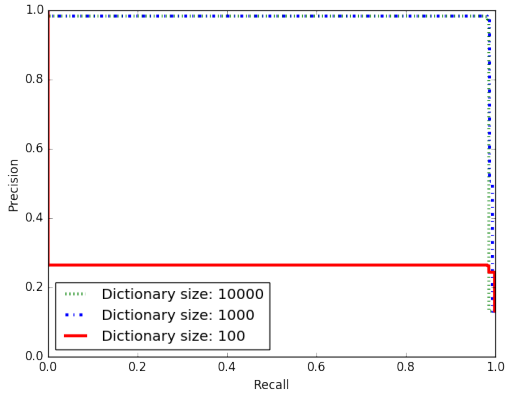


Figure 5: ROC curve of text classifier using KNN with different dictionary sizes. A dictionary size of 1,000 already provides most of the benefit.

7.2 Classifier Algorithms

Table 7 compares the results of the text classifier using different classifier algorithms. As the results show, there is a very small difference between the different classifiers. This is primarily due to the fact that we use a dictionary with a large number of features (10,000). Table 8 shows the results for the link classifier. In the case of the link classifier, random forest more clearly provides superior results than the other classifiers, including KNN. The link classifier is more sensitive to the classification algorithm, because it uses a smaller number of features. Figure 4 presents the ROC curve for four of the classifier algorithms that have a probabilistic output. The ROC curve shows the how each classifier can be tweaked to trade-off precision for recall. All four algorithms behave almost identically: they provide a high level of precision, until a recall level close to 99% where their precision drops. Note that to generate the ROC curves we ran the text classifier only on the emails that were already classified as impersonations. Therefore, its minimum precision in the ROC curve is equal to about 11.7%, which is equal to the precision of the impersonation classifier.

Org	TPs	FNs	Reason
A	31	1	Generic Sender Name
B	4	1	Misclassified Content
C	12	1	External Impersonation
D	8	1	External Impersonation
E	5	1	Misclassified Content
Total	60	5	

Table 9: True positives (TPs) and reported false negatives (FNs) among five organizations, where the administrator has reported at least one false negative.

To analyze the effect of the dictionary size on the classification, Figure 5 plots the efficacy of the text classifier using KNN with different dictionary sizes. The graph shows that most of the marginal benefit is achieved with a dictionary size of 1,000. We observed no noticeable difference in efficacy when using a dictionary larger than 10,000.

7.3 Evaluating Missed Attacks

A general limitation of evaluating imbalanced datasets is that it is difficult to accurately estimate the true false negative rate. In our evaluation dataset, we can only estimate the false negative rate in relation to the data that we labeled. If we missed an attack during labeling, and it was not detected by the classifiers, we would not count it as a false negative.

To deal with "unknown" attacks, our production system allows users to report attacks that it did not detect. We estimate the number of missed attacks *among organizations that have reported missed attacks*. We selected five random organizations that reported missed attacks, and analyzed their detections in the month during which they reported missed attacks. Table 9 provides the number of true and missed detections among these five organizations, as well as the reason for each false negative.

In organization A the attack was missed because the email did not impersonate an employee name, but rather the sender name had a generic title (e.g., "Accountant"). BEC-Guard only detects the impersonation of an employee's name. As we explained in our labeling assumptions (see §4.6), BEC-Guard is only designed to detect attacks that explicitly impersonate an employee name. We speculate that this type of email would be less successful, because the recipient might find it unusual to get an email from a sender name with a generic title, which is not normally used in their company. Nevertheless, our commercial product utilizes other detectors that find "generic titles" as well (see §6). In organization B and E the impersonation classifier successfully detected an impersonation, but the text classifier did not deem the text of the email as suspicious. In both instances, we have since retrained BEC-Guard's text classifiers using the reported emails. In the case of organization C and D, the reported missed email was due to the impersonation of an external colleague (e.g., a vendor the company works with that got impersonated). In §6 we

discuss how to extend BEC-Guard to detect such attacks.

8 Related Work

The growing threat of BEC is widely known and has been described in many in industry and government reports [13, 22, 23]. However, the existing academic work uses very small or synthetic datasets, and suffers from high false positives. In addition, since existing related work is based on limited datasets, it fails to address many of the real-world issues discussed in our paper, such as dealing with the imbalanced dataset, the usage of personal email addresses by employees or "legitimate" impersonations. We believe the reason for the small body of related work is that BEC primarily affects corporate users (not consumers), and it is generally difficult for academic researchers to obtain access to corporate email data.

EmailProfiler [10] builds a behavioral model on incoming emails in order to stop BEC. However, it is based on only 20 mailboxes, has no examples of real-world attacks and does not report false positive rates. In addition, there is prior work on systems that detect emails, which compromise employee credentials with a phishing link [20,45]. There is some overlap between BEC attacks and emails that compromise credentials: in our dataset, 40% of BEC attacks try to phish employee credentials with links. However, the remaining BEC attacks do not contain a phishing link that compromises credentials, and cannot be detected by these systems.

Gascon et al. [14] design a model to stop emails that spoof the domain of the receiver. Similar to BEC-Guard, they base their model on the historical communication patterns of senders. However, in our dataset, spoofing emails represent only about 1% of BEC attacks. Therefore, their model would not catch the other 99% of BEC attacks. The reason domain spoofing represents a small percentage of our dataset, is our dataset only contains emails that were already filtered by an existing spam filter (e.g., Office 365's default filter). Domain spoofing emails contain a mismatch between the sender and reply-to domains, or between the sender domain and the from email envelope. For this reason, traditional spam filters already stop a large number of spoofing emails [33]. In addition, their model is based on a dataset of only 92 mailboxes.

DAS [20] uses unsupervised learning techniques to identify that result in credential theft, which are a subset of BEC attacks. However, it cannot detect attacks that contain only plain text, and is based on a dataset from a single organization with only 19 known attacks. It also suffers from a 0.2% precision, and a much higher false positive rate than BEC-Guard. Similarly, IdentityMailer [45] tries to prevent employee credential compromise by modeling employee behavior, and detecting anomalies in outbound emails. Once an anomaly is detected, the employee is asked to re-authenticate with two-factor authentication. However, their technique suffers from very high false positive rates (1%-8%, compared with 1 in millions of emails in BEC-Guard), and the analysis is based on a small corpus of emails.

Another contemporaneous study done at Barracuda Networks by Ho et al. [18, 19] examines the behavior of attackers using compromised accounts and possible ways to detect account takeover incidents. The techniques presented in this paper are complimentary with the other study, and focus on a different type of attack.

Finally, there is a large body of work on adverserial learning in the context of spam detection [3, 4, 8, 21, 31, 32, 37, 50] that is relevant to our work. In the future, we plan to incorporate some of the evasion techniques introduced in past work, including randomization and the use of honey pots to trick adversaries.

9 Conclusions

BEC is a significant cyber security threat that results in billions of dollars of losses a year. We present the first system that detects a wide variety of BEC attacks at a high precision and false positives, and is used by thousands of organizations. BEC-Guard prevents these attacks in real-time using a novel API-based architecture combined with supervised learning.

One of the main lessons we have learned in developing and deploying BEC-Guard, is that attackers constantly adapt their tactics and approaches. While our supervised learning approach does require continuously retraining our classifiers, and is not fully generalizable, we have found the general approach of using historical email patterns via an API-based architecture has been very useful in quickly developing new classifiers for evolving threats. We have employed a similar approach to the one described in this paper in other contexts, such as detecting brand impersonation, generic sender names and account takeover.

Acknowledgments

We thank Grant Ho, our shepherd, Devdatta Akhawe, and the anonymous reviewers for their thoughtful feedback.

References

[1] R Anglen. First-time phoenix homebuyer duped out of $73k in real-estate scam, 2017. https://www.azcentral.com/story/news/local/arizona-investigations/2017/12/05/first-time-phoenix-homebuyer-duped-out-73-k-real-estate-scam/667391001/.

[2] Manos Antonakakis, Roberto Perdisci, David Dagon, Wenke Lee, and Nick Feamster. Building a dynamic reputation system for DNS. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 18–18, Berkeley, CA, USA, 2010. USENIX Association.

[3] Marco Barreno, Blaine Nelson, Anthony D. Joseph, and J. D. Tygar. The security of machine learning. *Machine Learning*, 81(2):121–148, Nov 2010.

[4] Marco Barreno, Blaine Nelson, Russell Sears, Anthony D. Joseph, and J. D. Tygar. Can machine learning be secure? In *Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security*, ASIACCS '06, pages 16–25, New York, NY, USA, 2006. ACM.

[5] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct 2001.

[6] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: Synthetic minority over-sampling technique. *J. Artif. Int. Res.*, 16(1):321–357, June 2002.

[7] A. Cidon. Threat spotlight: Spear phishing for mortgages. hooking a big one., 2017. https://blog.barracuda.com/2017/07/31/threat-spotlight-spear-phishing-for-mortgages-hooking-a-big-one/.

[8] Nilesh Dalvi, Pedro Domingos, Mausam, Sumit Sanghai, and Deepak Verma. Adversarial classification. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, pages 99–108, New York, NY, USA, 2004. ACM.

[9] Pedro Domingos. Metacost: A general method for making classifiers cost-sensitive. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 155–164. ACM, 1999.

[10] Sevtap Duman, Kubra Kalkan-Cakmakci, Manuel Egele, William Robertson, and Engin Kirda. EmailProfiler: Spearphishing filtering with header and stylometric features of emails. In *Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*, volume 1, pages 408–416. IEEE, 2016.

[11] Luca Invernizzi Elie Bursztein, Kylie McRoberts. Tracking desktop ransomware payments end to end. Black Hat USA 2017, 2017. https://www.elie.net/talk/tracking-desktop-ransomware-payments-end-to-end.

[12] FBI. Cyber-enabled financial fraud on the rise globally, 2017. https://www.fbi.gov/news/stories/business-e-mail-compromise-on-the-rise.

[13] FBI. Business email compromise, the 12 billion dollar scam, 2018. https://www.ic3.gov/media/2018/180712.aspx.

[14] Hugo Gascon, Steffen Ullrich, Benjamin Stritter, and Konrad Rieck. Reading between the lines: Content-agnostic detection of spear-phishing emails. In Michael Bailey, Thorsten Holz, Manolis Stamatogiannakis, and Sotiris Ioannidis, editors, *Research in Attacks, Intrusions, and Defenses*, pages 69–91, Cham, 2018. Springer International Publishing.

[15] John A Hartigan and Manchek A Wong. Algorithm AS 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[16] Alex Hern. Unicode trick lets hackers hide phishing URLs, 2017. https://www.theguardian.com/technology/2017/apr/19/phishing-url-trick-hackers.

[17] L Hernandez. Homebuyers lose life savings during wire fraud transaction, sue Wells Fargo, realtor and title company, 2017. https://www.thedenverchannel.com/money/consumer/homebuyers-lose-life-savings-during-wire-fraud-transaction-sue-wells-fargo-realtor-title-company.

[18] Grant Ho, Asaf Cidon, Lior Gavish, Marco Schweighauser, Vern Paxson, Stefan Savage, Geoffrey M. Voelker, and David Wagner. Detecting and characterizing lateral phishing at scale. In *26th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 2019.

[19] Grant Ho, Asaf Cidon, Lior Gavish, Marco Schweighauser, Vern Paxson, Stefan Savage, Geoffrey M. Voelker, and David Wagner. Detecting and Characterizing Lateral Phishing at Scale (Extended Report). In *arxiv*, 2019.

[20] Grant Ho, Aashish Sharma, Mobin Javed, Vern Paxson, and David Wagner. Detecting credential spearphishing in enterprise settings. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 469–485, Vancouver, BC, 2017. USENIX Association.

[21] Ling Huang, Anthony D. Joseph, Blaine Nelson, Benjamin I. P. Rubinstein, and J. Doug Tygar. Adversarial machine learning. In *AISec*, 2011.

[22] Infosec Institute. Phishing data – attack statistics, 2016. http://resources.infosecinstitute.com/category/enterprise/phishing/the-phishing-landscape/phishing-data-attack-statistics/.

[23] SANS Institute. From the trenches: Sans 2016 survey on security and risk in the financial sector, 2016. https://www.sans.org/reading-room/whitepapers/analyst/trenches-2016-survey-security-risk-financial-sector-37337.

[24] Nathalie Japkowicz. The class imbalance problem: Significance and strategies. In *Proc. of the Int'l Conf. on Artificial Intelligence*, 2000.

[25] M. Korolov. Report: Only 6% of businesses use DMARC email authentication, and only 1.5% enforce it, 2016. https://www.csoonline.com/article/3145712/security/.

[26] Miroslav Kubat, Robert C Holte, and Stan Matwin. Machine learning for the detection of oil spills in satellite radar images. *Machine learning*, 30(2-3):195–215, 1998.

[27] Miroslav Kubat, Stan Matwin, et al. Addressing the curse of imbalanced training sets: one-sided selection. In *ICML*, volume 97, pages 179–186. Nashville, USA, 1997.

[28] M. Lan, C. L. Tan, J. Su, and Y. Lu. Supervised and traditional term weighting methods for automatic text categorization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(4):721–735, April 2009.

[29] David D Lewis and Jason Catlett. Heterogeneous uncertainty sampling for supervised learning. In *Proceedings of the eleventh international conference on machine learning*, pages 148–156, 1994.

[30] Charles X Ling and Chenghui Li. Data mining for direct marketing: Problems and solutions. In *KDD*, volume 98, pages 73–79, 1998.

[31] Daniel Lowd. Good word attacks on statistical spam filters. In *Proceedings of the Second Conference on Email and Anti-Spam (CEAS*, 2005.

[32] Daniel Lowd and Christopher Meek. Adversarial learning. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, KDD '05, pages 641–647, New York, NY, USA, 2005. ACM.

[33] Microsoft. Anti-spoofing protection in Office 365, 2019. https://docs.microsoft.com/en-us/office365/securitycompliance/anti-spoofing-protection.

[34] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 3111–3119. Curran Associates, Inc., 2013.

[35] Yoav Nathaniel. ZeroFont phishing: Manipulating font size to get past Office 365 security, 2018. https://www.avanan.com/resources/zerofont-phishing-attack.

[36] C Northern. Nickname and diminutive names lookup, 2017. https://github.com/carltonnorthern/nickname-and-diminutive-names-lookup.

[37] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The limitations of deep learning in adversarial settings. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 372–387, March 2016.

[38] Michael Pazzani, Christopher Merz, Patrick Murphy, Kamal Ali, Timothy Hume, and Clifford Brunk. Reducing misclassification costs. In *Proceedings of the Eleventh International Conference on Machine Learning*, pages 217–225, 1994.

[39] N. Perlroth. Hackers are targeting nuclear facilities, Homeland Security Dept. and F.B.I. say, 2017. https://www.nytimes.com/2017/07/06/technology/nuclear-plant-hack-report.html.

[40] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[41] J.J. Roberts. Facebook and Google were victims of $100m payment scam, 2017. http://fortune.com/2017/04/27/facebook-google-rimasauskas/.

[42] G. Salton and M. J. Mcgill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[43] Z. Song and N. Roussopoulos. K-nearest neighbor search for moving query point. pages 79–96, 2001.

[44] United States Securities and Exchange Commission. Form 8-k, 2015. https://www.sec.gov/Archives/edgar/data/1511737/000157104915006288/t1501817_8k.htm.

[45] Gianluca Stringhini and Olivier Thonnard. That ain't you: Blocking spearphishing through behavioral modelling. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 78–97. Springer, 2015.

[46] Andrew Trask, Phil Michalak, and John Liu. sense2vec - A fast and accurate method for word sense disambiguation in neural word embeddings. *CoRR*, abs/1511.06388, 2015.

[47] Gary M Weiss and Haym Hirsh. Learning to predict rare events in event sequences. In *KDD*, pages 359–363, 1998.

[48] Colin Whittaker, Brian Ryner, and Marria Nazif. Large-scale automatic classification of phishing pages. In *NDSS '10*, 2010.

[49] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using CWSandbox. *IEEE Security Privacy*, 5(2):32–39, March 2007.

[50] Gregory L. Wittel and S. Felix Wu. On attacking statistical spam filters. In *Proceedings of the Conference on Email and Anti-Spam (CEAS)*, 2004.

[51] Gang Wu and Edward Y Chang. Class-boundary alignment for imbalanced dataset learning. In *ICML 2003 workshop on learning from imbalanced data sets II, Washington, DC*, pages 49–56, 2003.

Cognitive Triaging of Phishing Attacks

Amber van der Heijden
a.v.d.heijden@student.tue.nl
Eindhoven University of Technology

Luca Allodi
l.allodi@tue.nl
Eindhoven University of Technology

Abstract

In this paper we employ quantitative measurements of *cognitive vulnerability triggers* in phishing emails to predict the degree of success of an attack. To achieve this we rely on the cognitive psychology literature and develop an automated and fully quantitative method based on machine learning and econometrics to construct a triaging mechanism built around the cognitive features of a phishing email; we showcase our approach relying on data from the anti-phishing division of a large financial organization in Europe. Our evaluation shows empirically that an effective triaging mechanism for phishing success can be put in place by response teams to effectively prioritize remediation efforts (e.g. domain takedowns), by first acting on those attacks that are more likely to collect high response rates from potential victims.

1 Introduction

Phishing attacks represent a significant threat to organizations and their customers [36]. The problem of phishing detection has been addressed multiple times in the literature [16,18,32], yet classification is only part of the issue. A timely and efficient *reaction* to phishing attempts (e.g. performing takedown actions on phishing domains, blacklisting, or notifying customers) could save hundreds or thousands of customers from fraud or theft, and associated costs for all involved stakeholders. For this reason, most 'large enough' organizations operate a phishing-response team whose task is to promptly investigate potential impacts, identify rogue domains and attack vectors, and act to contain or neutralize the attack [8]. The size of this effort often requires the full time operation of several experts within the response team [46].

Unfortunately, these teams currently lack of an objective and quantitative way of prioritizing response activities, which can lead to large inefficiencies in the response process. Technical mechanisms are often in place to *a-posteriori* quantify the success of a phishing attack, but these are technically limited to attacks 'in scope' of the measuring mechanism (e.g. evaluating the requests for internal resources received by the organization's servers and originating from remote domains) and, importantly, cannot *predict* how successful the attack is likely to be if no immediate mitigation is put in place.

Key to predicting phishing success is the likelihood that a human will *comply* with whatever instruction is in the phishing email. Cialdini pioneered the definition of '*principles of influence*', namely `Reciprocity`, `Consistency`, `Social Proof`, `Authority`, `Liking`, and `Scarcity` as 'cognitive triggers' that, once engaged, can greatly impact the likelihood of a human's decision to comply with what he or she is being requested to do [6]. These principles have been used as a theoretical framework to investigate persuasion in different domains, such as sales and marketing [7], organizational behaviour [41], and wellbeing [51], as well as being linked to phishing effectiveness [53, 54] in (synthetic) experimental settings [55]; however no means to automatically measure the cognitive features of a phishing email, and estimate their relation to phishing success 'in the wild', currently exists.

In this paper we employ techniques from natural language processing and econometrics to build a method and estimation process to measure cognitive triggers in phishing emails, and to build a *cognitive triaging model* of how successful an attack can be expected to be. We demonstrate empirically that the resulting estimations can be used to efficiently prioritize phishing response actions, by addressing first the (few) attacks that are likely to be highly successful. To do this, we extensively analyze more than eighty thousand phishing emails received by the anti-phishing division of a very large European financial organization, quantify the 'cognitive vulnerability triggers' embedded in the attacks, and relate them to the number of accesses to the remote phish domain that the anti-phishing division measured. This allows us to empirically derive a triaging model that, only based on cognitive features of the incoming phishing email, can predict how many 'clicks' it can be expected to generate.

Scope and contribution of this work. With this work we aim at building a principled analysis that explains *why* one can expect a certain phishing email to be successful, as opposed to

building a method that 'blindly' maps mail bodies to success of attack. Importantly, with this work we do *not* aim to build a classifier to distinguish phishing from non-phishing emails; instead, we propose a method to *predict* to what extent a *known* phishing attack can be expected to lure users in falling for it. Our contributions can be summarized as follows:

- we provide the first empirical analysis of cognitive vulnerabilities as exploited in the wild by attackers launching phishing attacks;
- we employ a robust measurement methodology to identify cognitive vulnerability triggers in phishing emails, using supervised Latent Dirichlet Allocation, and a set of bootstrapped econometric simulations to build robust estimations of model coefficients and predictions;
- we show empirically the correlation between exploited cognitive factors and spoofed `From:` addresses with an objective evaluation of phishing success;
- we quantitatively show that triaging phishing emails to prioritize remediation action is possible and effective in an operational setting.

This paper proceeds as follows: Section 2 sets the background for this work in both the cognitive psychology and information security literature; Section 3 details the employed data and methodology, and Section 4 reports the exploratory and cognitive analysis of the data. The cognitive model and predictions are presented in Section 5. Section 6 provides a discussion of our results, and Section 7 concludes the paper.

2 Background and Related Work

The general objective of a phishing attack is to convince a target to comply with a request, such as clicking a link to a phishing domain, downloading malware, or providing personal credentials. The effectiveness of these attacks significantly relies on how quickly the message can generate the desired response [55]. Moreover, both cognitive [54, 56] and technical [27, 30, 42] features are employed to lure users into falling for the phish and are known to be relevant to explain phishing effectiveness.

2.1 Cognitive characterizations

Believability. Phishers apply several techniques to increase believability of their phishing messages. For example, they may craft their phishing messages to resemble communications of the impersonated organizations as closely as possible [56]. This is commonly done by duplicating the look and feel of these communications by including logos and other branded graphics extracted from their legitimate counterparts, and by adopting a formal writing style [15]. Furthermore, the context of phishing messages is generally highly personalized to appeal to the targeted population [55]. These practices are enhanced by more technical measures, such as spoofing of the phishing source address, and the use of shortened URLs to hide the destination of the embedded phishing link [24].

Persuasiveness. Persuasiveness is associated with the text content of the email. These techniques work by exploiting fundamental vulnerabilities of human cognition [31] that can be explained by 'shortcuts' in human cognitive processes that determine decisions on the basis of previous experiences, biases, or beliefs [48]. Despite the clear benefits of these mental-shortcuts, they can result in irrational decision-making as well [52]. Cialdini [6] identified several principles that explain how these mental shortcuts can be exploited for the persuasion of others (e.g. for marketing purposes). Indeed, these principles are applied regularly in multiple domains, including marketing (e.g. to purchase a product or solution) [7], organizational behaviour (e.g. to comply to policies) [41], and health and wellbeing (e.g. to adopt healthy lifestyles) [51]. As these are foundational to human decision-making processes [33], these principles may not be effectively applied to *distinguish* legitimate from illegitimate resources (e.g. a website, email, or conversation): any activity aiming at 'influencing' one's behaviour (that being through spam or organization policies, phishing or advertisement) will employ some variation of these principles. On the other hand, these provide a solid foundation to evaluate how *effective* an attempt at convincing a human can be expected to be. Table 1 provides examples and definitions of these principles.

Cialdini's principles of persuasion are strongly related to the successfulness of face-to-face social engineering efforts in the real world [44] as well. Akbar [2] performed a quantitative analysis on 207 unique phishing emails to identify the application of Cialdini's persuasion principles in phishing emails. The results show the `Authority`, `Scarcity` and `Liking` principles to be most popular. A similar study was performed by Ferreira et al. [12], who found the Liking principle to be most popularly used, followed distantly by the principles of `Scarcity` and `Authority`. Differences can be explained by different experimental settings and application domains. Several other studies [3, 13, 55] have addressed the prevalence and efficacy of Cialdini's principles in phishing attacks. Others have evaluated phishing campaigns against specific users [23], discussing some of the techniques used by phishers to lure their victims. Unlike these works, we integrate quantitative measures of cognitive attacks and measures of phishing success to predict attack effectiveness in operational settings.

2.2 Phishing effectiveness

Previous work considered the inclusion of forged quality marks, images, and logos from trusted organizations as well as other signals of credibility as means to increase the effectiveness of a phishing attack [9]. Other more technical measures are employed to enhance the credibility of phishing as well, for example spoofing of the source email address, adoption

Table 1: Definitions and examples of Cialdini's principles of influence in phishing emails

Principle	Definition [6]	Phishing text example [2]
Reciprocity	Tendency to feel obliged to repay favours from others."I do something for you, you do something for me."	"While we work hard to keep our network secure, we're asking you to help us keep your account safe."
Consistency	Tendency to behave in a way consistent with past decisions and behaviours. After committing to a certain view, company or product, people will act in accordance with those commitments.	"You agreed to the terms and conditions before using our service, so we ask you to stop all activities that violate them. Click here to unflag your account for suspension."
Social Proof	Tendency to reference the behaviour of others, by using the majority behaviour to guide their own actions.	"We are introducing new security features to our services. All customers must get their accounts verified again."
Authority	Tendency to obey people in authoritative positions, following from the possibility of punishment for not complying with the authoritative requests.	"Best regards, Executive Vice President of <company name>"
Liking	Preference for saying "yes" to the requests of people they know and like. People are programmed to like others who like them back and who are similar to them.	"We care for our customers and their online security. Confirm your identity .. so we can continue protecting you."
Scarcity	Tendency to assign more value to items and opportunities when their availability is limited, not to waste the opportunity.	"If your account information is not updated within 48 hours then your ability to access your account will be restricted."

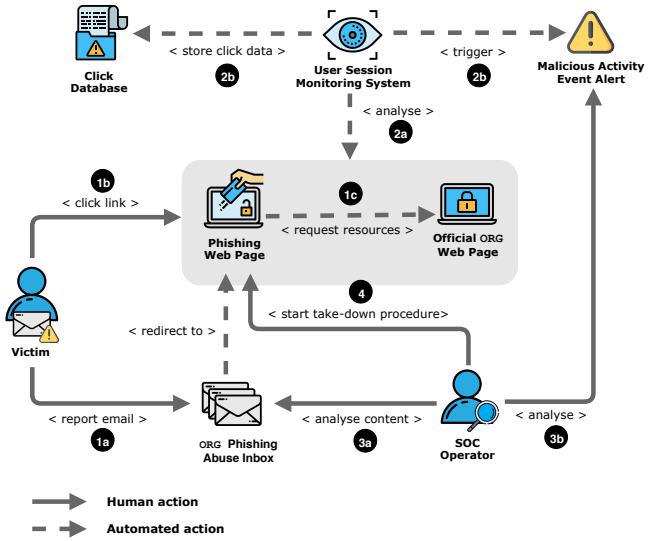
[2] Examples drawn from anti-phishing database at http://www.millersmiles.co.uk.

of HTTPS instead of HTTP to convince the user the webpage is 'safe' [36], or cloning of the original webpage. Several works have considered such visual similarities between phishing landing pages and their legitimate counterparts based on different features, including DOM tree structures [42], CSS styling [27, 30], content signatures [1, 17], and pixel and/or image properties [5, 10]. Whereas these technical features constitute additional relevant information for the identification of a phishing attack, in this study we focus on the cognitive attacks embedded in an email text (as opposed to the visual clues included in a landing webpage) that affect the human decision making. Additionally, a number of user-studies has been conducted on the impact of client-side detection-assistance tools [20, 57] and how people evaluate phishing web pages [9]. Various phishing detection mechanisms have been proposed based on technical features such as signatures of user email behaviour [49], email-header properties [16], impersonation limitations of attackers [28], search engine rankings [25], and botnet effects [35]. Additionally, [29] presents a set of research guidelines for design and evaluation of such detection systems. These works have predominantly focused on the detection of phishing domains and emails by means of technical traces in order to prevent phishing attacks from happening in the first place. Unlike these studies, we focus on the evaluation of the potential of those attacks that, despite the countermeasures in place, make it through and must be timely addressed.

On the cognitive-side we can consider the impact of user demographics. Oliveira et al. [34] found age to be an important feature, finding younger adults to be more susceptible to Scarcity, whereas older adults were more susceptible to Reciprocity. Other results of this study indicate the relevance of gender by finding older women to be most susceptible of all of the studied user groups. Furthermore, Wash and Cooper [53] demonstrated the impact of message presentation by showing how phishing training methods based on giving facts-and-advice were more effective when presented by an expert figure (Authority), whereas methods based on personal stories benefited more from presentation by people perceived as similar to the user (Liking). In the context of social media, user activity, consumption behaviour, and clicking norms in the social network were found to be important factors for phishing success [40]. As opposed to focusing on the characteristics of the individuals that receive the phishing (as this information for the population of customers is generally unknown to organizations, or may be impossible to collect due to legal and ethical challenges), in this work we consider the expected aggregate responses of the phishing recipients as a function of the phishing emails.

3 Methodology and Data collection

Our analysis relies on a unique dataset from a large phishing email database provided by Org, a large financial organization in Europe with more than 8 million customers and a multi-billion Euro turnover. Org customers that suspect they have received a phishing email in their personal email accounts are instructed by the organization to forward these emails to an internal Org functional mailbox. In parallel, Org's phishing response team runs a service to detect phishing domains (not necessarily linked with the received phishing emails) by means of internal heuristics and limited to external domains requesting resources internal to Org (e.g. images, forms, logos, CSS files/javascript, etc.). This data is generated by a third party service hired by Org that monitors *all* requests generated towards Org's resources. Through this mechanism Org can detect the number of visits to the detected domains by accounting for the unique sessions opened between the (rogue) external and the (legitimate) internal services. Access to this data allows us to perform a rich analysis of the arrival of phishing emails, their characteristics, and to evaluate how often users have accessed malicious domains as a proxy mea-



SOC Operators collect evidence on the maliciousness of the web domain under investigation such that an external party can perform the notice and take-down requests for the malicious domains.

Figure 1: Overview of phishing-related activities at `Org`

sure of 'phishing success'. Figure 1 depicts `Org`'s internal process to handle suspect phishing emails.

Overall, we extracted 115,698 reported emails and 11,936 alerts for malicious links between February 1st, 2018 and 15 December 2018, with the exception of the period August-September 2018 due to infrastructural limitations at `Org`. For this same reason, our sample only includes data for 'clicks' collected from end of July onwards.

Data limitations and ethical aspects. From the data structure, the link between a clicked URL and the specific email from which that click originated is not explicit and can only be reconstructed by exact match of the destination URL. This has the effect of limiting the scope of this study to the comparison of the effectiveness of cognitive influence techniques between phishing emails that are likely to have generated the click (as we cannot fully reproduce the process generating the detection of URLs that *could* have been clicked, but have not). This also limits the number of matches between URLs reported in event alerts and URLs linked in emails. Further, the results of this work are limited to the emails that have been *reported* (and therefore identified at least once) by `Org`'s customers. Despite the large number of active reporting customers, particularly well-crafted emails may not be represented in our dataset. Further, we can only observe data captured by the *User Session Monitoring System*, i.e. related to emails pointing to domains that 'call back' to `Org`'s systems. This may represent a limitation if emails that do not 'call back' also exploit different 'cognitive vulnerabilities', or with different distributions. However, an analysis on the available data does not show apparent biases between emails for which a 'click' has been recorded, and those for which we do not know of any (ref. Figure 10). These limitations are akin to those outlined by Pitsillidis *et al.* [37]. Aware of these, we compensate by means of the analysis methodology that explicitly accounts for the potential biases in the data. Finally, the collected data did not contain sensitive subject information and all data handling has been performed within allowance from `Org` and within the scope of work previously approved by the department IRB.

3.1 Data sanitization and processing

As our email dataset contains messages forwarded by users, we first sanitize the data by removing mobile text messages ($n = 18,817$) that likely result from erroneous forwards to the functional mailbox from a related banking service; as they are irrelevant in our setting, we discarded them. Further, users may have reported emails that target financial organizations different from `Org`. To capture this, we identify targeted organizations in our dataset by a string search operation within email bodies for the names of the most prominent financial organizations in the country where `Org` is located, and remove all records that do not belong to `Org` ($n = 15,623$). To identify phishing email subjects, dates, and recipient/sender information, we recursively searched through each raw email message to find header matches of the first original email arrived in the user's inbox,[1] and extract information on `From`, `To`, `Date`, and `Subject` values. Table 2 reports summary statistics of the final dataset.[2]

3.1.1 Identification of suspicious and landing URLs

Suspicious URLs. We check emails for the presence of suspicious URLs that point to any domain that does not belong to `Org`, as these would not normally appear in a legitimate email originated by the organization. We exclude from the heuristic general-purpose domains with no direct phishing correlation (e.g. `youtube.com`). Based on this classification we flag emails that contain at least one suspicious URL as `Suspicious`, whereas the remaining ones are considered uninteresting within our scope (as we can neither count nor estimate clicks for URLs that do not exist).

Landing URLs. These are landing URLs that load resources internal to `Org`, as detected and reported by the *User Session Monitoring System* (ref. Fig.1). Whereas they are related to a click on a suspicious URL, this relation is not immediate in the data and needs to be reconstructed.

[1] This is necessary as emails can be forwarded multiple times (e.g. if originally forwarded by the customer to an `Org` employee) before ending up in the phishing inbox.

[2] We notice that the upper 2.5% of the distribution of email length is disproportionally long w.r.t. the remainder of the distribution, suggesting a few outliers in the data. Manual inspection reveals malformed email corpora (e.g. with HTML tags embedded in the body); as no obvious 'upper limit' for email length is apparent, we keep these in the dataset for the sake of transparency.

Table 2: Descriptive statistics of the collected dataset

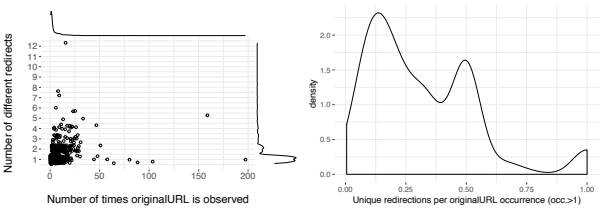
The column `type` indicates whether the variable is a factor (f) or numeric (n). The column `n` reports number of levels for factors, and number of records with at least one observation for numerical variables. We do not report summary statistics for factors. The standard deviation for variable `Date` is reported in days. All dates are in 2018 and in format $\%m-\%d$.

			Feb-Jul 2018						Oct-Dec 2018					
	Variable	type	n	Min	0.025q	Median	0.975q	Max	n	Min	0.025q	Median	0.975q	Max
	Language	f	3						2					
	To	f	38760						2239					
	From	f	1641						330					
	Date	n	69800	02-02	03-07	05-30	07-28	07-31	11458	10-01	10-01	11-29	12-11	12-11
	Length	n	69800	160	446	1068	3973	67246	11458	173	329	1320	5480	15685
Vuln. triggers	Reciprocity	n	69800	0	0	3	67	149	11458	0	0	2	37	153
Vuln. triggers	Consistency	n	69800	0	0	13	84	132	11458	0	0	19	88	176
Vuln. triggers	Social Proof	n	69800	0	0	2	17	52	11458	0	0	0	17	90
Vuln. triggers	Authority	n	69800	0	0	5	55	121	11458	0	0	5	27	83
Vuln. triggers	Liking	n	69800	0	0	0	7	504	11458	0	0	0	8	198
Vuln. triggers	Scarcity	n	69800	0	1	40	107	157	11458	0	0	11	91	189
	Spoof dist.	n	61911	0	0	7	14	23	10604	0	0	6	14	24
	Clicks	n	4	9	9	28.5	78	78	35	1	1	37	220	220
Emails	Reported	f	69800						11458					
Emails	of which susp.	f	61079						9419					
Emails	Unique	f	1293						424					
Emails	of which susp.	f	952						329					

3.1.2 Landing URL extraction

To reconstruct the association between Landing URLs and Suspicious URLs we adopt the following method:

1. First, we traverse the suspicious URL embedded in the phishing email (`suspiciousURL`) multiple times by visiting all URLs arriving to `Org`'s inbox. These typically generate a number of redirections (generally HTTP 3xx) that lead to a landing webpage, where the actual phishing resource is located. We record the association $\langle$ `suspiciousURL, landingUrl` $\rangle$ for *all* visited URLs, and for *all* emails; if the redirection mechanism is not deterministic, we obtain a 1 *to n* association between `suspiciousURL` and a set of `landingURL`s. As we cannot know how many 'redirection chains' exist from a single `suspiciousURL`, we traverse the URL opportunistically every time it appears in `Org`'s inbox. To minimize confoundings in the redirection, each visit session is independent from the previous. Figure 2 shows that the number of different redirections stops growing quickly regardless of how many time we traverse a given URL, suggesting that the dataset of collected `landingURL`s does not suffer from systematic censoring problems.
2. When `landingURL` is visited, a third party contractor of `Org` records a 'click' for `landingURL` (see Fig 1 and discussion in *data limitations*), and reports it to `Org`.
3. We link clicked `landingURL`s with the original email body by matching them with the `landingURL`s we found by traversing the `suspiciousURL`s in the mail corpus; if there are multiple clicked `landingURL`s for a single `suspiciousURL`, we keep record of all matches.



The redirection count for all observed `suspiciousURL`s (left) shows that new `landingURL`s stop appearing after only few `suspiciousURL` visits. The density plot on the right shows ratio of unique `landingURL` per `suspiciousURL` for `suspiciousURL`s visited more than once, and confirms that new redirects stop appearing regardless of number of visits.

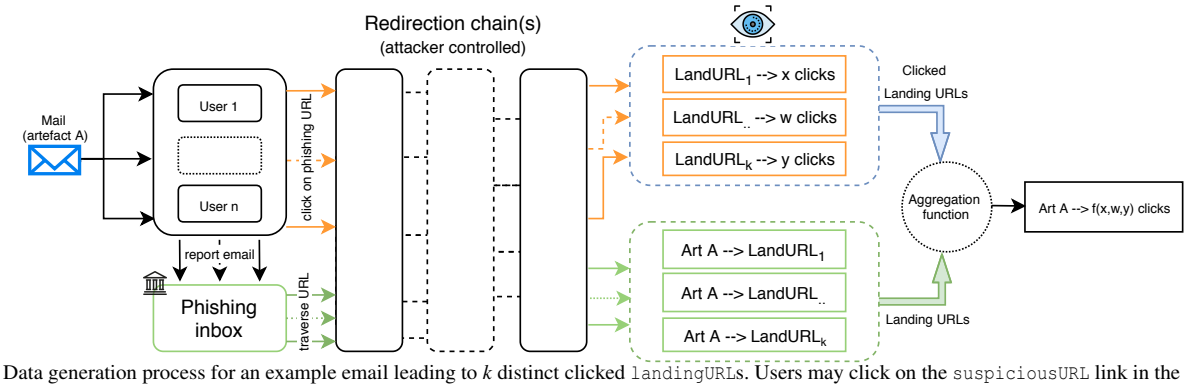
Figure 2: Redirection count (left) and density ratio (right) from observed `suspiciousURL`s.

4. To aggregate clicks to a single `suspiciousURL`, we considered: average, sum, and max no. of clicks across all `landingURL`s for a given `suspiciousURL`. We ran our experiments using all aggregation strategies, and obtained qualitatively identical results. In this paper we report average clicks as it is the most conservative choice to make (e.g. summing `landingURL` clicks is more susceptible to over-reporting multiple clicks by the same user).

Figure 3 provides a bird's eye view of the data generation process for the landing URL extraction.

3.1.3 Duplicate detection

One complexity of our unstructured dataset is the possible occurrence of multiple duplicates of the same suspect phishing email. In this paper we consider 'similar' emails received by



Data generation process for an example email leading to k distinct clicked `landingURL`s. Users may click on the `suspiciousURL` link in the email and, usually through a series of redirections, reach the phishing domain hosted at one of the `landingURL`s. Through the dynamics described in Figure 1, an association between each distinct `landingURL` and recorded number of clicks is reported. The same email can also be reported to `Org`'s phishing inbox. When it arrives, we opportunistically traverse the redirection chain and record the association between the original email and the final `landingURL`(s). To reconstruct the association between `suspiciousURL`s and clicked `landingURL`s, we aggregate the two datasets.

Figure 3: Data generation process for matched URLs and click data aggregation

users over long periods of time as belonging to the same 'campaign'.[3] Although the overall textual content of these duplicate emails is similar, they can still contain slight differences, for instance because of the presence of a recipient's name in the salutation of an email or other minor syntactic features. In order to detect, and subsequently remove, as many of these duplicate emails as possible, we used a fuzzy string matching approach to determine the pairwise similarity for each of the emails in our dataset. We employ a *bag-of-words* model to calculate, for each document, the frequency of each unique word in the document. We build the word-by-document matrix of our email corpora for the term frequency values for all emails in our dataset. As an additional pre-processing step all input was cleaned by removing special characters, urls, email addresses and line breaks from the text. We use L^2 normalization to the term frequencies to limit the impact of differences in email lengths [47].

To evaluate email similarity we employ a measure of cosine similarity. This similarity measure expresses the similarity between two vectors in terms of the cosine of the angle between the two vectors; the evaluation results in a score between $[0,1]$, where 0 constitutes low similarity, and 1 constitutes high similarity. To define the cutoff threshold for similar emails we manually marked 300 randomly sampled emails from the dataset and assigned them to 'similarity IDs' to track which emails were replicas of which others. We then performed a bootstrapped ($n = 100,000$) sensitivity analysis of the threshold level to determine the optimal level for the cutoff. This procedure tunes the categorization to very satisfactory sensitivity and specificity levels higher than 90%. Full details on procedure and results are reported in the Appendix.

The duplicate detection procedure identifies 1,293 and 424 unique emails in the data collection of Feb-Jul 2018 and Oct-Dec 2018 respectively (ref. Table 2). Of these 952 and 329 respectively are classified as 'suspicious'.[4]

[3] This is only based on the email text, and we use it as a term to group together emails that are likely to have a common denominator (e.g. a phishing tool, a specific market/phishing pool, or actual attacker).

Table 3: Topic model performance results

We perform LLDA using Gibbs sampling iterations for parameter estimation and inference initialised with hyper parameters $\alpha = 1.0$, $\beta = 0.001$, $k_{labels} = 6$ and $N_{iterations} = 1000$.

	Macro (sd)	Micro (sd)
Sensitivity	0.709 (±0.016)	0.807 (±0.016)
Specificity	0.714 (±0.042)	0.813 (±0.038)
Precision	0.718 (±0.025)	0.755 (±0.024)
F1	0.725 (±0.020)	0.760 (±0.020)

3.2 Cognitive evaluation

To identify the presence of cognitive vulnerabilities in email bodies and the *intensity* of the employed cognitive attacks, we construct a supervised topic model based on Labeled LDA [39] (LLDA). LLDA models each input document as a mixture of topics inferred from labeled input data and outputs probabilistic estimates of label-document distributions, i.e $P(label_t|document_m)$, and word counts of label-specific triggers for each input document. In our application the labels correspond to Cialdini's principles of influence, detailed in Table 1, whereas documents correspond to the emails.

For model training, we randomly sampled 99 emails (38 with clicks and 61 suspicious) out of the set of unique and suspicious emails in the dataset ($n = 1,281$),[5] and manually

[4] Note that otherwise identical emails may lead to different phishing domains.

[5] To have an indication of the effect of sample size on model performance, we first ran the training on 70 emails and added 29 (+40%) at a second time, obtaining virtually identical results. To rule out sampling issues, we also performed a cross-validation procedure (reported) which suggested stable

labelled them for presence of cognitive vulnerabilities. Due to language restrictions, we adopted a mixed approach whereby one author performed the labelling on the original data, and the second author blindly re-performed the labelling on an automatically-translated random sample (20 emails) of the labelled data. To assess model performance we performed a 5 times repeated 5-fold cross validation over the data. Numerous approaches exist to evaluate the performance of multilabel classification problems like ours. Following [43], we consider our problem as a label-pivoted binary classification problem, where the aim is to generate for each label strict yes/no predictions based on the document ranking for that label. For each label, we sort on the per document prediction values, and use the `PROPORTIONAL` method [14, 43] to define a rank-cutoff value that determines the top N ranked items that will receive a positive prediction. For each label, we set $TOPN_i$ equal to the expected number of positive predictions based on training-data frequencies: For label l_i, $TOPN_i = ceil\left(\frac{N^d_{test}}{N^d_{train}} * N^{train}_i\right)$ where N^d_{train} and N^d_{test} refer to the total number of training and testing documents and N^{train}_i is the number of training documents assigned label l_i.

We have aggregated the performance results of our topic model using the `PROPORTIONAL` rank-cutoff method in Table 3. Unlike other rank-cutoff methods, this approach relies solely on labeling information from the training set, which makes it appropriate for use in real-world production settings as well. We report both macro scores (averages computed over each result of the cross-validation procedure), and micro scores (computed over the aggregate of all cross-validation results). The obtained scores indicate a satisfactory fit over both projections. A manual analysis on randomly sampled emails confirms that the procedure appropriately assigns 'topics' to emails. The final model is trained on the complete set of 99 labeled training documents that were previously used in cross-validation, and then applied to the unseen and unlabeled remainder of the full dataset. Standard text cleaning procedures have been applied for removal of special characters and stop-words, sentence tokenization, and word stemming.

In this paper we refer to the 'topics' assigned by LLDA to an email as the **cognitive vulnerabilities** exploited in that text, and to the words associated with that topic and present in the text as the **vulnerability triggers** for that cognitive vulnerability. With this we aim at distinguishing the *presence* of a cognitive attack from its *intensity* in the email text.

Example of training results We report below an example of a phishing email (translated to English) and its association with different cognitive vulnerabilities. We have indicated the relevant vulnerability triggers in *italics* and refer to (1) `Liking`, (2) `Consistency`, (3) `Authority`, (4) `Social Proof`, (5) `Reciprocity` and (6) `Scarcity`:

> *(1) As a valued customer of* `Org` we always want to inform you of the latest updates and innovations in our system. We have recently switched to a new system that requires *(4) all current customers* to replace their *(2) current debit cards* by our newly-produced ones.
>
> In connection with the new changes to the *(3) European Safety Regulations*, `Org` wishes to alert all its customers to the availability of the new and improved debit cards that adhere to all *(3) environmental and safety regulations*.
>
> *(1)* `Org` *strives to be environmentally friendly*. Therefore, our service team will recycle all current debit cards by mounting your *(2) current AES Encryption Chip* on your renewed biological RFID payment card. For this reason, all current payment cards must be replaced. *(5) By participating in our recycling program, the new debit card can be requested free of charge. (6) After October 19th, 2018, a direct debit will be charged.*

Table 4: Example of extracted keywords for each topic

Reciprocity		Consistency		Social Proof	
Word	p	Word	p	Word	p
free	0.024	update	0.026	all	0.035
participate	0.016	improve	0.024	customer	0.011
program	0.011	recycle	0.018	current	0.005
request	0.010	renew	0.015	require	0.004
Authority		**Liking**		**Scarcity**	
Word	p	Word	p	Word	p
safety	0.017	valued	0.022	after	0.031
regulate	0.013	friendly	0.012	charge	0.027
european	0.010	strive	0.008	direct	0.020
must	0.007	environment	0.005	debit	0.019

From the example we can observe that the different cognitive vulnerabilities often appear alongside each other, and that a single vulnerability can even occur multiple times within an email body. Table 4 reports an excerpt of the classification results for the above message, and the learned keywords (translated in English) for each topic.[6]

4 Exploratory analysis

In this section we provide an exploratory analysis of the obtained email data set reported in Table 2.

We first give a look at the time of suspicious email arrivals in victims' inboxes. Figure 4 reports the CDF distribution of email arrivals to `Org`'s phishing inbox. We observe a steady arrival rate through April and the first cutoff date in July 2018, suggesting that email arrival is approximately constant and uniformly distributed in time. As per the time of day of their arrival (not depicted here for brevity) we observe that few suspicious emails arrive in the users' inboxes during the

results. Finally, manual checks on a random sample from the dataset of predicted labels found no obvious miscategorization.

[6] As the original text is not in English, to provide an accurate translation we report keyword matches for an example.

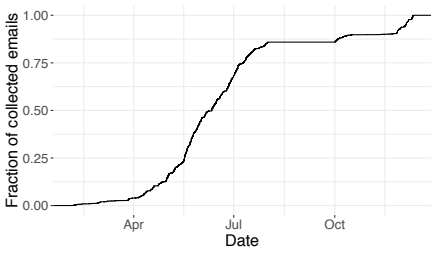


Figure 4: Arrival of notified emails to `Org`'s inbox

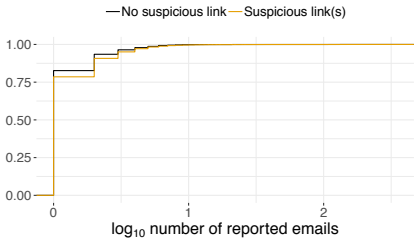


Figure 5: CDF of emails reported by victim addresses

weekend, with most phishing activity happening during the working days. This may suggest a strategic aspect of these campaigns aimed at increasing the credibility of the email source. On this same line, we find that most emails arrive between 9am and 5pm (business hours), and most arriving between 9am and 11am. Interestingly, these findings are all in line with optimal email send days and times for newsletters as reported by analyses from multiple popular online email marketing services [4, 26, 38, 45], and is an indication that attackers may follow similar strategies.

4.1 Spoofing and victimization

Figure 5 depicts the distribution of suspicious and non suspicious reported emails. The CDF is on a log scale to better represent the distribution's log tail. The vast majority of users report only one email, with almost all reporting less than 10 emails. This suggests that the distribution of phishing emails is uniform across victims, as is generally the case with untargeted phishing attacks [23, 36]. Only 122 addresses out of about 40 thousand report more than 10 emails, and only nine report more than 100 emails.

Figure 6 reports the distribution of spoofed and non-spoofed `From:` domains for reported emails with and without a suspicious URL in the body. An email is classified as spoofed based on the Levenshtein distance of the (spoofed) `From:` domain the original attack was sent to, w.r.t. the actual name of the organization. This captures exact string matches as well as small variations that may remain undetected by the user [50]. We find attacks employing a range of domains resembling `Org`'s: from less similar (e.g. `org-safety.com`,

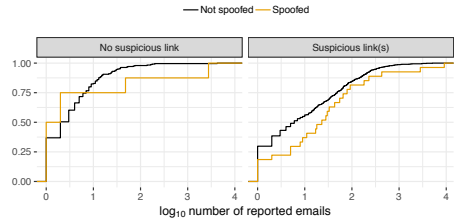
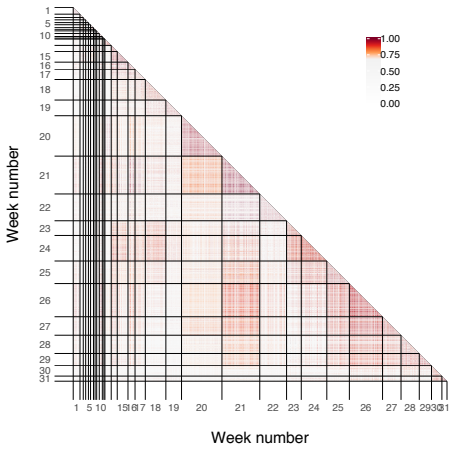


Figure 6: CDF of spoofed and non-spoofed `From:` domains



For visualization purposes we report random samples per week of 10% of the emails received in that week. Red represent high similaries above the threshold. We do not observe specific cycles of similar emails, suggesting that any sufficiently long period of time (3-4 weeks) would cover a diverse set of phishing attacks.

Figure 7: Pair-wise cosine similarity between email samples

`org-customersupport.com`), to more closely spoofed domain variations (e.g. `theorg.com`, `0rg.com`). We observe a clear differentiation, whereby emails with no suspicious URL are approximately as likely to have a spoofed `From:` address as a non-spoofed one. On the other hand, emails with suspicious URLs are more likely to be delivered from non-spoofed than from spoofed addresses, as can observed from the areas under the two curves. This is compatible with a model of a relatively unsophisticated attacker. Here it is also relevant to consider that the pool of 'spoofed' addresses is much smaller than the pool of 'non-spoofed' addresses (as there are many fewer viable choices similar to `Org` than otherwise), suggesting that as spoofed domains get blacklisted, attackers may be forced to move to less well-spoofed `From:` addresses.

4.2 Phishing campaigns

Figure 7 reports a visualization of the similarity scores between emails received during the observation period. Dark red indicates high similarity.[7] We do not observe specific and sys-

[7] For details on the identification of similar emails see the Appendix.

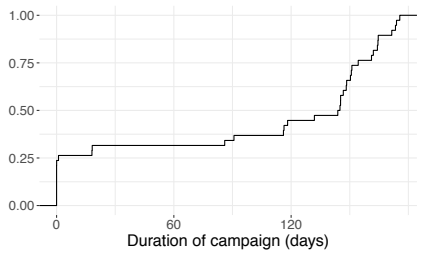
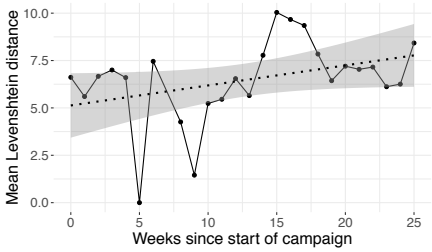


Figure 8: Duration of a phishing campaign



As phishing campaigns progress, the spoofed `From:` domains appear to be more dissimilar w.r.t. the original domain.

Figure 9: Average weekly decrease in similarity between spoofed domains and name of target organization

tematic cycles of campaigns emerging with repeating patterns across several weeks. This also suggests that any sufficiently long observation period (in the order of 3-4 weeks) may suffice to collect a diverse set of attacks for analysis. A first look suggests that some attacks seem to re-appear after a few weeks in slightly different forms, perhaps to increase chances of passing updated spam filters (see for example emails from week 21 reappearing slightly modified in week 26, or those from week 18 reappearing in week 24). To evaluate this, Figure 8 reports the distribution of suspicious emails that likely belong to the same campaign. Most campaigns are relatively long, with approximately 50% of similar emails arriving more than 120 days apart, and 25% of emails arriving more than 150 days apart with a relatively long left tail. From the distribution it appears that *single-day* campaigns are relatively common, whereas long campaigns extend for more than 100 days. Mid-range campaigns lasting between 2 and 100 days are by comparison only few, suggesting that attacks may either be extremely quick and disappear the next day, or last for long periods. Table 5 reports summary statistics of suspected phishing campaigns. We identify 38 distinct campaigns lasting on average 150 days (approx 5 months) and up to 175 days in the observation period.

To investigate how address spoofing evolves during campaigns, Figure 9 reports the weekly average similarity between the domain of the attacker `From:` address and the domain of the victim organization (measured as their Levenshtein distance) for `LONG` campaigns. Lower scores indicate more closely spoofed domains. We observe an average increase in dissimilarity between spoofed `From:` addresses and organization domain, which suggests an overall deterioration of a phishing campaign as it progresses or is replicated by phishers ($cor = 0.31, p = 0.08$). This is in line with the intuition that spoofed domains are limited in number, and attackers may therefore run out of options as domains get blacklisted as the campaign progresses.

4.3 Cognitive effects

Figure 10 reports the distribution of triggered cognitive vulnerabilities in each unique email (left) and the corresponding vulnerability triggers identified in the corpus (right). We observe a clear relation between the two plots: the most common vulnerabilities and triggers in emails appear to be linked to the `Consistency` and `Scarcity` vulnerabilities, regardless of whether a 'click' has been recorded for that link or not. `Liking` and `Social proof` triggers appear to be particularly rare on the average, with most emails targeting none.[8] This is consistent with the intuition that in one-shot interactions (as opposed to prolonged or repeated exchanges as in spear-phishing attacks [23]) cognitive attacks linked to the target's social context and personal preferences (ref. Table 1) are rare. By contrast, exploiting `Consistency` may only require reference to previous actions that the group of potential victims will have likely performed, such as buying an insurance or receiving a debit card from the organization. `Authority` appears to be a relatively common trigger in our sample, albeit not for all emails. Common triggers here refer to European and national-level legislation and often come together with the threat of a punishment if certain actions are not completed. Overall, we find that few cognitive triggers are present in the median email, suggesting that the median reported attack may not be highly effective, whereas few emails embed more 'intense' cognitive attacks.

Effect of cognitive vulnerablities on phishing success. To evaluate the effect of the cognitive features of the email(s) embedding the 'clicked' URL links, we first report in Figure 11 the distribution of average clicks generated by emails for which at least one click has been recorded ($n = 40$). Most emails generate fewer than 150 clicks, with two emails generating more than 200 clicks ($min = 1$, $median = 37$, $max = 220$, $sd = 51.9$). Figure 12 displays the relation between triggered cognitive vulnerabilities and generated clicks, for which we observe a clear positive relation.[9] Following common prac-

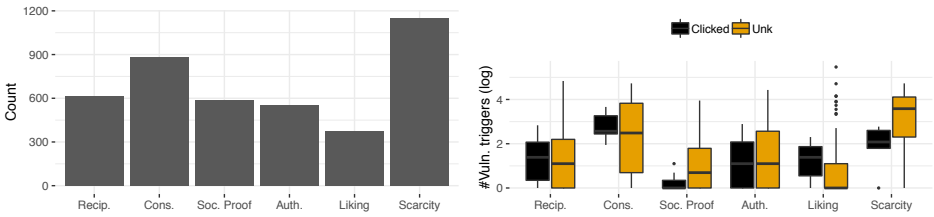
[8] The descriptive statistics reported in Table 2 also suggest stable distributions between the collection periods; for `Liking` we observe more extreme values (upper 97.5% = 7, $max = 504$ in the Feb-Jul data collection); this is caused by the outliers in the email corpora for which we measure disproportionate email lengths.

[9] A possibility is that some emails may be distributed to substantially more users than other emails, generating greater aggregate click counts. As we have no access to the victim's inboxes, we cannot directly measure this.

Table 5: Descriptive statistics of duration and intensity of phishing campaigns

SINGLE-DAY campaigns last up to one day; SHORT campaigns up to 100 days; LONG campaigns more than 100 days. Most phishing campaigns are either very short (one day) or long, with only a handful lasting more than one day but less than 100.

		Phishing samples (#reported emails)							Campaign duration (days)						
Type	n	Min	1stQ	Mean	Med	3rdQ	Max	sd	Min	1stQ	Mean	Med	3rdQ	Max	sd
SING.	10	1	1.0	1.3	1.0	1.0	3	0.7	0.0	0.0	0.1	0.0	0.0	1.0	0.3
SHORT	4	2	2.0	36.0	3.0	37.0	136	66.7	18.1	18.2	53.3	52.1	87.2	90.8	40.6
LONG	24	46	86.2	783.4	226.5	929.5	4827	1207.5	116.1	145.2	150.9	150.6	164.2	175.6	17.3



Most phishing attempts trigger Scarcity, Consistency, and Reciprocity vulnerabilities. Social Proof, Authority, and Liking are the least common. Relative frequency of cognitive vulnerabilities is reflected in the distribution of vulnerability triggers identified in the emails. We do not identify specific biases in presence of vulnerability triggers between emails for which a 'click' has been registerd, and emails for which it has not (i.e. that received an unknown number of clicks).

Figure 10: Distribution of triggered cognitive vulns. (left), and of vuln. triggers (right) for emails

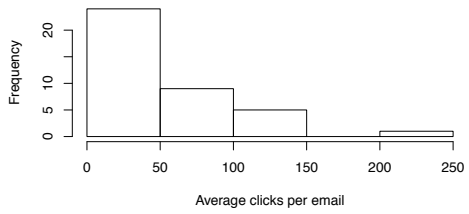
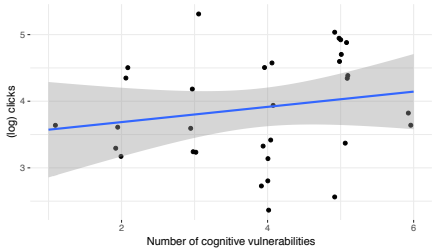


Figure 11: Histogram distribution of clicks per email



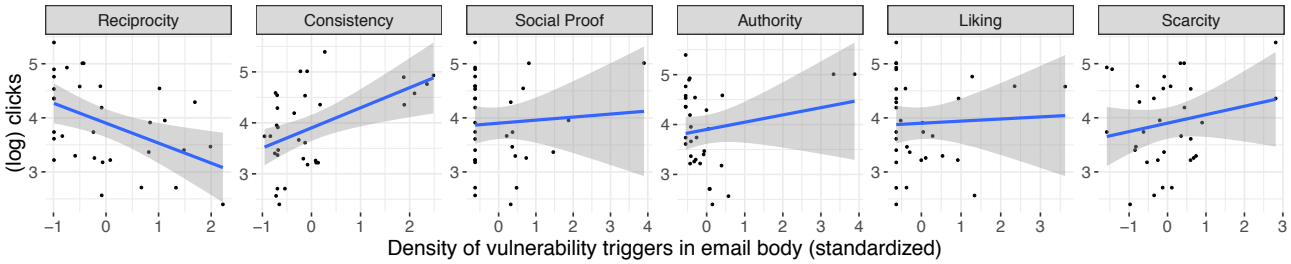
We observe a clear relation between the presence of exploited cognitive vulnerabilities and the clicks generated by the embedded URL(s). The shifted position of points in the pictures is to clear overlaps and is only presentational.

Figure 12: Relation between number of cognitive vulnerabilities in an email and average clicks (log_{10})

tice [21], to avoid dispersion we here only consider URLs clicked at least ten times, removing six emails. A simple Poisson regression of the form $log(clicks_i) = \alpha + \beta(cogvulns_i)$ reveals a strong positive correlation between the variables ($\beta = 0.12, p < 0.001$). This suggests that the more cognitive vulnerabilities are exploited in an email body, the more that email can be expected to generate compliant user behaviour, even when not considering the type of cognitive attack, or its intensity.

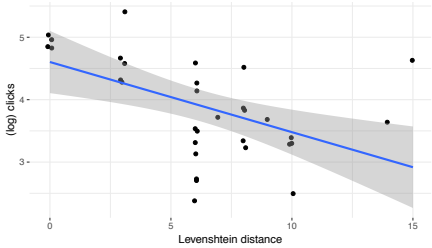
However, the data does not show specific biases in the likelihood of users reporting emails (Figure 5), suggesting that major skews are not realistic. This is consistent with previous findings in the literature [36, 58]. Further, due to the very low click-through rates of spam and phishing campaigns [19], this difference should be of several orders of magnitude to have a visible effect (as opposed to be undetectable noise in the data generation process). Regardless, in the Appendix we build a data generation model to evaluate the effect this bias would have in the data if present; our analysis finds no evidence.

Effect of vulnerability triggers. We now consider the relation of the intensity of each cognitive attack (i.e. measured by the presence of vulnerability triggers) with the measured 'success' of the phishing email. Figure 13 reports the results. The data reports a clear positive relation between Consistency, and Scarcity vulnerability triggers with the expected (log) number of clicks. Reciprocity shows a negative relationship. Additionally Social proof, Liking and Authority show no evident effect, whereby the majority of emails have relatively small counts of associated vulnerability triggers (see also Figure 10). On the other hand, looking at the right extreme of the scale, the few available data points are always related to highly-clicked emails; this may indicate that triggering these vulnerabilities (in this application domain) may be particularly difficult, for example as decisions related to



The data shows the effect of different cognitive vulnerability triggers on expected number of clicks. `Consistency` and `Scarcity` have a clear positive association with the expected number of clicks they generate. `Social proof`, `Authority` and `Liking` do not show any evident trend. Interestingly, we find that `Reciprocity` appears to be counterproductive.

Figure 13: Correlation between vulnerability triggers and observed clicks



We identify a negative relation between the dissimilarity of the spoofed `From:` domain in an email against the original one, and the expected number of clicks the email entices.

Figure 14: Relation between spoofing dissimilarity and average clicks (log_{10})

personal finance may have a smaller attached 'social' component, or as adding additional 'authoritative' effects in the banking domain may be challenging for an attacker.

Effect of spoofing distance. Apart from the cognitive vulnerabilities exploited in the text, a second relevant factor could be the similarity between the `From:` address displayed to a user and `Org`'s legitimate one. Figure 14 reports the relation between Levenshtein distance of the spoofed `From:` domain and the expected number of clicks. We find an inverse relation between the two variables, suggesting that the greater the dissimilarity between the spoofed and the original domain, the lower the average number of generated clicks ($\beta = -0.13, p < 0.001$). This suggests that both cognitive attacks and the degree of spoofing in an email may have an effect on the relative success of a phishing email and could be considered to build a triaging model for phishing emails.

5 Modelling phishing success

We now evaluate the relative impact of each cognitive variable in the collected dataset. We estimate coefficients for a Poisson process of the (aggregate) form:

$$log(clicks_i) = \alpha + \beta_1 cogvulns_i + \beta_2 spoofdist_i + \varepsilon_i \quad (1)$$

whereby, for each email i, *clicks* represents the number of measured clicks, *cogvulns* is the array of counts of the vulnerability triggers identified in the email body, and *spoofdist* indicates the degree of (dis-)similarity between the spoofed `From:` address and the original `Org` domain. ε_i is the error term. To monitor and account for overfitting problems related to the few available datapoints, we combine a step analysis of each model (M1..M7) with regression bootstrapping to generate robust confidence intervals for the coefficient estimations. For model selection we report coefficients, 95% confidence intervals, residual deviance, and Adjusted McFadden Pseudo-R^2, to reduce the statistical bias in the performance metrics for model selection.[10] Results are reported in Table 6.

All models have relatively stable coefficient estimations showing no evident interaction effects between the regressors (correlation matrix presented in Table 9 in the Appendix). Coefficients should be interpreted relative to each other as opposed to in absolute terms. Because of the relatively small sample size, we refrain from drawing direct conclusions on the model coefficients. For this reason statistical significance is better served in the analysis reported in Figure 13 and is only detailed in Table 6 for the reader's reference. Within our sample, model coefficients can be interpreted as the relative change in number of clicks for every additional vulnerability trigger of that type in an email. For example, the M7 coefficient for `Scarcity` (0.02) indicates an increase of 2% in the number of expected clicks for every new trigger of that category. Likewise, an increase in one point on the Levenshtein distance scale is related to a *decrease* in clicks of 10%. A first informal look at the McFadden's $Pseudo - R^2$s, `Reciprocity`, `Consistency`, and `Spoof dist.` appear to have the strongest effect in increasing the explanatory power

[10] Importantly, with this procedure we *do not* aim at identifying a definitive model and coefficients to forecast phishing success: regardless of the amount of observations in the dataset, that would not be possible because the 'click generation process' generating the observations necessarily varies from domain to domain (e.g. finance vs health), from organization to organization (e.g. national vs international), and from customer base to customer base (e.g. sensibility of application domain). Therefore, coefficient estimations out of this type of models cannot be '*plug-and-play*' across organizations and domains and will require tuning before being applied in-house.

Table 6: Regression results for Eq. 1

All model coefficients estimations are relatively stable across the seven models. Coefficients for the Poisson models are presented with 95% confidence intervals in parentheses. `Social proof` and `Spoof distance` of `From:` addresses appear to have the largest effects on predicted number of clicks. Higher spoof distances (i.e. higher dissimilarity between `From:` domain and original domain) result in a lower number of expected clicks. We only report coefficient significance (indicated by a * for significance at the 0.1% level) for the reader's reference; however due to the relatively small sample size coefficient estimations should only be interpreted relative to each other as opposed to in absolute terms. Model power w.r.t. the baseline model is reported by the adjusted McFadden Pseudo-R^2; a χ^2 test is employed for model comparison ($^\star: p \leq 0.001$; $^\dagger: 0.001 < p \leq 0.01$). Standard model checks do not reveal issues or biases in the model fit.

	M1	**M2**	**M3**	**M4**	**M5**	**M6**	**M7**
α	4.38*	3.89*	3.79*	3.63*	3.37*	3.37*	4.22*
	(4.33, 4.42)	(3.81, 3.97)	(3.71, 3.87)	(3.54, 3.73)	(3.17, 3.44)	(3.23, 3.51)	(4.02, 4.42)
`Reciprocity`	-0.02*	-0.01*	-0.02*	-0.02*	-0.02	-0.02*	-0.02*
	(-0.02, -0.02)	(-0.02, -0.01)	(-0.03, -0.02)	(-0.02, -0.01)	(-0.02, -0.01)	(-0.02, -0.01)	(-0.02, -0.01)
`Consistency`		0.02*	0.02*	0.02*	0.03*	0.03*	0.01*
		(0.02, 0.02)	(0.02, 0.02)	(0.02, 0.02)	(0.02, 0.03)	(0.02, 0.03)	(0.01, 0.02)
`Social proof`			0.14*	0.11*	0.04	0.04	0.10*
			(0.11, 0.16)	(0.08, 0.14)	(0.01, 0.08)	(0.01, 0.07)	(0.06, 0.13)
`Authority`				0.01*	0.02*	0.02*	0.00
				(0.01, 0.02)	(0.02, 0.03)	(0.02, 0.02)	(0.00, 0.01)
`Scarcity`					0.02*	0.02*	0.02*
					(0.02, 0.03)	(0.02, 0.03)	(0.01, 0.02)
`Liking`						-0.02*	0.04*
						(-0.04, -0.01)	(0.02, 0.06)
`Spoof dist.`							-0.10*
							(-0.12, -0.08)
Adj. Pseudo-R^2	0.09	0.23	0.28	0.30	0.33	0.33	0.41
Res. Dev.	1390*	1136*	1054*	1012*	958*	951†	814*
N	38	38	38	38	38	38	38

of the model. `Scarcity` appears to contribute modestly, whereas `Liking` appears to have the smallest effect on the model. The negative effect of `Reciprocity` as shown in Figure 13 is confirmed in the model as well.

5.1 Cognitive triaging of phishing success

We now extend the model evaluation to estimate the amount of clicks generated by other emails for which `Org` has detected no click (e.g. because no call-back to `Org` resources has originated from the phishing website, remaining therefore invisible to `Org`'s detection infrastructure, ref. Fig 1). Recall however that our model estimates are likely subject to overfitting issues due to the inevitably small sample size. This only means that predicted outcomes could be unreliable over arbitrarily diverse email corpora (i.e. not represented in the training data); on the other hand, predictions over *similar* emails to those provided to the fitted models will not suffer from unmodelled biases and will generate reliable estimations. For this reason we only limit our analysis to emails with a distribution of vulnerability triggers within plus or minus one standard deviation from the mean for that trigger in the model's respective training set.

To choose the model for the prediction we perform a set of ANOVA tests (χ^2), which indicate all factors add significant information to the model, albeit `Liking` only marginally. However, due to the statistical limitations of estimations in our dataset, we also consider a second model that considers only isolated factors for which we observe a clear effect as reported in Figure 13. Based on these observations we consider two different prediction models (PM), each with different regressors, namely: PM1: `Reciprocity`, `Consistency`, `Scarcity` and `Spoofing distance`; PM2 all six cognitive vulnerabilities + `Spoofing distance` (i.e. equal to M7 as suggested by the ANOVA tests). This leaves us with $n = 334$ and $n = 189$ suspicious emails on which to run the predictions for PM1 and PM2 respectively. To build robust confidence intervals around the estimations, we run a bootstrap simulation ($n = 5,000$). Table 7 reports median coefficients and 95% confidence intervals of the estimations. Notice that the estimated coefficients remain largely similar to those of the original models for most coefficients. PM1 shows much tighter confidence intervals

Table 7: Bootstrapped regression coefficients

	PM1			**PM2**		
	0.025q	Med	0.975q	0.025q	Med	0.975q
α	3.37	4.35	4.90	2.84	4.22	5.17
`Recip.`	-0.05	-0.01	0.00	-0.08	-0.02	0.00
`Cons.`	0.00	0.01	0.04	0.00	0.01	0.05
`Soc.Pr.`				-0.18	0.10	0.37
`Auth`				-0.03	0.00	0.05
`Scar.`	0.00	0.02	0.04	-0.03	0.02	0.05
`Liking`				-0.02	0.04	0.05
`Sp.dist.`	-0.17	-0.09	0.03	-0.12	-0.10	0.18

Table 8: Descriptive statistics of average predicted clicks
Estimations are generated from 50,000 simulations run on the bootstrapped model coefficients (Table 7).

PM1					
Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
30	48	54	56	62	99

PM2					
Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
26	43	50	53	60	129

for the estimated coefficients in comparison with PM2, suggesting more reliable predictions. Notice that the distribution of the coefficient estimations in PM1 tends to remain on the same side of zero, again suggesting statistically robust results for this model. This suggests the exclusion of `Liking`, `Authority` and `Social proof` in PM1 may lead to more realistic estimations.

We simulate model predictions for the undetected clicks by randomly sampling ($n = 50,000$) model coefficients from the two distributions and report aggregate statistics (Table 8) of the estimated number of generated clicks. Figure 15 reports the results. The simulation results indicate that the average 'undetected' email has potentially generated 50 - 55 clicks, with a long tail of (few) emails generating up to 100 clicks.[11] This suggests that prioritization efforts based on the cognitive characteristics of a phishing email could help in more efficiently addressing attacks (e.g. by means of takedown actions), by targeting first attacker resources that are likely to generate more impact on the organization's customer base: by targeting first the emails that are most likely to engage users in compliant behaviour, organizations can effectively triage the stream of incoming phishing attacks to minimize the impact on their customer base.

6 Discussion

The previous sections have demonstrated how quantitative measurements of cognitive vulnerabilities employed in phishing attacks can be used to develop a model to make predictions about the expected efficacy of these attacks. This characterization allows one to assess the threat of these attacks in an automated way such that instant prioritization of phishing incident responses becomes possible. This paper's contributions go beyond the scope of earlier works on cognitive factors for phishing by providing an empirical estimation and operable implementation to estimate phishing success.

[11] Notice that additional organization-specific features of the email (e.g. presence of the company logo), may also have an effect on the number of clicks. Whereas this is out of the scope of this paper, which only looks at the cognitive effects, a fully-operative model within an organization can easily integrate other factors in the prediction.

In this work we identified several correlations between different cognitive vulnerabilities and the average number of clicks an email can be expected to generate. In line with the hypothesis that the presence of any individual cognitive vulnerability increases user response to the phish, we found that `Consistency` and `Scarcity` exercise a clear positive effect on the number of generated clicks. We find no evident effect from `Social proof` and `Liking`, whereas `Authority` appears to have a positive effect albeit driven by only a few non-zero data points. Interestingly, `Reciprocity` even shows a counterproductive effect, albeit only marginal. This difference may well be explained by the specific application domain, as corporate customers subject to financial threats from phishing can generally be expected to have different sensitivity to specific principles of influence than other groups [22]. Although this suggests that full generalizability can not be expected for any one set of results, conclusions similar to ours could be drawn for specific contexts close in nature to the one in which `Org` operates. In particular, our finding of the reduced effect of `Authority` in the banking domain is in contrast with results from Wash and Cooper [53], who found `Authority` to be the most effective strategy for the presentation of certain phishing education materials. This difference may illustrate the context-dependency of the relative efficacy of these influence tactics, and indicates a need for careful consideration of such differences across domains. For example, the effect of `Authority` can be mediated here by the already relatively high authoritive position a bank has on its customers; this suggests that, on one side, depending on the domain it may be more difficult for an attacker to devise effective attacks *adding* to baseline cognitive effects; on the other, this also suggest that a relevant metric to evaluate could be the *relative increase* (or decrease) in the cognitive effect w.r.t. the baseline. A similar consideration could be drawn for `Reciprocity`, whereby we observe a negative effect on the generated 'clicks'. An explanation could be that these type of triggers rise a red flag in the context of banking operations, for example as a bank's 'environmental friendliness' may not be a convincing-enough reason to act on a request (e.g. to renew one's debit card).

These observations also provide useful input to training campaigns regularly run by medium and large organizations in an attempt to increase their customers and employee's awareness of the social engineering threat. Replications of this study in specific domains could reveal which principles of influence the 'average' customer of an organization is more vulnerable to; awareness campaigns run by the organization could then target those specific traits by providing specific examples or information material built ad-hoc for the consumer base (or targeting sections of it). For example, consumers particularly vulnerable to `Scarcity` may benefit from knowing the organization's policies in terms of change deadlines and processes, such that an email stating unrealistic and short cutoff dates to react lose credibility.

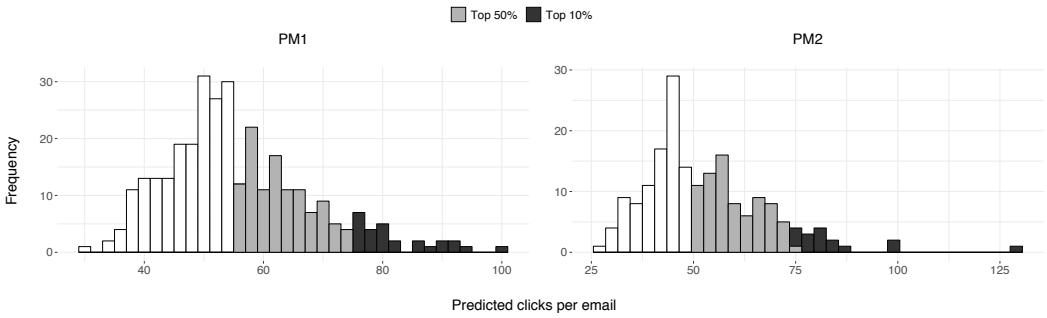


Figure 15: Distribution of predicted average clicks

Operationally, the presented procedure could be applied both client and server side to automate the risk evaluation of potential phishing emails for the enforcement of security policies; for example, mail client plugins or server-side processes could automatically divert or forward high-risk emails to phishing investigation and response teams for further evaluation, while delaying the delivery of messages waiting for a diagnosis. Furthermore, we have described how these observed effects can be used in the construction of a prediction model for the triaging of incoming phishing attacks. By enabling the triaging of incoming phishing attacks, our results will enable incident response teams to focus on the most prominent threats immediately, without having to manually filter out the noise from the bulk of low priority emails in their phishing abuse inbox, thereby minimizing reaction costs and increasing response effectiveness. The practicality of this is evidenced in Figure 15, where by addressing the small fraction of emails associated with the highest expected click counts one can mitigate a large fraction of potential attacks. This is critical to minimize overall victimization rates, as the short-lived nature of phishing domains stresses a need for prompt identification of which domains are most likely to be reached by customers falling for the phish. By contrast, addressing attacks in no particular order would most likely result in wasting valuable time and resources by addressing first the vast majority of attacks that are likely to generate few clicks only (ref. Figure 15 and Table 8).

Finally, our method opens up new opportunities in terms of automated incident handling and security orchestration, e.g. by enabling incident handlers to apply automated follow up procedures to incoming phishing attacks that fall within a certain threat range; for example, reported measures on the vulnerability triggers could be used to implement dynamic risk-based access control policies to limit immediate follow-up actions. Similarly, CSIRTs (*Computer Security Incident Response Team*) could implement automated network-level containment procedures based on the profile of incoming emails, and avoid additional (and unnecessary) victimization by delaying follow-up actions by the users until the risk is cleared.

7 Conclusions

In this work we presented an empirical method and evaluation of the effect of cognitive vulnerability triggers in phishing emails on the expected 'success' of an attack. We employed a unique dataset from a large European financial organization with data from their phishing response division. Our results indicate that response teams' operations, such as take-down actions against rogue phishing domains, could largely benefit from a (fully automated) cognitive assessment of the email body to predict relative success of the attack, given the relevant user base. Our findings and method could also be employed to deploy more effective training and awareness campaigns in response to the more prominent threats suffered by the potential victims. Future work could explore automated response strategies to contain potential attacks and/or delay user response where needed.

References

[1] Sadia Afroz and Rachel Greenstadt. PhishZoo: Detecting phishing websites by looking at them. In *Proc. of ICSC*, pages 368–375. IEEE, 2011.

[2] Nurul Akbar. *Analysing Persuasion Principles in Phishing Emails*. PhD thesis, 2014.

[3] Marcus Butavicius, Kathryn Parsons, Malcolm Pattinson, and Agata McCormac. Breaching the Human Firewall: Social engineering in Phishing and Spear-Phishing Emails. In *Proc. of ACIS*, pages 1–11, 2015.

[4] Campaign Monitor. What Our Data Told Us about the Best Time to Send Email Campaigns, 2014.

[5] Kuan-Ta Chen, Jau-Yuan Chen, Chun-Rong Huang, and Chu-Song Chen. Fighting Phishing with Discriminative Keypoint Features. *IEEE Internet Comput.*, pages 1–6, 2007.

[6] R Cialdini. *Influence: The Psychology of Persuasion*. 1984.

[7] R Cialdini and N Goldstein. The science and practise of persuasion. *Cornell Hosp. Q.*, 43(2):40–50, 2002.

[8] Paul Cichonski, Tom Millar, Tim Grance, and Karen Scarfone. Computer security incident handling guide. *NIST Special Publication*, 800(61):1–147, 2012.

[9] Rachna Dhamija, J. D. Tygar, and Marti Hearst. Why phishing works. In *Proc. of CHI*, page 581, 2006.

[10] Matthew Dunlop, Stephen Groat, and David Shelly. GoldPhish: Using images for content-based phishing analysis. In *Proc. of ICIMP*, pages 123–128. IEEE, 2010.

[11] Bradley Efron and Robert J. Tibshirani. *Introduction to the Bootstrap*. 1993.

[12] Ana Ferreira, Lynne Coventry, and Gabriele Lenzini. Principles of persuasion in social engineering and their use in phishing. In *Lecture Notes in Computer Science*, volume 9190, pages 36–47. 2015.

[13] Ana Ferreira and Gabriele Lenzini. An analysis of social engineering principles in effective phishing. In *Proc. of STAST*, pages 9–16, 2015.

[14] Johannes Fürnkranz, Klaus Brinker, Eneldo Loza Mencía, and Eyke Hüllermeier. Multilabel Classification via Calibrated Label Ranking. *Mach. Learn.*, 73(2):1–23, 2008.

[15] Matthew L. Hale, Rose F. Gamble, and Philip Gamble. CyberPhishing: A game-based platform for phishing awareness testing. In *Proc. of HICSS*, pages 5260–5269. IEEE, 2015.

[16] Grant Ho, Aashish Sharma Mobin Javed, Vern Paxson, and David Wagner. Detecting Credential Spearphishing Attacks in Enterprise Settings. In *Proc. of USENIX-Security*, pages 469–485, 2017.

[17] Chun Ying Huang, Shang Pin Ma, Wei Lin Yeh, Chia Yi Lin, and Chien Tsung Liu. Mitigate web phishing using site signatures. In *Proc. of TENCON*, pages 803–808. IEEE, 2010.

[18] Ankit Kumar Jain and B. B. Gupta. Phishing detection: Analysis of visual similarity based approaches. *Secur. Commun. Netw.*, 2017, 2017.

[19] Chris Kanich, Christian Kreibich, Kirill Levchenko, Brandon Enright, Geoffrey M Voelker, Vern Paxson, and Stefan Savage. Spamalytics: an empirical analysis of spam marketing conversion. In *Proc. of CCS*, pages 3–14, 2008.

[20] Ponnurangam Kumaraguru, Justin Cranshaw, Alessandro Acquisti, Lorrie Cranor, Jason Hong, Mary Ann Blair, and Theodore Pham. School of phish: a real-world evaluation of anti-phishing training. In *Proc. of SOUPS*, page 3, 2009.

[21] Jerald Franklin Lawless. Regression methods for Poisson process data. *J. of the Am. Stat. Assoc.*, 82(399):808–815, 1987.

[22] Patrick Lawson, Olga Zielinska, Carl Pearson, and Christopher B. Mayhorn. Interaction of personality and persuasion tactics in email phishing attacks. In *Proc. of HFES*, volume 61, pages 1331–1333, 2017.

[23] Stevens Le Blond, Adina Uritesc, Cédric Gilbert, Zheng Leong Chua, Prateek Saxena, and Engin Kirda. A Look at Targeted Attacks Through the Lense of an NGO. In *Proc. of USENIX-Security*, pages 543–558, 2014.

[24] Sophie Le Page, Guy Vincent Jourdan, Gregor V. Bochmann, Jason Flood, and Iosif Viorel Onut. Using URL shorteners to compare phishing and malware attacks. In *Proc. of eCrime*, pages 1–13, 2018.

[25] Gang Liu, Bite Qiu, and Liu Wenyin. Automatic detection of phishing target from phishing webpage. In *Proc. of ICPR*, pages 4153–4156, 2010.

[26] Mailchimp. Insights from Mailchimp's Send Time Optimization System, 2014.

[27] Jian Mao, Pei Li, Kun Li, Tao Wei, and Zhenkai Liang. BaitAlarm: Detecting phishing sites using similarity in fundamental visual features. In *Proc. of INCoS*, pages 790–795. IEEE, 2013.

[28] Samuel Marchal, Giovanni Armano, Tommi Gröndahl, Kalle Saari, Nidhi Singh, and N Asokan. Off-the-Hook: an efficient and usable client-side phishing prevention application. *IEEE Trans. Comput.*, 66(10):1717–1733, 2017.

[29] Samuel Marchal and N Asokan. On Designing and Evaluating Phishing Webpage Detection Techniques for the Real World. In *Proc. of USENIX-Security*, 2018.

[30] A Mishr and B B Gupta. Hybrid Solution to Detect and Filter Zero-day Phishing Attacks. In *Proc. of ERCICA*, pages 373–379, 2014.

[31] Kevin D. Mitnick and William L. Simon. *The Art of Deception: Controlling the Human Element in Security*. 2002.

[32] Mahmood Moghimi and Ali Yazdian Varjani. New rule-based phishing detection method. *Expert Syst. Appl.*, 53:231–242, 2016.

[33] Daniel J O'Keefe. Elaboration likelihood model. *The International Encyclopedia of Communication*, 2008.

[34] Daniela Oliveira, Harold Rocha, Huizi Yang, Donovan Ellis, Sandeep Dommaraju, Melis Muradoglu, D H Weir, and N C Ebner. Dissecting Spear Phishing Emails for Older vs Young Adults: On the Interplay of Weapons of Influence and Life Domains in Predicting Susceptibility to Phishing. In *Proc. of CHI*, pages 1–13, 2017.

[35] P. Pearce, V. Dave, C. Grier, K. Levchenko, S. Guha, D. McCoy, V. Paxson, S. Savage, and G. M. Voelker. Characterizing large-scale click fraud in zeroaccess. In *Proc. of CCS*, pages 141–152, 2014.

[36] PhishLabs. Phishing Trends & Intelligence Report: Hacking the Human. Technical report, 2018.

[37] A. Pitsillidis, C. Kanich, G. M. Voelker, K. Levchenko, and S. Savage. Taster's choice: A comparative analysis of spam feeds. In *Proc. of IMC*, pages 427–440, 2012.

[38] Propeller. The 2017 Email Marketing Field Guide: The Best Times and Days to Send Your Message and Get It Read, 2017.

[39] Daniel Ramage, David Hall, Ramesh Nallapati, and Christopher D. Manning. Labeled LDA. In *Proc. of EMNLP*, volume 1, page 248, 2009.

[40] Elissa M Redmiles, Neha Chachra, and Brian Waismeyer. Examining the Demand for Spam: Who Clicks? In *Proc. of CHI*, pages 1–10, 2018.

[41] Jennifer L Robertson and Julian Barling. Greening organizations through leaders influence on employees pro-environmental behaviors. *J. of Organ. Behav.*, 34(2):176–194, 2013.

[42] Angelo P.E. Rosiello, Engin Kirda, Christopher Kruegel, and Fabrizio Ferrandi. A layout-similarity-based approach for detecting phishing pages. In *Proc. of SecureComm*, pages 454–463, 2007.

[43] Timothy N. Rubin, America Chambers, Padhraic Smyth, and Mark Steyvers. Statistical topic models for multi-label document classification. *Mach. Learn.*, 88(1-2):157–208, 2012.

[44] Brad J. Sagarin and Kevin D. Mitnick. The Path of Least Resistance. In *Six Degrees Of Social Influence: Science, Application, and the Psychology of Robert Cialdini*, chapter 3. 2012.

[45] SendInBlue. Best Time to Send an Email: User Data Study by Industry, 2017.

[46] Ankit Shah, Rajesh Ganesan, Sushil Jajodia, and Hasan Cam. Understanding tradeoffs between throughput, quality, and cost of alert analysis in a csoc. *IEEE Transactions on Information Forensics and Security*, 14(5):1155–1170, 2019.

[47] Amit Singhal, Chris Buckley, and Mandar Mitra. Pivoted Document Length Normalization. *ACM SIGIR Forum*, 51(2):21–29, 2011.

[48] K E Stanovich and R F West. Individual differences in reasoning: Implications for the ratinality debates? *Behav. Brain Sci.*, 23:645–726, 2000.

[49] Gianluca Stringhini and Olivier Thonnard. That ain't you: Blocking spearphishing through behavioral modelling. In *Proc. of DIMVA*, pages 78–97, 2015.

[50] Janos Szurdi, Balazs Kocso, Gabor Cseh, Mark Felegyhazi, and Chris Kanich. The Long Taile of Typosquatting Domain Names. In *Proc. of USENIX-Security*, pages 191–206, 2014.

[51] Louis Tay, Kenneth Tan, Ed Diener, and Elizabeth Gonzalez. Social Relations, Health Behaviors, and Health Outcomes: A Survey and Synthesis. *Appl. Psychol. Health Well-Being*, 5(1):28–78, 2013.

[52] Amos Tversky and Daniel Kahneman. Judgment under Uncertainty: Heuristics and Biases. *Science*, 185(4157):141–162, 1974.

[53] Rick Wash and Molly M Cooper. Who Provides Phishing Training? Facts, Stories, and People Like Me. In *Proc. of CHI*, pages 1–12, 2018.

[54] Michael Workman. Wisecrackers: A Theory-Grounded Investigation of Phishing and Pretext Social Engineering Threats to Information Security. *J. of Am. Soc. Inf. Sci.*, 59(4):1–12, 2008.

[55] Ryan T. Wright, Matthew L. Jensen, Jason Bennett Thatcher, Michael Dinger, and Kent Marett. Influence techniques in phishing attacks: An examination of vulnerability and resistance. *Inf. Syst. Res.*, 25(2):385–400, 2014.

[56] Ryan T. Wright and Kent Marett. The Influence of Experiential and Dispositional Factors in Phishing: An Empirical Investigation of the Deceived. *J. of Manag. Inf. Syst.*, 27(1):273–303, 2010.

[57] Min Wu, Robert C. Miller, and Simson L. Garfinkel. Do security toolbars actually prevent phishing attacks? In *Proc. of CHI*, page 601, 2006.

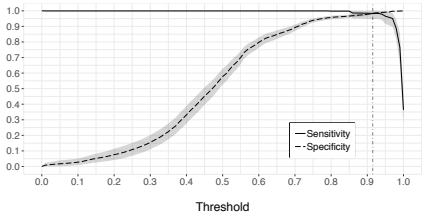
[58] Michael Yip, Nigel Shadbolt, and Craig Webber. Why forums? An empirical analysis into the facilitating factors of carding forums. In *Proc. of WebSci*, 2013.

Appendix

Table 9: Correlations between regression variables
We find no evident correlations between regressors. Only `Scarcity` and `Social proof`, and `Liking` and `Spoof dist.` show a higher than average correlation of 0.50 and 0.57 respectively, which is unlikely to affect estimation results.

	(1)	(2)	(3)	(4)	(5)	(6)	(7)
(1) `Reciprocity`	1.00	-0.19	0.13	-0.06	-0.11	-0.09	-0.17
(2) `Consistency`		1.00	-0.08	-0.24	0.10	-0.09	-0.41
(3) `Social proof`			1.00	0.25	-0.04	0.50	0.13
(4) `Authority`				1.00	0.06	-0.08	-0.10
(5) `Liking`					1.00	0.09	0.57
(6) `Scarcity`						1.00	0.24
(7) `Spoof dist.`							1.00

Bootstrap analysis for similar email detection



The optimal threshold was found at 0.91 based on the intersection of the mean sensitivity and specificity metrics at all decimal thresholds in [0,1] across 10,000 bootstrap simulations with sample size 300.

Figure 16: Simulated optimal cosine similarity threshold for duplicate detection

After computation of the full pairwise similarity matrix for all suspect emails in our dataset, a threshold value was used to determine the lower-bound for the similarity score of emails we consider to be duplicates. In order to determine the most optimal threshold value for our specific dataset we performed a *bootstrap analysis* [11]. A bootstrap analysis generally involves repeatedly running simulations on samples drawn with replacement from an original sample set in order to estimate statistics on a larger population. This is a fitting solution for problems concerning dataset of large sizes like ours, which do not generally allow for efficient derivation of the full set of results that qualify as *"ground-truth"*.

We started our bootstrap analysis with a random sample of 300 suspect phishing emails for which we made a manual assessment of all pairwise similarities to test the performance of our cosine similarity algorithm across different thresholds. Then, we repeatedly ($n = 10,000$) drew samples with replacement of size 300 from our manually classified sample and computed the pairwise cosine similarity matrix for all decimal thresholds in the interval [0,1]. For each combination of bootstrap sample and threshold value we computed the performance using the sensitivity (true positive rate) and specificity metrics (true negative rate). A high sensitivity score refers to a high probability of duplicate detection, measured by the proportion of actual duplicates that are correctly identified as being similar, whereas a high specificity score refers to a high probability of non-duplicate rejection, measured by the proportion of actual non-duplicates that are correctly identified as not being similar. The intersections of the mean results for these two performance measures indicate that 0.91 is the optimal threshold value for our dataset, as is visualized more elaborately in Figure 16.

We use this threshold to calculate the pairwise similarity matrix for all emails in our dataset and assign emails that are found to be similar the same *duplicate ID* to allow us to filter for unique emails.

Distribution of phishing emails by user

We perform a robustness check to evaluate the robustness of our results against large biases in the distribution of phishing emails across potential victims (whereby higher reported clicks may relate to higher email delivery volumes). We base the following on the sole assumption that attackers "sample" victims from the same pool (i.e. `Org` customers). As we of course cannot measure email delivery rates in user inboxes, our aim is to ask how would the dataset look like if large sample biases were present, and look for evidence in the data.

We developed the following data generation model to formalize and test this: a phishing email $e \in E$ can reach a user $u \in U$ with probability $P_e(u)$. A suspicious email will be detected by a user with a certain probability $P^{det}_{u,e} = P_u(DET|e)$. Notice that this depends on the email and the specific user that receives it, as different users may have different sensibilities to emails with different characteristics. The probability of remaining undetected is simply the complement, and is defined as $P^{und}_{u,e} = 1 - P_u(DET|e)$.

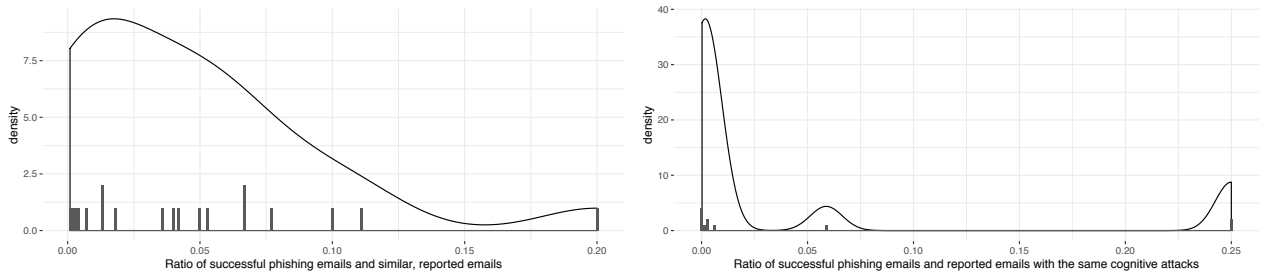
Further, each user has a certain probability $P_u(notify|DET,e)$ and $P_u(click|\neg DET,e)$ of, respectively, notifying a detected email, and clicking on a link if the email is not detected. Hence, the probability of an email being reported by a user is $P_e(u) \cdot P_u(DET|e) \cdot P_u(notify|DET,e)$. Conversely, $P_e(u) \cdot (1 - P_u(DET|e)) \cdot P_u(click|\neg DET,e)$ is the probability of a click for each $e \in E$ and $u \in U$.

Let C be the set of clicked emails, and D the set of reported emails, we'd then have:

$$|C| = |E| \sum_{\forall e \in E} \sum_{\forall u \in U} P_e(u) \cdot P^{und}_{u,e} \cdot P_u(click|\neg DET,e) \quad (2)$$

$$|D| = |E| \sum_{\forall e \in E} \sum_{\forall u \in U} P_e(u) \cdot P^{det}_{u,e} \cdot P_u(notify|DET,e) \quad (3)$$

$|D|$ corresponds to the whole set of suspicious emails reported in the organization's phishing inbox. $|C|$ corresponds to the set of emails clicked (of which we observe $C' = D \cap C$).



The distribution is calculated over similar emails (as per their cosine similarity, left), and over emails that share the same cognitive attacks (right). We observe a stable ratio across all emails, suggesting no significant skew in the probability of an email reaching a user's inbox.

Figure 17: Estimation of rates of arrival of phishing emails to users

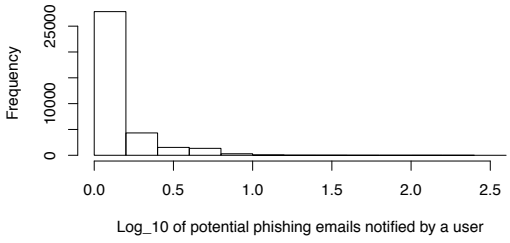


Figure 18: Number of users reporting $|D_u|$ emails (log_{10})

Notice that $P_e(u)$ (i.e. the probability of an email arriving to a user's inbox) is the only variable that is outside of the direct influence of the user. On the contrary, $P^{det}_{u,e}$, $P^{und}_{u,e}$, $P_u(notify|DET,e)$ and $P_u(click|\neg DET,e)$ directly depend on the characteristics of the user and of the email. Hence, we would expect them to be approximately constant as long as within the comparison the users are the same, and the emails are similar to each other. As we can control for 'similar' emails (Section 3.1.3) and users are all pooled from the set of `Org`'s clients, we can isolate large effects on $|C'|$ and $|D|$ as being caused by large fluctuations in $P_e(u)$.

Specifically, we would expect $|C'_{similar}|/|D_{similar}| \approx constant$ **iff** also $P_e(u) \approx constant\ \forall e \in E_{similar}$ as all other terms in the two equations will remain approximately the same for similar emails and the users sampled from the same pool. Hence, we measure the ratio of $|C'_{similar}|/|D_{similar}|$ as a proxy to estimate how much $P_e(u)$ can be expected to vary across emails. This holds under the sole assumption that emails in $D \setminus C$ are indistinguishable (from the perspective of the user) from those in $D \cap C = C'$; this is uncontroversial as the detection mechanism for the inclusion of an email in C' only depends on the phishing webpage and not on the email per se.

Figure 17 reports the ratio distribution calculated over emails with cosine similarity above the defined threshold (left), and over emails employing the same cognitive attacks (right). The ratios are small (i.e. only a small fraction of reported emails of a certain type can be expected to generate at least one click). This is qualitatively and quantitatively in line with previous findings in the literature [19]. Importantly, we observe that under both measures of similarity the rate is essentially constant and settles around 1-5% for all emails. This observation is incompatible with a significantly skewed distribution of emails per user. Breaking down the figure by users receiving the phishing email does not reveal any additional pattern as most users report only a few emails each (ref Figure 5 and Figure 18).

This is in line with previous literature on phishing attacks [9, 16] suggesting that no specific pre-selection of users charcterizes untargeted phishing attacks. We therefore do not expect significant biases in the analysis to emerge by the otherwise unmeasurable distribution of emails in users' inboxes.

Users Really Do Answer Telephone Scams

Huahong Tu[1], Adam Doupé[2], Ziming Zhao[3], and Gail-Joon Ahn[2,4]

[1]*University of Maryland, hh2@umd.edu*
[2]*Arizona State University, {doupe, gahn}@asu.edu*
[3]*Rochester Institute of Technology, zxzics@rit.edu*
[4]*Samsung Research*

Abstract

As telephone scams become increasingly prevalent, it is crucial to understand what causes recipients to fall victim to these scams. Armed with this knowledge, effective countermeasures can be developed to challenge the key foundations of successful telephone phishing attacks.

In this paper, we present the methodology, design, execution, results, and evaluation of an ethical telephone phishing scam. The study performed 10 telephone phishing experiments on 3,000 university participants without prior awareness over the course of a workweek. Overall, we were able to identify at least one key factor—spoofed Caller ID—that had a significant effect in tricking the victims into revealing their Social Security number.

1 Introduction

The rise of telephone spam, scams, fraud, phishing, or vishing, is a significant and growing problem. According to FTC reports for 2018, phone impersonation scams have increased significantly in the recent years. The national Do-Not-Call Registry received more than 5.78 million unwanted call complaints [1], with fraud and imposer scam in the top spots and more than 69% of all reported frauds were attempted over the phone [2].

With the growing dissatisfaction of telephone scams, however, little research has been done to study *why* people fall for telephone scams and how to combat the problem. In this paper, inspired by the work of Tischer et al. [3] on USB drives, we present the results of an empirical telephone phishing study, designed to systematically measure different attributes in relation to the success rate of telephone scams. Although the current understanding of telephone scams might be accepted as conventional wisdom, no prior work has specifically validated such claims with a systematic study. From this study, we hope to dispel some myths about what is "scammy" and what is not. With the understanding of the key attributes that make a scam convincing, the research community can focus on developing prevention methods to challenge the fundamentals of telephone phishing attacks. The key takeaway from this study is that caller ID spoofing is an incredibly effective feature in telephone scams, and, therefore, authenticated caller ID [4,5] is likely to be an important countermeasure.

The main contributions of this paper are the following:

- We describe a systematic approach to test the significance of various telephone phishing scam attributes and conduct an empirical study.
- We present our evaluation of the phishing study and provide our recommendations for combating the telephone phishing problem.

2 Background

With the emergence of distribution technology, decreasing economic cost, high reachability, and automation, the telephone has become an attractive medium for disseminating unsolicited information. As with any form of spam, there are three key ingredients: the recipient list, the content, and the distribution channel [6]. Telephone scams rely on distributing *deceitful* voice content, whereas telephone spam or telemarketing primarily distributes marketing and advertising content. In telephone scams, fraud, phishing, or vishing, the goal of the voice content is to trick the human victim into performing harmful actions for the benefit of the attacker (while other types of fraud are possible on telephone networks [7–9]).

Compared to other forms of phishing, such as email and website phishing [10–15], telephone phishing differs by having the potential to make the scam more convincing by falsifying both visual and auditory perceptions to induce the victims into falling for the scam. Visually, the scam can be made more convincing by altering the caller ID, such as by spoofing the caller ID, manipulating the area code (e.g., in "neighbor spoofing" attacks [16]), and impersonating a familiar contact name. Once the recipient has answered the call, the attacker then switches to using deceitful voice content to exploit the human recipient [17,18]. Within the voice content, an attacker can spoof or duplicate the speech from a known organization

or a familiar personal contact. To provide a motivation for the recipient to divulge confidential or personal information, the scammer can present a demanding scenario that forces the victim to divulge sensitive information.

By looking at telephone phishing from a perspective that can be characterized by the visual and voice attributes which it embodies, a systematic approach can be used to study and understand why some scams work better than others. Understanding why telephone phishing works can help us design solutions that challenge the core foundations of telephone scams.

3 Study Design

The goal of the study is to design a systematic approach that can reveal the effective factors in telephone scams by conducting our own telephone phishing scam. Our approach to designing the study is to first identify the attributes that could lead to an effective telephone phishing scam. After that, we design a set of experiments and procedures that allow comparison of different variations of an attribute. Each experiment followed a standardized procedure that was conducted on each group simultaneously (all calls were distributed in a randomized order throughout the experiment). Finally, we provide a discussion on what could be learned from the analysis and provide our recommended solutions for combating the telephone phishing problem. The study was conducted with significant ethical consideration and with IRB approval (see Section 3.5 for an in-depth discussion of ethics).

3.1 Attributes

To identify the telephone scam attributes, we gathered and reviewed more than 150 existing real-world telephone scam samples from various Internet sources, including the FTC website, IRS website, news websites, YouTube, SoundCloud, user comments, and industry surveys. While reviewing the scams, we identified the following attributes used in telephone scams:

Area Code: In North America, the area code is the first three digits on the caller ID. The area code specifies the geographic location associated with the caller's phone number, e.g., 202 is associated with Washington, DC. In addition, a toll-free phone number is also identified by the three-digit prefix similar to a geographic area code, e.g., 800, 888, 877, etc. According to reports of real-world IRS impersonation scams [19, 20], many scammers appeared to have either spoofed or obtained a 202 area code or toll-free area code on their caller IDs to make it appear as if the IRS is calling. To test the hypothesis that the area code could effect telephone phishing success, in our experiments we varied the caller ID area code between: 202 (Washington, DC), 800 (Toll-free), and 480 (local area code of the university location).

Caller Name: Today, most telephone terminals have the capability of associating a name with a telephone number. With a stored contact, an incoming call from the stored contact would show the name associated with the caller ID. To perform a spear phishing attack [21, 22], a malicious caller could spoof the caller ID of a known stored contact. A known stored contact can be identified for an organization by studying the publicly available phone numbers or for an individual by manually analyzing social network information. For legal, ethical, and IRB approval reasons, we did not actually spoof a known caller name. Instead, we asked our telephone service department to temporarily create a new contact in the university's internal phone directory and associated a legitimate sounding name with the telephone number. We used that telephone number in our scam experiments to produce a similar effect to caller name spoofing.

Voice Production: According to reports of real-world telephone scams, some used a robotic (synthesized) voice, while others used a pre-recorded human voice [20, 23]. To test the effect of synthesized voices vs. human voices, we recreated known scams using a text-to-speech synthesizer to generate a speech similar to the real-world scams. To mimic the human voice version of the scams, we recorded human voices speaking the exact same announcement message.

Gender: From listening to recordings of actual telephone scams, some used a male voice, and some used a female voice. To test if the vocal gender of the voice could have an effect on the telephone scam, we varied the voice gender between male and female in the text-to-speech synthesizer.

Accent: From the reports of telephone scams, some spoke with an Indian accent, and some others spoke with an American accent. It seems possible that recipients would be more wary of scams that speak in a foreign accent, and would be less suspicious of scams that speak in an American accent. To test if this could have an effect on the telephone scam, we varied the recorded voice accent between Indian and American in our experiments.

Entity: From gathering real-world telephone scams, two types of scams stood out in terms of the number of reports: IRS impersonation scams [24] and HR impersonation scams [25]. In these scams, the scammer claimed to be from the IRS or the company's HR department. While the IRS scams can affect any taxpayer in the US, the HR scams are usually targeted toward people in a specific company. Intuitively, it seems that a more targeted attack would have more success. Thus, we varied the impersonated entity of our scams between the IRS and ASU's HR department[1]. To simulate the real-world HR scams as closely as possible, we initially wanted to impersonate our university's HR department. However, our HR department had strong objections about using their name to conduct the scam experiments. As a compromise, our experiments claimed to be from a fake

[1] ASU is the university acronym for Arizona State University

but legitimate-sounding HR-like department called the "W-2 Administration"[2].

Scenario: Real-world telephone scams create various scenarios to motivate their victims to fall for the scam, such as tax lawsuits, payroll issues, or credit card verification. The type of motivation are generally either *fear-based* or *reward-based*. In our study, we crafted a fear-based and a reward-based scenario related to each entity. These scenarios were inspired by real-world IRS scams and HR scams. To test each type of scenario, our message announcements varied between Tax Lawsuit (IRS fear-based), Unclaimed Tax Return (IRS reward-based), Payroll Withheld (HR fear-based), and Bonus Issued (HR reward-based).

3.2 Experiments

To test these attributes, we designed the experiments such that variations of each attribute can be compared under similar environmental conditions. When performing experiments under the same environmental conditions, one of the design issues is to decide whether to counterbalance the environmental conditions such that all variations of background attributes are tested. This would theoretically avoid possible interference due to a specific set of background conditions.

However, performing a counterbalanced measures design does not come without costs. Counterbalancing the conditions is performed by splitting the experiments into groups of every possible order of attribute conditions. Given the large number of attributes that we have identified, and of each attribute with 2–4 variations that we have identified, would require us to create 384 separate groups of experiments. This is unfeasible for an empirical study with real-world time and resource constraints.

As a solution to this problem, instead of experimenting with a large number of background conditions, we compare variations of each attribute under a specific set of background conditions that seem to be the most popular in the real world. We decided on a standard background condition: a phishing scam with area code 202, with no caller name, speaking in a synthesized, male voice, in an American accent, impersonating the IRS, motivating the recipient with a tax lawsuit. The set of 10 experiments and the variations of each attribute are listed in Table 1.

3.3 Population

To comply with legal requirements [26], our own ethical considerations, and our IRB (Section 3.5), we conducted experiments on our university's internal population (rather than the general population). This population was unaware of our study (and we discuss the ethical implications of this deceptive non-consent study in Section 3.5). The population of the study were work telephone numbers that are associated with university staff and faculty. We decided on a population of 3,000 recipients (300 per experiment) for the study. To compile the list of telephone numbers, we wrote a custom tool to download the university's internal phone directory. For a real-world scammer, our university's phone directory is also publicly available for crawling.

To minimize selection bias, the telephone numbers were randomly chosen from the university telephone directory, and then the chosen contacts were randomly put into one of the 10 experiment groups. The sample selection procedure was as follows: (1) Compile the list of work telephone numbers associated with university staff and faculty, (2) remove telephone numbers of people already aware of the study, and (3) randomly assign 300 numbers to each of the 10 experiments.

3.4 Procedure

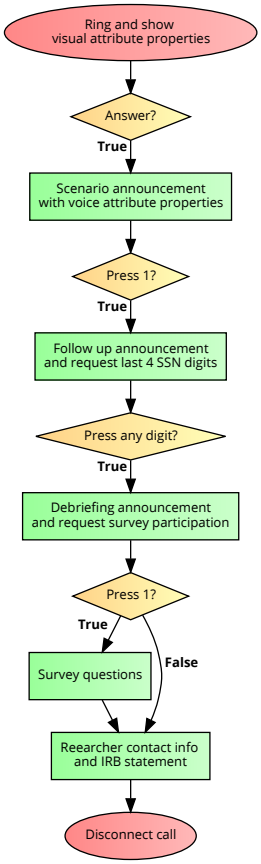


Figure 1: Procedure of each experiment.

Several considerations went into the design of the procedure. First, we need to ensure that the procedure is standardized across all experiments, such that the results are directly comparable to each other. Second, we need to ensure that the process minimizes false positives and false negatives, otherwise, the study results could be unreliable. Finally, the

[2]The W-2 is the income tax form currently used in the United States, so this name has associations with payroll and taxes.

No.	Caller ID	Area Code Location	Caller Name	Voice Production	Gender	Accent	Entity	Scenario
E1	202-869-XXX5	Washington, DC	N/A	Synthesizer	Male	American	IRS	Tax Lawsuit
E2	800-614-XXX9	Toll-free	N/A	Synthesizer	Male	American	IRS	Tax Lawsuit
E3	480-939-XXX6	University Location	N/A	Synthesizer	Male	American	IRS	Tax Lawsuit
E4	202-869-XXX0	Washington, DC	N/A	Synthesizer	Female	American	IRS	Tax Lawsuit
E5	202-869-XXX2	Washington, DC	N/A	Synthesizer	Male	American	IRS	Unclaimed Tax Return
E6	202-849-XXX7	Washington, DC	N/A	Human	Male	American	IRS	Tax Lawsuit
E7	202-869-XXX4	Washington, DC	N/A	Human	Male	Indian	IRS	Tax Lawsuit
E8	480-462-XXX3	University Location	N/A	Synthesizer	Male	American	ASU	Payroll Withheld
E9	480-462-XXX5	University Location	W-2 Administration	Synthesizer	Male	American	ASU	Payroll Withheld
E10	480-462-XXX7	University Location	N/A	Synthesizer	Male	American	ASU	Bonus Issued

Table 1: Table of all experiments and their attributes.

procedure also must be carried out ethically and minimize potential harm to the participants.

To ensure that the procedure is standardized, we used an autodialer to automate the process of sending out the telephone calls and collecting the recipients' responses.

Every experiment followed a standard procedure that is summarized in Figure 1. The procedure has several steps that require inputs from the recipient. The purpose of this action is to reduce the likelihood of recipients making random input actions without hearing the announcement. The action also helps to filter out answers from answering machines. Note that a recipient could break off from the procedure at any point by simply disconnecting the phone, hence not every recipient follows the procedure until the end.

The procedure first begins with a ring on the recipient's work phone (the recipient does not expect the call). When the phone is ringing, the incoming call screen shows the caller ID and, in experiment E9, the caller name. An example of the incoming call screen is shown in Figure 2a. In all of our experiments, the caller ID showed up as 91XXXXXXXXXX, where XXXXXXXXXX is the caller ID used in the respective experiment. Our university's work phone adds a 91 prefix to every incoming phone call from an external source as all of the calls were distributed from an external telephone service provider, similar to what a real-world scammer would do.

For Experiment 9, the incoming call screen also shows a caller name as shown in Figure 2b. This experiment was designed to simulate a scammer spoofing a known caller name. For legal and ethical reasons, we did not actually spoof a phone number. Instead, we asked our telephone service department to temporarily create a new contact in the university's internal phone directory and associated a legitimate sounding HR department name "W-2 Administration" with the telephone number. In a normal external call, there is no caller ID displayed, however, IT was able to help us create the caller ID shown in Figure 2b. While a scammer would not be able to create a new name, they can spoof the caller ID of a known caller with a targeted spearphishing scam.

If the call is answered, it starts by playing a prerecorded scenario announcement message (which is different for each scenario). The prerecorded scenario announcement message incorporates the voice attribute properties of each particular experiment. We crafted the four different announcement messages to mimic what a real-world scammer would say by using words and sentences from our collected scam samples.

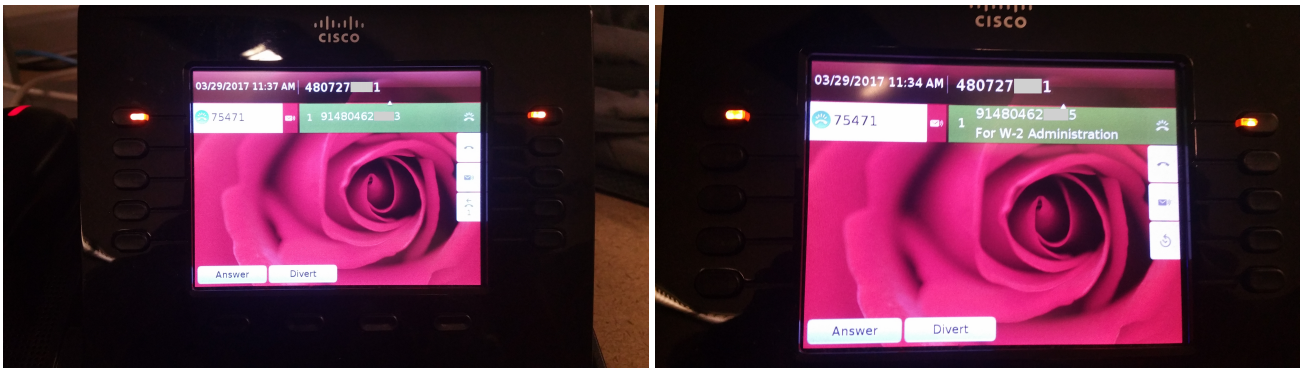
In the *Tax Lawsuit* scenario, we claimed to be the IRS and presented a scenario where the recipient had to act because of a tax lawsuit. The transcript of the announcement message is in Appendix A.1. In the *Unclaimed Tax Return* scenario, we claimed to be the IRS and presented a scenario where the recipient had to act because of an unclaimed tax return. The transcript of the announcement message is in Appendix A.2. In the *Payroll Withheld* scenario, we claimed to be ASU "HR" department and presented a scenario where the recipient had to act because pay would be withheld. Our university has a publicly available payroll calendar on the HR department's website[3], hence a real-world scammer could also use this information to craft an announcement message based on the payroll information. The transcript of the announcement message is in Appendix A.3. In the *Bonus Issued* scenario, we claimed to be ASU "HR" department and presented a scenario where the recipient had to act because a performance bonus was issued. The transcript of the announcement message is in Appendix A.4.

Every scenario announcement message requests the recipient to enter `1` to continue to the next step for a follow-up message (same for every participant). After pressing `1`, the follow-up message asks the recipient to enter the last four digits of their Social Security number and mimics the process of connecting the phone call to a live agent. The transcript of the follow-up announcement message is in Appendix B.

In the real world, the last four digits of the Social Security number can be used to perpetrate financial and identity fraud [27]. Other parts of the Social Security number can also be inferred from the recipient's phone number [28]. To minimize potential risk to the recipient (with cooperation and consultation with our IRB), *we did not record which digits were pressed*, we instead recorded only if any digit was pressed.

This then led to a debriefing announcement and a request to participate in our phone survey. The transcript of the debriefing message is in Appendix C. To emphasize the fact that whatever they listened to was not a real scam, the debriefing announcement and survey questions were recorded with the researcher's real voice. The post-debriefing survey

[3] https://cfo.asu.edu/payroll-calendars



(a) All experiments except experiment E9

(b) Experiment E9 with caller name displayed

Figure 2: Incoming call screen of different experiments.

consisted of two questions: (1) a survey question that asked whether the recipient was convinced by the scam (transcript in Appendix D.1) and, depending on how they responded, (2) asked what factor convinced them of the scam (Appendix D.2) or convinced them not to believe the scam (Appendix D.3). We recorded the participant's voice recording for the second question. After the second survey question, the autodialer system plays an ending message stating the researcher's contact information (transcript in Appendix E).

In summary, during each step of the procedure, the autodialer was configured to collect the following inputs from the recipient: Continued, Entered SSN, Convinced, Unconvinced, and Recording.

3.5 Ethics

These experiments were a deceptive study on involuntary participants, and therefore we deeply considered the ethical issues. To address the ethical issues inherent in our experiments, we carefully designed the experiments and worked with our university's IRB, to not simply obtain approval but to conduct the study minimizing harm. This is important because, to have scientifically valid results, we could not obtain informed consent (this would bias the results of the study) and we must deceive the participants (they would need to believe that the call was an actual scam call). To protect our participants, we implemented several safeguards in the experimental design.

The nature of this experiment, studying telephone phishing attacks, involves deception as well as involuntary participation. Both aspects are critical to receiving scientifically valid results—informing the participants of the study would significantly bias the results. However, the use of deception can harm the recipients, by wasting their time, confusing them, or leading them to believe they fell victim to a scam. Therefore, our debriefing served to not only inform the participants of the study, but to also educate recipients about the dangers of telephone scams. In addition, we only called each participant once throughout the entire study duration (to minimize the disruption).

Before proceeding with the study, we also worked with our university's IT security group to provide them with information that would help to alleviate the concerns of our participants. This IT security group at ASU is responsible for the security of all aspects of the university. We shared with the security group the experiment contact list, the experimental design, and the incoming phone numbers (that we used to send the calls) so that the help desk personnel could be prepared to handle any requests and reports. In this way, our participants who reported the scam calls to IT would be assured that it was part of a study.

In recording the results, we also strove to do so ethically and in accordance with established IRB protocols. One of the major safeguards is that we did not record the Social Security number. While a spammer would typically want the Social Security number, all that we record is the fact that they pressed any digit. In fact, we did not even ask for the full Social Security number, and we performed no analysis to see if they provided nonsensical last four Social Security numbers. This has the drawback of decreasing the validity of our data—participants may have felt safe to input only the last four of their Social Security number (when they would not input the full number) or they input fake last four digits of their Social Security number. Although these measures may diminish the strength of our data, we believe ethics is a more important aspect of designing a telephone phishing study.

3.6 Dissemination

We ran the previously described procedure using the 10 described experiments during a workweek in the late March of 2017, during core working hours of 10:00am–5:00pm each day. We used an Internet-hosted autodialer[4] to automate the process of sending out the telephone calls to the 3,000 recipients. Each experiment's calls were simultaneously distributed

[4] https://www.callfire.com/

during the experiment period at a rate of 1–3 live calls per experiment.

We associated each experiment with a unique caller ID. In all experiments, the vast majority of the outbound calls did not reach a live recipient and were answered by a voicemail answering machine. If a recipient could not answer the phone, the recipient could use the caller ID in their call history to call us back. As each experiment had a unique caller ID, the return call would be directed to that particular experiment's procedure. When a recipient called back, the same procedure was administered where a prerecorded scenario announcement message is first played.

While disseminating the phone calls, several unexpected events impacted our study.

The ASU school of journalism and mass communication identified the scam call incidents only 2 hours and 45 minutes from the launch of the experiments on the first day. Instead of reporting it to the university help desk (who were prepared and aware of our study), the school sent out a mass email warning all journalism staff and faculty at 4 hours 28 minutes from launch. However, we did not notice a significant dip in the number of recipients that continued with our scam calls as the portion of work phones at the journalism department represents less than 2% of our sample population.

At 4 hours and 22 minutes from the launch of the experiments, our university's telephone service office also started blocking some of our phone calls as they were receiving system alerts of too many incoming phone calls exhausting the telephone trunk routes. We worked with the telephone service office to get our calls unblocked within the next 4 hours as we decreased the simultaneous call rate of our phone calls to one per experiment.

The IRB office also received some complaints (we were not told exactly how many) regarding the scam call experiments, which resulted in our experiments being paused for roughly 12 hours (start to finish) starting on day 2, as we waited for the IRB committee to review the complaints. The IRB examined our procedures and decided that, as our study was originally designed, the beneficence outweighed the harm (as evidenced by the complaints) and allowed the study to proceed.

A summary showing how these events affected our calls is shown in Figure 3. In the end, despite the unexpected events, we finished sending out the telephone calls to the 3,000 recipients as planned before the end of the workweek.

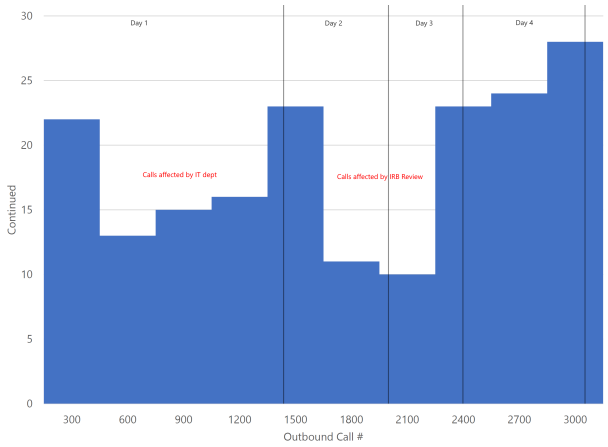


Figure 3: No. of recipients pressed 1 to continue the received calls over the experiment time.

4 Results and Analysis

The input data collected from the 3,000 recipients are presented in Table 2. Across all 10 experiments of 3,000 total recipients, 8.53% (256/3000) of all recipients continued after listening to the scam scenario announcement, 3.73% (112/3000) of all recipients called back after receiving the initial call from us, 4.93% (148/3000) of all recipients entered at least a digit when requested to enter the last four digits of their Social Security number, 1.17% (35/3000) of all recipients explicitly stated that they were convinced by the scam, and 1.23% (27/3000) of all recipients explicitly stated that they were not convinced by the scam.

Before presenting our analysis of the experiments, we first discuss our methodology to systematically analyze their relative effectiveness. The first step of performing the analysis is to decide on metric(s) that will be used as the standard of measurement. To chose an ideal metric, we believe a good metric should not only be quantifiable but also be a proxy for what ultimately matters. From the telephone scammers' perspective, the ultimate goal is to collect as many Social Security numbers as possible for the purpose of conducting identity fraud.

We could use the metric of *Entered SSN*, which is the number of participants that entered any value for their Social Security number (SSN). However, as discussed in Section 3.5, we did not collect the SSNs input by the user. Although this seems to be an ideal metric to estimate the number of SSNs collected, there is still the possibility that the recipient may have tried to enter a fake Social Security number. In fact, in some of the recordings, a few recipients stated that they did not enter their real Social Security number information.

Therefore, we need to derive a metric that could provide a reasonable estimate of the actual number of real SSNs given to us in each experiment. *Convinced* is the metric of the number of recipients that explicitly stated that they were convinced by the scam after the first survey question. This metric is the most conservative for estimating attack success. However, with the low number of responses, participants rarely made it to that step. Using this metric would exclude a large number of recipients that fell for the scam but declined to participate in the phone survey after the debriefing announcement.

Because we cannot assume that all SSNs entered were real, to reduce these types of false positives, we could create a new metric and remove the participants that entered their SSNs and then subsequently stated that they were unconvinced by

No.	Continued		Callbacks		Entered SSN		Convinced		Recordings		Unconvinced		Recordings	
E1	12	4.00%	7	2.33%	6	2.00%	0	0.00%	0	0.00%	4	1.33%	2	0.67%
E2	19	6.33%	7	2.33%	15	5.00%	3	1.00%	0	0.00%	3	1.00%	3	1.00%
E3	13	4.33%	6	2.00%	8	2.67%	1	0.33%	1	0.33%	2	0.67%	1	0.33%
E4	23	7.67%	14	4.67%	13	4.33%	2	0.67%	0	0.00%	3	1.00%	2	0.67%
E5	9	3.00%	3	1.00%	2	0.67%	1	0.33%	0	0.00%	1	0.33%	1	0.33%
E6	9	3.00%	7	2.33%	8	2.67%	2	0.67%	2	0.67%	2	0.67%	1	0.33%
E7	13	4.33%	8	2.67%	9	3.00%	3	1.00%	1	0.33%	5	1.67%	4	1.33%
E8	53	17.67%	22	7.33%	30	10.00%	8	2.67%	3	1.00%	9	3.00%	8	2.67%
E9	60	20.00%	15	5.00%	35	11.67%	7	2.33%	3	1.00%	4	1.33%	3	1.00%
E10	45	15.00%	25	8.33%	22	7.33%	8	2.67%	7	2.33%	4	1.33%	2	0.67%
Total	256	8.53%	112	3.73%	148	4.93%	35	1.17%	17	0.57%	37	1.23%	27	0.90%

Table 2: Summary of recipient inputs from all experiments.

the scam during the survey process. This metric, which we call *Possibly Tricked*, provides a reasonable estimate of the actual number of recipients that fell for the scam by entering the last four digits of their Social Security number. Compared to the previous metrics, this metric provides a good balance of conservativeness and sample size, and, therefore, we use this metric for our analysis.

No.	Entered SSN	Unconvinced	Possibly Tricked	
E9	35	4	31	10.33%
E8	30	9	21	7.00%
E10	22	4	18	6.00%
E2	15	3	12	4.00%
E4	13	3	10	3.33%
E3	8	2	6	2.00%
E6	8	2	6	2.00%
E7	9	6	3	1.00%
E1	6	4	2	0.67%
E5	2	1	1	0.33%
Total	148	37	111	3.70%

Table 3: Estimating the number of recipients possibly tricked into entering their real SSN information

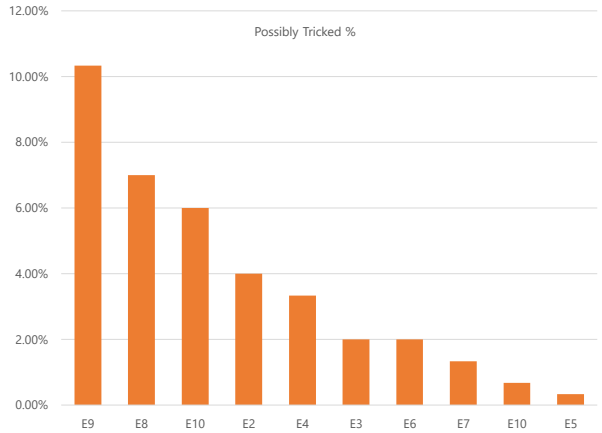


Figure 4: Recipients possibly tricked into entering their real SSN information.

Figure 4 presents a view of the number of *possibly tricked* recipients for each experiment, ranked from most successful to least successful. The tabulated data is in Table 3. Comparing the possibly tricked result between experiments, experiment E9 (spoofed caller ID) had the highest possibly tricked rate among all experiments, with an estimate of 10.33% (31/300) of recipients possibly tricked into entering the last four digits of their Social Security number. Experiment 5 (202 area code, unclaimed tax return) had the lowest success rate among all experiments, with an estimate of only 0.33% (1/300) of recipients possibly tricked into entering the last four digits of their Social Security number.

Attribute	Property	Linear Regression Coefficient
Area Code	Washington, DC	-2.22
	Toll Free	7.78
	Local	1.78
Caller Name	Unknown	-1.32
	Known	8.68
Voice Production	Synthetic	1.68
	Human	5.68
Gender	Male	-0.32
	Female	7.68
Accent	American	5.18
	Indian	2.18
Entity	IRS	-0.99
	ASU	8.34
Scenario	Tax Lawsuit	0.00
	Unclaimed Tax Return	-1.00
	Payroll Withheld	5.67
	Bonus Issued	2.67

Table 4: Linear regression coefficients of all attribute properties overfitted on possibly tricked.

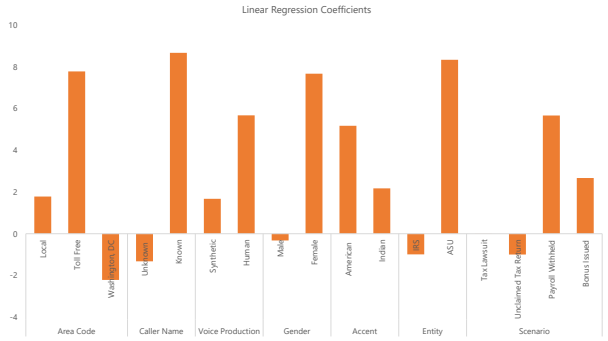


Figure 5: Linear regression coefficients of all attribute properties overfitted on possibly tricked.

The next step is to decide on an appropriate method of data analysis on the chosen metric. With a myriad of possible data analysis methods, we decided to use both linear regression and statistical hypothesis testing analysis. Linear regression is a model-based analysis can produce a model that can fit an optimal mapping of attribute properties to the results (i.e. possibly tricked). However, such method tend to overfit the spurious correlations that occur in training data since it is a

Small Data problem [29]. Furthermore, the attribute properties used in our experiments are also not conditionally independent. Nonetheless, the results of linear regression analysis are shown in Table 4 and Figure 5.

Alternatively, we used a statistical hypothesis testing approach for analysis. Before doing statistical hypothesis testing, we asked, "what are the hypothesis questions that our data can provide an answer for?" We will provide a discussion on the hypothesis questions we decided to ask and how we applied a data analysis method to provide a contextual answer to the hypothesis questions. Because we are testing several hypotheses, we perform the Holm-Bonferroni step-down correction [30] on the significance tests. The results are shown in Table 5 sorted by the individual p-value.

Can manipulating the area code have a significant effect on the attack success of a telephone scam?

In the real world, we observed that telephone scammers used area code manipulation in many instances (in particular in Neighbor Spoofing scams [16]). To provide an answer to this question, we can compare the number of possibly tricked between similar experiments that used different area codes, i.e., E1, E2, and E3. We see that E1 had 0.67% possibly tricked, E2 had 4% possibly tricked, and E3 had 2% possibly tricked.

In our question concerning the significance of area code, since E1 and E2 have the greatest difference in the number of possibly tricked recipients, we test if using a toll-free area code is significantly more effective than Washington, DC area code in the context of the IRS scam example. So we perform a right-tailed p-value hypothesis testing approach on the chosen experiment groups (E1 vs. E2) using the adjusted p-value corrected with Holm-Bonferroni's step-down method [30].

The use of right-tailed p-value statistical hypothesis testing approach is a method to answer if it is "likely" or "unlikely" to observe the improved alternative hypothesis (i.e. E2 possibly tricked) – assuming that the null hypothesis is true (i.e. probability distribution of E1 possibly tricked).

With regards to the choice of using Bayesian vs. Frequentist methods, since we are aware of no similar prior experiments, we can only use Frequentist methods to calculate the statistical significance on the underlying truths using only data from the current experiment.

In addition, not only do we want to know if the improvement to attack success is significant, it is also important to know the magnitude of improvement. To avoid making statements such as "E2 is 5 times more effective than E1", instead of measuring the relative difference, we calculated Cohen's d to measure the effect size for comparison between the two groups.

Using the right-tailed p-value approach, we have a χ^2 statistic of 7.314 and an adjusted p-value of 0.00684. Using an arbitrary confidence level of 95%, it is very likely that using a toll-free area code can result in a more successful attack than using a Washington, DC area code in the context of the IRS scam example. The two groups also have a Cohen's d of 0.222, which suggests it has a small effect according to Cohen [31] and has a somewhat educationally significant effect according to Wolf [32]. Therefore, we could say that the area code can have a statistically significant yet somewhat minor effect on the attack success of telephone phishing scam.

Can manipulating the type of voice production have a significant effect on the attack success of a telephone scam?

To provide an answer to this question, we can compare the number of possibly tricked between similar experiments that used different types of voice production, i.e., E1 and E6. In our question concerning the significance of voice production, we test if using a recorded human voice is significantly more effective than using synthesized voice in the context of the IRS scam example.

Using the same right-tailed p-value approach, we have a χ^2 statistic of 2.027 and an adjusted p-value of 0.155. Using an arbitrary confidence level of 95%, we cannot conclude that using a recorded human voice can result in a more successful attack than using synthesized voice in the context of the IRS scam example. The two groups have a Cohen's d of 0.117, which also suggests the effect size is very small and not educationally significant. Therefore, we are not able to conclude at this time if the type of voice production has a significant effect on the attack success of a telephone phishing scam.

Can manipulating the voice gender have a significant effect on the attack success of a telephone scam?

For the telephone scammer, the voice gender of the voice synthesizer can be easily changed with a simple option click in the autodialer. To provide an answer to this question, we compare the number of possibly tricked between similar experiments that used different voice genders, i.e., E1 and E4. In our question concerning the significance of voice gender, we test if using a female synthesized voice is significantly more effective than using male synthesized voice in the context of the IRS scam example.

Using the same right-tailed p-value approach, we have a χ^2 statistic of 5.442 and an adjusted p-value of 0.0197. Using an arbitrary confidence level of 95%, it is unlikely that using a female synthesized voice can result in a more successful attack than using a male synthesized voice in the context of the IRS scam example. The two groups have a Cohen's d of 0.192, which suggests the effect size is small and not educationally significant. Therefore, it is difficult for us to conclude at this time if the voice gender has a significant effect on the attack success of a telephone phishing scam.

Can manipulating the voice accent have a significant effect on the attack success of a telephone scam?

To provide an answer to this question, we compare the number of possibly tricked between similar experiments that used different accents, i.e., E6 and E7. In our question concerning the significance of voice accent, we test if speaking with an American accent is significantly more effective than speaking with an Indian accent in the context of the IRS scam example.

Hypothesis	Group A	Possibly Tricked	Group B	Possibly Tricked	p-value	Adjusted p-value[1]	Significant[1]	Cohen's d	Effect Size[2]	Conclusive
Entity Scenario (IRS vs. HR)	E1 + E5	3/600	E8 + E9	39/600	1.56E-8	1.09E-7	Yes	0.331	Small & educationally significant	Yes
Area Code (202 vs. 800)	E1	2/300	E2	12/300	0.00684	0.0410	Yes	0.222	Small & somewhat educationally significant	Somewhat
Voice Gender (Male vs. Female)	E1	2/300	E4	10/300	0.0197	0.0985	No	0.192	Small & not educationally significant	No
Caller Name (Unknown vs. Known)	E8	21/300	E9	31/300	0.147	0.588	No	0.119	Very small & not educationally significant	No
Voice Production (Synthetic vs. Human)	E1	2/300	E6	6/300	0.155	0.465	No	0.117	Very small & not educationally significant	No
Voice Accent (Indian vs. American)	E7	3/300	E6	6/300	0.314	0.628	No	0.082	Very small & not significant	No
Motivation (Reward vs. Fear)	E5 + E10	19/600	E1 + E8	23/600	0.530	0.530	No	0.036	Very small & not significant	No

Table 5: Summary of statistical hypothesis testing results ordered individual p-value.
[1]Using p-values corrected with Holm-Bonferroni's step-down method [30].
[2]Using effect size descriptors by Cohen [31] & Wolf [32]

Using the same right-tailed p-value approach, we have a χ^2 statistic of 1.015 and an adjusted p-value of 0.314. Using an arbitrary confidence level of 95%, we cannot conclude that speaking with an American accent can result in a more successful attack than speaking with an Indian accent in the context of the IRS scam example. The two groups also have a Cohen's d of 0.082, which suggests the effect size is very small and not educationally significant. Therefore, we are not able to conclude at this time if the voice accent has a significant effect on the attack success of a telephone phishing scam.

Can spoofing a known caller name have a significant effect on the attack success of a telephone scam?

To provide an answer to this question, we compare the number of possibly tricked between similar experiments that show a difference in the display of a caller name, i.e., E8 and E9. In our question concerning the significance of spoofing caller name, we test if displaying a HR-department caller name "W-2 Administration" is more effective than not displaying a caller name in the context of the HR scam example.

Using the same right-tailed p-value approach, we have a χ^2 statistic of 2.106 and an adjusted p-value of 0.147. Using an arbitrary confidence level of 95%, we cannot conclude that displaying a HR-department caller name can result in a more successful attack than displaying a caller name in the context of the HR scam example. The two groups also have a Cohen's d of 0.119, which suggests the effect size is very small and not educationally significant. Therefore, we are not able to conclude at this time if spoofing a known caller name has a significant effect on the attack success of a telephone phishing scam.

Can manipulating the entity scenario have a significant effect on the attack success of a telephone scam?

Any form of spear phishing involves impersonating an internal entity that the recipient is familiar with. The scammer has to create a spoofed caller ID and devise a scenario that is tailored to the entity, as the "Entity" cannot be set independently from "Scenario". To provide an answer to the hypothesis question, we compare the number of possibly tricked between similar experiments that used different entity-scenarios, i.e. comparing E1 and E5 with E8 and E10. In our question concerning the significance of impersonating an internal entity, we test if impersonating an internal entity is more effective than impersonating the IRS with the context of the scenarios tested.

Using the same right-tailed p-value approach, we have a χ^2 statistic of 31.976 and an adjusted p-value of 1.56E-8. Using an arbitrary confidence level of 95%, it is likely that impersonating an internal entity can result in a more successful attack than impersonating the IRS with the context of the scenarios tested. The two groups also have a Cohen's d of 0.331, which suggests the effect size is small and educationally significant. Therefore, we could say that impersonating an internal entity had a significant effect on the attack success of a telephone phishing scam.

Can manipulating the type of motivation have a significant effect on the attack success of a telephone scam?

To motivate the recipient into taking some harmful action, the scammer could either use fear or reward. To provide an answer to the hypothesis question, we compare the number of possibly tricked between similar experiments that used different types of motivation, i.e., comparing E1 and E8 with E5 and E10. In our question concerning the significance of the type of motivation, we test if fear-based scenarios are more effective than reward-based scenarios the context of the entities tested.

Using the same right-tailed p-value approach, we have a χ^2 statistic of 0.395 and an adjusted p-value of 0.530. Using an arbitrary confidence level of 95%, we cannot conclude that fear-based scenarios can result in a more successful attack than reward-based scenarios with the context of the entities tested. The two groups also have a Cohen's d of 0.036, which suggests the effect size is very small and not educationally significant. Therefore, we are not able to conclude at this time if manipulating the type of motivation has a significant effect on the attack success of a telephone phishing scam.

Summary

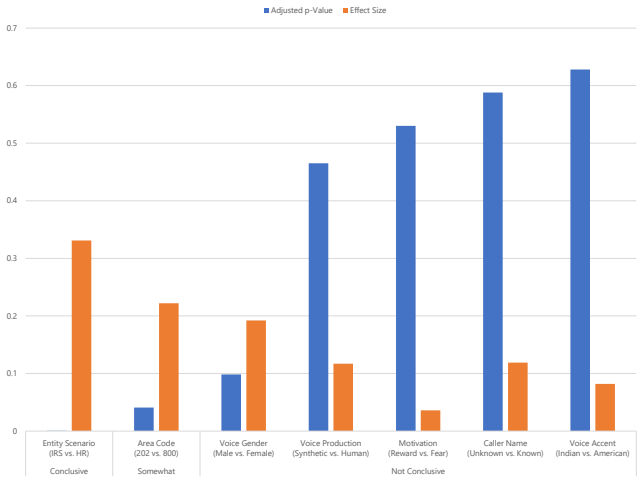


Figure 6: Summary of statistical hypothesis testing results.

The summary of our statistical hypothesis testing results is shown in Figure 6. Based on the statistical hypothesis results, we found that impersonating an internal entity had the most significant effect on the attack success of a telephone phishing scam. We also found that manipulating the area code (using a toll-free vs. a 202 area code) can have a somewhat significant effect.

On the contrary, manipulating the type of motivation, voice production, voice accent, and caller name, individually had an insignificant effect on the attack success. It is also difficult for us to conclude whether manipulating the voice gender has a significant effect even though the result was statistically significant.

5 Survey Responses

In this section, we highlight the recorded survey responses that asked the participants for the reasons they were convinced or unconvinced to enter the last four digits of their Social Security number. We listened to all 44 recorded voice responses and tabulated their responses in Table 6.

Based on the voice responses from the survey respondents, no one provided an explicit voice response on why they were convinced by the IRS scams. The four recordings we received were either silent or contained no useful information. We believe that participants were less willing to report the reasons why they were convinced by the scam after they were explicitly told that they had fallen victim to an attack.

On why the IRS scams were unconvincing, most of the survey respondents stated that they already knew that the IRS would not make a call like this or that they were already vigilant about IRS scam calls. This is understandable because there are numerous media reports about the IRS scams, and the IRS posted many public warnings not to trust these types of scams. This further supports the hypothesis that the impersonated identity and the corresponding scenario was the most significant factor. In experiment E7, two respondents also mentioned that the Indian accent was one of the reasons they were unconvinced.

On why the ASU imposer scams convinced them, most of the survey respondents described something related to the scam scenario, which means that the impersonated entity and the scenario were the key factors. Three respondents also believe that the caller ID was from ASU and stated caller ID was one of the reasons they believed in the scam, even though none of the caller IDs were actually from ASU.

On why the ASU scams did not convince them, most of the survey respondents stated that they were quite certain that ASU would not make a call like this or they were already vigilant about giving their SSN information over an incoming call. Two respondents in experiment E9 mentioned that the scenario only asked for the last four digits of their SSN, and should have asked for their complete SSN if it was really payroll related, which quite possibly meant that those two might have given out their complete SSNs if the phishing scam had asked for it. The external caller ID and synthetic voice were also mentioned as factors that made the survey respondents suspicious.

6 Limitations

The experiments were conducted in a university setting where the recipients are university staff and faculty. The demographics of the recipients in our study are not representative of the general population of telephone users in the US.

The experiments only sent out calls to a specific brand of work phones. The type of phone in our study is not representative of the entire population of telephones in the US. The vast majority of telephones in the US are mobile phones [33], and it is possible that these have different actual tricked rates than work phones. In addition, the participants had to be in their office when receiving the phone call (or to return our call if they listened to the voicemail), which is a different usage behavior compared to mobile phones.

The experiments requested only partial SSN information without storing it. The experiments had several safeguards, and the process was carefully designed and tightly regulated to ensure risks and harm to the human research subjects were minimized. This prevented us from collecting any actual Social Security numbers from the recipients. Collecting actual Social Security numbers might have changed the results of the study: more people might be willing to give out their full Social Security numbers, or more people could be skeptical of providing their full Social Security number.

As we did not collect the Social Security numbers directly, we derived a metric called “possibly tricked.” While the goal is to provide an estimate of the number of Social Security numbers that a real scammer would collect, this metric may

No.	Reasons Convinced	Reasons Unconvinced
E1		Would never enter SSN on incoming call; No name mentioned for the charge
E2		IRS won't make a call like this (x2); Already aware of scams like this
E3		IRS won't make a call like this
E4		IRS won't make a call like this; Didn't sound legitimate
E5		IRS won't make a call like this
E6		IRS won't make a call like this; Already aware of scams like this
E7		IRS won't make a call like this (x4); Indian accent (x2)
E8	To get paid (x2); Sounded legitimate; Trusted work phone; Only asked for last 4 SSN; Caller ID showed local ASU number	ASU won't make a call like this (x5); Not from ASU number (x2); Synthetic voice;
E9	Sounded legitimate; Only asked for last 4 digits of SSN; Caller ID showed ASU W-2	Should have asked for complete SSN (x2); Would never enter SSN on incoming call
E10	To get bonus (x2); Trusted work phone; From ASU number; Asked to do so	ASU won't make a call like this; Not ASU number

Table 6: Summary of recorded survey responses.

be under or overestimating the number of real collected Social Security numbers. With the data presented in this paper (Table 2), others can choose to use different metrics to calculate significance. These new metrics and hypotheses should be corrected to prevent *p*-hacking.

7 Discussion

Our results show that automated telephone phishing attacks can be effective. One experiment, E9, which simulated a targeted phishing attack with caller name spoofing, achieved a 10.33% possibly tricked rate, where recipients possibly divulged the last four digits of their Social Security numbers.

We have also validated some potential key attributes that can have a significant effect on the scam effectiveness: impersonating an internal entity and announcing a relevant scenario. Manipulating the caller ID to a toll-free area code may also somewhat improve the scam effectiveness for certain scams. Other attribute properties such as human voice, female voice, American accent, caller name spoofing, and fear-based scenario also improved the scam effectiveness in our empirical study, however, at this time we are not able to conclusively demonstrate that they have a significant effect. Nonetheless, given how easy it is for a scammer to manipulate all these attributes, a scammer would seek to incorporate all attribute properties that made an improvement to the attack success, i.e. a phishing scam with toll-free area code, spoofing known a caller name, speaking in a recorded human, female voice, in American accent, impersonating an internal entity, motivating the recipient with a relevant fear-based scenario.

To prevent falling victim to these types of phishing scams, we believe that the key is to target and prevent *impersonation*. Our statistical results have shown that impersonating an internal entity had a significant effect on the scam. To address the impersonation issue, feedback from our survey participants suggests that vigilance was an important reason for not falling for a scam. Many surveyed subjects expressed distrust towards our scam calls when they were already vigilant about the scam scenario. Therefore, we recommend education and awareness of telephone phishing as a countermeasure.

On technical solutions, we recommend a similar approach to help the subjects stay vigilant against phishing calls. There are solutions that can provide early warnings against impersonated calls, such as, caller ID authentication [4, 34, 35], which has strong safeguards against caller ID impersonation and could help to warn the users against malicious calls with a reputation system.

8 Related Work

To our knowledge, there have been no prior empirical user studies on telephone phishing. The most similar work we found was by Aburrous et al. [36], who performed a phone phishing experiment on a group of 50 employees contacted by female colleagues assigned to lure them into giving away their personal e-banking usernames and passwords. They were able to deceive 32% of the employees to give out their e-banking credentials. In the experiment, the 50 employees already knew the female colleagues that contacted them, which suggests an insider attack rather than an impersonation attack.

Other related work studied phishing using different channels. Dhamija et al. performed a website phishing study on 22 university participants and their best phishing site was able to fool more than 90% of participants [37]. Egelman et al. performed an email and website phishing experiment on 60 in-person participants to test the effectiveness of various web browser phishing warnings at that time, and it was found that 79% of Internet Explorer 7.0 participants heeded the active phishing warnings and only 13% of them obeyed the passive warnings [38]. Jagatic et al. performed a social media spear phishing study on 481 targeted Indiana University student emails obtained by crawling social network websites and it had a 72% success rate of recipients authenticating themselves on a redirected website [13]. Vidas et al. performed a QR code phishing study where the experiment distributed 139 posters containing QR codes at various locations at Carnegie Mellon University and the city of Pittsburgh, the experiment was able to trick 225 individuals to visit the associated website in four weeks [39].

9 Conclusion

This paper presented the methodology, design, execution, results, analysis, and evaluation of a study exploring why tele-

phone phishing works and how to defend against it. The study was executed using 10 experiments simulating telephone phishing attacks, administered to 3,000 work phones of university staff and faculty over the course of a workweek. The results were collected from the inputs and survey responses of the phone recipients. We analyzed the results by performing linear regression and statistical hypothesis testing methods on a chosen metric derived from the inputs, and we were able to identify at least one attribute that had a significant effect. We provided a discussion on how to effectively prevent such types of telephone phishing scams, and we believe that the best countermeasures should target impersonation and instill vigilance.

References

[1] Federal Trade Commission, "National Do Not Call Registry Data Book for Fiscal Year 2018," 2018.

[2] Federal Trade Commission, "Consumer Sentinel Network Data Book for January - December 2018," 2019.

[3] M. Tischer, Z. Durumeric, S. Foster, S. Duan, A. Mori, E. Bursztein, and M. Bailey, "Users Really Do Plug in USB Drives They Find," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2016.

[4] H. Tu, A. Doupé, Z. Zhao, and G.-J. Ahn, "Toward Authenticated Caller ID Transmission: The Need for a Standardized Authentication Scheme in Q.731.3 Calling Line Identification Presentation," in *ITU Kaleidoscope 2016 - ICTs for a Sustainable World*, ITU Telecommunication Standardization Sector (ITU-T), Institute of Electrical and Electronics Engineers (IEEE), Nov. 2016.

[5] I. S. working group, "Secure telephone identity revisited (stir)." https://datatracker.ietf.org/wg/stir/about/. (Accessed on 04/30/2019).

[6] H. Tu, A. Doupé, Z. Zhao, and G.-J. Ahn, "SoK: Everyone Hates Robocalls: A Survey of Techniques against Telephony Spam," in *Proceedings of the 37th IEEE Symposium on Security and Privacy*, IEEE, 2016.

[7] M. Sahin, A. Francillon, P. Gupta, and M. Ahamad, "SOK: Fraud in telephony networks," in *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroS&P'17), EuroS&P*, vol. 17, 2017.

[8] C. Peeters, H. Abdullah, N. Scaife, J. Bowers, P. Traynor, B. Reaves, and K. Butler, "Sonar: Detecting SS7 Redirection Attacks Via Call Audio-Based Distance Bounding," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2018.

[9] B. Reaves, L. Blue, D. Tian, P. Traynor, and K. R. Butler, "Detecting SMS Spam in the Age of Legitimate Bulk Messaging," in *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pp. 165–170, ACM, 2016.

[10] A. Oest, Y. Safaei, A. Doupé, G.-J. Ahn, B. Wardman, and G. Warner, "Inside a Phisher's Mind: Understanding the Anti-phishing Ecosystem Through Phishing Kit Analysis," in *Proceedings of the Symposium on Electronic Crime Research (eCrime)*, May 2018.

[11] R. C. Dodge Jr, C. Carver, and A. J. Ferguson, "Phishing for user security awareness," *Computers & Security*, vol. 26, no. 1, pp. 73–80, 2007.

[12] P. Kumaraguru, Y. Rhee, A. Acquisti, L. F. Cranor, J. Hong, and E. Nunge, "Protecting people from phishing: the design and evaluation of an embedded training email system," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 905–914, ACM, 2007.

[13] T. N. Jagatic, N. A. Johnson, M. Jakobsson, and F. Menczer, "Social phishing," *Communications of the ACM*, vol. 50, no. 10, pp. 94–100, 2007.

[14] P. Kumaraguru, S. Sheng, A. Acquisti, L. F. Cranor, and J. Hong, "Teaching johnny not to fall for phish," *ACM Transactions on Internet Technology (TOIT)*, vol. 10, no. 2, p. 7, 2010.

[15] N. Miramirkhani, O. Starov, and N. Nikiforakis, "Dial one for scam: A large-scale analysis of technical support scams," in *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2017.

[16] E. Fletcher, "That's not your neighbor calling." https://www.consumer.ftc.gov/blog/2018/01/thats-not-your-neighbor-calling, Jan. 2018.

[17] V. B. Payas Gupta, Bharat Srinivasan and M. Ahamad, "Phoneypot: Data-driven Understanding of Telephony Threats," in *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2015.

[18] A. Marzuoli, H. A. Kingravi, D. Dewey, A. Dallas, T. Calhoun, T. Nelms, and R. Pienta, "Call me: Gathering threat intelligence on telephony scams to detect fraud," *Black Hat*, 2016.

[19] I. R. Service, "Irs repeats warning about phone scams." https://www.irs.gov/uac/newsroom/irs-repeats-warning-about-phone-scams, 8 2014. (Accessed on 04/20/2017).

[20] Andrew Johnson, Division of Consumer and Business Education, FTC, "Voicemail from an irs imposter? | consumer information." https://www.consumer.ftc.gov/blog/voicemail-irs-imposter, 9 2016. (Accessed on 04/20/2017).

[21] G. Stringhini and O. Thonnard, "That ain't you: Blocking spearphishing through behavioral modelling," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 78–97, Springer, 2015.

[22] J. Hong, "The state of phishing attacks," *Communications of the ACM*, vol. 55, no. 1, pp. 74–81, 2012.

[23] J. Pavia, "Sadly, irs phone scams are very successful 'businesses'." http://www.cnbc.com/2016/10/18/sadly-irs-phone-scams-are-very-successful-businesses.html, 10 2016. (Accessed on 04/20/2017).

[24] I. R. Service, "Irs warns of pervasive telephone scam." https://www.irs.gov/uac/newsroom/irs-warns-of-pervasive-telephone-scam, 10 2013. (Accessed on 04/17/2017).

[25] I. R. Service, "IRS Alerts Payroll and HR Professionals to Phishing Scheme Involving W-2s." https://www.irs.gov/uac/newsroom/irs-alerts-payroll-and-hr-professionals-to-phishing-scheme-involving-w2s, 3 2016. (Accessed on 04/17/2017).

[26] Federal Communications Commission, "Telephone Consumer Protection Act 47 U.S.C. § 227," 1991.

[27] K. Queen, "Guard the last 4 digits of your social security number: they're all id thieves need." http://blogs.creditcards.com/2015/11/social-security-last-4-digits.php, 19 2015. (Accessed on 08/30/2017).

[28] A. Acquisti and R. Gross, "Predicting social security numbers from public data," *Proceedings of the National academy of sciences*, vol. 106, no. 27, pp. 10975–10980, 2009.

[29] A. E. Deeb, "What to do with "small" data? rants on machine learning." https://medium.com/rants-on-machine-learning/what-to-do-with-small-data-d253254d1a89, 5 2015. (Accessed on 09/25/2017).

[30] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian journal of statistics*, pp. 65–70, 1979.

[31] J. Cohen, "Statistical power analysis for the behavioral sciences (revised ed.)," 1977.

[32] F. M. Wolf, *Meta-analysis: Quantitative methods for research synthesis*, vol. 59. Sage, 1986.

[33] "CTIA - The State of Wireless 2018." https://www.ctia.org/news/the-state-of-wireless-2018. (Accessed on 05/02/2019).

[34] C. Jennings, J. Peterson, and E. Rescorla, "Authenticated Identity Management in the Session Initiation Protocol (SIP)," *IETF*, 2016.

[35] B. Reaves, L. Blue, and P. Traynor, "AuthLoop: End-to-End Cryptographic Authentication for Telephony over Voice Channels," in *Proceedings of the USENIX Security Symposium (USENIX)*, 2016.

[36] M. Aburrous, M. A. Hossain, K. Dahal, and F. Thabtah, "Experimental case studies for investigating e-banking phishing techniques and attack strategies," *Cognitive Computation*, vol. 2, no. 3, pp. 242–253, 2010.

[37] R. Dhamija, J. D. Tygar, and M. Hearst, "Why Phishing Works," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 581–590, ACM, 2006.

[38] S. Egelman, L. F. Cranor, and J. Hong, "You've been warned: an empirical study of the effectiveness of web browser phishing warnings," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 1065–1074, ACM, 2008.

[39] T. Vidas, E. Owusu, S. Wang, C. Zeng, L. F. Cranor, and N. Christin, "Qrishing: The susceptibility of smartphone users to qr code phishing attacks," in *International Conference on Financial Cryptography and Data Security*, pp. 52–69, Springer, 2013.

Appendix

A Announcement Messages

A.1 Tax Lawsuit scenario

This is an urgent notice! This is a final notice from the IRS. The reason for this call is to inform you that the IRS is filing a lawsuit against you. Your action is required immediately, or a penalty will be assessed. To speak to an IRS agent and get more information about this case, please press 1 on your phone now.

A.2 Unclaimed Tax Return scenario

This is an urgent notice! This is a final notice from the IRS. The reason for this call is to inform you that the IRS has an unclaimed tax return for you that is due to expire within three days. Your action is required immediately. To speak to an IRS agent and get more information about claiming your tax refund, please press 1 on your phone now.

A.3 Payroll Withheld scenario

Dear ASU employee. This is an urgent notice! This is a final notice from the ASU W-2 administration office. The reason for this call is to inform you that to process your next Friday payroll, you are required to update your 2017 tax information immediately. To speak to a staff agent and get more information, please press 1 on your phone now.

A.4 Bonus Issued scenario

Dear ASU employee. This is an urgent notice! This is a final notice from the ASU W-2 administration office. The reason for this call is to inform you a performance bonus has been issued to your account. Your action is required immediately. To speak to a staff agent and get more information, please press 1 on your phone now.

B Follow-up Message

Please wait for the next available agent. Thank you for holding. Your call will be connected shortly. Please enter the last four digits of your Social Security number on your phone now.

C Debriefing Message

Hi, I am [redacted for anonymity] in the Department of Computer Science at Arizona State University. I am conducting a research study to measure the effectiveness of telephone phishing. The reason you are receiving this message is because I would like to inform you that what you just did could potentially lead you becoming exploited in a real telephone scam. However, I would like to assure you that this is not an actual scam, none of your social security information was actually collected.

We would like to invite you to participate in our phone survey, to help us better understand your thoughts about the scam. You will be able to listen to the survey questions right after this message. Your participation in this survey is voluntary. There are no foreseeable risks for your participation. If you choose not to participate or to withdraw from the survey at any time, there will be no penalty. Your responses will be anonymous. The results of this study may be used in reports, presentations, or publications but your identity will not be used. Please press 1 to listen to the survey questions or participate in the phone survey.

D Survey Questions

D.1 Did the scam convince you

Thank you. Could you please help us understand if the scam was able to convince you to enter your Social Security number? Please use the number on your keypad to answer this question. If "yes", please press 1. If "no" please press 0.

D.2 What factor made the scam convincing

Thank you. Could you please help us understand what was the most important factor that made the scam convincing? We will record your voice response for this question. At the tone, please state briefly what you thought was the most important factor. When you are finished, please press the pound key to end recording.

D.3 What reason made the scam unconvincing

Thank you. Could you please help us understand what was the most important reason you did not believe in the scam? We will record your voice response for this question. At the tone, please state briefly what you thought was the most important reason. When you are finished, please press the pound key to end recording.

E Ending Message

Thank you. This is the end of the research experiment. If you have any questions concerning the research study, please contact the research team at [redacted for anonymity]. If you have any questions about your rights as a participant in this research, or if you feel you have been placed at risk, you can contact the Chair of the Human Subjects Institutional Review Board, through the ASU [redacted for anonymity], at [redacted for anonymity]. Thank you for your participation. Goodbye.

Platforms in Everything: Analyzing Ground-Truth Data on the Anatomy and Economics of Bullet-Proof Hosting

Arman Noroozian[1] ✉, Jan Koenders[2], Eelco van Veldhuizen[2],
Carlos H. Ganan[1], Sumayah Alrwais[3], Damon McCoy[4] and Michel van Eeten[1]

[(1)] *Delft University of Technology,* [(2)] *Dutch National High-Tech Crime Unit,*
[(3)] *King Saud University and International Computer Science Institute,* [(4)] *New York University*

Abstract

This paper presents the first empirical study based on ground-truth data of a major Bullet-Proof Hosting (BPH) provider, a company called MaxiDed. BPH allows miscreants to host criminal activities in support of various cybercrime business models such as phishing, botnets, DDoS, spam, and counterfeit pharmaceutical websites. MaxiDed was legally taken down by law enforcement and its backend servers were seized. We analyze data extracted from its backend databases and connect it to various external data sources to characterize MaxiDed's business model, supply chain, customers and finances. We reason about what the "inside" view reveals about potential chokepoints for disrupting BPH providers. We demonstrate the BPH landscape to have further shifted from agile resellers towards marketplace platforms with an oversupply of resources originating from hundreds of legitimate upstream hosting providers. We find the BPH provider to have few choke points in the supply chain amendable to intervention, though profit margins are very slim, so even a marginal increase in operating costs might already have repercussions that render the business unsustainable. The other intervention option would be to take down the platform itself.

1 Introduction

"Bullet-proof" hosting (BPH) is a part of the hosting market where its operators knowingly enable miscreants to serve abusive content and actively assist in its persistence. BPH enables criminals to host some of their most valuable resources, such as botnet command-and-control (C&C) assets, exploit-kits, phishing websites, drop sites, or even host child sexual abuse material [1–5]. The name refers to the fact that BPH provides "body armor" to protect miscreants against interventions and takedown efforts by defenders and law enforcement.

Much of the prior work in this area has focused on how to identify such malicious providers. Initially, BPH providers served miscreants directly from their own networks, even though this associated them with high levels of abuse. Famous examples of such providers include McColo Corp. [6], the Russian Business Network (RBN) [7], Troyak [3] and Freedom Hosting [8]. This operational model enabled AS-reputation based defenses, such as Fire [9], BGP Ranking [10] and ASwatch [11]. These defenses would identify networks with unusually high concentrations of abuse as evidence for the complicity of the network owner, and thus of BPH.

AS-reputation defenses became largely ineffective when a more "agile" form of BPH emerged. In this new form, providers would rent and resell infrastructure from various legitimate upstream providers, rather than operate their own "monolithic" network. Concentrations of abuse were diluted beyond detection thresholds by mixing it with the legitimate traffic from the ASes of the upstream providers.

In response, researchers developed a new detection approach, which searched for concentrations of abuse in sub-allocated IP blocks of legitimate providers [4, 5]. This approach assumes that honest upstream providers update their WHOIS records when they delegate a network block to resellers. It also assumes that the BPH operator functions as a reseller of the upstream providers.

A key limitation of this prior work is that it is based on external measurements. This means that we have little inside knowledge of how BPH operations are actually run and whether assumptions behind the most recent detection approaches are valid. A second, and related, limitation is the lack of ground-truth data on the actions of the provider. There are minor exceptions, but even those studies contain highly sparse and partial ground-truth data [2, 5].

This paper presents the first empirical study of BPH based on comprehensive internal ground-truth data. The data pertains to a provider called MaxiDed, a significant player in the BPH market. It unearths a further, and previously unknown, evolution in the provisioning of BPH, namely a shift towards platforms. Rather than MaxiDed renting and reselling upstream resources on its own, it offered a platform where external merchants could offer, for a fee, servers of upstream providers to MaxiDed customers, while explicitly indicating what kinds of abuse were allowed. By operating as a platform, MaxiDed externalizes to the merchants the cost and risk of acquiring and abusing infrastructure from legitimate upstream providers. The merchants, in turn, externalize the risk of customer acquisition, contact and payment handling to the marketplace. This new BPH model is capable of evading the state-of-the-art detection methods. Our analysis shows that

in most cases, there are no sub-allocations visible in WHOIS that can be used to detect abuse concentrations, rendering the most recent detection method [5] much less effective.

Before we can develop better detection and mitigation strategies, we need an in-depth empirical understanding of how this type of provider operates and what potential chokepoints it has. To this end, we analyze a unique dataset captured during the takedown of MaxiDed by Dutch and Thai law enforcement agencies in May 2018 [12]. The confiscated data includes over seven years of records (Jan 2011 – May 2018) on server packages on offer, transactions with customers, provisioned servers, customer tickets, pricing, and payment instruments. In addition to the confiscated systems, two men were arrested: allegedly the owner and admin of MaxiDed.

The central question of this paper is: *how can we characterize the anatomy and economics of an agile BPH provider and what are its potential chokepoints for disruption*? We first describe how the supply chain is set up. Then, we characterize and quantify the supply, demand, revenue, payment instruments and profits of the BPH services offered by MaxiDed. All of this will be analyzed longitudinally over seven years. We also explore what MaxiDed's customers used servers for.

Our main contributions may be summarized as follows:

- We provide the first detailed empirical study of the anatomy and economics of an agile BPH provider based on ground-truth data.
- We map the supply of BPH services and find a highly diversified ecosystem of 394 abused upstream providers.
- Contrary to conventional wisdom, we find that the provider's BP services are not expensive and priced at a 40-54 % markup to technically similar non-BP offers.
- We quantify demand for BPH services and find it resulting in a revenue of 3.4M USD over 7 years. We conclude the market to be constrained by demand, not by supply, i.e. demand for this type of agile BPH seems limited.
- We estimate profits to amount to significantly less than 280*K* USD over 7 years. This belies the conventional wisdom of BPH being a very lucrative business.
- We find disruptable pressure points to be limited. Payment instruments were sensitive to disruption, but a recent shift to crypto-currencies limits this option. We identified 2 merchants and a set of 15 abused upstream hosting providers as pressure points though their identifcation would have been difficult based on external measurements. The only remaining viable options are raising operational costs and taking down the provider's platform.

We should note that the "bullet-proof" metaphor seems less suited for this new model of BPH provider that we study. Commonly, BPH is understood to include two aspects: (i) intentionally enabling abuse, and (ii) providing resilience against takedowns. The BP metaphor directs attention to the resilience. This new business model, however, primarily focuses on the agile enabling of abuse at low cost. MaxiDed and its external merchants provide servers for abuse at close to the market price for legitimate servers. Customers then prepay the rent for these servers. This means that the risk of takedown, in terms of a prepaid server being prematurely shut down by the upstream provider, is borne by the customer. Most customers manage this risk by opting for short lease times and treating servers as disposable and cheaply replaceable resources. They take care of the resilience of their services themselves, using these disposable resources. Some forms of resilience – e.g., reinstalling an OS and moving files to a new server – are provided by the BPH provider as a premium service for an additional fee. The 'bullet-proof' metaphor is less suitable for this business model. A more fitting alternative may be "agile abuse enabler". That being said, in this paper we retain the existing term. The market of intentionally provisioning hosting services for criminals is still widely referred to as BPH and we want to maintain the connection with prior work.

The remainder of this paper is structured as follows. First, we provide a high-level overview of MaxiDed 's business (𝕊.2). We then discuss the ethical issues related to our study (𝕊.3). Next, we describe our datasets (𝕊.4) and the integrity checks we performed to ensure the validity of our analysis (𝕊.5). We then outline MaxiDed's anatomy and business model (𝕊.6). Next, we turn to the substantive findings and analyze the supply and demand around MaxiDed's platform, with a specific focus on identifying choke points (𝕊.7). We also analyze MaxiDed's customer population (𝕊.8). We then take a look at longitudinal patterns in terms of use and abuse of BP servers by customers (𝕊.9). The final part of the analysis is on MaxiDed's revenue, costs and profits (𝕊.10). We conclude by locating our study within the related work(𝕊.11) and by discussing its implications for the problem of BPH (𝕊.13). Additional material are provided in Appendices (𝕊.14)

2 Background

MaxiDed Ltd. was a hosting company legally registered in the Commonwealth of Dominica, an island state in the West Indies that is also known for its offshore banking and payments processing companies. MaxiDed's operators publicly advertised the fact that customers were allowed to conduct certain abusive activities upon purchasing its hosting solutions. While WHOIS information of the MaxiDed domain shows that it has existed since 2008, web archive data suggest that initially it was just a small hosting provider with no mention of allowing illicit activities. It underwent a major transformation in 2011 towards becoming an agile BPH service. MaxiDed does not have its own Autonomous System, nor does it have any IP address ranges assigned to it by RIRs, according to our analysis of WHOIS data at the time of its disruption. This implies that IP addresses are provisioned to customer servers by upstream providers, rather than by MaxiDed. This underlines

	Advertised BPH Services			
BPH	**Dedicated Servers**	**VPS**	**Shared Hosting**	**Total**
66host	0	0	3	3
outlawservers	1	6	4	11
abusehosting	47	5	3	55
bpw	5	4	0	9
bulletproof-web	7	9	0	16
MaxiDed	1,855	1,066	0	2,921

Table 1: MaxiDed in comparison with previously studied BPH by Alrwais et al.[5] that appear to be still operational

MaxiDed's agile nature, i.e., its reliance on reselling upstream infrastructure. Table 1 compares MaxiDed with several previously studied agile BPH providers in terms of the quantity and types of services they offered. It highlights that its scale of operations is around two orders of magnitude larger. It is reasonable to view the provider as a major player in this market which others have similarly pointed to [13].

3 Ethics

Our data is similar in nature to that used in prior studies of criminal backends [14–16]. It originates from legal law enforcement procedures to seize infrastructure. Using such data raises ethical issues. We operated in compliance with and under the approval of our institution's IRB. We discuss further issues using the principles identified in the *Menlo Report* [17].

(**Respect for persons.**) The data contains personally identifiable information (PII) on customers, merchants and employees. Access has been controlled and limited to authorized personnel within the investigative team, and later granted to several of the co-authors. Since 'participation' in this study is not voluntary and cannot be based on informed consent, we took great care not to analyze PII on customers, because they form the most vulnerable party involved and not all of them may have used servers for illicit purposes. We only compiled aggregate statistics. For merchants, we have masked identities using pseudonyms to prevent identifiability. We did not analyze the data in terms of MaxiDed employee names.

(**Beneficence.**) We believe that our analysis does not create further harm. We did not purchase services from the provider and thus did not contribute to any criminal revenue. The authors and police investigators believe the benefits of a better understanding of BPH operations, most notably in terms of better countermeasures, outweigh the potential cost of making this kind of knowledge more widely known, as the model of agile BPH itself is already well-documented in prior work.

(**Justice.**) The benefits of the work are distributed to the wider public, in terms of helping to reduce crime. It especially helps to protect persons who are more vulnerable to being victimized. We see no impact to persons from being included in the study itself.

(**Respect for law and public interest.**) This study has been conducted with the approval of, and in collaboration with, the investigative team and public prosecutors. It is important to note, that while captured information may point to certain illegal conduct, establishing legal proof of criminal conduct is *not* the purpose of this study.

4 Data

From the servers seized during the takedown, the Dutch investigative team has been able to resurrect MaxiDed's administrative backend (CRM and database). They have granted us access to the data and corresponding source code. We analyzed the source code to ensure correct interpretation of the stored data. We observed how various resurrected administrative pages queried specific records to display information.

The revived single-instance Postgres database contains longitudinal information on several key aspects of MaxiDed's operations. On the supply side, it includes data on what server packages were on offer, which merchants were offering these packages, and the internal and externally-advertised prices of each package. On the demand side, there is customer contact information, order placements, rented servers, server assigned IP addresses, financial transactions, and type of payment instruments used and available over time.

Communications between MaxiDed operators, customers, merchants, and upstream providers were captured as CRM system tickets. Ticket contents and email communications also include instances of abuse complaint emails that MaxiDed administrators received and forwarded to their customers. We should note that the operators also operated a live-chat channel for customers on the site. They were also known to use ICQ, Jabber and Skype contact channels at some point in time. These communications were not stored on the seized servers, if they were stored at all. Communications data, often the most sensitive, have not been analyzed in favor of the ethical principles that we followed.

Overall, the retrieved data represents information over the course of MaxiDed's life span from Jan.- 2011 to May-2018, when its operation was disrupted. High level statistics and descriptions of the ground-truth data is presented in Table 2.

To enrich the ground-truth data, we deployed several additional data sources. Domain-based resources operating from the customer IPs, were identified using historical passive DNS data collected via Farsight Security's (DNSDB [18]). To identify upstream providers of servers and IPs, we used historical WHOIS IP allocation data from Maxmind [19]. A set of domain and IP-based blacklists have been used to gain further insights into abuse emanating from customer servers.

5 Data Integrity

Since we did not gather the information ourselves, we need to evaluate its accuracy and authenticity: how do we know that MaxiDed admins did not manipulate data, for reasons of operational security or otherwise?

Our data resulted from the legal seizure of servers, in close coordination with apprehension of two individuals who had

Data on	Description	Total Nr.
Suppliers	60 directly listed upstream hosters and 14 listed merchants supplying server packages	74
Server Packages	Customizable server packages on offer during 2011-2018	56113
Payment Instruments	Supported payment instruments/methods	23
Orders	Customer placed orders for various server packages and other administrative services	66886
Users	Number of registered users	308396
Transactions	Financial transactions including 30938 received payments and 33124 payments made to other entities	64602
Tickets	CRM system tickets capturing communications between various entities	26562

Table 2: High-level statistics of MaxiDed backend data

administrative control over these systems. This ensured that the data was not manipulated during or after the seizure. To ensure that data was not manipulated in the course of MaxiDed's operation, we have examined data integrity in several ways. We first discuss the correspondence of the seized data with external (third-party) data. Next, we analyze the internal consistency of the seized data itself.

The strongest indicator of integrity is that the seized server data was consistent with the data that was collected via legal intercept prior to the takedown. A wiretap had been running for over two years on the backend CRM server.

We also compared the data to snapshots of MaxiDed's webshop archives on Internet Archive between 2015-2018. We extracted all server package IDs that were on offer. All these IDs were present in our back-end data as well.

For a sample of over 50 server packages on sale in April 2018, we compared the internally recorded price with the prices of the entities listed as the upstream providers. These included packages from a Dutch and a German upstream hosting provider. For each package, we visited the supplier's website, customized a server package to match, and found its price to be correctly reflected by the internal price.

For the payment data, we were able to compare the WebMoney transactions logged in the database with data that was subpoenaed by Dutch law enforcement from WebMoney on transactions during a period of 10 days involving one particular WebMoney wallet address. Of 31 internally recorded transactions during this period via WebMoney, 17 were matched with the external data.

Together, these external checks provide confidence that the internal data has not been manipulated. Multiple *internal* data consistency checks were also carried out. We cross referenced customer order placements against server package data, to determine if all order placements consistently point to an existing package. Of the 14,702 customer orders for servers, we found 431 referencing package IDs that were not listed, indicating a 2.9% proportion of inconsistent order placement records. These references point to a set of 306 unique server packages (a 0.5% proportion of all server packages).

We also cross referenced MaxiDed operators' payments to their merchants, against server package data. These indirectly referenced specific server packages, thereby indicating what each payment is for. Of the 33,124 outgoing payments, we found 345 referencing packages that were not listed among the set of offered server packages (a 1.0% proportion of inconsistent payment records). Cross referencing the same payment data against customer orders, we found 474 outgoing payments referencing servers that were not listed among the orders of customers (a 1.5% of inconsistent payment records).

The timestamps of order placement and transactions were also analyzed, to check for suspicious gaps in the timeline. The longest gap was observed to be 76 days from 2011-03-31 to 2011-06-15. All remaining gaps (37) were at most 2 days long. Approximately an average number of 26 order placements per day were observed. For payment events, the longest timeline gap was observed to be 135 days pertaining to the data from the period between 2011-01-29 and 2011-06-13. The remaining gaps (5) were no longer than 1 day. An average number of 24 transactions per day were observed in the payment data.

The minor inconsistencies and timeline gaps for the most part relate to records from 2011 and 2012, a period corresponding to the initial set up and early growth phase of MaxiDed. A certain amount of inconsistency in database records is to be expected, but more so during the initial set up and growth phase of any organization. All in all, the internal and external consistency of the data merits confidence in its validity for the purposes of characterizing the overall anatomy and economics of MaxiDed 's BPH operation.

6 Anatomy of MaxiDed 's business

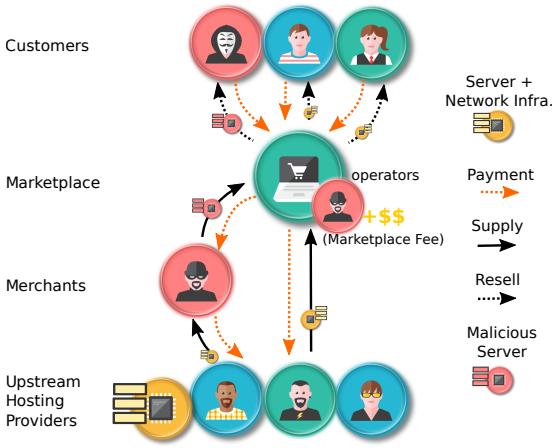


Figure 1: MaxiDed in a glance.

Figure 1 provides a high-level overview of MaxiDed 's anatomy and business model. We take a close look at each of its components.

6.1 Hosting Business Components

(**Marketplace**) MaxiDed was a marketplace which connected merchants offering server packages that allowed abuse, with

customers looking for an abuse-tolerant provider. It captured a fixed 20% fee from each sale between a merchant and a customer. Customers did not see the merchants' identities or even that an offer came from a separate entity. All they knew was that they contracted with MaxiDed. The merchants advertised server packages from legitimate upstream providers and put these on the MaxiDed market with a markup. Server packages specified default server configurations that were further customizable by customers. In addition to the technical specification, each package indicated what type of abuse, if any, was allowed. The majority of the packages explicitly allowed certain forms of abuse. MaxiDed itself also put server packages from certain upstream providers for sale in the webshop, de facto operating as merchant on its own platform. For its own packages, profits varied between 0 to 40% of the cost of packages at the upstream providers. What's more, MaxiDed also operated as a customer on its own platform, acquiring offers from merchants for its side business, a highly permissive and lucrative file sharing service called DepFile. This file sharing service was a major hub for distributing child sexual abuse material.

The platform approach means MaxiDed can externalize the cost and risks of acquiring and supplying upstream server infrastructure to third-party merchants. As such it is decoupled from the upstreams. The advantage for merchants, on the other hand, was that they could externalize the responsibility and risks of acquiring customers and processing their payments. Beside the fee that MaxiDed charged on top of the merchant's price, it also charged customers for performing additional administrative tasks, like re-installing servers after a takedown by the upstream provider. From these fees, it needed to recoup the cost of its staff and backend systems.

The main components of the marketplace were a frontend webshop, a backend Customer Relationship Management (CRM) system, accounts for merchants who could offer server packages on in the webshop, and payment handling of customers paying to MaxiDed and, in turn, MaxiDed paying the merchants when their offers resulted in a sale. The CRM, a series of webpages implemented in PHP, was used by both MaxiDed and merchants to create the server packages displayed on the webshop. It was also used to facilitate communications between customers and merchants through customer tickets. Merchants were responsible for handling customer tickets of their own server packages. Communications also took place through multiple MaxiDed support email addresses which were automatically imported into the backend database and live-chat functionality which was not retrievable from our data.

Different payment options have been supported over time by MaxiDed; 23 in total. Some from third-party payment providers like Paypal and WebMoney to cryptocurrencies such as Bitcoin and Zcash.

(**Merchants**) Third-party merchants supplied server packages that were re-branded and sold, with a mark-up, under MaxiDed's name. Many offered packages were directly scraped by the merchants from retail auction sites run by certain upstream providers. As far as we could tell, most merchants had no established reseller relationship with the upstream provider and no delegation was visible in IP WHOIS. (We explore this more systematically in 𝕊.7.3.) This invalidates a key assumption in prior work, i.e., that agile BPH providers operate on the basis of established reseller relationships that are visible in sub-allocations. In some cases, merchants did establish reseller relationships with an upstream provider. This allowed them to hook into an API and automate the importing and advertising process of upstream packages, rather than having to manually scrape other hosting provider's websites, in addition to receive certain discounts.

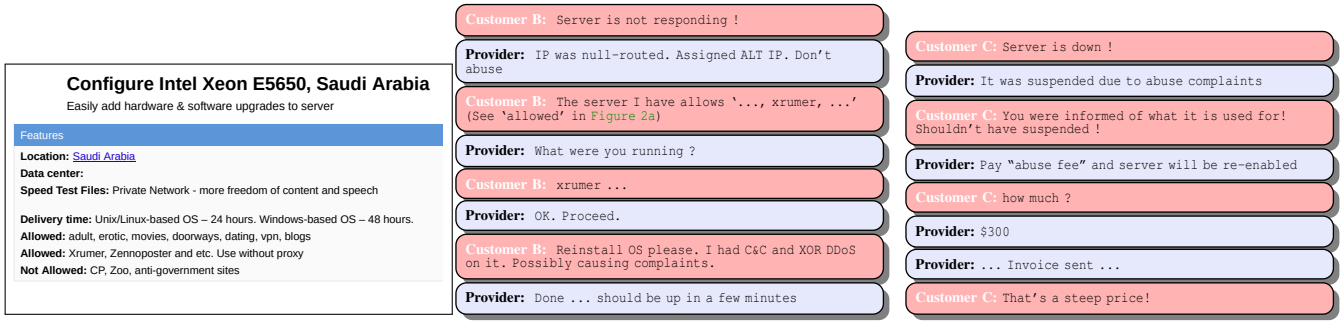
(**Upstream Providers**) These are legitimate hosting companies that offer server packages, via retail channels, auctions or reseller programs, which are put into the MaxiDed marketplace by the merchants. Once sold, the merchant acquires the package from the upstream provider. In 𝕊.7.3, we use WHOIS IP allocation information to infer from which upstream providers the merchants bought their packages.

(**Customers**) Customers were elicited for their preferences and guided towards server packages upon visiting MaxiDed's webshop. This occurred via standard search filters or via live chat with administrators. Customers were able to request more powerful hardware, additional IP addresses, pre-installation of a specific OS, and decide on the physical location of the servers. Figure 15 (see 𝕊.14 Appendix-A) provides an excerpt of a live chat conducted by one of the authors with MaxiDed operators prior to its takedown demonstrating this process.

Customers would first deposit funds into a USD denominated "wallet" and then use these wallet funds to pay for the invoices that MaxiDed issued to them. In other words, purchases were prepaid. This structure allows merchants to place orders only after receiving payments and to shift the risks of premature contract termination to customers as they have received payments in full. Customers were not reimbursed for lost server-day usage due to premature service suspension at the upstream.

6.2 Side Business

MaxiDed's administrators also operated a file sharing platform, known as DepFile [13, 20], run on servers which they rented through the MaxiDed marketplace. Some of these servers were also seized during the law enforcement action. Data shows that DepFile infrastructure was acquired using a single MaxiDed customer account which never paid its invoices. Over time, the account accrued approximately 400,000 USD in debt. DepFile allowed its customers to host and access content, some of which included child sexual abuse material, on a monthly subscription basis. Our separate analysis of internal DepFile data, suggest that it resembled a so called "affiliate program" [15, 21, 22] with affiliates bringing in new subscribers. The profits from subsequent sign-ups



(a) **(b)** **(c)**

Figure 2: Examples of MaxiDed's bullet-proof behavior. (a) screenshot of server publicly advertised to customers. (b) and (c) are excerpts of a conversation between customer and administrator (edited for readability).

were shared between DepFile (a.k.a. MaxiDed) and the affiliates. As an aside: these profits were much higher than those of MaxiDed. One could argue that the MaxiDed was more valuable to its owners as a way to acquire cheap and risk-free server infrastructure than as its own profit model.

6.3 Examples of Bullet-Proof Behavior

Figure 2a shows a screenshot of one of MaxiDed's publicly advertised server packages along with descriptions of its location, network/IP-address information, price, in addition to explicit descriptions of abusive activities that were (dis-)allowed upon purchasing. Figure 2b illustrates a conversation (lightly edited for spelling) that took place between an admin and a customer in the context of a CRM ticket. XRumer is a tool aimed at boosting search engine rankings by auto-registering accounts and posting link spam. It demonstrates that MaxiDed operators were not only explicitly tolerating abuse, but that they were informed about the abusive activities of their customers and actively supported them. This is also the case for DepFile. It knows the file sharing service is supporting illegal content, including child sexual abuse material. The customer interaction also shows the admin ignoring abuse complaints, then assisting the customer by migrating resources to a different network location. Figure 2c is another example of a (lightly-edited) conversation excerpt, demonstrating that certain customers were asked to pay an 'abuse fee' to continue accessing their rented server upon receiving abuse complaints.

7 Supply and Demand for BPH

MaxiDed's operations deviate from certain assumptions underlying recent detection techniques. This warrants a more detailed analysis of its characteristics to understand if this new form of agile BPH exhibits chokepoints that allow for disruption. Most disruption strategies rely either on taking down the provider as a whole or on cutting off the supply of resources that it needs: servers, connectivity, payment instruments, customers. In MaxiDed's case, the former occurred. These kinds of takedowns however, are rare and hard to scale. This section explores the alternative strategy: squeezing potential chokepoints in the supply chain.

7.1 Merchants

In a period of seven years, merchants offered 56,113 different server packages. Around a quarter of all packages (14,931) explicitly allowed certain kinds of abuse. We refer to these as bullet-proof (BP) packages. Note that non-BP packages were also abused, as we learned from customer tickets when servers were suspended. Admins frowned on this practice. Not because of the abuse itself, but because these customers should have purchased a more expensive abuse-allowing package. MaxiDed admins listed offers as well in the role of a merchant on their own platform. We label MaxiDed as *merchant zero* (mc_0) and 14 third-party merchants as $mc_{1...14}$, identified by connecting MaxiDed's user and supplier database tables.

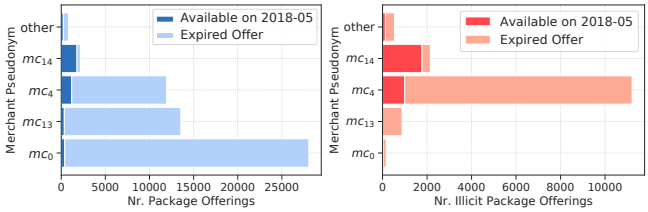


Figure 3: Merchant Package Offerings. (**left**) All packages; (**right**) Subset of illicit packages

Figure 3 (left) illustrates the total number of server packages offered by the top 4 merchants, which accounted for 98% of all packages. At the moment of takedown (May 2018), there were 3,957 available packages. Of these, 2,921 (74%) explicitly allowed abuse. Packages expired when corresponding upstream provider packages expired or when operators no longer maintained relationships with the upstreams.

Figure 3 (right) shows the subset of server packages that allowed abuse, from the same top four merchants. This figure highlights that two merchants, mc_4 and mc_{14} were responsible for 89% of all the BP packages offered on MaxiDed's platform and 94% of the BP packages available at the moment of the takedown. Interestingly, MaxiDed itself (mc_0) supplied only 29 BP packages (1%), relying almost exclusively on its merchants to supply BP infrastructure. This fits with our interpretation that moving to a platform model allowed MaxiDed to externalize the risk and cost of managing the relationships with upstream providers around abusive practices.

Of the 14,931 BP packages on offer, only 3,066 (20%)

were ever sold. There were 9,439 customer orders for these. This indicates that there was an oversupply of BP packages on MaxiDed. Sales followed a similar distribution to supply, with mc_4 and mc_{14} accounting for 70% of all sales. (Of the packages that did not explicitly allow abuse, 2,006 were sold 4,832 times.)

In sum, only around 20% of offers were ever sold, showing that the market for BPH is, unfortunately, not supply-constrained. MaxiDed externalized the supply of BP packages to merchants and two of these were dominant, in terms of supply and sales. Merchants mc_4 and mc_{14} would have been viable candidates for disrupting the supply chain of the marketplace as a whole, had they been identified prior to MaxiDed's takedown. This might be feasible if, as prior work assumed, they are resellers of upstream providers and WHOIS records are updated to show which network blocks are delegated to them. We later discuss evidence that, in most cases, there is no such delegation. The takedown of MaxiDed itself is unlikely to have disrupted these merchants. They may have taken some losses from outstanding due payments from MaxiDed. Except for these losses, merchants could migrate to other marketplaces, resulting in a game of whack-a-mole. This demonstrates the advantages of merchants externalizing part of their risks to the MaxiDed platform.

7.2 BP Package Categories

BP packages were differentiated in terms of what types of abuse was allowed. The platform pre-defined 12 categories of abusive activities. Merchants could tick the boxes of whatever categories they were comfortable with for their packages. The activities ranged from the distribution of pornographic content or copyrighted material, to Internet-wide scanning, running counterfeit pharmacies, running automated spamming software such as Xrumer, and doing IP spoofing, typically to conduct amplification DDoS attacks. Table 3 lists these activities along with associated category labels $C_{1..12}$.

We suspect merchant choices for certain types of abuse to have been partly driven by what they could handle in terms of their relationship with the upstream provider of a package. Some forms of abuse trigger more backlash than others. Plus, certain upstreams might be less vigilant regarding certain forms of abuse, depending on jurisdiction or other factors.

To analyze the relationships among the allowed forms of abuse, we calculate the correlations between all categories. In other words, if category 'c_X' is allowed, what is the probability that category 'c_Y' is also allowed? The results are plotted in Figure 4. Five groups of server packages can be identified, each with a different type of abuse profile, which roughly corresponds to a certain risk profile. At the top end of the risk profile is "spoofing" ($x = c_{12}$). Where this was allowed, everything else was also allowed with high probability (i.e., all values along the y-axis indicate high probability for $x = c_{12}$). As such a highest risk group label G_5 was assigned to packages that allow "spoofing". One step down are packages that allow "scanning" ($x = c_{11}$): everything else is typically allowed, except "spoofing" ($x = c_{11}, y = c_{12}$), which has a lower probability. This is group G_4. Next, G_3 was assigned to a group composed of 4 categories, $C_{7..10}$ which were allowed in conjunction with a high probability, and disallowed the higher risk $c_{11..12}$ categories with a high probability. The remaining groups were created using a similar logic.

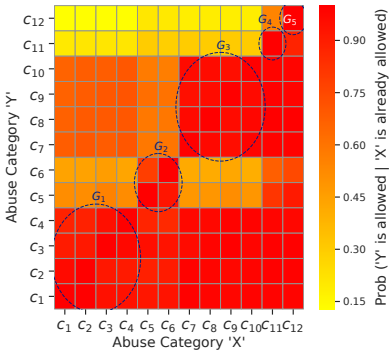


Figure 4: Correlation of abuse categories. (See Table 3 for c_i labels).

Cat.	Description	All packages	Avail. before takedown	Risk Group	Avail. per-group
C_1	File Sharing	12,344	2,724	G_1	404
C_2	Content Streaming	11,891	2,629		
C_3	WAREZ	11,856	2,615		
C_4	Adult Content	10,732	2,557		
C_5	Double VPN	10,099	1,529	G_2	630
C_6	Seedbox	8,835	1,298		
C_7	Gambling	2,663	1,862	G_3	1,279
C_8	Xrumer	3,120	1,849		
C_9	DMCA ignore	2,978	1,841		
C_{10}	Pharma	2,620	1,821		
C_{11}	Scanning	629	565	G_4	254
C_{12}	Spoofing	396	354	G_5	354

Table 3: Statistics on packages allowing each category of illicit activity and associated risk groups

For each risk group, Table 3 lists the abuse types and the number of packages that allowed it, over the whole period of MaxiDed ('all packages') or at the moment of the takedown ('Avail. before takedown'). Note that packages are counted multiple times, as they often allowed multiple forms of abuse. The last column, 'Avail. per group', counts each package as belonging uniquely to one group, namely the group with the highest risk profile – e.g., if a package allows spoofing, it will be counted in G5, but not in others, even though it likely also allows those types of activities. We can see that MaxiDed had a significant amount of supply in each category, with a clear peak in group 3.

A side note: the tickets and live chats clearly showed that other types of abuse were also allowed, such as running botnet C&C servers. The admins did not wish to list these forms of abuse publicly (see Figure 15 in 𝕊.14 Appendix-A).

7.3 Merchant Upstream Providers

To understand how MaxiDed's supply of BP infrastructure was distributed over legitimate upstream providers, we narrowed our analysis to 5 merchants, namely mc_0, mc_4, mc_{10}, mc_{12},

and mc_{14}, who jointly had 94% of the BP package sales.

Merchant mc_{14} sold most of the servers associated with risk groups G_3 or higher, the others sold mostly packages of group G_3 and below. So mc_{14} appears to have specialized in higher risk packages.

We determined each merchant's set of upstream providers by first extracting from the data the IP addresses provisioned once the server was sold. Maxmind's historical IP WHOIS data was then used to lookup organizations to which these IP address belonged. This way, we could see how each merchant's supply chain was composed of multiple upstream providers. The variance was significant. The two dominant merchants (mc_{10} and mc_{14}) abused 134 and 276 upstream providers, respectively. The others connected with 4 to 26 upstreams. Overall, `MaxiDed`'s supply chain comprised of servers at 394 upstream providers.

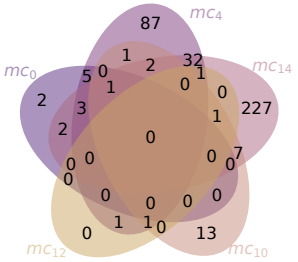


Figure 5: Upstream Overlaps

Figure 5 show how much, or rather how little, the supply chains of merchants overlapped in terms of upstreams. Figure 6 shows a CDF of how each merchant's sold BP servers were distributed across its own set of upstream providers. Across all merchants, 15 upstream hosted 50% of all sold BP servers and 57 account for 80% of all sold servers.

At first glance, the concentration in 15 upstream providers suggests a chokepoint that could be leveraged, but the long tail of available upstreams makes this strategy not very promising. Merchants could shift supply to those hundreds of alternatives. The 15 top ones might have certain advantages in terms of location, price and quality, but only 5 of them are shared between the two top merchants, so there does not seem to be a unique advantage to these providers.

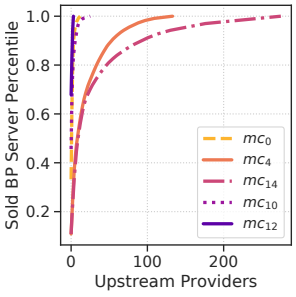


Figure 6: BP Server Distribution over Upstream Providers

Recent BPH detection approaches [5] have relied on upstream providers updating WHOIS records when they delegate network blocks to resellers. As stated, our data suggested that merchants often do not enter into reseller agreements with upstream. That would seriously undermine the effectiveness of these detection methods. To test this more systematically, we looked at the set of upstream providers that hosted 80% of the BP servers (57). In this set, we found 22 which are reputable upstream providers and more likely to reflect sub-allocations to their clients in WHOIS. We randomly sampled 10 BP servers for each of these 22 providers and manually inspected their IP WHOIS information. In only 24% of the cases did the WHOIS information reflect sub-allocation to downstream entities. Note that these downstream entities might also be legitimate resellers who sold to the merchants, rather than being the merchants themselves. Also, none of the records pointed to `MaxiDed`. This means that in 76% of the cases, the BP activities could not be associated with a sub-allocation, thus evading the current best detection method. Abuse on these addresses would be counted against the upstream provider, typically diluting the detectable concentration of abuse. Establishing a relationship between the upstream provider, their downstream customers, merchants and, ultimately, `MaxiDed`, would have been impossible with this kind of data.

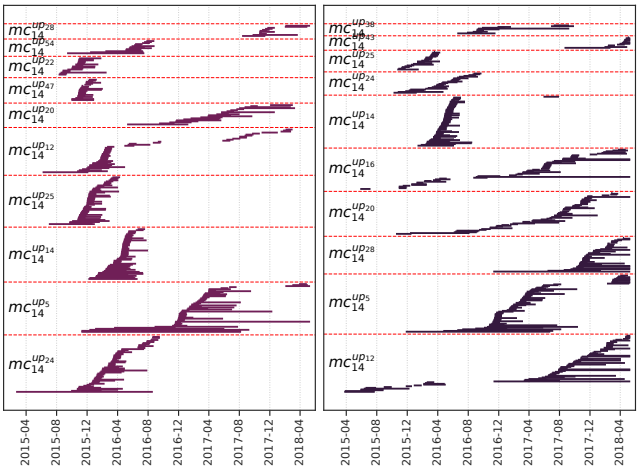


Figure 7: 10 most misused upstream providers via which mc_{14} provisioned BP servers of risk group G_4 (allowing "scanning" - left) and G_5 ("spoofing' - right'), plotted against server lifespans at each provider. Each colored line represents the lifespan of one server.

We next examined the distribution of each merchants' sold BP servers and server life spans across their corresponding upstream providers longitudinally. We visualize some of the results for mc_{14}, who was specialized in selling higher risk BP servers. Figure 7 plots the lifespan of mc_{14}'s sold BP servers that allowed "scanning" (left) and "spoofing" (right) for its 10 most misused upstream providers.

Figure 7 demonstrates that the merchant's BP customer servers were spatially as well as temporally spread across multiple upstream providers. It also shows that at no point in time, was there a shortage in the supply of servers even for the higher risk server packages. We observe no timeline gap during which servers of a particular group were not provisioned and active. We clearly observe a supply chain that was diversified, yet proportionally concentrated on a limited set of upstream providers. This approach of the merchant seems to be driven by a combination of efficiency in working with a limited set of upstreams and the flexibility of migrating from one upstream to the next, once the cost of working with that provider went up, perhaps because of mounting abuse complaints.

7.4 Payment Instruments

Next, we analyze the various payment instruments to identify potential chokepoints. From analyzing the source code of the webshop and the transactions in the database, we know that MaxiDed accepted payments via 23 different instruments. Three of these were actually never used by customers: Bitcoin Gold, Electroneum and Kubera Coin. Eight payment options were provided for a limited time and then discontinued by MaxiDed. At the moment of its takedown, 12 payment options were available. Some of these instruments, e.g., Paypal, were later restricted to specific groups of customers. Payments through Yandex Money were generally restricted to clients from Russia.

Figure 8 reconstructs transaction volumes over time for 20 payment instruments based on timestamps of financial transactions in the data. It plots a logscale of the number of transactions in each month. The Y-axes are the same for all instruments. First, we see that WebMoney has been a consistent and reliable payment provider for MaxiDed, basically from the start. Other instruments from that period proved more problematic. For example, Paypal became much more difficult to use in the course of 2015 and was abandoned completely in early 2018. We can see the operators deploying new ones and also abandoning some of them again. This process seems to suggest responding to potential or manifest disruptions via payment providers. Consistent with this interpretation is the increase in options to pay with cryptocurrencies. We first see a major shift to bitcoin at the end of 2013. Then, around the end of 2017, MaxiDed added 8 new cryptocurrencies. A preference to move to cryptocurrencies was also observed in backend data, where MaxiDed's operators maintained an explicit preference order for the different payment methods.

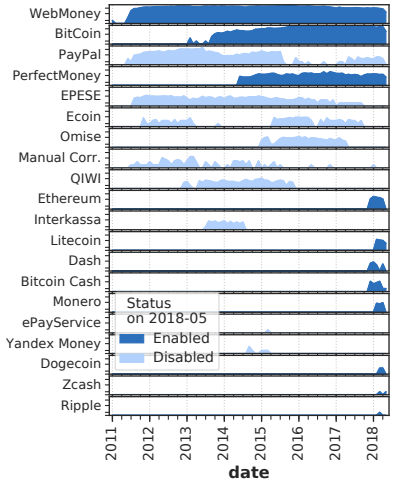


Figure 8: Payment instrument monthly transaction volume

Figure 9 plots the cumulative generated revenue for the top 5 most popular payment instruments. While WebMoney had brought in the most revenue, the total amount of bitcoin payments was growing rapidly and poised to overtake the leading position, until the takedown happened.

All in all, MaxiDed's revenue was generated through a small set of payment methods. The bulk of their customers used only one payment method. Disruption of MaxiDed's payment flow via WebMoney would have been a viable chokepoint in earlier phases. The self-imposed limits on using Paypal probably reflect the fact that those payments were vulnerable to countermeasures by Paypal. The shift towards cryptocurrency payments demonstrates that MaxiDed recognized this dependency, as well as illustrates how it was attempting to remediate it. It is clear that this shift makes disruption more difficult, though it is hard to gauge how resilient the bitcoin payment option actually was. This would require a study of the blockchain and the role of currency exchanges, which is out of scope for this study. That being said, the proliferation of cryptocurrency options might counteract the vulnerabilities associated with each specific instrument.

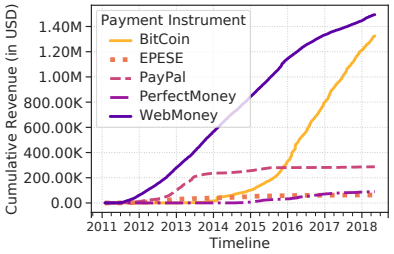
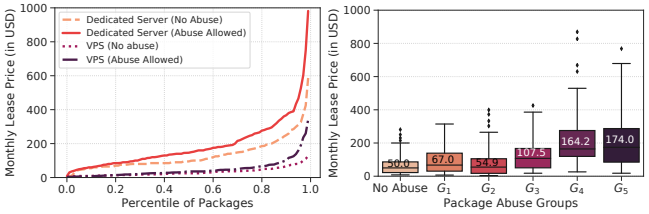


Figure 9: Revenue

7.5 Package Pricing

BPH businesses are typically understood as charging customers high markup prices for allowing illicit activities and offering protection against takedowns. There is anecdotal evidence (e.g., [2, 5]) that suggests prices are well above those for bonafide services. Our data, however, questions this widely-held understanding.

We first distinguished VPS packages from physical dedicated servers. In each category, we then compared the distribution of the monthly lease price of packages that allowed abuse versus those that did not. The results are plotted in Figure 10a. We observe that indeed abuse-enabling servers cost more, but the difference are modest across most of the distribution. For dedicated servers, the median price was 95.00 USD for non-BP packages and 146.00 USD for BP packages. For virtual servers, the median prices were 25.00 USD versus 35.00 USD. These numbers suggest that customers payed a median markup ranging from 40% to 54% for being allowed to abuse. This includes both the fee of MaxiDed as well as the margin of the merchant. The rest goes to the upstream provider.



(a) Price per package type **(b)** Price per risk group

Figure 10: Package pricing (See Table 3 for risk group labels).

We also compared package prices based on associated risk

groups of their packages. Figure 10b illustrates the results with median group prices indicated in the plot. Here, we observe larger prices differences. The median price of the highest risk packages are 3.5 times higher than those for the non-abuse packages.

The limited markup seen in the lower risk packages might reflect the fact that the platform has an oversupply of BP packages. Many packages never got sold. The platform also sets up the merchants to compete with each other. All of this might push prices down, towards the cost of the upstream package. Relatively low markup might also reflect less cost on the side of the merchant and marketplace because of takedown. Low prices may also be the result of MaxiDed's business model which pushes takedown risks to customers by requiring prepayment.

8 Customers

Law enforcement takedowns of online anonymous markets (a.k.a., dark markets) have targeted the platforms, the supply chains, but also the customers on these platforms, in an attempt to disrupt the demand side. The most ambitious operation was the coordinated Alphabay-Hansa market action, which de-anonymized many merchants and buyers [23]. As of yet, it is unclear if these actions will have any impact on the demand for these services. Nevertheless, we will take a closer look at the population of MaxiDed customers to understand how demand has evolved over time and whether it offers starting points for disruption.

MaxiDed's registration data shows that 308,396 unique users signed up to its platform. Figure 11 plots the cumulative number of registered, active and paying users over time. We find three outlier events, during which a large number of users appear to have been artificially created, that distort the numbers. Only 6,782 of the user population ever purchased server packages. Of these, 4,498 users were active in the sense that they logged into the platform's CRM at least once after having signed up. On average, the platform saw a daily growth of 3 user sign ups, excluding the three outlier events.

Cross referencing the user data, customer orders, and server package data, we find that the majority of the customers were interested in and may have engaged in abusive activities. This is observable in Figure 12 (left) which plots the cumulative number of customers, separating out those that eventually ended up purchasing BP servers. In the earlier stage of MaxiDed 's evolution, they still had a significant number of customers who never bought BP packages. A few years in, they attract an increasing number of users that do buy BP packages. At the time of its disruption, 66% of all customers ever to register had purchased BP packages. The remaining 34% was a mix of bonafide customers and customers who may have undertaken abusive activities on non-BP packages.

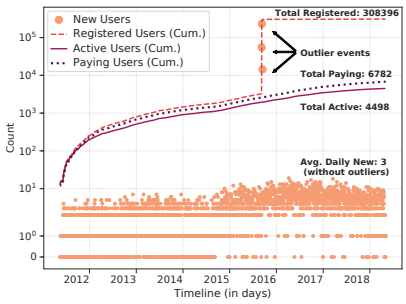


Figure 11: User number over time

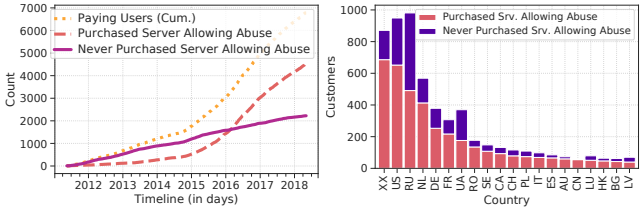


Figure 12: (**left**) Customer types; (**right**) Customer locations (XX = Location not specified)

Customers could specify language preferences in their profile: 5,085 selected English and 1,697 selected Russian. They were also asked to supply location information. Assuming that user-specified locations are correct, a crude assumption, then most users came from 3 countries, namely RU, US and NL (see Figure 12 - right), followed by a long tail of other countries.

9 Use and Abuse

Next, we explore server use and abuse by customers. We examine how customers manage takedown risks transferred to them by MaxiDed and look at the measure of last-resort, namely blacklisting BP servers once they are detected.

9.1 In Demand Abuse Categories

Our data contains timestamps of when servers were provisioned and when they were taken offline. Servers were deactivated when their lease expired or when abuse complaints caused the upstream provider to terminate the lease early. Figure 13 plots the number of active servers across various risk profiles. It shows what customers mostly sought to purchase.

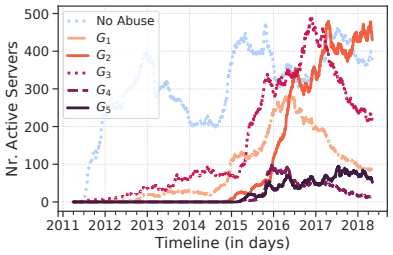


Figure 13: Active servers

After a start as a legitimate provider, BP servers become dominant over time (see Figure 13). Initially, customers were interested in spamming, operating phishing domains (which triggered DMCA complaints), running counterfeit pharma and gambling sites (risk profile G_3). Then we see a steady growth in demand for G_1: file sharing, streaming, adult content, and WAREZ forums. The rapid growth of MaxiDed, starting around the end of 2014, saw a diversification of the abuse and an increase of VPNs and seedboxes for file sharing (G_2),

scanning (G_4), and spoofing (G_5). These shifts reflect a wider trend towards commoditization of cybercrime services, such as the provisioning of DDoS-as-a-Service [1]. At its peak, MaxiDed administered 1,620 active BP and non-BP servers.

9.2 Abusive Server Uptime

MaxiDed and its merchants shifted the risk of takedown to their customers. They required prepayment, offered no reimbursements, and provided minimal resilience support with considerable attached "abuse fees".

Risk Profile	Payment Cycle (days)	Premature Termination (%)	Expired (%)	Extended (%)	Lost Usage (Median # days)	Total (# servers)
No Abuse	91.0	15.69	38.77	45.54	10	4,831
G_1	92.0	18.23	47.39	34.38	23	1,437
G_2	90.0	23.04	52.22	24.74	28	2,834
G_3	61.0	19.59	45.86	34.55	13	3,792
G_4	46.0	15.41	48.39	36.20	3	558
G_5	31.0	19.15	54.73	26.12	6	804

Table 4: Server lifespan statistics

How do customers deal with this risk? In essence: by choosing shorter lease periods for more risky activities. Table 4 lists the median lease periods that customers opt for across various risk groups. The more risky the abuse, i.e., the higher the probability of a takedown, the shorter the lease time. The table also provides statistics on the proportions of BP servers that were prematurely terminated due to abuse complaints, proportions of lease expirations, extensions, in addition to the number of usage days that customers lost from termination of their lease. Customers with the most risky activities manage to mitigate the cost of takedown to a median of 6 lost days.

We also see that at most 23% of the BP servers were prematurely taken down. Most BP server ran uninterrupted for their entire lease period. This speaks to the low rate of blacklisting, questioning the effectiveness of this practices in disincentivizing abuse. An interesting pattern is that customers also abused servers that did not allow abuse. 15% of these servers were also taken down.

Overall 2,656 servers were deactivated prior to the expiry of their lease plan. Another 6,483 active servers were deactivated when they reached their normal expiry term. 5,117 servers remained active beyond their initial lease plan.

9.3 Detected Abusive Resources

We next explore a final chokepoint: blocking the BP servers and abusive content hosted on them once they are discovered.

We triangulated these results by looking directly at several blacklists. We used three years of passive DNS data from Farsight Security's DNSDB to identify domain based resources on MaxiDed's IP addresses: fully qualified domain names (FQDNs) and 2nd-level domains (2LDs). Table 5 lists the quantities of resources associated with MaxiDed from 2016 to 2018. This period corresponds to when MaxiDed had the highest number of active servers. We examined the intersection between these resources and those flagged or blocked by several leading industry abuse feeds. The feeds capture a mix of spam, phishing, malware and botnet C&C abuse. Detailed information on these feeds is provided in Table 5. The quantities of flagged MaxiDed customer resources within each of these abuse feeds are also listed in the table. When no historical feed data was available, we left the cell empty.

While coverage of blacklists is known to be limited, it is quite disappointing to see the small fraction of the abuse that gets picked up by the feeds. This confirms, with ground truth, the observation in prior work that blacklisting is generally ineffective in disrupting abuse.

10 Marketplace Finances

Disruption of BPH is also determined by how profitable the business is. Lower margins mean that the provider is more vulnerable to raised operating costs in the supply chain. In this section, we analyze MaxiDed's revenue, costs and profits. To get a sense of the company as a whole, we include both BP and non-BP services.

(Revenue.) From the 23 different payment instruments employed by MaxiDed, most of its revenue was received via WebMoney payments (1,493,876 USD) followed by direct BitCoin payments (1,324,449 USD, MaxiDed itself logged these in USD). Around 577,118 USD was received through the remaining payment instruments. The total amount of revenue from 2011 up to May 2018, adds up to 3.4*M* USD.

(Operating Costs.) We have no data on personnel cost at MaxiDed. Here, we analyze the outgoing payments to merchants, upstreams and outstanding debts recorded in the database.

i) *Payments to Merchants.* A main component of MaxiDed's cost structure consists of payments to merchants. Merchant payments were exclusively deposited on WebMoney and Epayments wallets. After MaxiDed took their 20% fee, the remaining 80% went to the merchants. Analyzing outgoing MaxiDed payments show 11 of the 14 operating merchants to have received payments, adding up to 1,588,810 USD. Figure 14 illustrates the distribution of payments made to each merchant. The two largest suppliers of server packages, mc_4 and mc_{14}, received the bulk of the earnings. Most of the merchants were completely unsuccessful. The lowest earners, combined, generated less than 190K USD over all years.

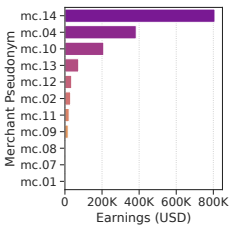


Figure 14: Payments to merchants.

ii) *Payments to Upstreams.* We cannot see the payments of third-party merchants to their upstreams, only the payments where MaxiDed is itself a merchant on the platform (mc_0). Data shows that mc_0 payments to their upstreams add up to 1,526,015 USD, paid via WebMoney

	Hosted resources			Number flagged resource in abuse feed																	
	IPs	FQDN	2LD	PHTK[1]			APWG[2]			SBW[3]			GSB[4]			DBL[5]			CMX[6]		
Year				(IP)	(FQDN)	(2LD)	(IP)	(FQDN)	(2LD)	(IP)	(FQDN)	(2LD)	(IP)	(FQDN)	(2LD)	(IP)	(FQDN)	(2LD)	(IP)	(FQDN)	(2LD)
2016	985	9,902	3,378	2	1	32	29	45	75	12	10	23	.	.	.	.	.	.	85	185	201
2017	906	15,494	3,573	5	2	18	1	4	23	.	.	.	4	63	71	40	644	696	22	20	51
2018	145	416	280	0	0	2	0	0	5	.	.	.	0	0	4	20	23	22	.	.	.

***Sources*:PHTK**: Phishtank[24], **APWG**: Anti-Phishing Working Group[25], **SBW**: StopBadware[26], **GSB**: Google Safe Browsing[26], **DBL**: Spamhaus[27], **CMX**: Clean-MX[28].
Notes: (1) Phishing; (2) Phishing; (3, 4) Malware drive-by; (5) SPAM, Malware, Phishing, botnet C&C; (6) Malware and Phishing.

Table 5: Statistics on flagged or blocked MaxiDed customer resources

and PayPal. Note that 99% of these payments were not for BP servers, as those were almost exclusively provided by the third-party merchants.

iii) *Debtors.* The final component of MaxiDed's costs structure is that of outstanding debts due from its customers. The operators have vigilantly banned customers with outstanding debts. One customer was the exception to this rule. Actually, this was not a real customer, but a customer account through which MaxiDed operators themselves purchased servers from merchants on their platform. These were used to host DepFile, their large file-sharing platform side-business. This customer entity accumulated debts amounting to 399,123 USD.

(Profits.) Table 6 details MaxiDed's yearly finances, alongside finances of their side business DepFile. Despite the common understanding of BPH services being lucrative, we clearly observe MaxiDed's earnings to be modest and declining. In total, over seven years, MaxiDed made just over 280K USD in profit. If we take out the debt incurred for the DepFile side-business ($399,123 + 280,618$), then the profit would have been 679,741 USD. This is still an underwhelming figure for 7 years of operating a BPH platform. Recall that the cost of personnel, office space, and equipment also has to be taken from this amount. These combined costs would have to be substantially lower than 100K USD per year to leave even a tiny profit on the balance sheet.

	MaxiDed			DepFile			
Year	Revenue	Costs	Prof_{mx}	Revenue	Costs	Prof_{dp}	(Σ Prof.$_i$)
2011	79,987	1,312	78,675	.	.	.	78,675
2012	345,213	72,418	272,794	.	.	.	272,794
2013	458,028	17,9761	278,266	334,540	248,307	86,233	364,499
2014	419,739	328,757	90,981	1,646,568	712,442	934,125	1,025,106
2015	615,046	570,895	44,150	2,205,687	1,396,820	808,867	853,017
2016	733,151	726,040	7,111	3,153,553	2,188,634	964,919	972,030
2017	566,471	872,520	-306,048	3,998,244	2,841,322	1,156,922	850,874
2018	177,806	363,118	-185,312	1,547,078	1,129,586	417,492	232,180
Total	3,395,444	3,114,825	**280,618**	12,885,673	8,517,113	**4,368,560**	**4,649,178**

Note: (*mx*: MaxiDed) (*dp*: DepFile)

Table 6: Yearly finances

The side-business DepFile, on the other hand, generated much better margins. We could even speculate that MaxiDed was more valuable to its owners as a way to acquire cheap and risk-free server infrastructure than as its own profit model.

11 Related Work

(Underground Ecosystems.) Several ecosystems and marketplaces of a malicious nature have been studied in the literature via captured datasets. Stone-Gross et al. analyzed credential stealing malware [29] and spam botnets [14] by taking over part of the botnet infrastructure to understand their inner workings. Wang et al. studied SEO campaigns to sell counterfeit luxury goods and the effectiveness of various interventions to combat such activities [30]. Alrwais et al.[34] investigate illicit activities in the domain parking industry by interacting with the services to collect ground truth data. Christin [31] analyzed the Silk Road marketplace by running daily crawls of its webservices for 6 months to understand merchants, customers, and what was being sold. A followup study by Soska and Christin [32] examined 16 anonymous market places also by periodically crawling their webservices and found that marketplace takedowns may be less effective than pursuing key merchants that may migrate to others. Another followup study by Wegberg et al. [33] augments previous studies by examining evidence for commoditization of entire cybercrime value-chains in underground marketplaces and finds that only niche value-chain components are on offer.

Datasets on the underground can also be leaked by criminal competitors. McCoy et al. used leaked databases of three affiliate programs to study pharmaceutical affiliate programs [15]. More recently, Brunt et al.[35] analyzed data from a DDoS-for-hire service and found that disrupting their regulated payment channel reduced their profitability but that they were still profitable by switching to unregulated cryptocurrency payments. Hao et al. [16] analyzed a combination of leaked and legally seized data to understand the ecosystem for monetizing stolen credit cards. Our dataset resulted from the aftermath of the legal takedown of the BPH provider MaxiDed. To the best of our knowledge, there has been no prior academic work on BPH using such ground-truth data. Our study uniquely provides a comprehensive picture of the supply, demand and finances of the entire BPH operation.

(Bulletproof hosting.) Earlier efforts on detecting BPH have relied heavily on identifying autonomous systems. Fire [9] was one of the first systems for detecting BP ASes by temporally and spatially aggregating information from multiple blacklists in order to detect elevated concentrations of persistent abuse within an AS's IP blocks. Shue et al. [36] noted that BP ASes often fast-flux their BGP routing information to evade detection. ASwatch [11] leveraged fast-fluxing

BGP routing as strong indicator of a BP AS to build a classifier and detect BP ASes before they appear on blacklists. Others have developed security metrics to compare concentrations of abuse on various hosting networks and to identify negligent providers that may be suspected of operating BPH services [37, 38], while Tajalizadehkhoob et al. developed techniques to analyze abuse concentration on the hosting market as a whole by identifying providers from their WHOIS information rather than BGP data [39]. BPH however, has evolved over time. Alrwais, et al.[5] studied a recent approach of BPH abusing legitimate hosting providers through reseller packages to provide a more agile BP infrastructure. Our work complements this work by providing a unique perspective into to the the ecosystem of BPH. Based on our analysis, we can better reason about which mitigation techniques might be effective and which are likely ineffective for undermining modern agile BPH marketplaces.

12 Limitations and Future Work

In comparison to other underground marketplaces studied previously (cf. [32, 33]), MaxiDed may be seen as a specialized marketplace for provisioning BP servers. While comparisons with other underground markets may be drawn, direct comparisons are difficult due to differences in how MaxiDed's marketplace operated. For example its customers were not aware that merchants were involved in supplying the marketplace with resources. This also explains why in comparison no reputation mechanisms were in place for customers to differentiate packages based on their quality (or differentiate good/bad merchants).

Despite such differences, we do still observe patterns similar to what other studies of criminal endeavors have reported. For example, we have observed a concentrated supply pattern around a handful of merchants in MaxiDed's case, which is a similar to what other studies of underground market places have observed ([32, 33]). We have also observed demand to gravitate towards the resources supplied by successful merchants. The number of successful merchants being limited, also agrees with studies of other criminal operations, e.g. in studying spam botmasters and their operations [14].

Given that this study has focused on an in-depth analysis of the anatomy and economics of MaxiDed, future work may draw more systematic comparisons to better understand the implications of what we has been reported here. Furthermore, MaxiDed's prominence within the ecosystem has also not been systematically explored in our study, albeit the limited comparisons with other BPH providers in addition to anecdotal evidence [4, 13] suggest that MaxiDed may be reasonably considered as a major provider within the ecosystem. Nevertheless, some of our findings, particularly those relating to the economics and profitability of BPH services may require further research to better understand the BPH ecosystem as a whole.

13 Discussion and Implications

(Discussion.) We found MaxiDed to have developed a new agile model in response to detection and disruption strategies. Its operations had matured to the point of a new innovation, namely operating a marketplace-like platform for selling BPH services. This model transfers the risks of acquiring the BP server infrastructure from upstream providers to merchants. MaxiDed 's main role was to take on the risks of acquiring customers, communicating with them and processing their payments. The 14 merchants on the platform (over)-supplied the market with more than 50K different server packages, many of which expired without being purchased. They abused a total of set 394 different upstream providers, thus allowing merchants to spread out and rotate abuse across many different legitimate networks.

We see some concentration in this supply chain, with 15 upstreams providing infrastructure for over 50% of the BP servers sold. Most of these upstream resources are not shown to be delegated in WHOIS, drastically curtailing the effectiveness of the most recent detection approaches. Another point of concentration is in the merchant pool: two merchants offered 89% of all BP servers and made 94% of the BP packages sales. Most other MaxiDed merchants failed to generate any meaningful sales. The platform deployed 23 different instruments to transact with customers over various periods. Revenue was initially largely processed by one payment settlement system: WebMoney. We also saw an increased volume of BitCoin payments and the adoption of other cryptocurrencies in response to disruptions in other instruments, such as PayPal. A lack of product differentiation on the market is likely to have created a fierce price competition across the merchants which in turn has led a great proportion of merchants to fail. This competition also decreases the profits of not only the merchants, but also of MaxiDed itself. Its profits, over seven years, amounted to a mere 280K USD (or 680K USD if we ignore cross subsidies to their other business, DepFile). The actual profits are even lower, as this amount also has to cover the cost of personnel, office space and equipment, on which we had no data.

(Implications.) Bullet-proof hosting (BPH) companies remain a difficult problem as their operators adapt to evade detection and disruption. Prior work in this area has largely relied on external measurements and generally lacks ground-truth data on the internal operations of such providers. Recent detection techniques rely on certain assumptions, namely that agile BPH operates under reseller relationships, and that upstream providers accurately reflect such relationships in their WHOIS information. We found MaxiDed to deviate from both assumptions, thus rendering detection less effective.

Prior BPH instances were mainly disrupted by pressuring upstream providers to sever ties with downstream BPH providers. Given the number of available substitute upstream providers of MaxiDed, this is unlikely to be an effective choke-

point. Drawing parallels with other underground markets suggest that, other than taking down the platform itself, disruption may also be achieved by pressuring other chokepoints: merchants, revenue and demand. MaxiDed's dominant merchants would have been a viable chokepoint, yet, identifying them most likely required internal operational knowledge as their existence and identities were not externally visible. As for disrupting payment channels, the transition to mostly unregulated cryptocurrencies payments suggest that this is no longer a straightforward option. Surprisingly, MaxiDed's low profits indicate that an increase in transaction or operating costs may be viable a pressure point to disrupt revenue and demand. Future work could explore how to raise these costs. Being aware of the threat of criminal prosecution might, ironically, be one way.

The final remaining pressure point would be to take down the platform. Such takedowns however are hard to replicate, let alone scale. That being said, MaxiDed explicitly marketed bullet proof services on the clear web. Even in cases when criminal prosecution itself is not feasible, if the threat can be made plausible, it might force the company to operate within higher op sec requirements, raising the cost of doing business. This suggests that what appears the more difficult strategy might actually be the best option in light of the supply chain becoming even more agile and evasive. Our hope is that by further studying and understanding of these emerging agile BPH services we can inform new and potentially more effective directions for mitigating this threat. To orient future work in this area, researchers might be better off deprecating the increasingly misleading metaphor of "bullet-proof" hosting in favor of a term like "agile abuse enablers".

Acknowledgments The authors would like to thank the anonymous reviewers of our study for their feedback and suggestions to improve the quality of our manuscript. We greatly appreciate the data sharing efforts of Farsight Security, and other organizations including Phishtank, APWG, Stopbadware, Spamhaus and CleanMX that have provided us with passive DNS and the abuse data on which parts of this study are based. We would like to thank the Dutch National High-Tech Crime Police unit for making this study possible as well as the Dutch Ministry of Economic Affairs and SIDN for supporting our research. Finally, we acknowledge funding support under NSF award number 1717062, DHS S&T FA8750-19-2-0009, and gifts from Comcast and Google.

References

[1] Kurt Thomas, Danny Yuxing, Huang David, Thomas J Holt, Christopher Kruegel, Damon Mccoy, Elie Bursztein, Chris Grier, Stefan Savage, and Giovanni Vigna. "Framing Dependencies Introduced by Underground Commoditization". In: *WEIS*. 2015.

[2] Brian Krebs. *Inside the Gozi Bulletproof Hosting Facility*. 2013. URL: https://krebsonsecurity.com/2013/01/inside-the-gozi-bulletproof-hosting-facility/.

[3] Danny Bradbury. "Testing the defences of bulletproof hosting companies". In: *Network Security* 2014.6 (2014), pp. 8–12.

[4] Dhia Mahjoub and Sarah Brown. *Behaviors and Patterns of Bulletproof and Anonymous Hosting Providers*. 2017. URL: https://www.usenix.org/conference/enigma2017/conference-program/presentation/mahjoub.

[5] Sumayah Alrwais, Xiaojing Liao, Xianghang Mi, Peng Wang, XiaoFeng Wang, Feng Qian, Raheem Beyah, and Damon McCoy. "Under the Shadow of Sunshine : Understanding and Detecting Bulletproof Hosting on Legitimate Service Provider Networks". In: *Proc. of IEEE S&P (Oakland)*. 2017.

[6] Brian Krebs. *Host of Internet Spam Groups Is Cut Off*. 2008. URL: http://www.washingtonpost.com/wp-dyn/content/article/2008/11/12/AR2008111200658.html.

[7] Brian Krebs. *Shadowy Russian Firm Seen as Conduit for Cybercrime*. 2007. URL: http://www.washingtonpost.com/wp-dyn/content/article/2007/10/12/AR2007101202461.html.

[8] Patrick Howell O'Neill. *An in-depth guide to Freedom Hosting, the engine of the Dark Net*. 2013. URL: https://www.dailydot.com/news/eric-marques-tor-freedom-hosting-child-porn-arrest/.

[9] Brett Stone-Gross, Christopher Kruegel, Kevin Almeroth, Andreas Moser, and Engin Kirda. "FIRE: FInding Rogue nEtworks". In: *ACSAC*. 2009, pp. 231–240.

[10] C. Wagner, J. François, R. State, A. Dulaunoy, T. Engel, and G. Massen. "ASMATRA: Ranking ASs providing transit service to malware hosters". In: *Integrated Network Management*. 2013, pp. 260–268.

[11] Maria Konte, Roberto Perdisci, and Nick Feamster. "ASwatch: An AS Reputation System to Expose Bulletproof Hosting ASes". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication - SIGCOMM '15*. ACM Press, 2015, pp. 625–638.

[12] Dutch-Police. *Nederlandse en Thaise politie pakken bulletproof hoster aan*. URL: https://www.politie.nl/nieuws/2018/mei/16/11-nederlandse-en-thaise-politie-pakken-bulletproof-hoster-aan.html.

[13] Catalin Cimpanu. *Police Seize Servers of Bulletproof Provider Known For Hosting Malware Ops*. URL: https://www.bleepingcomputer.com/news/security/police-seize-servers-of-bulletproof-provider-known-for-hosting-malware-ops/ (visited on 05/28/2019).

[14] Brett Stone-gross, Thorsten Holz, Gianluca Stringhini, and Giovanni Vigna. "The Underground Economy of Spam: A Botmaster's Perspective of Coordinating Large-Scale Spam Campaigns". In: *USENIX LEET*. 2011.

[15] Damon McCoy, A Pitsillidis, G Jordan, N Weaver, C Kreibich, B Krebs, G M Voelker, S Savage, and K Levchenko. "PharmaLeaks: Understanding the Business of Online Pharmaceutical Affiliate Programs". In: *USENIX Security 2012* (2012), pp. 1–16.

[16] Shuang Hao, Kevin Borgolte, Nick Nikiforakis, Gianluca Stringhini, Manuel Egele, Michael Eubanks, Brian Krebs, and Giovanni Vigna. "Drops for Stuff: An Analysis of Reshipping Mule Scams". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15* (2015), pp. 1081–1092.

[17] Michael Bailey, David Dittrich, Erin Kenneally, and Doug Maughan. "The Menlo report". In: *IEEE Security and Privacy* 10.2 (2012), pp. 71–75.

[18] *DNSDB*. URL: https://www.dnsdb.info.

[19] *Maxmind GeoIP2 DB*. URL: https://www.maxmind.com/en/geoip2-isp-database.

[20] Annelie Langerak. *Groot pedonetwerk opgerold*. 2018. URL: https://www.telegraaf.nl/nieuws/2043709/groot-pedonetwerk-opgerold.

[21] K. Levchenko, A. Pitsillidis, N. Chachra, B. Enright, M. Felegyhazi, C. Grier, T. Halvorson, C. Kanich, C. Kreibich, D. McCoy, N. Weaver, V. Paxson, G. M. Voelker, and S. Savage. "Click Trajectories: End-to-End Analysis of the Spam Value Chain". English. In: *2011 IEEE Symposium on Security and Privacy*. IEEE, 2011, pp. 431–446.

[22] Damon Mccoy, Hitesh Dharmdasani, Christian Kreibich, Geoffrey M Voelker, and Stefan Savage. "Priceless : The Role of Payments in Abuse-advertised Goods". In: *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), pp. 845–856.

[23] Andy Greenberg. *Operation Bayonet: Inside the Sting That Hijacked an Entire Dark Web Drug Market*. URL: https://www.wired.com/story/hansa-dutch-police-sting-operation/ (visited on 11/01/2018).

[24] *Phishtank*. URL: https://www.phishtank.com/index.php.

[25] *APWG*. URL: https://www.antiphishing.org/.

[26] *StopBadware*. URL: https://www.stopbadware.org/data-sharing.

[27] *SpamHaus DBL*. URL: https://www.spamhaus.org/dbl/.

[28] *CleanMX*. URL: https://support.clean-mx.com.

[29] Brett Stone-Gross, Marco Cova, Lorenzo Cavallaro, Bob Gilbert, Martin Szydlowski, Richard Kemmerer, Christopher Kruegel, and Giovanni Vigna. "Your botnet is my botnet". In: *Proceedings of the 16th ACM conference on Computer and communications security - CCS '09*. New York, New York, USA: ACM Press, 2009, p. 635.

[30] David Y Wang, Matthew Der Mohammad, Lawrence Saul, Damon Mccoy, Stefan Savage, and Geoffrey M Voelker. "Search + Seizure : The Effectiveness of Interventions on SEO Campaigns". In: *IMC*. 2014, pp. 359–372.

[31] Nicolas Christin. "Traveling the silk road". In: *Proceedings of the 22nd international conference on World Wide Web - WWW '13*. New York, New York, USA: ACM Press, 2013, pp. 213–224.

[32] Kyle Soska and Nicolas Christin. "Measuring the Longitudinal Evolution of the Online Anonymous Marketplace Ecosystem". In: *Usenix Sec*. 2015, pp. 33–48.

[33] Rolf van Wegberg, Samaneh Tajalizadehkhoob, Kyle Soska, Ugur Akyazi, Carlos Hernandez Ganan, Bram Klievink, Nicolas Christin, and Michel van Eeten. "Plug and Prey? Measuring the Commoditization of Cybercrime via Online Anonymous Markets". In: *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 2018, pp. 1009–1026.

[34] Sumayah Alrwais, Kan Yuan, Eihal Alowaisheq, Zhou Li, and Xiaofeng Wang. "Understanding the Dark Side of Domain Parking". In: *23rd USENIX Security Symposium (USENIX Security '14)*. 2014.

[35] Ryan Brunt, Prakhar Pandey, and Damon McCoy. "Booted: An Analysis of a Payment Intervention on a DDoS-for-Hire Service". In: *Workshop on the Economics of Information Security (WEIS)* (2017).

[36] Craig A. Shue, Andrew J. Kalafut, and Minaxi Gupta. "Abnormally Malicious Autonomous Systems and Their Internet Connectivity". In: *IEEE/ACM TON* 20.1 (2012), pp. 220–230.

[37] Arman Noroozian, Maciej Korczynski, Samaneh Tajalizadehkhoob, and Michel van Eeten. "Developing Security Reputation Metrics for Hosting Providers". In: *USENIX CSET*. 2015.

[38] Arman Noroozian, Michael Ciere, Maciej Korczynski, Samaneh Tajalizadehkhoob, and Michel Van Eeten. "Inferring the Security Performance of Providers from Noisy and Heterogenous Abuse Datasets". In: *WEIS*. 2017.

[39] Samaneh Tajalizadehkhoob, Maciej Korczynski, Arman Noroozian, Carlos Ganan, and Michel van Eeten. "Apples, oranges and hosting providers: Heterogeneity and security in the hosting market". In: *Proc. of NOMS*. IEEE, 2016.

14 Appendices

A - Customer Preference Elicitation



Figure 15: Chat excerpt illustrating customer preference elicitation.

Figure 15 illustrates an excerpt of a live chat (edited for readability) conducted by one of the authors with MaxiDed

operators prior to its takedown. It shows the process of preference elicitation by MaxiDed operators.

The conversation was conducted using the live-chat functionality on their webshop. It demonstrates that MaxiDed operators may have also allowed other forms of abuse which they did not publicly mention on their webshop along side the various BP server packages that the platform advertised.

B - Geographical distribution of Customer Servers

In analyzing MaxiDed's platform, we also examined where its customer servers were located. We used Maxmind's commercial historical geo-location data for this purpose. This data is available on a weekly basis. For each customer server we first found the closest matching Maxmind IP geolocation database with the timespan during which the server was active. We then determined were each server was located based on its IP address and Maxmind's datasets. Figure 16 plots the top-20 locations for MaxiDed's customer servers. We found that the majority of the BP servers geolocated to Moldova followed by Russia, the US, Ukraine, the Netherlands and a long tail of other countries. Figure 16 also displays the number of non-BP servers in each of these top-20 locations. We observed that the Netherlands in particular hosted a substantial number of the non-BP servers.

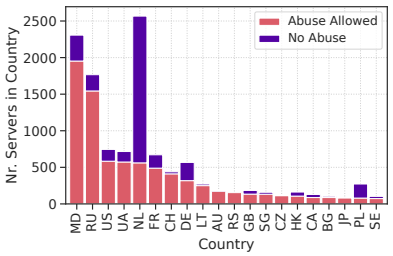


Figure 16: Top-20 locations for MaxiDed customer servers

Protecting Cloud Virtual Machines from Commodity Hypervisor and Host Operating System Exploits

Shih-Wei Li John S. Koh Jason Nieh
Department of Computer Science
Columbia University
{shihwei,koh,nieh}@cs.columbia.edu

Abstract

Hypervisors are widely deployed by cloud computing providers to support virtual machines, but their growing complexity poses a security risk as large codebases contain many vulnerabilities. We have created HypSec, a new hypervisor design for retrofitting an existing commodity hypervisor using microkernel principles to reduce its trusted computing base while protecting the confidentiality and integrity of virtual machines. HypSec partitions the hypervisor into an untrusted host that performs most complex hypervisor functionality without access to virtual machine data, and a trusted core that provides access control to virtual machine data and performs basic CPU and memory virtualization. Hardware virtualization support is used to isolate and protect the trusted core and execute it at a higher privilege level so it can mediate virtual machine exceptions and protect VM data in CPU and memory. HypSec takes an end-to-end approach to securing I/O to simplify its design, with applications increasingly using secure network connections in the cloud. We have used HypSec to retrofit KVM, showing how our approach can support a widely-used full-featured hypervisor integrated with a commodity operating system. The implementation has a trusted computing base of only a few thousand lines of code, many orders of magnitude less than KVM. We show that HypSec protects the confidentiality and integrity of virtual machines running unmodified guest operating systems while only incurring modest performance overhead for real application workloads.

1 Introduction

The availability of cost-effective, commodity cloud providers has pushed increasing numbers of companies and users to move their data and computation off site into virtual machines (VMs) running on hosts in the cloud. The hypervisor provides the VM abstraction and has full control of the hardware resources. Modern hypervisors are often integrated with a host operating system (OS) kernel to leverage existing kernel functionality to simplify their implementation and maintenance effort. For example, KVM [44] is integrated with Linux and Hyper-V [56] is integrated with Windows. The result is a huge potential attack surface with access to VM data in CPU registers, memory, I/O data, and boot images. The surge in outsourcing of computational resources to the cloud and away from privately-owned data centers further exacerbates this security risk of relying on the trustworthiness of complex and potentially vulnerable hypervisor and host OS infrastructure. Attackers that successfully exploit hypervisor vulnerabilities can gain unfettered access to VM data, and compromise the privacy and integrity of all VMs—an undesirable outcome for both cloud providers and users.

Recent trends in application design and hardware virtualization support provide an opportunity to revisit hypervisor design requirements to address this crucial security problem. First, modern hardware includes virtualization support to protect and run the hypervisor at a higher privilege level than VMs, potentially providing new opportunities to redesign the hypervisor to improve security. Second, due to greater security awareness because of the Snowden leaks revealing secret surveillance of large portions of the network infrastructure [49], applications are increasingly designed to use end-to-end encryption for I/O channels, including secure network connections [29, 50] and disk encryption [14]. This is decreasing the need for hypervisors to themselves secure I/O channels since applications can do a better job of providing an end-to-end I/O security solution [68].

Based on these trends, we have created HypSec, a new hypervisor design for retrofitting commodity hypervisors to significantly reduce the code size of their trusted computing base (TCB) while maintaining their full functionality. The design employs microkernel principles, but instead of requiring a clean-slate rewrite from scratch—a difficult task that limits both functionality and deployment—applies them to restructure an existing hypervisor with modest modifications. HypSec partitions a monolithic hypervisor into a small trusted core, the *corevisor*, and a large untrusted host, the *hostvisor*. HypSec leverages hardware virtualization support to isolate and protect the corevisor and execute it at a higher privilege

level than the hostvisor. The corevisor enforces access control to protect data in CPU and memory, but relies on VMs or applications to use end-to-end encrypted I/O to protect I/O data, simplifying the corevisor design.

The corevisor has full access to hardware resources, provides basic CPU and memory virtualization, and mediates all exceptions and interrupts, ensuring that only a VM and the corevisor can access the VM's data in CPU and memory. More complex operations including I/O and interrupt virtualization, and resource management such as CPU scheduling, memory management, and device management are delegated to the hostvisor, which can also leverage a host OS. The hostvisor may import or export encrypted VM data from the system to boot VM images or support hypervisor features such as snapshots and migration, but otherwise has no access to VM data. HypSec redesigns the hypervisor to improve security but does not strip it of functionality. We expect that HypSec can be used to restructure existing hypervisors by encapsulating much of their codebase in a hostvisor and augmenting security with a corevisor.

We have implemented a HypSec prototype by retrofitting KVM. Our approach works with existing ARM hardware virtualization extensions to provide VM confidentiality and integrity in a full-featured commodity hypervisor with its own integrated host OS kernel. Our implementation requires only modest modifications to Linux and has a TCB of only a few thousand lines of code (LOC), many orders of magnitude less than KVM and other commodity hypervisors. HypSec significantly reduces the TCB of an existing widely-used hypervisor and improves its security while retaining the same hypervisor functionality, including multiprocessor, full device I/O, multi-VM, VM management, and broad ARM hardware support. We also show that HypSec provides strong security for VMs running unmodified guest operating systems while only incurring modest performance overhead for real application workloads.

2 Assumptions and Threat Model

Assumptions. We assume VMs use end-to-end encrypted channels to protect their I/O data. We assume hardware virtualization support and an IOMMU similar to what is available on x86 and ARM servers in the cloud. We assume a Trusted Execution Environment (TEE) provided by secure mode architectures such as ARM TrustZone [7] or a Trusted Platform Module (TPM) [38] is available for trusted persistent storage. We assume the hardware, including a hardware security module if applicable, is bug-free and trustworthy. We assume the HypSec TCB, the corevisor, does not have any vulnerabilities and can thus be trusted. Given the corevisor's modest size as shown in Section 6.3, it may be possible to formally verify the codebase. We assume it is computationally infeasible to perform brute-force attacks on any encrypted VM data, and any encrypted communication protocols are assumed to be designed to defend against replay attacks. We assume the system is initially benign, allowing signatures and keys to be sealed in the TEE before a compromise of the system.

Threat Model. We consider an attacker with remote access to a hypervisor and its VMs, including administrators without physical access to the machine. The attacker's goal is to compromise the confidentiality and integrity of VM data, which includes: the VM boot image containing the guest kernel binary, data residing in memory addresses belonging to guests, guest memory copied to hardware buffers, data on VM disks or file systems, and data stored in VM CPU registers. VM data does not include generic virtual hardware configuration information, such as the CPU power management status or the interrupt level being raised. An attacker could exploit bugs in the hostvisor or control the VM management interface to access VM data. For example, an attacker could exploit bugs in the hostvisor to execute arbitrary code or access VM memory from the VM or hypervisor host. Attackers may also control peripherals to perform malicious memory access via direct memory access (DMA). We consider it out of scope if the entire cloud provider, who provides the VM infrastructure, is malicious.

A remote attacker does not have physical access to the hardware, so the following attacks are out of scope: physical tampering with the hardware platform, cold boot attacks [31], memory bus snooping, and physical memory access. These threats are better handled with on-site security and tamper-resistant hardware; cloud providers such as Google go to great lengths to ensure the physical security of their data centers and restrict physical access even for administrators [28]. We also do not defend against side-channel attacks in virtualized environments [39, 53, 65, 93, 94], or based on network I/O [10]. This is not unique to HypSec and it is the kernel's responsibility to obfuscate such patterns with defenses orthogonal to HypSec.

We assume a VM does not voluntarily reveal its own sensitive data whether on purpose or by accident. A VM can be compromised by a remote attacker that exploits vulnerabilities in the VM. We do not provide security features to prevent or detect VM vulnerabilities, so a compromised VM that involuntarily reveals its own data is out of scope. However, attackers may try to attack other hosted VMs from a compromised VM for which we provide protection.

3 Design

HypSec introduces a new hypervisor design that reduces the TCB necessary to protect VM confidentiality and integrity while retaining full-fledged hypervisor functionality. We observe that many hypervisor functions can be supported without any access to VM data. For example, VM CPU register data is unnecessary for CPU scheduling. Based on this observation, HypSec leverages microkernel design principles to split a monolithic hypervisor into two parts, as depicted in Figure 1: a trusted and privileged corevisor with full access to VM data, and an untrusted and deprivileged hostvisor delegated with most hypervisor functionality. Unlike previous microkernel approaches [1, 13, 51], HypSec is designed

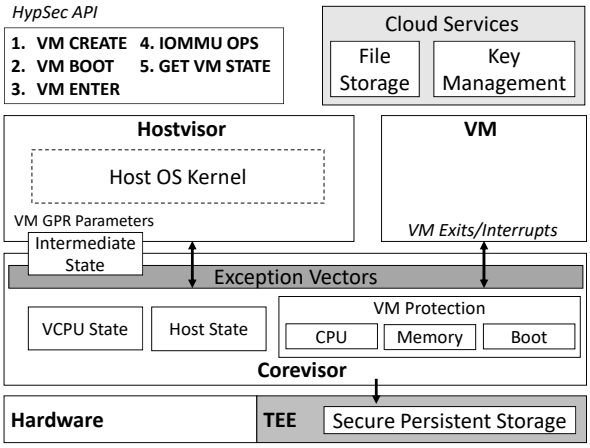


Figure 1: HypSec Architecture

specifically to restructure existing hypervisors with modest modifications as opposed to requiring a clean-slate redesign. Splitting the hypervisor this way results in a significantly smaller TCB that is still flexible enough to implement modern hypervisor features, as discussed in Section 4.

The corevisor is kept small by only performing VM data access control and hypervisor functions that require full access to VM data: secure VM boot, CPU virtualization, and page table management. With applications increasingly using secure communication channels to protect I/O data, HypSec takes an end-to-end approach to simplify its TCB and allows the hostvisor to provide I/O and interrupt virtualization. The hostvisor also handles other complex functions which do not need access to VM data, including resource management such as CPU scheduling and memory allocation. The hostvisor may even incorporate a full existing OS kernel to support its features.

HypSec leverages modern hardware virtualization support in a new way to enforce the hypervisor partitioning. HypSec runs the corevisor in a higher privileged CPU mode designed for running hypervisors, giving it full control of hardware, including virtualization hardware mechanisms such as nested page tables (NPTs).[1] The corevisor deprivileges the hostvisor and VM kernel by running them in a less privileged CPU mode. For example, in HypSec's implementation using ARM Virtualization Extensions (VE) shown in Figure 3, the corevisor runs in hypervisor (EL2) mode while the hostvisor and VM kernel run in a less privileged kernel (EL1) mode. The corevisor interposes on all exceptions and interrupts, enabling it to provide access control mechanisms that prevent the hostvisor from accessing VM CPU and memory data. For example, the corevisor has its own memory and uses NPTs to enforce memory isolation between the hostvisor, VMs, and itself. A compromised hostvisor or VM can neither control hardware virtualization mechanisms nor access corevisor memory and thus cannot disable HypSec.

HypSec Interface. As shown in Figure 1, the corevisor exposes a simple API to the hostvisor and interposes on all hostvisor and VM interactions to ensure secure VM execution throughout the lifecycle of a VM. The life of a VM begins when the hostvisor calls the corevisor's *VM CREATE* and *VM BOOT* calls to safely bootstrap it with a verified VM image. The hostvisor is deprivileged and cannot execute VMs. It must call *VM ENTER* to request the corevisor to execute a VM. When the VM exits execution because an interrupt or exception occurs, it traps to the corevisor, which examines the cause of the exit and if needed, will return to the hostvisor. The corevisor provides the *IOMMU OPS* API to device drivers in the hostvisor for managing the IOMMU, as discussed in Section 3.3. While the hostvisor has no access to VM data in CPU or memory, it may request the corevisor to provide an encrypted copy of VM data via the *GET VM STATE* hypercall API. The hostvisor can use the API to support virtualization features that require exporting VM data to disk or across the network, such as swapping VM memory to disk or VM management functions like VM snapshot and migration. The corevisor only uses encryption to export VM data. It never uses encryption, only access control, to protect VM data in CPU or memory.

3.1 Boot and Initialization

Corevisor Boot. HypSec ensures that the trusted corevisor binary is booted and the bootstrapping code itself is secure. To ensure only the trusted corevisor binary is booted, HypSec relies on Unified Extensible Firmware Interface (UEFI) firmware and its signing infrastructure with a hardware root of trust. The hostvisor and corevisor are linked as a single HypSec binary which is cryptographically ("digitally") signed by the cloud provider, similar to how OS binaries are signed by vendors like Red Hat or Microsoft. The HypSec binary is verified using keys in secure storage provided by the TEE, guaranteeing that only the signed binary can be loaded.

To ensure the bootstrapping code is secure, HypSec could implement it in the trusted corevisor, but does not. Bare-metal hypervisors implement bootstrapping, but this imposes a significant implementation and maintenance burden. The code must be manually ported to each different device, making it more difficult to support a wide range of systems. Instead, HypSec relies on the hostvisor bootstrapping code to install the corevisor securely at boot time since the hostvisor is initially benign. At boot time, the hostvisor initially has full control of the system to initialize hardware. The hostvisor installs the corevisor before entering user space; network and serial input services are not yet available, so remote attackers cannot compromise the corevisor's installation. After its installation, the corevisor gains full control of the hardware and subsequently deprivileges the hostvisor, ensuring the hostvisor can never control the hardware or access the corevisor's memory to disable HypSec. Using information provided at boot time, the corevisor is self-contained and can operate without any external data structures.

VM Boot. HypSec also guarantees the confidentiality and in-

[1] Intel's Extended Page Tables or ARM's stage 2 page tables.

tegrity of VM data during VM boot and initialization. HypSec keeps its TCB small by delegating complicated boot processes to the untrusted hostvisor, and verifying any loaded VM images in the corevsor before they are run. As shown in Figure 1, when a new VM is created, the hostvisor participates with the corevisor in a verified boot process. The hostvisor calls *VM CREATE* to request the corevisor to allocate VM state in corevisor memory, including an NPT and VCPU state, a per virtual CPU (VCPU) data structure. It then calls *VM BOOT* to request the corevisor to authenticate the loaded VM images. If successful, the hostvisor can then call *VM ENTER* to execute the VM. In other words, the hostvisor stores VM images and loads them to memory, avoiding implementing this complex procedure in the corevisor. The corevisor verifies the cryptographic signatures of VM images using public key cryptography, avoiding any shared secret between the user and HypSec.

Both the public keys and VM image signatures are stored in TEE secure storage prior to any attack, as shown in Figure 1. If the VM kernel binary is detached and can be mapped separately to memory, the hostvisor calls the corevisor to verify the image. If the VM kernel binary is in the VM disk image's boot partition, HypSec-aware virtual firmware bootstraps the VM. The firmware is signed and verified like VM boot images. The firmware then loads the signed kernel binary or a signed bootloader such as GRUB from the cleartext VM disk partition. The firmware then calls the corevisor to verify the VM kernel binary or bootloader. In the latter case, the bootloader verifies VM kernel binaries using the signatures on the virtual disk; GRUB already supports this. GRUB can also use public keys in the signed GRUB binary. The corevisor ensures only images it verified, either a kernel binary, virtual firmware, or a bootloader binary, can be mapped to VM memory. Finally, the corevisor sets the VM program counter to the entry point of the VM image to securely boot the VM.

As discussed in Section 3.5, HypSec expects that VM disk images are encrypted as part of an end-to-end encryption approach. HypSec ensures that any password or secret used to decrypt the VM disk is not exposed to the hostvisor. Common encrypted disk formats [6, 57] use user-provided passwords to protect the decryption keys. HypSec can store the encrypted key files locally or remotely using a cloud provider's key management service (KMS) [5, 58]. The KMS maintains a secret key which is preloaded by administrators into hosts' TEE secure storage. The corevisor decrypts the encrypted key file using the secret key, and maps the resulting password to VM memory, allowing VMs to obtain the password without exposing it to the hostvisor. The same key scheme is used for VM migration; HypSec encrypts and decrypts the VM state using the secret key from the KMS.

3.2 CPU

Hypervisors provide CPU virtualization by performing four main functions: handling traps from the VM; emulating privileged CPU instructions executed by the guest OS to ensure the hypervisor retains control of CPU hardware; saving and restoring VM CPU state, including GPRs and system registers such as page table base registers, as needed when switching among VMs and between a VM and the hypervisor; and scheduling VCPUs on physical CPUs. Hypervisors typically have full access to VM CPU state when performing any of these four functions, which can pose a problem for VM security if the hypervisor is compromised.

HypSec protects VM CPU state from the hostvisor while keeping its TCB small by restricting access to VM CPU state to the corevisor while delegating complex CPU functions that can be done without access to VM CPU state to the hostvisor. This is done by having the corevisor handle all traps from the VM, instruction emulation, and world switches between VMs and the hostvisor, all of which require access to VM CPU state. VCPU scheduling is delegated to the hostvisor as it can be done without access to VM CPU state.

The corevisor configures the hardware to route all traps from the VM, as well as interrupts as discussed in Section 3.4, to go to the corevisor, ensuring that it retains full hardware control. It also deprivileges the hostvisor to ensure that the hostvisor has no access to corevisor state. Since all traps from the VM go to the corevisor, the corevisor can trap and emulate CPU instructions on behalf of the VM. The corevisor multiplexes the CPU execution context between the hostvisor and VMs on the hardware. The corevisor maintains VCPU execution context in the VCPU state in-memory data structure allocated on *VM CREATE*, and maintains the hostvisor's CPU context in a similar *Host state* data structure; both states are only accessible to the corevisor. On VM exits, the corevisor first saves the VM execution context from CPU hardware registers to VCPU state, then restores the hostvisor's execution context from Host state to the CPU hardware registers. When the hostvisor calls to the corevisor to re-enter the VM, the corevisor first saves its execution context to Host state, then restores the VM execution context from VCPU state to the hardware. All saving and restoring of VM CPU state is done by the corevisor, and only the corevisor can run a VM.

The hostvisor handles VCPU scheduling, which can involve complex scheduling mechanisms especially for multiprocessors. For example, the Linux scheduler code alone is over 20K LOC, excluding kernel function dependencies and data structures shared with the rest of the kernel. VCPU scheduling requires no access to VM CPU state, as it simply involves mapping VCPUs to physical CPUs. The hostvisor schedules a VCPU to a physical CPU and calls to the corevisor to run the VCPU. The corevisor then loads the VCPU state to the hardware.

HypSec by default ensures that the hostvisor has no access to any VM CPU state, but sometimes a VM may execute instructions that requiring sharing values with the hostvisor that may be stored in general purpose registers (GPRs). For example, if the VM executes a hypercall that includes some parameters and

the hypercall is handled by the hostvisor, it will be necessary to pass the parameters to the hostvisor, and those parameters may be stored in GPRs. In these cases, the instruction will trap to the corevisor. The corevisor will identify the values that need to be passed to the hostvisor, then copy the values from the GPRs to an in-memory per VCPU intermediate VM state structure that is accessible to the hostvisor. Similarly, hostvisor updates to the intermediate VM state structure can be copied back to GPRs by the corevisor to pass values back to the VM. Only values from the GPRs explicitly identified by the corevisor for parameter passing are copied to and from intermediate VM state; values in other CPU registers are not accessible to the hostvisor.

The corevisor determines if and when to copy values from GPRs, and the GPRs from which to copy, based on the specific CPU instructions executed. The set of instructions are those used to execute hypercalls and special instructions provided by the architecture to access virtual hardware via model-specific registers (MSRs), control registers in the x86 instruction set, or memory-mapped I/O (MMIO). There are typically only a few specific CPU instructions that involve parameter passing to the hostvisor via GPRs, but the specific cases are architecture dependent.

For example, on ARM, HypSec copies selected GPRs to and from intermediate VM state for power management hypercalls to the virtual firmware interface and selected MMIO accesses to virtual hardware. For power management hypercalls, the guest kernel passes input parameters in GPRs, and the corevisor copies only those GPRs to intermediate VM state to make the parameters available to the hostvisor. Upon returning to the VM, the hostvisor provides output data as return values to the power management hypercalls, which the corevisor copies from intermediate VM state back to GPRs to make them available to the VM. As discussed in Sections 3.4 and 3.5, values stored and loaded in GPRs on MMIO accesses to the virtual interrupt controller interface or I/O devices are also copied between the selected GPRs and the intermediate VM state to make them available to the hostvisor.

3.3 Memory

Hypervisors provide memory virtualization by performing three main functions: memory protection to ensure VMs cannot access unauthorized physical memory, memory allocation to provide physical memory to VMs, and memory reclamation to reclaim physical memory from VMs. Other advanced memory management features may also be performed that build on these functions. All of these functions rely on NPTs. A guest OS manages the traditional page tables to map guest virtual memory addresses (gVA) to guest physical memory addresses (gPA). The hypervisor manages the NPTs to map from gPAs to host physical memory addresses (hPA) so it can virtualize and restrict a VM's access to physical memory. The hypervisor has full access to physical memory so it can manage VM memory either directly [11] or via a host OS kernel's [23]

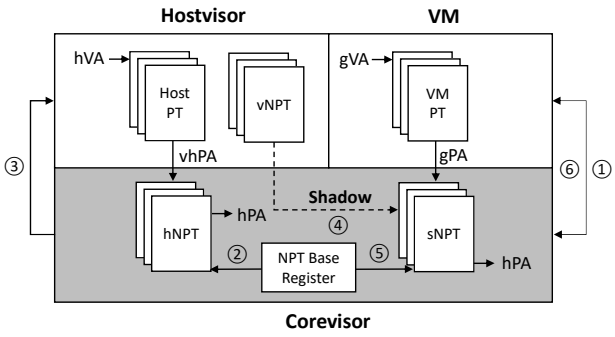


Figure 2: HypSec Memory Virtualization

memory management APIs. A compromised hypervisor or host OS kernel thus has unfettered access to VM memory and can read and write any data stored by VMs in memory.

HypSec protects VM memory from the hostvisor while keeping its TCB small by restricting access to VM memory to the corevisor while delegating complex memory management functions that can be done without access to actual VM data in memory to the hostvisor. The corevisor is responsible for memory protection, including configuring NPT hardware, while memory allocation and reclamation is largely delegated to the hostvisor. HypSec memory protection imposes an additional requirement, which is to also protect corevisor and VM memory from the hostvisor.

Memory Protection. The corevisor uses the NPT hardware in the same way as modern hypervisors to virtualize and restrict a VM's access to physical memory, but in addition leverages NPTs to isolate hostvisor memory access. The corevisor configures NPT hardware as shown in Figure 2. The hostvisor is only allowed to manage its own page tables (Host PT) and can only translate from host virtual memory addresses (hVAs) to what we call virtualized host physical memory addresses (vhPAs). vhPAs are then in turn translated to hPAs by the Host Nested Page Table (hNPT) maintained by the corevisor. The corevisor adopts a flat address space mapping; each vhPA is mapped to an identical hPA. The hostvisor, if granted access, is given essentially the same view of physical memory as the corevisor. The corevisor prevents the hostvisor from accessing corevisor and VM memory by simply unmapping the memory from the hNPT to make the physical memory inaccessible to the hostvisor. Any hostvisor accesses to corevisor or VM memory will trap to the corevisor, enabling the corevisor to intercept unauthorized accesses. Physical memory is statically partitioned between the hostvisor and corevisor, but dynamically allocated between the hostvisor and VMs as discussed below. The corevisor allocates NPTs from its own memory pool which is not accessible to the hostvisor. All VCPU state is also stored in corevisor memory.

The corevisor also protects corevisor and VM memory against DMA attacks [75] by retaining control of the IOMMU. The corevisor allocates IOMMU page tables from its memory and exports the *IOMMU OPS* API to device drivers in the

hostvisor to update page table mappings. The corevisor validates requests and ensures that attackers cannot control the IOMMU to access memory owned by itself or the VMs.

Memory Allocation. Memory allocation for VMs is largely done by the hostvisor, which can reuse memory allocation functions available in an integrated host OS kernel to dynamically allocate memory from its memory pool to VMs. Traditional hypervisors simply manage one NPT per VM. However, HypSec's memory model disallows the hostvisor from managing VM memory and therefore NPTs. The hostvisor instead manages an analogous Virtual NPT (vNPT) for each VM, and HypSec introduces a Shadow Nested Page Table (sNPT) managed by the corevisor for each VM as shown in Figure 2. The sNPT is used to manage the hardware by shadowing the vNPT. The corevisor multiplexes the hardware NPT Base Register between hNPT and sNPT when switching between the hostvisor and a VM.

Figure 2 also depicts the steps in HypSec's memory virtualization strategy. When a guest OS tries to map a gVA to an unmapped gPA, a nested page fault occurs which traps to the corevisor (step 1). If the corevisor finds that the faulted gPA falls within a valid VM memory region, it then points the NPT Base Register to hNPT (step 2) and switches to the hostvisor to allocate a physical page for the gPA (step 3). The hostvisor allocates a virtualized physical page identified by a vhPA and updates the entry in its vNPT corresponding to the faulting gPA with the allocated vhPA. Because the vhPA is mapped to an identical hPA, the hostvisor is able to implicitly manage host physical memory. The hostvisor then traps to the corevisor (step 4), which determines the faulting gPA and identifies the updates made by the hostvisor to the vNPT. The corevisor verifies the resulting vhPA is not owned by itself or other VMs, the latter by tracking ownership of physical memory using a unique VM identifier (VMID), and copies those updates to its sNPT. The corevisor unmaps the vhPA from the hNPT, so that the hostvisor no longer has access to the memory being allocated to the VM. The corevisor updates the NPT Base Register to point to the sNPT (step 5) and returns to the VM (step 6) so that the VM has access to the allocated memory identified by the hPA that is identical to the vhPA. Although possible, HypSec does not scrub pages allocated to VMs by the hostvisor. Guest OSes already scrub memory allocated from their free list before use for security reasons, so the hostvisor cannot allocate pages that contain malicious content to VMs.

HypSec's use of shadow page tables differs significantly from previous applications of it to collapse multi-level page tables down into what is supported by hardware [2, 11, 16, 52, 82]. In contrast, HypSec uses shadowing to protect hardware page tables, not virtualize them. The corevisor does not shadow guest OS updates in its page tables; it only shadows hostvisor updates to the vNPT. HypSec does not introduce additional traps from the VM for page table synchronization. Overshadow [16] maintains multiple shadow page tables for a given VM that provide different views (plaintext/encrypted) of physical memory to protect applications from an untrusted guest OS. In contrast, HypSec manages one shadow page table for each VM that provides a plaintext view of gPA to hPA. The shadowing mechanism in HypSec is also orthogonal to recent work [19] that uses shadow page tables to isolate kernel space memory from user space.

Memory Reclamation. HypSec supports VM memory reclamation in the hostvisor while preserving the privacy and integrity of VM data in memory in the corevisor. When a VM voluntarily releases memory pages, such as on VM termination, the corevisor returns the pages to the hostvisor by first scrubbing them to ensure the reclaimed memory does not leak VM data, then mapping them back to the hNPT so they are accessible to the hostvisor. To allow the hostvisor to reclaim VM memory pages without accessing VM data in memory, HypSec takes advantage of ballooning [82]. Ballooning is widely supported in common hypervisors, so only modest effort is required in HypSec to support this approach. A paravirtual "balloon" device is installed in the VM. When the host is low on free memory, the hostvisor requests the balloon device to inflate. The balloon driver inflates by getting pages from the free list, thereby increasing the VM's memory pressure. The guest OS may therefore start to reclaim pages or swap its pages to the virtual disk. The balloon driver notifies the corevisor about the pages in its balloon that are ready to be reclaimed. The corevisor then unmaps these pages from the VM's sNPT, scrubs the reclaimed pages to ensure they do not leak VM data, and assigns the pages to the hostvisor, which can then treat them as free memory. Deflating the balloon releases memory pressure in the guest, allowing the guest to reclaim pages.

HypSec also safely allows the hostvisor to swap VM memory to disk when it feels memory pressure. The hostvisor uses *GET VM STATE* to get access to the encrypted VM page before swapping it out. Later, when the VM page is swapped in, the corevisor unmaps the swapped-in page from hNPT, decrypts the page, and maps it back to the VM's sNPT.

Advanced VM Memory Management. HypSec by default ensures that the hostvisor has no access to any VM memory, but sometimes a VM may want to share its memory, after encrypting it, with the hostvisor. HypSec provides the `GRANT_MEM` and `REVOKE_MEM` hypercalls which can be explicitly used by a guest OS to share its memory with the hostvisor. As described in Section 3.5, this can be used to support paravirtualized I/O of encrypted data in which a memory region owned by the VM has to be shared between the VM and hostvisor for communication and efficient data copying. The VM passes the start of a guest physical frame number (GFN), the size of the memory region, and the specified access permission to the corevisor via the two hypercalls. The corevisor enforces the access control policy by controlling the memory region's mapping in hNPT. Only VMs can use these two hypercalls, so the hostvisor cannot use it to request access to arbitrary VM pages.

HypSec can support advanced memory virtualization features such as merging similar memory pages, KSM [46]

in Linux, by splitting the work into the simple corevisor functions which require direct access to VM data, and the more complicated hostvisor functions which do not require access to VM data. For example, to support KSM, the hostvisor requests the corevisor for the hash values of a VM's memory pages and maintains the data structure in its address space to support the merging algorithm. The corevisor validates the hostvisor's decision for the pages to be merged, updates the corresponding VM's sNPT, and scrubs the freed page before granting the hostvisor access. While KSM does not provide the hostvisor or other VMs direct access to a VM's memory pages, it can be used to leak some information such as whether the contents of memory pages are the same across different VMs. To avoid this kind of information leakage, HypSec disables KSM support by default.

3.4 Interrupts

Hypervisors trap and handle physical interrupts to retain full control of the hardware while virtualizing interrupts for VMs. Accesses to the interrupt controller interface can by done via MSRs or MMIO. Hypervisors provide a virtual interrupt controller interface and trap and emulate VM access to the interface. Virtual devices in the hypervisors can also raise interrupts to the interface. However, giving hypervisors full control of hardware poses a problem for VM security if the hypervisor is compromised.

To protect against a compromised hostvisor, the corevisor configures the hardware to route all physical interrupts and trap all accesses to the interrupt controller to the corevisor, ensuring that it retains full hardware control. However, to simplify its TCB, HypSec delegates almost all interrupt functionality to the hostvisor, including handling physical interrupts and providing the virtual interrupt controller interface. Before entering the hostvisor to handle interrupts, the corevisor protects all VM CPU and memory state, as discussed in Sections 3.2 and 3.3.

The hostvisor has no access to and requires no VM data to handle physical interrupts. However, VM accesses to the virtual interrupt controller interface involve passing parameters between the VM and the hostvisor since the hostvisor provides the interface. On ARM, this is done using only MMIO via the intermediate state structure discussed in Section 3.2. On an MMIO write to interrupt controller interface, the VM passes the value to be stored in a GPR. The write traps to the corevisor, which identifies the instruction and memory address as corresponding to the interrupt controller interface. The corevisor copies the value to be written from the GPR to the intermediate VM state to make the value available to the hostvisor. For example, when the guest OS in the VM sends an IPI to a destination VCPU by doing an MMIO write to the virtual interrupt controller interface, the identifier of the destination VCPU is passed to the hostvisor by copying the value from the respective GPR to the intermediate VM state. Similarly, on an MMIO read from the interrupt controller interface, the read traps to the corevisor, which identifies the instruction and memory address as corresponding to the interrupt controller interface. The corevisor copies the value from the intermediate VM state updated by the hostvisor to the GPR the VM is using to retrieve the value, updates the PC of the VM to skip the faulting instruction, and returns to the VM.

3.5 Input/Output

To ease the burden of supporting a wide range of virtual devices, modern hypervisors often rely on an OS kernel and its existing device drivers to support I/O virtualization, which significantly increase the hypervisor TCB. Similar to previous work [16,33], HypSec assumes an end-to-end I/O security approach, relying on VMs for I/O protection. VMs can leverage secure communication channels such as TLS/SSL for network communications and full disk encryption for storage. This allows the corevisor to relax its I/O protection requirements, simplifying the TCB. HypSec offloads the support of I/O virtualization to the untrusted hostvisor. Since I/O data is already encrypted by VMs, a compromised hostvisor would at most gain access to encrypted I/O data which would not reveal VM data.

HypSec, like other modern hypervisors, supports all three classes of I/O devices: emulated, paravirtualized, and passthrough devices; the latter two provide better I/O performance. Emulated I/O devices are typically supported by hypervisors using trap-and-emulate to handle both port-mapped I/O (PIO) and MMIO operations. In both cases, HypSec configures the hardware to trap the operations to the corevisor which hides all VM data other than actual I/O data and then allows the hostvisor to emulate the operation. For example, to support MMIO, the corevisor zeroes out the mappings for addresses in the VM's sNPT corresponds to virtual device I/O regions. Any subsequent MMIO accesses from the VM result in a memory access fault that traps to the corevisor. The corevisor then securely supports MMIO accesses as discussed in Section 3.4. We assume security aware users disable the use of emulated devices such as the serial port, keyboard, or mouse to avoid leaking private information to a compromised hostvisor.

Paravirtualized devices require that a front-end driver in the VM coordinate with a back-end driver in the hypervisor; the two drivers communicate through shared memory asynchronously. HypSec allows back-end drivers to be installed as part of the untrusted hostvisor. To support shared memory communication, the front-end driver is modified to use `GRANT_MEM` and `REVOKE_MEM` hypercalls to identify the shared data structure and I/O memory buffers as accessible to the hostvisor back-end driver. Since the I/O data is encrypted, hostvisor access to the I/O memory buffers does not risk VM data.

Passthrough devices are assigned to a VM and managed by the guest OS. To support passthrough I/O, HypSec configures the hardware to trap sensitive operations such as Message Signaled Interrupt (MSI) configuration in BAR to trap to the corevisor for secure emulation, while granting VMs

	Xen	KVM	HypSec
Boot and Initialization			
Secure Boot	○	○	○
Secure VM Boot	⊖	⊖	○
CPU			
VM Symmetric Multiprocessing (SMP)	○	○	○
VCPU Scheduling	○	○	○
Memory			
Dynamic Allocation	○	○	○
Memory Reclamation - Ballooning	○	○	○
Memory Reclamation - Swapping	○	○	⊘
DMA	○	○	○
Same Page Merging	○	○	⊘
Interrupts Virtualization			
Hardware Assisted	○	○	○
I/O			
Device Emulation	○	○	○
Paravirtualized (PV)	○	○	○
Device Passthrough	○	○	○
VM Management			
Multi-VM	○	○	○
VM Snapshot	○	○	○
VM Restore	○	○	○
VM Migration	○	○	○

Table 1: Supported features comparison. (○ = Supported, ⊖ = Not applicable, ⊘ = Not implemented.)

direct access to the non-sensitive device memory region. The corevisor controls the IOMMU to enforce inter-device isolation, and ensures the passthrough device can only access the VM's own I/O buffer. Since we assume the hardware is not malicious, passthrough I/O can be done securely on HypSec.

4 Implementation

We demonstrate how HypSec can improve the security of existing commodity hypervisors by applying our approach to the mainline Linux KVM/ARM [22, 23] hypervisor, given ARM's increasing popularity in server systems [4, 63, 87]. Table 1 compares commodity hypervisors with the current HypSec implementation, showing that this security improvement comes without compromising on hypervisor features. Since KVM is a hosted hypervisor tightly integrated with a host OS kernel, retrofitting KVM also demonstrates the viability of HypSec in supporting an entire OS kernel as part of the hostvisor.

HypSec requires a higher-privileged CPU mode, nested page tables for memory virtualization, and an IOMMU for DMA protection. These requirements are satisfied by the ARM architecture. ARM VE provides Hyp (EL2) mode for hypervisors that is strictly more privileged than user (EL0) and kernel (EL1) modes. EL2 has its own execution context defined by register and control state, and can therefore switch the execution context of both EL0 and EL1 in software. Thus, the hypervisor can run in an address space that is isolated from EL0 and EL1. ARM VE provides stage 2 page tables

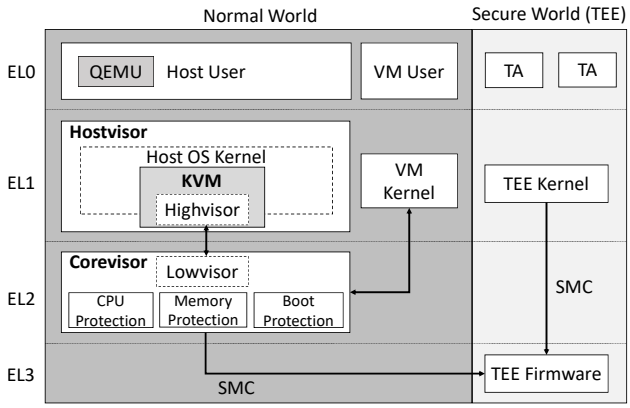


Figure 3: HypSec on KVM/ARM

which are nested level page tables configured in EL2 that affect software in EL0 and EL1. ARM provides the System Memory Management Unit (SMMU) [8] to protect DMA.

HypSec's corevisor is initialized at machine bootup and runs in EL2 to fully control the hardware. HypSec's code is embedded in the Linux kernel binary, which is verified and loaded via UEFI. The kernel boots in EL2 and installs a trap handler to later return to EL2. The kernel then enters EL1 so the hostvisor can bootstrap the machine. The hostvisor allocates resources and configures the hardware for the corevisor. The hostvisor then makes a hypercall to the corevisor in EL2 to enable HypSec.

The HypSec ARM implementation leverages KVM/ARM's split into an EL2 lowvisor and an EL1 highvisor to support the ARM virtualization architecture. This is done because EL2 is necessary for controlling hardware virtualization features, but Linux and KVM are designed to run in kernel mode, EL1. Thus, the lowvisor manages hardware virtualization features and VM-hypervisor switches, while the highvisor contains the rest of the hypervisor and Linux. However, the lowvisor cannot protect VM data if any other part of Linux or KVM are compromised; with KVM/ARM, the Linux host has unfettered access to all VM data.

As shown in Figure 3, the corevisor encapsulates the KVM lowvisor and runs in EL2. The hostvisor, including the KVM highvisor and its integrated Linux OS kernel, runs in EL1. The hostvisor has no access to EL2 registers and cannot compromise the corevisor or disable VM protection. HypSec leverages ARM VE to force VM operations that need hypervisor intervention to trap into EL2. The corevisor either handles the trap directly to protect VM data or world switches the hardware to EL1 to run the hostvisor if more complex handling is necessary. When the hostvisor finishes its work, it makes a hypercall to trap to EL2 so the corevisor can securely restore the VM state to hardware. The corevisor interposes on every switch between the VM and hostvisor, thus protecting the VM's execution context. Our implementation ensures that the hostvisor cannot invoke arbitrary corevisor functions via hypercalls.

HypSec leverages ARM VE's stage 2 memory translation support to virtualize VM memory and prevent accesses to protected physical memory. The corevisor routes stage 2 page faults to EL2 and rejects illegal hostvisor and VM memory accesses. The corevisor allocates hNPTs and VMs' sNPTs from its protected physical memory and manages the page tables.

To secure DMA, the corevisor uses trap-and-emulate on hostvisor accesses to the SMMU. HypSec ensures only the corevisor has access to the SMMU hardware. The corevisor manages the SMMU page tables in its protected memory to ensure hostvisor devices cannot access corevisor or VM memory, and devices assigned to the VM can only access VM memory.

HypSec leverages the hardware features from VGIC and KVM/ARM's existing support to virtualize interrupts. Our implementation supports ARM GIC 2.0. HypSec relies on QEMU and KVM's virtual device support for I/O virtualization. Our implementation supports emulated devices via MMIO, paravirtualized devices via virtio [67], and passthrough devices. For virtio, we modified front-end drivers to use `GRANT/REVOKE_MEM` hypercalls to share memory with the hostvisor back-end drivers. To support passthrough devices, HypSec configures the hardware to grant VMs direct access to them. We modified the front-end virtio-balloon driver to notify the corevisor about the pages allocated for the balloon device. The corevisor scrubs and assigns these pages to the hostvisor, allowing it to safely reclaim memory as needed. Our current implementation does not support page swapping and KSM, which are both left as future work.

HypSec supports secure VM boot using ARM TrustZone-based TEE frameworks such as OP-TEE [61] to store the signatures and keys securely. HypSec tasks QEMU to load the VM boot images to VM memory, but the corevisor requires QEMU to participate with its verified boot process. The corevisor retrieves the VM boot image signatures and the user public key from TrustZone for verifying the VM images remapped to its address space. The corevisor uses Ed25519 [62] to verify the boot images. HypSec builds the VM's stage 2 page table with mappings to the verified VM boot image. If the verification fails, HypSec stops the VM boot process. The same scheme can also verify VM firmware and other binaries. HypSec also retrieves the encrypted password which protects the VM's encrypted disk from either TrustZone or from the cloud provider's key management service. A small AES implementation [45] ported to run in EL2 performs the decryption. We include only two small yet sufficient crypto libraries in EL2 to keep the TCB small. This limits the number of crypto algorithms, but avoids including comprehensive but excessively large crypto libraries such as OpenSSL. HypSec leverages AES to support encrypted VM migration and snapshot, and ensures only encrypted VM data is exposed to the hostvisor.

HypSec's hardware requirements can also be satisfied on Intel's x86 architecture by using Virtual Machine Extensions (VMX) [35] and the IOMMU. Existing x86 hypervisors can be retrofitted to run the corevisor in VMX root operation which allows control of virtualization features for deprivileging the hostvisor. The hostvisor runs in VMX non-root operation to provide resource management and virtual I/O. The corevisor protects VM execution state by managing a Virtual-Machine Control Structure (VMCS) per CPU, and VM memory by using Extended Page Tables (EPT) and controlling the IOMMU.

5 Security Analysis

We present five properties of the HypSec architecture, then discuss how their combination provides a set of security properties regarding HypSec's ability to protect the integrity and confidentiality of VM data.

Property 1. *HypSec's corevisor is trusted during the system's lifetime against remote attackers.*

HypSec leverages hardware secure boot to ensure only the signed and trusted HypSec binary can be booted. This prevents an attacker from trying to boot or reboot the system to force it to load a malicious corevisor. The hostvisor securely installs the corevisor during the boot process before network access and serial input service are available. Thus, remote attackers cannot compromise the hostvisor prior to or during the installation of the corevisor. The corevisor protects itself after initialization. It runs in a privileged CPU mode using a separate address space from the hostvisor and the VMs. The corevisor has full control of the hardware including the virtualization features that prevent attackers from disabling its VM protection. The corevisor also protects its page tables so an attacker cannot map executable memory to the corevisor's address space.

Property 2. *HypSec ensures only trusted VM images can be booted on VMs.*

Based on Property 1, the trusted corevisor verifies the signatures of the VM images loaded to VM memory before they are booted. The public keys and signatures are stored using TEE APIs for persistent secure storage. A compromised hostvisor therefore cannot replace a verified VM with a malicious one.

Property 3. *HypSec isolates a given VM's memory from all other VMs and the hostvisor.*

Based on Property 1, HypSec prevents the hostvisor and a given VM from accessing memory owned by other VMs. The corevisor tracks ownership of physical pages and enforces inter-VM memory isolation using nested paging hardware. A compromised hostvisor could control a DMA capable device to attempt to access VM memory or compromise the corevisor. However, the corevisor controls the IOMMU and its page tables, so the hostvisor cannot access corevisor or VM memory via DMA. VM pages reclaimed by the hostvisor are scrubbed by the corevisor, so they do not leak VM data. HypSec also protects the integrity of VM nested page tables. The corevisor manages shadow page tables for VMs. The MMU can only walk the shadow page tables residing in a protected memory region only accessible to the corevisor. The corevisor manages

and verifies updates to the shadow page tables to protect VM memory mappings.

Property 4. *HypSec protects a given VM's CPU registers from the hostvisor and all other VMs.*

HypSec protects VM CPU registers by only granting the trusted corevisor (Property 1) full access to them. The hostvisor cannot access VM registers without permission. Attackers cannot compromise VM execution flow since only the corevisor can update VM registers including program counter (PC), link register (LR), and TTBR.

Property 5. *HypSec protects the confidentiality of a given VM's I/O data against the hostvisor and all other VMs assuming the VM employs an end-to-end approach to secure I/O.*

Based on Properties 3 and 4, HypSec protects any I/O encryption keys loaded to VM CPU registers or memory, so a compromised hostvisor cannot steal these keys to decrypt encrypted I/O data. The same protection holds against other VMs.

Property 6. *HypSec protects the confidentiality and integrity of a given VM's I/O data against the hostvisor and all other VMs assuming the VM employs an end-to-end approach to secure I/O and the I/O can be verified before it permanently modifies the VM's I/O data.*

Using the reasoning in Property 5 with the additional assumption that I/O can be verified before it permanently modifies I/O data, HypSec also protects the integrity of VM I/O data, as any tampered data will be detected and can be discarded. For example, a network endpoint receiving I/O from a VM over an encrypted channel with authentication can detect modifications of the I/O data by any intermediary such as the hostvisor. If verification is not possible, then HypSec cannot prevent compromises of data availability that result in destruction of I/O data, which can affect data integrity. As an example, HypSec cannot prevent an attacker from arbitrarily destroying a VM's I/O data by blindly overwriting all or parts of a VM's local disk image; both the VM's availability and integrity are compromised since the data is destroyed. Secure disk backups can protect against permanent data loss.

Property 7. *Assuming a VM takes an end-to-end approach for securing its I/O, HypSec protects the confidentiality of all of the VM's data against a remote attacker, including if the attacker compromises any other VMs or the hostvisor itself.*

Based on Properties 1, 3, and 4, a remote attacker cannot compromise the corevisor, and any compromise of the hostvisor or another VM cannot allow the attacker to access VM data stored in CPU registers or memory. This combined with Property 5 allows HypSec to ensure the confidentiality of all of the VM's data.

Property 8. *Under the assumption that a VM takes an end-to-end approach for securing its I/O and I/O can be verified before it permanently modifies any VM data, HypSec protects the integrity of all of the VM's data against a remote attacker, including if the attacker compromises any other VMs or the hostvisor itself.*

Based on Properties 1, 3, and 4, HypSec ensures a remote attacker cannot compromise the corevisor, and that any compromise of the hostvisor or another VM cannot allow the attacker to access VM data stored in CPU registers or memory, thereby preserving VM CPU and memory data integrity. This combined with Property 6 allows HypSec to ensure the integrity of all of the VM's data.

Property 9. *If the hypervisor is benign and responsible for handling I/O, HypSec protects the confidentiality and integrity of all of the VM's data against any compromises of other VMs.*

If both the hostvisor and corevisor are not compromised and the hostvisor is responsible for handling I/O, then the confidentiality and integrity of a VM's I/O data will be protected against other VMs. This combined with Properties 3 and 4 allows HypSec to ensure the confidentiality and integrity of all of the VM's data. This guarantee is equivalent to what is provided by a traditional hypervisor such as KVM.

6 Experimental Results

We quantify the performance and TCB of HypSec compared to other approaches, and demonstrate HypSec's ability to protect VM confidentiality and integrity. All of our experiments were run on ARM server hardware with VE support, specifically a 64-bit ARMv8 AMD Seattle (Rev.B0) server with 8 Cortex-A57 CPU cores, 16 GB of RAM, a 512 GB SATA3 HDD for storage, an AMD 10 GbE (AMD XGBE) NIC device, and an IOMMU (SMMU-401) to support control over DMA devices and direct device assignment. The hardware did not support ARM Virtualization Host Extensions [20, 21]. For client-server experiments, the clients ran on an x86 machine with 24 Intel Xeon CPU 2.20 GHz cores and 96 GB RAM. The clients and the server communicated via a 10 GbE unsaturated network connection.

To provide comparable measurements across the approaches, we kept the software environments across all platforms as uniform as possible. We compared KVM with our HypSec modifications versus standard KVM, both in Linux 4.18 with QEMU 2.3.50. In both cases, KVM was configured with its standard VHOST virtio network, and with `cache=none` for its virtual block storage devices [30, 47, 77]. All hosts and VMs used Ubuntu 16.04 with the same Linux 4.18 kernel, except for HypSec changes. All VMs used paravirtualized I/O, typical of cloud infrastructure deployments such as Amazon EC2.

We ran benchmarks both natively on the hardware and in VMs. Each physical or VM instance was configured as a 4-way SMP with 12 GB of RAM to provide a common basis for comparison. This involved two configurations: (1) native Linux capped at 4 cores and 12 GB RAM, and (2) a VM using KVM with 8 cores and 16 GB RAM with the VM capped at 4 virtual CPUs (VCPUs) and 12 GB RAM. We measure multi-core configurations to reflect real-world server deployments. For VMs,

Name	Description
Hypercall	Transition from the VM to the hypervisor and return to the VM without doing any work in the hypervisor. Measures bidirectional base transition cost of hypervisor operations.
I/O Kernel	Trap from the VM to the emulated interrupt controller in the hypervisor OS kernel, and then return to the VM. Measures a frequent operation for many device drivers and baseline for accessing I/O devices supported by the hypervisor OS kernel.
I/O User	Trap from the VM to the emulated UART in QEMU and then return to the VM. Measures base cost of operations that access I/O devices emulated in the hypervisor OS user space.
Virtual IPI	Issue a virtual IPI from a VCPU to another VCPU running on a different PCPU, both PCPUs executing VM code. Measures time between sending the virtual IPI until the receiving VCPU handles it, a frequent operation in multi-core OSes.

Table 2: Microbenchmarks

Name	Description
Kernbench	Compilation of the Linux 4.9 kernel using `allnoconfig` for ARM with GCC 5.4.0.
Hackbench	`hackbench` [66] using Unix domain sockets and 100 process groups running in 500 loops.
Netperf	`netperf` v2.6.0 [41] running `netserver` on the server and the client with its default parameters in three modes: TCP_STREAM (throughput), TCP_MAERTS (throughput), and TCP_RR (latency).
Apache	`Apache` v2.4.18 Web server running `ApacheBench` [80] v2.3 on the remote client, which measures number of handled requests per second when serving the 41 KB `index.html` file of the GCC 4.4 manual using 100 concurrent requests.
Memcached	`memcached` v1.4.25 using the `memtier` benchmark v1.2.3 with its default parameters.
MySQL	`MySQL` v14.14 (distrib 5.7.24) running `SysBench` v.0.4.12 using the default configuration with 200 parallel transactions.

Table 4: Application Benchmarks

we pinned each VCPU to a specific physical CPU (PCPU) and ensured that no other work was scheduled on that PCPU. All of the host's device interrupts and processes were assigned to run on other PCPUs. For client-server benchmarks, the clients ran natively on Linux and used the full hardware available.

6.1 Microbenchmark Results

We first ran microbenchmarks to quantify the cost of low-level hypervisor operations. We used the KVM unit test framework [48] listed in Table 2 to measure the cost of transitioning between the VM and the hypervisor, initiating a VM-to-hypervisor OS kernel I/O request, emulating user space I/O with QEMU, and sending virtual IPIs. We slightly modified the test framework to measure the cost of virtual IPIs and to obtain cycle counts on ARM to ensure detailed results by configuring the VM with direct access to the cycle counter.

Microbenchmark	KVM	HypSec
Hypercall	2,896	3,202
I/O Kernel	3,831	4,563
I/O User	9,288	10,704
Virtual IPI	8,816	10,047

Table 3: Microbenchmark Measurements (cycles)

Table 3 shows the microbenchmarks measured in cycles for both standard KVM and HypSec. HypSec introduces roughly 5% to 19% overhead over KVM. HypSec does not increase the number of traps in the operations we measured. The corevisor interposes on exisiting traps to add additional logic to protect VM data, so the cost is relatively small. The I/O Kernel, I/O User, and Virtual IPI measurements show relatively higher overhead than Hypercall on HypSec because of the cost involved to secure data transfers between the VM and hostvisor for I/O and interrupt virtualization.

6.2 Application Workload Results

Next we ran real application workloads to evaluate HypSec compared to standard KVM. Table 4 lists the workloads which are a mix of widely-used CPU and I/O intensive benchmarks. To evaluate VM performance with end-to-end I/O protection, we used five configurations: (1) Native unmodified Linux host kernel without Full Disk Encryption, (2) Unmodified KVM and guest kernel without FDE (KVM), (3) Unmodified KVM and guest kernel with FDE (KVM-FDE), (4) HypSec and paravirtualized guest kernel without FDE (HypSec), (5) HypSec and paravirtualized guest kernel with FDE (HypSec-FDE). For FDE, we use dm-crypt to create a LUKS-encrypted root partition of the VM filesystem. We measure with and without FDE to separately quantify its extra costs. We leveraged the TLS/SSL support in Apache and MySQL and evaluated VM performance on HypSec with end-to-end network encryption.

Figure 4 shows the relative overhead of executing in a VM in our four VM configurations compared to natively. We normalize the results so that a value of 1.00 means the same performance as native hardware. Lower numbers mean less overhead. The performance on real application workloads shows modest overhead overall for HypSec compared to standard KVM. The overhead for HypSec in many cases is less than 10%, even with FDE enabled.

The worst overhead for HypSec occurs for some of the network workloads. Our current implementation of the front-end network virtio driver applies grant/revoke hypercalls on a per transaction basis to make data available to the back-end driver in the hostvisor. Therefore, HypSec's performance is sub-optimal in workloads where the virtio driver can batch multiple transactions without trapping to the hypervisor, most notably in TCP_MAERTS. TCP_MAERTS measures the bandwidth of a VM sending packets to a client. The virtio driver batches multiple sends to avoid traps to hypervisor, while in the implementation measured in the paper, the driver traps additionally

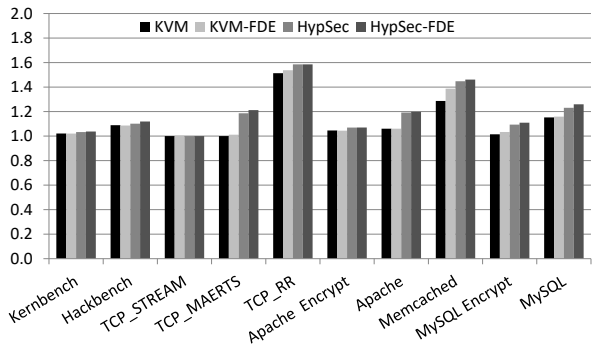


Figure 4: Application Benchmark Performance

on sending network data, resulting in higher overhead. Note that other network workloads such as TCP_STREAM have negligible overhead as the granularity at which the additional traps happen is large enough that the performance impact is negligible. To avoid extra traps to the hypervisor, our implementation can be optimized by batching the effect of the grant/revoke calls at the same level of granularity as used by the virtio driver to batch multiple transactions. This is an area of future work.

6.3 TCB Implementation Complexity

We ran cloc [24] against our implementation's corevisor to measure the TCB, as shown in Table 5. The total is roughly 8.5K LOC of which just under 4.5K LOC is from the Ed25519 and AES crypto libraries. The rest of the HypSec TCB is less than 4.1K LOC, consisting of mostly CPU/memory protection and existing KVM lowvisor code. Overall, we modified or added a total of 8,695 LOC in the mainline Linux kernel v4.18 across both the corevisor and hostvisor. More than 1.3K LOC were in existing Linux files, and around 7.3K LOC were in new files for HypSec, including around 4.5K LOC in the crypto libraries and slightly less than 2.8K LOC for corevisor functions. Finally, less than 70 LOC were added to QEMU to support secure boot and VM migration. These results demonstrate that HypSec can retrofit existing hypervisors with modest implementation effort.

For comparison purposes, we also used cloc to measure KVM's TCB in Linux v4.18 and Xen v4.9 for ARM64 support when running Linux v4.18 on Dom0, shown in Table 6. For KVM, we counted its LOC for the specific Linux v4.18 codebase running on the ARM64 server used in our experiments. KVM's massive TCB with access to VM data consists of more than 1.8M LOC and includes QEMU, the KVM module, core Linux functions such as CPU scheduling, ARM64 architectural support, and the device drivers used on the server.

To provide a fair comparison, we assumed the same threat model for each system and that VMs encrypt their I/O. Even under this assumption, KVM, including its I/O kernel code, must be entirely trusted to protect VM data since a compromised KVM can steal encryption keys from VM CPU and memory

Components	LOC
Ed25519 library	4,074
AES library	403
CPU protection	1,883
Memory protection	1,727
Secure boot	232
Helper	247
HypSec TCB	8,566

Table 5: HypSec TCB

Hypervisor	LOC
HypSec	8,566
KVM	1,857,575
Xen	71,604
Xen + Dom0	2,054,756

Table 6: TCB size comparison with KVM and Xen

state. By retrofitting KVM with HypSec to protect VM CPU and memory against the rest of the KVM codebase, we show that the TCB of KVM can be reduced by more than 200 times.

Using the same assumption, Xen's TCB should include both its hypervisor code and Dom0, a special privileged VM used to reuse existing Linux drivers to support I/O for user VMs. Although Dom0 is not part of the hypervisor, Xen provides it with a management interface that can request the hypervisor to dump entire VM state, thereby giving a compromised Dom0 full access to encryption keys. Xen's resulting TCB including Dom0, which has a full copy of Linux, is therefore larger than KVM and hundreds of times larger than HypSec. If we conservatively assume features in Xen's management stack that expose VM state such as VM dump and migration are disabled, so that we can exclude Dom0 from Xen's TCB and only count Xen ARM hypervisor code in EL2, Xen's TCB is then 71K LOC as listed in Table 6. This is roughly an order of magnitude larger than HypSec because Xen still has to do its own bootstrapping, CPU and memory resource management, and completely support memory and interrupt virtualization.

We estimated HypSec's TCB for an equivalent x86 implementation, assuming HypSec is also applied to KVM Linux v4.18 for x86 hardware with VMX support. We ran cloc against the C files that encapsulate the KVM functions for CPU and memory virtualization to conservatively measure HypSec's TCB size. The total is less than 27K LOC. Although the TCB size for HypSec on x86 would be larger than HypSec on ARM, we believe the resulting TCB on x86 would still result in a substantial reduction as KVM's TCB on x86 is also larger than on ARM, at roughly 10M LOC including x86 device drivers; this is an area of future work.

6.4 Evaluation of Practical Attacks

We evaluated HypSec's effectiveness against a compromised hostvisor by analyzing CVEs and identifying the cases where HypSec protects VM data despite any compromise, assuming an equivalent implementation of HypSec for x86 platforms. We analyzed CVEs related to Linux/KVM, which are listed in Tables 7 and 8. The CVEs consider two cases: a malicious VM who exploits KVM functions supported by the hostvisor, and an unprivileged host user who exploits bugs in Linux/KVM. Among the selected CVEs, 16 of them are x86-specific, one is specific to ARM, while the rest are independent of architecture. An attacker's goal is to exploit these CVEs to obtain

Bug	Description	KVM	HypSec
CVE-2015-4036	Memory Corruption: Array index error in hostvisor.	No	Yes
CVE-2013-0311	Privilege Escalation: Improper handling of descriptors in vhost driver.	No	Yes
CVE-2017-17741	Info Leakage: Stack out-of-bounds read in hostvisor.	No	Yes
CVE-2010-0297	Code Execution: Buffer overflow in I/O virtualization code.	No	Yes
CVE-2014-0049	Code Execution: Buffer overflow in I/O virtualization code.	No	Yes
CVE-2013-1798	Info Leakage: Improper handling of invalid combination of operations for virtual IOAPIC.	No	Yes
CVE-2016-4440	Code Execution: Mishandling of virtual APIC state.	No	Yes
CVE-2016-9777	Privilege Escalation: Out-of-bounds array access using VCPU index in interrupt virtualization code.	No	Yes
CVE-2015-3456	Code Execution: Memory corruption in virtual floppy driver allows VM user to execute arbitrary code in hostvisor.	No	Yes
CVE-2011-2212	Privilege Escalation: Buffer overflow in the virtio subsystem allows guest to gain privileges to the host.	No	Yes
CVE-2011-1750	Privilege Escalation: Buffer overflow in the virtio subsystem allows guest to gain privileges to the host.	No	Yes
CVE-2015-3214	Code Execution: Out-of-bound memory access in QEMU leads to memory corruption.	No	Yes
CVE-2012-0029	Code Execution: Buffer overflow allows VM users to execute arbitrary code in QEMU	No	Yes
CVE-2017-1000407	Denial-of-Service: VMs crash hostvisor by flooding the I/O port with write requests.	No	No
CVE-2017-1000252	Denial-of-Service: Out-of-bounds value causes assertion failure and hypervisor crash.	No	No
CVE-2014-7842	Denial-of-Service: Bug in KVM allows guest users to crash its own OS.	No	No
CVE-2018-1087	Privilege Escalation: Improper handling of exception allows guest users to escalate their privileges to its own OS.	No	No

Table 7: Selected Set of Analyzed CVEs - from VM

Bug	Description	KVM	HypSec
CVE-2009-3234	Privilege Escalation: Kernel stack buffer overflow resulting in ret2usr [43].	No	Yes
CVE-2010-2959	Code Execution: Integer overflow resulting in function pointer overwrite.	No	Yes
CVE-2010-4258	Privilege Escalation: Improper handling of get_fs value resulting in kernel memory overwrite.	No	Yes
CVE-2009-3640	Privilege Escalation: Improper handling of APIC state in hostvisor.	No	Yes
CVE-2009-4004	Privilege Escalation: Buffer overflow in hostvisor.	No	Yes
CVE-2013-1943	Privilege Escalation, Info Leakage: Mishandling of memory slot allocation allows host users to access hostvisor memory.	No	Yes
CVE-2016-10150	Privilege Escalation: Use-after-free in hostvisor.	No	Yes
CVE-2013-4587	Privilege Escalation: Array index error in hostvisor.	No	Yes
CVE-2018-18021	Privilege Escalation: Mishandling of VM register state allows host users to redirect hostvisor execution.	No	Yes
CVE-2016-9756	Info Leakage: Improper initialization in code segment resulting in information leakage in hostvisor stack.	No	Yes
CVE-2013-6368	Privilege Escalation: Mishandling of APIC state in hostvisor.	No	Yes
CVE-2015-4692	Memory Corruption: Mishandling of APIC state in hostvisor.	No	Yes
CVE-2013-4592	Denial-of-Service: Host users cause memory leak in hostvisor.	No	No

Table 8: Selected Set of Analyzed CVEs - from host user

hostvisor privileges and compromise VM data. The CVEs related to our threat model could result in information leakage, privilege escalation, code execution, and memory corruption in Linux/KVM. While KVM does not protect VM data against any of these compromises, HypSec protects against all of them. HypSec does not guarantee availability and cannot protect against CVEs that allow VMs or host users to cause denial of service in the hostvisor. Vulnerabilities that allow unprivileged guest users to attack their own VMs like CVE-2014-7842 and CVE-2018-1087 are unrelated to HypSec's threat model; protection against CVEs of these types is an area of future work.

We also executed attacks representative of information leakage to show that HypSec protects VM data even if an attacker has full control of the hostvisor. First, we simulated an attacker trying to read or modify VMs' memory pages. We added a hook to KVM which modifies a page that a targeted gVA maps to. As expected, the compromised KVM (without HypSec) successfully modified the VM page. Using HypSec, the same attack causes a trap to the corevisor which rejects the invalid memory access.

Second, we simulated a host that tries to tamper with a VM's nested page table by redirecting a gPA's NPT mapping to host-owned pages. This is in contrast to the prior attack of modifying VM pages, but shares the same goal of accessing VM data in memory. We added a hook to the nested page fault handler in KVM; the hook allocates a new zero page in the host OS's address space, which in a real attack could contain arbitrary code data. The hook associates a range of a VM's gPAs with this zero page. As expected, this attack succeeds in KVM but fails in HypSec. First, the attacker has no access to the sNPT walked by the MMU. Second, the corevisor synchronizes the vNPT to sNPT mapping on the gPA's initial fault during VM boot, so malicious vNPT modifications do not propagate to sNPT.

7 Related Work

The idea of retrofitting a commodity hypervisor with a smaller core was inspired by KVM/ARM's split-mode virtualization [22, 23], which introduced a thin software layer to enable Linux KVM to make use of ARM hardware virtualization extensions without significant changes to Linux, but did nothing to reduce the hypervisor TCB. HypSec builds on this work to leverage ARM hardware virtualization support to run the corevisor with special hardware privileges to protect VM

data against a compromised hostvisor. More recently, Nested Kernel [25] used the idea of retrofitting a small TCB into a commodity OS kernel, FreeBSD, to intercept MMU updates to enforce kernel code integrity. Both HypSec and Nested Kernel retrofit commodity system software with a small TCB that mediates accesses to critical hardware resources and strengthens system security guarantees with modest implementation and performance costs. Nested Kernel focuses on a different threat model and does not protect against vulnerabilities in existing kernel code in part because both its TCB and untrusted components run at the highest hardware privilege level. In contrast, HypSec deprivileges the hostvisor and uses its TCB to provide data confidentiality and integrity even in the presence of hypervisor vulnerabilities in the hostvisor.

Bare-metal hypervisors often claim a smaller TCB as an advantage over hosted hypervisors, but in practice, the aggregate TCB of the widely-used Xen [11] bare-metal hypervisor includes Dom0 [18, 92] and therefore can be no smaller than hosted hypervisors like KVM. Some work thus focuses on reducing Xen's attack surface by redesigning Dom0 [15, 18, 59]. Unlike HypSec, these approaches cannot protect a VM against a compromised Xen or Dom0. We believe Xen can be restructured using HypSec by moving resource management, interrupt virtualization, and other hardware-specific dependencies, along with Dom0, into a hostvisor to further reduce Xen's TCB to protect VM data.

Microhypervisors [32, 74] take a microkernel approach to build clean-slate hypervisors from scratch to reduce the hypervisor TCB. For example, NOVA [74] moves various aspects of virtualization such as CPU and I/O virtualization to user space services. The virtualization services are trusted but instantiated per VM so that compromising them only affects the given VM. Others simplify the hypervisor to reduce its TCB by removing [72] or disabling [60] virtual device I/O support in hypervisors, or partitioning VM resources statically [42, 73]. Although a key motivation for both microhypervisors and HypSec is to reduce the size of the TCB, HypSec does not require a clean-slate redesign, and supports existing full-featured commodity hypervisors without removing important hypervisor features such as I/O support and dynamic resource allocation while preserving confidentiality and integrity of VM data even if the hostvisor is compromised.

HyperLock [86], DeHype [88], and Nexen [70] focus on deconstructing existing monolithic hypervisors by segregating hypervisor functions to per VM instances. While this can isolate an exploit of hypervisor functions to a given VM instance, if a vulnerability is exploitable in one VM instance, it is likely to be exploitable in another as well. Nexen builds on Nested Kernel to retrofit Xen in this manner, though it does not protect against vulnerabilities in its shared hypervisor services. In contrast to HypSec, these systems focus on availability and do not fully protect the confidentiality and integrity of VM data against a compromised hypervisor or host OS.

CloudVisor [92] uses a small, specialized host hypervisor to support nested virtualization and protect user VMs against an untrusted Xen guest hypervisor, though Xen modifications are required. CloudVisor encrypts VM I/O and memory but does not fully protect CPU state, contrary to its claims of "providing both secrecy and integrity to a VM's states, including CPU states." For example, the VM program counter is exposed to Xen to support I/O. As with any nested virtualization approach, performance overhead on application workloads is a problem. Furthermore, CloudVisor does not support widely used paravirtual I/O. CloudVisor has a smaller TCB by not supporting public key cryptography, making key management problematic. In contrast, HypSec protects both CPU and memory state via access control, not encryption, making it possible to support full-featured hypervisor functionality such as paravirtual I/O. HypSec also does not require nested virtualization, avoiding its performance overhead.

To protect user data in virtualization systems, others enable and require VM support for specialized hardware such as Intel SGX [36] or ARM TrustZone. Haven [12] and S-NFV [71] use Intel SGX to protect application data but unlike HypSec, cannot protect the whole VM including the guest OS and applications against an untrusted hypervisor. Although HypSec relies on a TEE to support key management, it fundamentally differs from other approaches which extensively use TEEs for much more than storing keys. Others [34, 96] run a security monitor in ARM TrustZone and rely on ARM IP features such as TrustZone Address Space Controller to protect VMs. vTZ [34] virtualizes TrustZone and protects the guest TEE against an untrusted hypervisor, but does not protect the normal world VM. HA-VMSI [96] protects the normal world VM against a compromised hypervisor but supports limited virtualization features. In contrast, HypSec protects the entire normal world VM against an untrusted hypervisor without requiring VMs to use specialized hardware. HypSec leverages ARM VE to trap VM exceptions to EL2 while retaining hypervisor functionality. Others [40, 78, 90] propose hardware-based approaches to protect VM data in CPU and memory against an untrusted hypervisor. However, without actual hardware implementations, these works implement the proposed changes by modifying either Xen [40] or QEMU [90], or on a simulator [78]. Some of them [40, 78] do not support commodity hypervisors. In contrast, HypSec leverages existing hardware features to protect virtual machine data and supports KVM on ARM server hardware.

Recent architectural extensions [3, 37] proposed hardware support on x86 for encrypted virtual machines. Fidelius [89] leverages AMD's SEV (Secure Encrypted Virtualization) [3] to protect VMs. Unlike these encryption-based approaches, HypSec primarily uses access control mechanisms.

Some projects focus on hardening the hypervisor to prevent exploitation. They improve hypervisor security by either enforcing control flow integrity [84] or measuring runtime hypervisor integrity [9, 26]. These approaches can be applied to HypSec to further strengthen VM security. XMHF [81]

verifies the memory integrity of its hypervisor codebase but supports single VM with limited virtualization features. Verification of HypSec's TCB is an area of future work.

Various projects extend a trusted hypervisor to protect software within VMs, including protecting applications running on an untrusted guest OS in the VM [16, 17, 33, 55, 91], ensuring kernel integrity and protecting against rootkits and code injection attacks or to isolate I/O channels [64, 69, 83, 85, 95], and dividing applications and system components in VMs then relying on the hypervisor to safeguard interactions among secure and insecure components [27, 54, 76, 79]. Overshadow [16] and Inktag [33] have some similarities with HypSec in that they use a more trusted hypervisor component to protect against untrusted kernel software. Overshadow and Inktag also assume applications use end-to-end encrypted network I/O, though they protect file I/O by replacing it with memory-mapped I/O to encrypted memory. HypSec has three key differences with these approaches. First, instead of memory encryption, HypSec primarily uses access control, which is more lightweight and avoids the need to emulate functions that are problematic when memory is encrypted. Second, instead of instrumenting or emulating complex system calls, HypSec relies on hardware virtualization mechanisms to interpose on hardware events of interest. Finally, instead of protecting against guest OS exploits, HypSec protects against hypervisor and host OS exploits, which none of the other approaches do.

8 Conclusions

We have created HypSec, a new approach to hypervisor design that reduces the TCB necessary to protect virtual machines. HypSec decomposes a monolithic hypervisor into a small, trusted corevisor and untrusted hostvisor, the latter containing the vast majority of hypervisor functionality including an entire host operating system kernel. The corevisor leverages hardware virtualization support to execute at a higher privilege level and provide access control mechanisms to restrict hostvisor access to VM data. It can be simple because it only needs to perform basic CPU and memory virtualization. When VMs use secure I/O channels, HypSec can protect the confidentiality and integrity of all VM data. We have demonstrated that HypSec can support existing commodity hypervisors by retrofitting KVM/ARM. The resulting TCB is orders of magnitude less than the original KVM/ARM. HypSec provides strong security guarantees to VMs with only modest performance overhead for real application workloads.

Acknowledgments

Steve Bellovin, Christoffer Dall, and Nathan Dautenhahn provided helpful comments on earlier drafts of this paper. This work was supported in part by NSF grants CNS-1717801 and CNS-1563555.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer USENIX Conference (USENIX Summer 1986)*, pages 93–112, Atlanta, GA, June 1986.

[2] K. Adams and O. Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2006)*, pages 2–13, San Jose, CA, Oct. 2006.

[3] Advanced Micro Devices. Secure Encrypted Virtualization API Version 0.16. https://support.amd.com/TechDocs/55766_SEV-KM%20API_Spec.pdf, Feb. 2018.

[4] Amazon Web Services, Inc. Introducing Amazon EC2 A1 Instances Powered By New Arm-based AWS Graviton Processors. https://aws.amazon.com/about-aws/whats-new/2018/11/introducing-amazon-ec2-a1-instances/, Nov. 2018.

[5] Amazon Web Services, Inc. AWS Key Management Service (KMS). https://aws.amazon.com/kms/, May 2019.

[6] ArchWiki. dm-crypt. https://wiki.archlinux.org/index.php/dm-crypt, Apr. 2018.

[7] ARM Ltd. ARM Security Technology - Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, Apr. 2009.

[8] ARM Ltd. ARM System Memory Management Unit Architecture Specification - SMMU architecture version 2.0. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0062d.c/IHI0062D_c_system_mmu_architecture_specification.pdf, June 2016.

[9] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky. HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS 2010)*, pages 38–49, Chicago, IL, Oct. 2010.

[10] M. Backes, G. Doychev, and B. Kopf. Preventing Side-Channel Leaks in Web Traffic: A Formal Approach. In *20th Annual Network and Distributed System Security Symposium (NDSS 2013)*, San Diego, CA, Feb. 2013.

[11] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 164–177, Bolton Landing, NY, Oct. 2003.

[12] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI 2014)*, pages 267–283, Broomfield, CO, Oct. 2014.

[13] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP 1995)*, pages 267–283, Copper

Mountain, CO, Dec. 1995.

[14] Business Wire. Research and Markets: Global Encryption Software Market (Usage, Vertical and Geography) - Size, Global Trends, Company Profiles, Segmentation and Forecast, 2013 - 2020. https://www.businesswire.com/news/home/20150211006369/en/Research-Markets-Global-Encryption-Software-Market-Usage, Feb. 2015.

[15] S. Butt, H. A. Lagar-Cavilla, A. Srivastava, and V. Ganapathy. Self-service Cloud Computing. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS 2012)*, pages 253–264, Raleigh, NC, Oct. 2012.

[16] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: A Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2008)*, pages 2–13, Seattle, WA, Mar. 2008.

[17] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic. SecureME: A Hardware-software Approach to Full System Security. In *Proceedings of the 25th International Conference on Supercomputing (ICS 2011)*, pages 108–119, Tucson, AZ, May 2011.

[18] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield. Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP 2011)*, pages 189–202, Cascais, Portugal, Oct. 2011.

[19] J. Corbet. KAISER: hiding the kernel from user space. https://lwn.net/Articles/738975/, Nov. 2017.

[20] C. Dall, S.-W. Li, J. Lim, J. Nieh, and G. Koloventzos. ARM Virtualization: Performance and Architectural Implications. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA 2016)*, pages 304–316, Seoul, South Korea, June 2016.

[21] C. Dall, S.-W. Li, and J. Nieh. Optimizing the Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC 2017)*, pages 221–234, Santa Clara, CA, July 2017.

[22] C. Dall and J. Nieh. KVM/ARM: Experiences Building the Linux ARM Hypervisor. Technical Report CUCS-010-13, Department of Computer Science, Columbia University, June 2013.

[23] C. Dall and J. Nieh. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2014)*, pages 333–347, Salt Lake City, UT, Mar. 2014.

[24] A. Danial. cloc: Count Lines of Code. https://github.com/AlDanial/cloc, May 2019.

[25] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2015)*, pages 191–206, Istanbul, Turkey, Mar. 2015.

[26] L. Deng, P. Liu, J. Xu, P. Chen, and Q. Zeng. Dancing with Wolves: Towards Practical Event-driven VMM Monitoring. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2017)*, pages 83–96, Xi'an, China, Apr. 2017.

[27] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-based Platform for Trusted Computing. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 193–206, Bolton Landing, NY, Oct. 2003.

[28] Google. Google Cloud Security and Compliance Whitepaper - How Google protects your data. https://static.googleusercontent.com/media/gsuite.google.com/en//files/google-apps-security-and-compliance-whitepaper.pdf, Sept. 2017.

[29] Google. HTTPS encryption on the web – Google Transparency Report. https://transparencyreport.google.com/https/overview, Apr. 2018.

[30] S. Hajnoczi. An Updated Overview of the QEMU Storage Stack. https://events.linuxfoundation.org/slides/2011/linuxcon-japan/lcj2011_hajnoczi.pdf, June 2011.

[31] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *Proceedings of the 17th USENIX Security Symposium (USENIX Security 2008)*, pages 45–60, San Jose, CA, July 2008.

[32] G. Heiser and B. Leslie. The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors. In *Proceedings of the 1st ACM Asia-pacific Workshop on Workshop on Systems (APSys 2010)*, pages 19–24, New Delhi, India, Aug. 2010.

[33] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. InkTag: Secure Applications on an Untrusted Operating System. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013)*, pages 265–278, Houston, TX, Mar. 2013.

[34] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and Haibing. vTZ: Virtualizing ARM Trustzone. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security 2017)*, pages 541–556, Vancouver, BC, Canada, Aug. 2017.

[35] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, 325462-044US, Aug. 2012.

[36] Intel Corporation. Intel Software Guard Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, Oct. 2014.

[37] Intel Corporation. Intel Architecture Memory Encryption Technologies Specification. https://software.intel.com/sites/default/files/managed/a5/16/Multi-Key-Total-Memory-Encryption-Spec.pdf, Dec. 2017.

[38] International Organization for Standardization and International Electrotechnical Commission. ISO/IEC 11889-1:2015 - Information technology – Trusted platform module library. https://www.iso.org/standard/66510.html, Sept. 2016.

[39] G. Irazoqui, T. Eisenbarth, and B. Sunar. S$A: A Shared Cache Attack That Works Across Cores and Defies VM Sandboxing

– and Its Application to AES. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP 2015)*, pages 591–604, San Jose, CA, May 2015.

[40] S. Jin, J. Ahn, S. Cha, and J. Huh. Architectural Support for Secure Virtualization Under a Vulnerable Hypervisor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*, pages 272–283, Porto Alegre, Brazil, Dec. 2011.

[41] R. Jones. Netperf. https://github.com/HewlettPackard/netperf, June 2018.

[42] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. NoHype: Virtualized Cloud Infrastructure Without the Virtualization. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA 2010)*, pages 350–361, Saint-Malo, France, June 2010.

[43] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis. kGuard: Lightweight Kernel Protection against Return-to-User Attacks. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security 2012)*, pages 459–474, Bellevue, WA, Aug. 2012.

[44] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux Virtual Machine Monitor. In *In Proceedings of the 2007 Ottawa Linux Symposium (OLS 2007)*, Ottawa, ON, Canada, June 2007.

[45] kokke. kokke/tiny-aes-c: Small portable aes128/192/256 in c. https://github.com/kokke/tiny-AES-c, 2018.

[46] KVM Contributors. Kernel Samepage Merging. https://www.linux-kvm.org/page/KSM, July 2015.

[47] KVM Contributors. Tuning KVM. http://www.linux-kvm.org/page/Tuning_KVM, May 2015.

[48] KVM Contributors. KVM Unit Tests. http://www.linux-kvm.org/page/KVM-unit-tests, May 2019.

[49] S. Landau. Making Sense from Snowden: What's Significant in the NSA Surveillance Revelations. *IEEE Security and Privacy*, 11(4):54–63, July 2013.

[50] Let's Encrypt. Let's encrypt stats - let's encrypt. https://letsencrypt.org/stats/, Apr. 2018.

[51] J. Liedtke. On Micro-kernel Construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP 1995)*, pages 237–250, Copper Mountain, CO, Dec. 1995.

[52] J. Lim, C. Dall, S.-W. Li, J. Nieh, and M. Zyngier. NEVE: Nested Virtualization Extensions for ARM. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP 2017)*, pages 201–217, Shanghai, China, Oct. 2017.

[53] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-Level Cache Side-Channel Attacks Are Practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP 2015)*, pages 605–622, San Jose, CA, May 2015.

[54] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS 2015)*, pages 1607–1619, Denver, CO, Oct. 2015.

[55] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP 2010)*, pages 143–158, Oakland, CA, May 2010.

[56] Microsoft. Hyper-V Technology Overview. https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/hyper-v-technology-overview, Nov. 2016.

[57] Microsoft. BitLocker. https://docs.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-overview, Jan. 2018.

[58] Microsoft Azure. Key Vault - Microsoft Azure. https://azure.microsoft.com/en-in/services/key-vault/, May 2019.

[59] D. G. Murray, G. Milos, and S. Hand. Improving Xen Security Through Disaggregation. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2008)*, pages 151–160, Seattle, WA, Mar. 2008.

[60] A. Nguyen, H. Raj, S. Rayanchu, S. Saroiu, and A. Wolman. Delusional Boot: Securing Hypervisors Without Massive Re-engineering. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys 2012)*, pages 141–154, Bern, Switzerland, Apr. 2012.

[61] OP-TEE. Open Portable Trusted Execution Environment. https://www.op-tee.org/, 2017.

[62] orlp. Ed25519. https://github.com/orlp/ed25519, 2017.

[63] Reuters. Cloud companies consider Intel rivals after the discovery of microchip security flaws. https://www.cnbc.com/2018/01/10/cloud-companies-consider-intel-rivals-after-security-flaws-found.html, Jan. 2018.

[64] R. Riley, X. Jiang, and D. Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID 2008)*, pages 1–20, Cambridge, MA, Sept. 2008.

[65] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009)*, pages 199–212, Chicago, IL, Nov. 2009.

[66] R. Russell. Hackbench. http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c, Jan. 2008.

[67] R. Russell. virtio: Towards a De-Facto Standard for Virtual I/O Devices. *SIGOPS Operating Systems Review*, 42(5):95–103, July 2008.

[68] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end Arguments in System Design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, Nov. 1984.

[69] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP 2007)*, pages 335–350, Stevenson, WA, Oct. 2007.

[70] L. Shi, Y. Wu, Y. Xia, N. Dautenhahn, H. Chen, B. Zang, and J. Li. Deconstructing Xen. In *24th Annual Network and Distributed System Security Symposium (NDSS 2017)*, San Diego, CA, Feb. 2017.

[71] M.-W. Shih, M. Kumar, T. Kim, and A. Gavrilovska. S-NFV:

Securing NFV States by Using SGX. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization (SDN-NFV Security 2016)*, pages 45–48, New Orleans, LA, Mar. 2016.

[72] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato. BitVisor: A Thin Hypervisor for Enforcing I/O Device Security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2009)*, pages 121–130, Washington, DC, Mar. 2009.

[73] Siemens. jailhouse - Linux-based partitioning hypervisor. https://github.com/siemens/jailhouse, May 2019.

[74] U. Steinberg and B. Kauer. NOVA: A Microhypervisor-based Secure Virtualization Architecture. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys 2010)*, pages 209–222, Paris, France, Apr. 2010.

[75] P. Stewin and I. Bystrov. Understanding DMA Malware. In *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2012)*, pages 21–41, Heraklion, Crete, Greece, July 2013.

[76] R. Strackx and F. Piessens. Fides: Selectively Hardening Software Application Components Against Kernel-level or Process-level Malware. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS 2012)*, pages 2–13, Raleigh, NC, Oct. 2012.

[77] SUSE. Performance Implications of Cache Modes. https://www.suse.com/documentation/sles11/book_kvm/data/sect1_3_chapter_book_kvm.html, Sept. 2016.

[78] J. Szefer and R. B. Lee. Architectural Support for Hypervisor-secure Virtualization. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*, pages 437–450, London, England, UK, Mar. 2012.

[79] R. Ta-Min, L. Litty, and D. Lie. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 279–292, Seattle, WA, Nov. 2006.

[80] The Apache Software Foundation. ab - Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/2.4/programs/ab.html, Apr. 2015.

[81] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta. Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP 2013)*, pages 430–444, San Francisco, CA, May 2013.

[82] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, pages 181–194, Boston, MA, Dec. 2002.

[83] X. Wang, Y. Chen, Z. Wang, Y. Qi, and Y. Zhou. Secpod: A Framework for Virtualization-based Security Systems. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC 2015)*, pages 347–360, Santa Clara, CA, July 2015.

[84] Z. Wang and X. Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP 2010)*, pages 380–395, Oakland, CA, May 2010.

[85] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering Kernel Rootkits with Lightweight Hook Protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009)*, pages 545–554, Chicago, IL, Nov. 2009.

[86] Z. Wang, C. Wu, M. Grace, and X. Jiang. Isolating Commodity Hosted Hypervisors with HyperLock. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys 2012)*, pages 127–140, Bern, Switzerland, Apr. 2012.

[87] C. Williams. Microsoft: Can't wait for ARM to power MOST of our cloud data centers! Take that, Intel! Ha! Ha! https://www.theregister.co.uk/2017/03/09/microsoft_arm_server_followup/, Mar. 2017.

[88] C. Wu, Z. Wang, and X. Jiang. Taming Hosted Hypervisors with (Mostly) Deprivileged Execution. In *20th Annual Network and Distributed System Security Symposium (NDSS 2013)*, San Diego, CA, Feb. 2013.

[89] Y. Wu, Y. Liu, R. Liu, H. Chen, B. Zang, and H. Guan. Comprehensive VM Protection Against Untrusted Hypervisor Through Retrofitted AMD Memory Encryption. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA 2018)*, pages 441–453, Vienna, Austria, Feb. 2018.

[90] Y. Xia, Y. Liu, and H. Chen. Architecture Support for Guest-transparent VM Protection from Untrusted Hypervisor and Physical Attacks. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA 2013)*, pages 246–257, Shenzhen, China, Feb. 2013.

[91] J. Yang and K. G. Shin. Using Hypervisor to Provide Data Secrecy for User Applications on a Per-page Basis. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2008)*, pages 71–80, Seattle, WA, Mar. 2008.

[92] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP 2011)*, pages 203–216, Cascais, Portugal, Oct. 2011.

[93] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS 2012)*, pages 305–316, Raleigh, NC, Oct. 2012.

[94] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-Tenant Side-Channel Attacks in Paas Clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS 2014)*, pages 990–1003, Nov. 2014.

[95] Z. Zhou, M. Yu, and V. D. Gligor. Dancing with Giants: Wimpy Kernels for On-Demand Isolated I/O. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP 2014)*, pages 308–323, San Jose, CA, May 2014.

[96] M. Zhu, B. Tu, W. Wei, and D. Meng. HA-VMSI: A Lightweight Virtual Machine Isolation Approach with Commodity Hardware for ARM. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2017)*, pages 242–256, Xi'an, China, Apr. 2017.

WAVE: A Decentralized Authorization Framework with Transitive Delegation

Michael P Andersen, Sam Kumar, Moustafa AbdelBaky
Gabe Fierro, John Kolb, Hyung-Sin Kim, David E. Culler, Raluca Ada Popa
University of California, Berkeley

Abstract

Most deployed authorization systems rely on a central trusted service whose compromise can lead to the breach of millions of user accounts and permissions. We present WAVE, an authorization framework offering *decentralized trust*: no central services can modify or see permissions and any participant can delegate a portion of their permissions autonomously. To achieve this goal, WAVE adopts an expressive authorization model, enforces it cryptographically, protects permissions via a novel encryption protocol while enabling discovery of permissions, and stores them in an untrusted scalable storage solution. WAVE provides competitive performance to traditional authorization systems relying on central trust. It is an open-source artifact and has been used for two years for controlling 800 IoT devices.

1 Introduction

Authorization and authentication are fundamental components of many systems. Most authorization systems today *rely on centralized services* such as centralized credential stores (e.g., [15, 19, 56]), Access Control Lists (ACLs), Active Directory, and OAuth [4]. For example, in a calendar application, a central service stores which users have access to what calendars, and users authenticate to it, e.g. via username and password. In these systems, delegation is critical: for instance, allowing an assistant to edit your calendar, and letting the assistant further delegate restrictive view access to an event organizer. These forms of delegation are typically implemented as changes to a centralized ACL.

However, this approach presents two fundamental problems. First, a centralized service is a central point of attack: a single attack can simultaneously compromise many user accounts and permissions. There have been numerous such breaches [39], and attackers even managed to log in as the victim users. Second, the operator of the central server has a complete view of the private permission data for all users (thus seeing users' social relationships [54]), and can modify permissions [2].

Responding to the weaknesses of centralized systems, recent security systems are increasingly avoiding a trusted central service. This approach has been adopted by end-to-end encryption systems [25], such as WhatsApp and Signal, blockchains (e.g., Bitcoin, Ethereum, Zcash), or ledgers (e.g., IBM's Hyperledger [17], Certificate Transparency [41], Key Transparency [32]). Our goal is to build a scalable decentralized authorization system, permitting delegation under a similar threat model.

We propose a decentralized authorization system that *does not rely on a trusted service*, *WAVE ("**W**AVE is an **A**uthorization **V**erification **E**ngine")*. WAVE offers decentralized trust: each user's WAVE client manages the permissions of that user and can delegate access to other users. WAVE enforces delegation *cryptographically*, not via a trusted service. It aims to capture a wide range of authorization policies and to provide an alternative to traditional systems, such as OAuth [4] and Active Directory.

Importantly, in providing decentralized transitive delegation, WAVE facilitates applications that span multiple trust domains. For example, IoT orchestration applications like If This Then That (IFTTT) [3] tie together multiple vendors and users, but IFTTT's design relies on several central points of attack: the vendor OAuth servers and the IFTTT token storage servers. The compromise of any one of these servers may affect hundreds of thousands of users. Using WAVE, greater cross-administrative-domain orchestration can be achieved with no central authorization servers, reducing the trust that each domain must place in the others.

1.1 Usage Scenarios

While authorization plays a key role in the security of almost any system today, the benefits of decentralized authorization are most pronounced in systems that are inherently distributed, where the prevailing centralized authorization schemes undermine what would otherwise be a resilient system. Our deployment of WAVE over the past two years has focused on securing distributed IoT devices and services used to monitor and control over twenty small to medium-sized commercial and residential buildings; hence, we will use smart buildings as a running example.

Consider a set of campuses, each owned by a property manager. Each campus is composed of multiple buildings, with portions of each building leased out to tenants by the property manager. The property manager within each campus is the authority for the cyberphysical resources associated with the buildings in the campus, but they must *delegate* permission to the individual building managers who must further delegate permissions to the tenants, allowing them to control the portions of the buildings that they rent. Any of these principals may then further delegate permissions to IoT devices, long-running analytics or control services operating on their behalf, perhaps provided by the utility. The building manager and/or tenant will also grant ephemeral permissions on subsets of the building infrastructure to contractors (like HVAC commissioning teams) and, especially in our case, to

researchers.

A similar structure occurs in small residential buildings where a homeowner installs smart devices such as lights and thermostats and needs to delegate permission on those devices to their partner, guest, nanny, or children.

Cross-administrative-domain delegation is present in both examples. In larger buildings, we see the boundary between the property owner and the tenants. In residential buildings, this is most evident when using orchestration tools like IFTTT, where an organization, distinct from the owner of the devices, runs the controller service and needs to obtain permission from the owner.

WAVE is not limited to IoT. It provides general purpose delegable authorization and can, for example, be used in place of OAuth to remove the risk of the centralized token-issuing server and allow for richer delegation semantics.

1.2 High-Level Security Goal & Threat Model

At a high level, our objective is to design a system where the compromise of an authorization server does not compromise all the users' permissions. Namely, even if an adversary has compromised any authorization servers and users, it should not be able to:

1. Grant permissions on behalf of uncompromised users.
2. See permissions granted in the system, beyond those potentially relevant to the compromised users. See §4 and §B for our definition of relevant.
3. Undetectably modify the permissions received/granted/revoked by uncompromised users from uncompromised users, or undetectably prevent uncompromised users from granting/receiving/revoking permissions to/from uncompromised users.

1.3 Failure Of Existing Systems

Existing authorization systems fall short in two general areas: they do not meet our Security Goals or they do not provide the features required for IoT usage scenarios. More concretely, we summarize the following six requirements that are not simultaneously met by any existing system (as illustrated in Table 4):

No reliance on central trust. For example, in the smart buildings scenario, the status quo has certain devices (e.g LiFx light bulbs) perform their authorization on the vendor's server in the cloud. If that server is compromised, all of those devices in all of the customer buildings are compromised. In this case, the adversary can violate all three Security Goals.

Transitive delegation. The smart building scenario illustrates the necessity for transitive delegation and revocation where, for example, a tenant can further delegate their permissions to a control service or guest and have those permissions predicated on the tenant's permissions. If the tenant moves out, all of the permissions they granted should be automatically revoked, even if the building manager is unaware of the grants the tenant has made. This form of transitive delegation is not found in widely-deployed systems like LDAP or OAuth: where delegation exists, it does not have this transitive predication property. In contrast, this property is well developed in academic work [49, 13, 43, 45, 29, 14, 51, 11].

Protected permissions. Parties should be able to see only the permissions that are potentially relevant to them. Even though the property manager is the authority for all the buildings, they must not be able to see the permissions that the tenants grant (Security Goal #2). Existing systems do not offer a solution to this requirement: in many centralized systems, for example, whoever operates the server can see all the permissions. We elaborate further in §9.

Decentralized verification. Some existing decentralized systems (e.g. SDSI/SPKI [49] and Macaroons [12]) allow only the authority to verify that an action is authorized. This is adequate in the centralized service case where the authority is the service provider, but it does not work in the IoT case where the root authority (the property manager) has nothing to do with the devices needing to verify an action is authorized (for example a thermostat). Any participant must be able to verify that an action is authorized.

No ordering constraints. Delegations must be able to be instantiated in any chronological order. For example, a participant can delegate permissions in anticipation of being granted sufficient ones for the delegation to be useful. We have found this to be critical in our deployments. As a further example, when the building manager's key needed to be replaced (e.g. it expired or was compromised), they created a new key and the property manager had to grant replacement permissions to this new key. In many existing systems (e.g. Macaroons [12]), this necessitates every tenant re-creating their entire permission trees, as all grants must happen in sequence, following the grants to the replacement key. This is not tractable in practice as it requires the coordination of many people and hundreds of devices, leading to extended downtime. Furthermore, when we had such ordering constraints in our prior deployments we observed users choosing insecure long expiry times or broad permissions to avoid this re-issue. As a result, we require that the system enables permission grants to occur out of order, so that permissions grants can be modified (revoked / re-issued) or any key can be "replaced" without re-issuing subsequent delegations. We have also found that this capability leads to safer user practices as "mistakes" like overly narrow permissions and short expiry times are easy to correct.

Offline participants. Not all participants have a persistent online presence. A device may be offline at the time that it is granted permissions (e.g. during installation) and it must be able to discover that it received permissions when it comes online. This is trivial to solve with a centralized authorization system, but is not solved in existing decentralized systems (e.g SDSI/SPKI [49], Macaroons [12] and [13, 43, 45, 29, 27, 44, 59, 18, 57, 50]).

While many existing systems meet some of these require-

ments, no existing work meets all of the requirements concurrently, as shown in §9.

1.4 Challenges and Approach

Compatible authorization model. The first challenge is identifying a model for authorization that is compatible with these requirements. We examined many authorization models [12, 49, 24, 13, 43, 45, 29, 27, 58, 37, 30, 48, 19, 15, 56, 44, 59, 18, 57, 50], but most of them cannot be enforced without a centralized authority or are incompatible with the other requirements. Nevertheless, we found that representing the authorization model as a graph, such as in SDSI/SPKI [49, 24] where a proof of authorization is a path through a graph, is compatible with our requirements, even though the existing systems implementing it fall short.

Consequently, WAVE maintains a global graph of delegations between entities (Fig. 1a), which are associated with participants. An *entity* is a collection of public and private key pairs and can correspond to a user, service, or group. An edge indicates that an entity grants another entity access according to a *policy*, which is one or more permissions along with a description of the resources for which the permissions are granted, and the expiry of the grant. This enables fine-grained transitive delegation with revocation and expiry.

To enforce the policy cryptographically, each edge, from *issuer* to *subject* entity, is a signed certificate recording the delegation of permissions, which we call an *attestation*. A path from an entity to another entity grants access equal to the *intersection* of the policies on that path. The graph enables entities to *prove* they have some permission `P` by revealing a path through the graph from an authority entity to themselves where all the edges of the path grant a superset of `P`. This path is called a *proof*. The graph construction allows permissions to be granted in any order, including delegation of permissions one does not yet possess but expects to receive in the future.

While WAVE's authorization graph and proofs are structurally similar to SDSI/SPKI, WAVE differs in three important aspects: (1) while in SDSI/SPKI only a central authority (holding an ACL) can verify a proof, in WAVE anyone can independently (with no communication) verify a proof yielding an authorization policy. (2) WAVE provides a trustworthy, scalable storage solution for attestations that enables discoverability with offline participants and out of order grants, which is out of scope for SDSI/SPKI. (3) Attestations are encrypted in WAVE whereas they are visible in SDSI/SPKI. These differences enable meeting the requirements in §1.3.

Scalable untrusted storage. To support granting permissions to offline participants, we use a storage system that enables participants to discover attestations when they later come online. To meet the requirements above, the storage must be able to prove its integrity cryptographically, so as not to compromise Security Goal #3.

Our first design of WAVE [9] was built on Ethereum, which has these properties. Unfortunately, our experiments showed that a blockchain-based system will not scale to a global size, even though changing permissions is far less common than accessing data.

We present a new type of transparency log, the *Unequivocable Log Derived Map (ULDM)*. Unlike Certificate Transparency [41], which cannot form a proof of nonexistance needed for revocations, or Key Transparency [32], which requires users to audit every object at every epoch, a ULDM is both capable of handling revocations and is efficiently auditable. The ULDM forms the foundation of a horizontally scalable storage tier with cryptographically proven integrity, which could also be useful outside of WAVE. Our current design, described in §5, allows for a shared-nothing architecture of storage servers with independent auditors that need only communicate periodically (e.g., once a day) with clients to verify the correct operation of the storage. The resulting architecture is arbitrarily horizontally scalable with each node having a higher capacity and lower latency than a blockchain, as we show in §8.

Confidentiality of permissions. To meet the requirement of protected permissions and Security Goal #2 despite the public ULDM storage tier, there must be a mechanism to prevent the storage servers or the general public from seeing the permissions, while ensuring that parties forming and verifying proofs can see the necessary permissions. The challenge lies in preserving confidentiality while enabling out of order delegation and offline participants. We overcome this challenge with a novel technique called *reverse-discoverable encryption* (RDE, §4) used to encrypt attestations. RDE allows entities to efficiently discover and decrypt the attestations that they can use in a valid proof while using policy-aware encryption to hide most other attestations. Importantly, RDE does not introduce additional constraints on the ordering of delegations or liveness of participants.

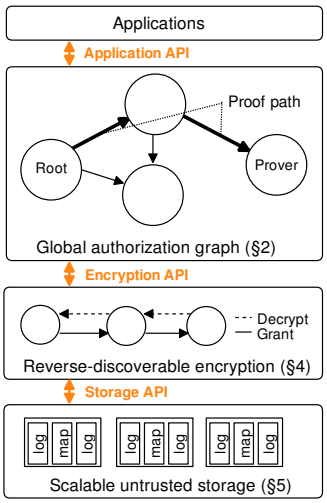
Our implementation of WAVE is a real-world *open-source* artifact [7]. We have deployed and operated various versions of WAVE over the past two years. During this time, WAVE has been used to control more than 20 buildings containing more than 800 IoT devices. We discuss lessons from our deployment in §8.4; in particular, this has allowed us to confirm that the authorization and delegation model presented here is useful in practice. Further, WAVE has offered performance comparable to traditional authorization systems, validating real proofs in 1–4 ms, depending on the depth of delegation.

2 WAVE Overview

WAVE runs as a service that can be logically divided into three layers (Fig. 1a) each providing an API (Fig. 1b).

2.1 Global Authorization Graph

Recall that the global authorization graph consists of entities, which are bundles of public and private keys, and attestations, which are the permission grants between them. The client (representing a user, device, or service) inter-



(a) The WAVE stack

Subsystem	API
Application	CreateEntity() ⟹ (privEnt, pubEnt)
	Delegate(issuer:privEnt, subject:pubEnt, policy) ⟹ attestation
	CreateProof(subject:privEnt, policy) ⟹ (proof)
	VerifyProof(proof) ⟹ (subject:pubEnt, policy)
	NewName(issuer: privEnt, subject:pubEnt, name) ⟹ (nameDecl)
	ResolveName(resolver: privEnt, name) ⟹ (nameDecl)
	Revoke(issuer:privEnt, object:attestation/pubEnt)
Encryption	EncryptAttest(attestation, partition) ⟹ attCiphertext
	DecryptAttest(perspective:privEnt, attCiphertext) ⟹ attestation
Storage	Put(object, server) ⟹ hash
	Get(hash, server) ⟹ object
	Enqueue(list:hash, entry:hash, server)
	IterQueue(list:hash, cursor) ⟹ (entry:hash, newCursor)

(b) The API provided by WAVE's stack.

Figure 1: An overview of WAVE

acts through the WAVE service with the global authorization graph. Clients can create new entities (e.g., for a service they are deploying).

To grant permissions to other entities, clients use the WAVE service to construct an attestation signed by the granting entity containing a policy describing the permissions. An attestation A consists of:

- A.issuer: the entity that wishes to grant permissions to another entity,
- A.subject: the entity receiving the permissions,
- A.policy: an expression of permissions, for example, RTree described in §2.4, and
- a revocation commitment described in §6.1
- signature(s) from the issuer.

When accessing a service or controlling a device, clients request a proof from the WAVE service; the WAVE service will search for a path through the global authorization graph from the authority for the service or device in question to the client's entity, where each edge grants a superset of the required permissions. The representation of this path is a self-standing proof of authorization that can be verified without communication with the proving entity. The receiving service or device can use the WAVE service to validate a proof, yielding the authorization policy it permits.

The WAVE service also allows for mapping human readable names to entity public keys to make the system more usable, as we elaborate in §6.2.

2.2 Reverse-Discoverable Encryption (RDE)

To ensure the privacy of permissions, the WAVE service uses our protocol, Reverse-Discoverable Encryption (described in §4) to encrypt the attestations. The encryption layer is transparent to clients: the WAVE service will discover and decrypt the portion of the global graph that concerns the client automatically. The only time a client interacts with the encryption layer is when they use RDE to encrypt messages for application-level end-to-end encryption, which is beyond the scope of this paper.

2.3 Scalable Untrusted Storage

When the client instructs the WAVE service to create an entity or an attestation, the WAVE service will place the public keys (for entities) or RDE ciphertext (for attestations) into the scalable untrusted storage (§5). As with RDE, the placement into storage is transparent to clients: clients operate only at the level of granting permissions, creating proofs and verifying proofs. The WAVE client will interact with the storage to discover and decrypt the portion of the global graph necessary for performing those actions without the client manually publishing or retrieving objects.

2.4 Resource Tree Authorization Policy

Although WAVE's design is agnostic to the specific mechanism used for expressing the authorization policy (i.e., it is compatible with existing policy languages such as [10, 12]), in our IoT deployments we use a simple yet widely applicable model: a resource tree (*RTree*) modelled roughly after SPKI's pkpfs tags [24].

An RTree policy manages permissions on a hierarchically organized set of resources. A resource is denoted by a URI pattern such as `company-entity/building/device` or `user-entity/albums/holiday/*`. The first element of a URI (e.g. `company-entity`) is called the *namespace authority* or just *namespace*, which specifies the entity who is the *root of authorization* for that resource (the entity who has permission on that policy without having received permission from someone else). The global authorization graph has many different RTrees with namespace authorities, ideally with one per intrinsic authority, e.g. homeowner or company. This lets the system be as decentralized as the naturally occurring authority structure, unlike the single-authority-per-vendor model, used in most systems today, which forces centralization. Depending on the structure of a given resource

hierarchy, there may be a minimum length for the resource URI. This often occurs where the first few elements are used to capture boundaries that exist naturally, such as a department, building or project. These elements that can be relied upon to exist, if present for a given RTree, are called the *resource prefix*. An RTree policy consists of:

- A set of permissions (strings such as "schema::read")
- A URI pattern describing a set of resources
- A time range describing when the grant is valid
- An *indirections* field, which limits re-delegation

For example, a building manager entity might grant `hvac::actuate` on `bldgnamespace/floor4/*` over a time range corresponding with the lease terms, allowing further delegation, to a tenant entity.

2.5 How WAVE Meets the Requirements

WAVE's global authorization graph, RDE, and storage layer allow it to achieve the requirements established in §1.3:

No reliance on central trust. WAVE achieves decentralization via three design features. First, the permission delegations are cryptographically enforced without a verifying authority. Secondly, any participant can create an RTree namespace, mimicking the natural ownership of resources. Finally, our Unequivocable Log Derived Map §5 allows participants to detect if the untrusted storage servers violate integrity. Although the storage server is centralized for availability, it is not a point of central trust as its behavior is cryptographically enforced.

Transitive Delegation. The graph-based authorization model efficiently captures transitive delegation. To delegate permissions, any entity can create an attestation that captures which subset of their permissions they wish to delegate. Since a proof is represented by a path through the graph, if an entity higher up in the delegation tree is revoked, all entities beneath it will no longer be able to prove they have permissions, even though the party revoking the entity may have been unaware of the delegations lower in the tree. This gives us the transitive delegation property.

Protected permissions. Through the Reverse-Discoverable Encryption scheme in §4, no party can decrypt attestations that are not potentially relevant to them. In our example, the property manager cannot decrypt attestations that the tenant makes, and the party running the storage servers cannot read any of the attestations.

Decentralized verification. WAVE proofs can be verified by anyone, unlike in SDSI/SPKI [49] or Macaroons [12]. This enables an IoT device to verify all messages it receives without communicating with an external service (with the exception of revocation checks, as detailed in §6.1).

No ordering constraints. An entity can grant any permissions at any time, including those that it has not yet received (although the recipient won't be able to form a proof yet). Consequently, attestations can be replaced anywhere in the hierarchy without requiring re-issue of subsequent delegations. Furthermore, our privacy mechanism preserves this property because an attestation can be encrypted under a policy-specific key before the issuer has been granted the permissions corresponding to the policy.

Offline participants. Attestations are disseminated through the ULDM storage tier (§5) which allows for entities to discover permissions they have been granted while they were offline and removes the need for any out-of-band online communication between entities.

3 Security Guarantees and Roadmap

WAVE must fulfill three security goals (§1.2). Regarding Security Goal #1, WAVE guarantees the following:

Guarantee 1. *An attacker* Adv *can form a proof of authorization on a policy if and only if the authority for that policy is compromised or has delegated access, directly or indirectly, to a compromised entity.*

This guarantee follows directly from the fact that each attestation is signed by its issuer. A WAVE proof can be thought of as a certificate chain. Given that existing systems like SDSI/SPKI [49] use a similar construction, we do not explore this further.

To achieve the other two security goals, WAVE introduces two new techniques: Reverse-Discoverable Encryption (§4) to satisfy Security Goal #2, and Unequivocable Log-Derived Maps (§5) to satisfy Security Goal #3. The following sections introduce these techniques and state formal security guarantees.

4 Encrypting Attestations

We encrypt attestations such that entities can decrypt attestations they can use in a valid proof. Entities cannot learn the policy (i.e., what permissions are granted) or the issuer (i.e., who created the attestation) of most other attestations. Our technique, *reverse-discoverable encryption* (RDE), does not require out-of-band communication between entities and works even if attestations are created out of order.

We present our solution incrementally: §4.1 formalizes the problem that RDE solves. §4.2 presents a simplified design of RDE, based on traditional public-key encryption, that provides a weak but useful security guarantee called "structural security." §4.3 augments the simplified RDE with *policy-aware* encryption to provide a significantly stronger notion of security, at the expense of making discoverability of attestations inefficient. §4.4 presents our final protocol, which provides both efficient discovery of attestations and a significantly stronger guarantee than structural security.

For all the security guarantees stated in this section, we assume that the attacker Adv is computationally-bounded, and that standard cryptographic assumptions hold.

4.1 Graph-based Formalization

We formalize the problem in terms of the global authorization graph; an example is shown in Fig. 2. For **correctness**, we require that each entity can decrypt all attestations that

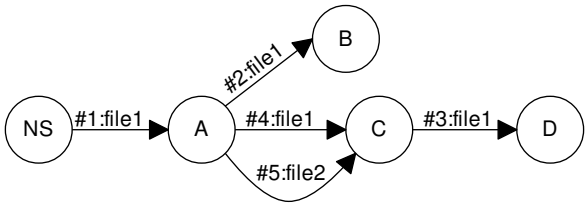


Figure 2: The number to the left of each colon indicates when the attestation was created. The string to the right denotes the resource on which it grants permission.

it can use to form a valid proof where it is the subject. In Fig. 2, entity D should be able to see attestations #1, #4, and #3. Correctness does *not* require D to be able to see attestation #2, as there is no path from B to D granting access to file1. Similarly, correctness does *not* require D to be able to see attestation #5, as there is no path from C to D granting access to file2. For **security**, we would like each entity to see as few additional attestations as possible.

4.2 Structural RDE

This section explains a simplified (yet weaker) version of RDE that is helpful to understand the main idea behind our technique. For this version alone, assume there are no revoked/expired attestations.

Each entity has an additional public-private keypair used only for encrypting/decrypting attestations, separate from the keys used to sign attestations. This keypair is governed by two rules: when an entity grants an attestation, it (1) attaches its private key to the attestation, and (2) encrypts the attestation, including the attached private key, using the public key of the attestation's subject (recipient). For example, in Fig. 2, #3 contains sk_C and is encrypted under pk_D (i.e., $\mathrm{Enc}(\mathrm{pk}_D; \#3||\mathrm{sk}_C)$).

This meets the correctness goal; D can decrypt #3 as #3 is encrypted under pk_D. In decrypting #3, it obtains sk_C, which it can use to decrypt #4. This works even though attestation #4 was issued after #3. In decrypting #4, it obtains sk_A, which it can use to decrypt #1. Essentially, each entity can see the attestations it can use in a proof by decrypting them in the reverse order as they would appear in a proof.

This achieves a simple security guarantee called **structural security**, which allows an entity e to see any attestation A for which there exists a path from $A.\mathsf{subject}$ to e. We call it "structural" security because only the structure of the graph, not the policies in attestations, affects whether A is visible to e. While structural RDE uses traditional public-key encryption, it differs from systems like PGP in that entities include their long-lived private keys in the attestations they encrypt.

4.3 Policy-Aware RDE

Structural security only takes into account the structure of the graph, not the policy of each attestation (i.e., the resources and the expiry). For example, structural RDE allows D to decrypt #5, though this is not necessary to meet the correctness goal; D cannot form a valid proof containing #5 because its policy differs from #4's (they delegate access to different files). With policy-aware RDE, we achieve a stronger notion of security that prevents D from decrypting #5 by making two high-level changes to structural RDE.

First, whereas structural RDE encrypts each attestation A according to only $A.\mathsf{subject}$, policy-aware RDE encrypts each attestation A according to both $A.\mathsf{subject}$ and $A.\mathsf{policy}$. Second, whereas structural RDE includes a key in A that can decrypt *all* attestations immediately upstream of A, policy-aware RDE includes a key in A that can only decrypt upstream attestations with *policies compatible with* $A.\mathsf{policy}$.

Choosing a suitable encryption scheme. Because the policy of an attestation determines how it is encrypted, the encryption scheme must be *policy-aware*. In particular, traditional public-key encryption is insufficient for policy-aware encryption (except for a boolean policy). We use the RTree policy type to explain our policy-aware RDE, although the technique applies to other policy types.

We identify Wildcard Identity-Based Encryption (WIBE) [5] as a suitable policy-aware encryption scheme to implement RDE for the RTree policy type. Typically, IBE [16] (or an IBE variant such as WIBE) is instantiated with a single centralized Private Key Generator (PKG) that issues private keys to all participants. This does not meet the goals of WAVE, because the PKG is a central trusted party. In RDE, however, our insight is to *instantiate a WIBE system for every entity, so there is no central PKG*.

A WIBE system consists of a master secret and public key pair $(\mathrm{WIBE.msk}, \mathrm{WIBE.mpk})$. A message m is encrypted using the master public key WIBE.mpk and a fixed-length vector of strings, called an ID: $\mathrm{WIBE.Enc}(\mathrm{WIBE.mpk}, \mathsf{ID}; m)$. Using msk, one can generate a secret key for a set of IDs. This set is expressed as an ID with some components replaced by wildcards, denoted ID^*. The secret key $\mathsf{sk}_{\mathsf{ID}^*}$ can decrypt an encrypted message, $\mathrm{WIBE.Enc}(\mathrm{WIBE.mpk}, \mathsf{ID}; m)$, if ID^* and ID match in all non-wildcard components.

Every policy p has an associated WIBE ID called a *partition*. The partition corresponding to policy p is denoted $P(p)$. When issuing an attestation A, an entity encrypts it using $P(A.\mathsf{policy})$, in the WIBE system of $A.\mathsf{subject}$: $\mathrm{WIBE.Enc}(\mathrm{WIBE.mpk}_{A.\mathsf{subject}}, P(A.\mathsf{policy}); A)$. Furthermore, the issuing entity generates secret keys in its own WIBE system, suitable to decrypt messages encrypted under $P(A.\mathsf{policy})$, and includes them in the attestation. Let $Q(A.\mathsf{policy}) = \{\mathsf{ID}^*_i\}_i$ represent the set of IDs suitable for decrypting attestations encrypted under $P(p)$ for p compatible with $A.\mathsf{policy}$, then A includes $W = \{\mathrm{WIBE.KeyGen}(\mathrm{WIBE.msk}_{\mathsf{Issuer}}; \mathsf{ID}^*_i)\}_{\mathsf{ID}^*_i \in Q(A.\mathsf{policy})}$. Below, we develop the *partition map* for RTree, which derives a partition from an RTree policy (i.e., functions P and Q).

Partition map for RTree. To define P, consider that an RTree policy consists of a resource prefix as defined in §2.4 (matching multiple resources) and a time range during which

the permission is valid. To express the start and end of this range as a WIBE ID, we define a time-partitioning tree of depth k over the entire supported time range; now any time in the supported time range can be represented as a *vector* representing a path in the tree from root to leaf. A WIBE ID is a length-n vector: to represent attestations with a certain time range, we choose k of those n components to encode the valid-after time, and another k components to encode the valid-before time. The remaining $n-2k$ components are used for the resource prefix. When granting an attestation for an RTree policy, the issuer encrypts the attestation contents under the resulting WIBE ID $= P(A.\mathsf{policy})$. Note that for a time tree of depth k, and a resource prefix of length ℓ, WIBE must be instantiated with at least $n = 2k+\ell$.

The issuer must also include the policy-specific WIBE keys from their own system in the attestations, generated with ID*s $Q(A.\mathsf{policy})$, so that upstream attestations with compatible policies can be discovered. We define Q for RTree as: let E be a set of subtrees, each represented as a *prefix* of a time vector (i.e., a vector where unused components are wildcards), that covers the time range from the earliest possible encryption start time to the end of the time range of the attestation's validity. Let S be a set of subtrees that covers the time range from the start of the attestation's time range to the latest possible encryption time. Attestations have a maximum validity of three years so this limits how long the start and end ranges need to be. Q returns ID*s corresponding to the Cartesian product $S \times E$ with each ID* also containing the policy's resource prefix. This allows any upstream attestation with an overlapping time range and compatible resource prefix to be decrypted by one of the secret keys in this attestation.

4.4 Efficient Discoverability

In the scheme above, attestations are encrypted under the partition in the subject's WIBE system. Unfortunately, it is subject to two major shortcomings. First, a WIBE ciphertext hides the message that was encrypted, but not the ID used to encrypt it; an attacker who guesses the ID of a ciphertext can efficiently verify that guess. Thus, every encrypted attestation leaks its partition. The second and more serious problem is that attestations are not efficiently discoverable. To understand this, suppose that Bob has issued many attestations $A_1,\ldots,A_n$ for Alice, with different policies. After this, an attestation B is granted to Bob. Alice might be able to form a proof using B and one of the A_i, but she does not know which of the A_i has a policy that intersects with $B.\mathsf{policy}$. As a result, she does not know which private key to use to decrypt B, and has to try *all* of the private keys conveyed by the A_i. This is infeasible if n is large, and becomes a vector for denial of service attacks.

If Alice knows B's partition, then the problem is solved—Alice can locally index the private keys she has from Bob's system, and efficiently look up a key that can decrypt B. However, B cannot include its own partition in plaintext, because it may leak part of $B.\mathsf{policy}$.

We solve this by encrypting the partition and storing it in the attestation. For this outer layer of encryption we use a more standard identity-based encryption (denoted IBE) that does not permit extracting the identity from the ciphertext [46, 42] because we do not need wildcards. As with the WIBE scheme, every entity has its own system, removing the centralized PKG. The ID used to encrypt the partition is called the *partition label*, and is denoted $L(A.\mathsf{policy})$. For the RTree policy type, it is the RTree namespace of $A.\mathsf{policy}$. We expect users to have far fewer unique keys for this outer layer, so they can feasibly try all the keys they have.

We also move the WIBE ciphertext under this IBE encryption so that the partition cannot be extracted. Finally, we include IBE keys from the issuer's IBE system, to allow the subject to discover the partition of upstream attestations. We denote the ID*s corresponding to these keys as $M(A.\mathsf{policy})$. Because the partition label is simpler in structure than the partition, defining $M(A.\mathsf{policy}) = \{L(A.\mathsf{policy})\}$ is sufficient. So far, what gets stored in the attestation is:

$$\begin{aligned}&\mathrm{IBE.Enc}(\mathrm{IBE.mpk}_{A.\mathsf{subject}}, L(A.\mathsf{policy}); P(A.\mathsf{policy})\,||\\ &\quad \mathrm{WIBE.Enc}(\mathrm{WIBE.mpk}_{A.\mathsf{subject}}, P(A.\mathsf{policy}); W\,||I))\end{aligned} \tag{1}$$

where W is defined as above, and

$$I = \mathrm{IBE.KeyGen}(\mathrm{IBE.msk}_{\mathsf{Issuer}}; L(A.\mathsf{policy}))$$

denotes the IBE secret key from the issuer's system.

4.5 Security Guarantees

We explain here at a high level how the policy-aware RDE restricts the visibility of attestations when used with RTree. Formal guarantees are given in Appendix B. In summary, for each attestation A granting permission on a namespace: entities who have not been granted permissions in that namespace in a path from $A.\mathsf{subject}$ can only see the subject and revocation commitment. Entities who have been granted some permissions in the namespace in a path from $A.\mathsf{subject}$ can see the partition (in essence the identifier of the key required to decrypt it). An entity e can decrypt an attestation A and use it in a proof if there exists a path, from $A.\mathsf{subject}$ to e where adjacent attestations (including A) have intersecting partitions. Issuers can encrypt under IDs before the corresponding private keys exist, so we introduce no ordering requirements and no interactivity requirements.

Thus, even though policy-aware RDE permits some entities to see more attestations than strictly needed to create a proof of authorization, it still provides a significant reduction in visibility when compared to structural security. We formalize the security guarantees of RDE in Appendix B.

A number of potential side channels are out of scope for WAVE, and can be addressed via complementary methods. Our storage layer does not provide any additional confidentiality, so compromised storage servers can see the time of each operation (e.g., when encrypted attestations are stored),

which encrypted attestations are fetched, as well as networking information of the packets arriving at the storage servers (which could be protected via Tor [1], a proxy, or other anonymous/secure messaging methods [21]).

Revocation. Although revoked attestations cannot be used in a proof due to the commitment revocation scheme described in §6.1, they still confer the ability to decrypt upstream attestations. Therefore we consider them part of the graph in the formal guarantees (Appendix B). This can be mitigated by keeping expiry times short and reissuing the attestations. As there are no ordering or interactivity requirements, short expiries are easy to implement. For example, if attestation `#1` in Fig. 2 were to expire and be reissued, it would not require the reissue of any other attestation.

Integrity. Finally, to maintain integrity, the issuer signs the attestation with a single-use ephemeral key $(\mathsf{pk}_e, \mathsf{sk}_e)$: $s_1 = \text{Sign}(\mathsf{sk}_e; A \setminus s_1)$, where $A \setminus s_1$ denotes the entire attestation except for s_1. Then, the issuer includes s_1 in the attestation in plaintext. The use of an ephemeral key ensures the signature does not reveal the issuer's public key. The issuer includes the outer signature in the plaintext header of the attestation. The issuer signs the ephemeral key pk_e with their entity private key, $s_2 = \text{Sign}(\mathsf{sk}_{\mathsf{Issuer}}; \mathsf{pk}_e))$, creating a short signature chain that ensures the attestation cannot be modified or forged. The issuer includes s_2 in the attestation *encrypted*, to avoid revealing the issuer's public key. In forming a proof, the verifier is allowed to decrypt s_2, allowing the verifier to verify s_2 and then s_1.

4.6 Reducing Leakage in Proofs

The methods discussed above ensure that a prover is able to decrypt all the attestations that it requires to build a proof. However, if a participant simply assembles a list of decrypted attestations into a proof and gives those attestations to a verifier, the verifier learns not only the attestations in that proof, but also the WIBE keys in those attestations, which it can use to decrypt other attestations not in the proof. To solve this, we split the attestation information into two *compartments*, one for the prover (that includes keys it needs to decrypt other attestations) and one for both the prover and the verifier (that includes the policy, issuer, expiry, etc.). We encrypt the prover compartment with k_{prover} and the prover/verifier compartment with k_{verifier}, both symmetric keys freshly sampled for each attestation. k_{prover} and k_{verifier} are encrypted with WIBE. This allows the prover to reveal to the verifier the necessary parts of an attestation by sending it the AES verifier key, without allowing the verifier to decrypt other attestations. The final structure of the attestation is in Fig. 3.

4.7 Discovering an Attestation

Each user's WAVE client maintains a *perspective subgraph* with respect to the user's entity, which is the portion of the global authorization graph visible to it. For each vertex (entity) in the perspective subgraph, the client "listens" for new attestations whose subject is that vertex (entity), using the

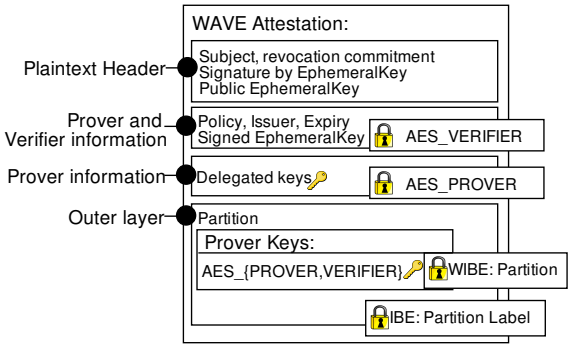


Figure 3: Encrypted WAVE attestation structure. The locks indicate the key used to encrypt the content.

Get and IterQueue API calls to the storage layer. For every attestation A received, the WAVE client does the following:

1. The client adds edge A to the perspective subgraph.
2. The client searches its local index for IBE keys received via attestations from A.subject, and tries to decrypt A's outer layer using each key. If none of the keys work, it marks A as **interesting** and stops processing it.
3. Having decrypted the outer layer in the previous step, the client can see A.partition. It searches its index for a WIBE key received via attestations from A.subject that are at least as general as A.partition. Unlike the previous step, this lookup is indexed. If the client does not have a suitable key, it marks A as **partition-known** and stops processing A.
4. Having completed the previous step, the client marks A as **useful** and can now see all fields in A. The client adds WIBE and IBE keys delegated via A to its index, as keys in the systems of A.issuer.
5. If the vertex A.issuer is not part of the perspective subgraph, then the client adds it and requests the storage layer for all attestations whose subject is A.issuer. They are processed by recursively invoking this algorithm, starting at Step 1 above.
6. If A.issuer is already in the perspective subgraph:
   - For each IBE key included in A, the client searches its local index for **interesting** attestations whose subject is A.issuer, and processes them starting at Step 2 above.
   - For each WIBE key, the client searches its local index for matching **partition-known** attestations whose subject is A.issuer, and processes them starting at Step 3.

This constitutes a depth-first traversal to discover newly visible parts of the authorization graph revealed by A.

4.8 Extensions

Our RDE construction for RTree is performant but allows an entity to see attestations not required for correctness (i.e. partition-compatible attestations that are not usable in a proof, as defined in Appendix B). This can be marginally improved by including an additional set of WIBE keys in the attestations to allow for the full resource (not just the prefix) to be captured by P and Q but this increases the number

of included keys by a factor of ℓ. Additionally, using KP-ABE [35] instead of WIBE would result in smaller attestations, but higher decryption times.

Aside from different encryption schemes, the RDE technique also generalizes beyond the RTree policy described above. Careful selection of P and Q, coupled with the use of a more expressive encryption scheme such as KP-ABE [35] allows for the realization of a more expressive policy (e.g. those discussed in §9) at the cost of decreased performance. While we have not found this trade-off warranted in our setting, this extension is straightforward and still meets our security goals. The formalism in Appendix B largely generalizes to other policy types, but the semantics of compatibility (Note 1) will change depending on the encryption schemes used and on the choice of P, Q, L, and M.

5 Scalable Untrusted Storage

To avoid centralized trust when storing attestations, we contribute a storage tier that enforces integrity cryptographically. This tier is physically decentralized: it is spread over multiple servers owned by different parties. Importantly, these individual servers are trusted to maintain availability, but not integrity (in the spirit of Certificate Transparency [41]) or privacy (achieved by RDE, §4). Thus, users and services can interact with storage servers that anybody operates, without trusting the servers' operators, except for availability.

The storage API (Fig. 1b) consists of four functions: Get and Put are used for placing/retrieving entities, attestations, name declarations (§6.2) and revocation secrets (§6.1) in storage; Enqueue places an object hash at the end of a named queue, and IterQueue allows retrieval from a queue. The queue functions facilitate discovery, allowing an entity to notify another entity that a new attestation has been granted to them or a new name declaration has been published.

A blockchain is a natural candidate for such a storage tier. Multiple servers are responsible for maintaining a blockchain, and, due to the underlying Merkle tree data structure, any one server can prove the integrity of its responses to state queries according to a specific Merkle tree root hash, meeting the requirements.

Prior versions of WAVE used an Ethereum blockchain, but extended use and experimentation revealed this solution to be inadequate for three reasons: (1) A blockchain introduces significant latency when adding objects to storage (up to a minute for a confirmed addition in Ethereum). (2) Participating in a blockchain requires constant network bandwidth and CPU time. (3) The blockchain does not scale past a few tens of transactions per second [22], so it could not store attestations for a global authorization system permitting thousands of delegations per second.

Although this problem appears solvable with existing transparency logs such as Certificate Transparency (CT) [41] or Key Transparency (KT) [32], neither of those is appropriate. CT cannot efficiently prove an object does not exist, needed for revocations, and KT is not efficiently auditable in our context (§9).

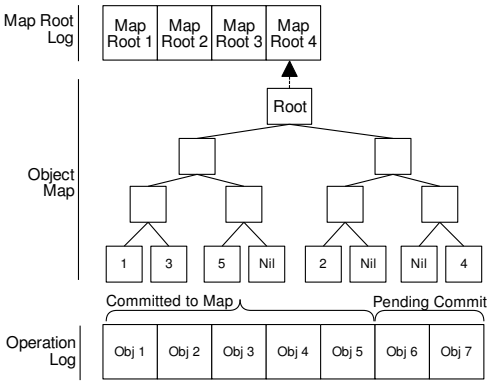


Figure 4: An Unequivocable Log Derived Map (ULDM) built from two Merkle tree logs and a Merkle tree map

Instead, we propose an *Unequivocable Log Derived Map (ULDM)*, a transparency log based on the Verifiable Log Backed Map (VLBM) [23]. A VLBM allows the storage server to form proofs of integrity. The VLBM whitepaper is brief and incomplete: it does not discuss auditing, such as which proofs are exchanged or how they are published, so it is unclear how the VLBM prevents equivocation (i.e., presenting different internally consistent views to different clients). To our knowledge, there is no complete open-source VLBM implementation (the code in the repository [34] only implements a subset of the paper, omitting the log of map roots), so we could not build upon the VLBM or infer its scheme from the code. The ULDM is our approach to filling in the missing pieces, such as an auditing scheme to prevent equivocation and secure batching to increase performance.

A ULDM is constructed using *three* Merkle trees, each serving a different purpose, as shown in Fig. 4. The first tree is the Operation Log, which stores every Put and Enqueue operation and can prove the log is append-only. These operations are then processed in batches into the second tree, the Object Map. This is used to satisfy queries and prove that objects exist or do not exist within the map. The ULDM Object Map is different from [23] as it only stores the hashes of the objects. Finally, every map root created when a batch is processed is inserted into the third Merkle tree, the Map Root Log. This makes the data structure efficiently auditable, as we discuss in §5.4.

In what follows, for every reply that the storage server provides, the storage server provides a signature on the reply along with the relevant version of the Map Root Log.

5.1 Inserting Values

To insert a value, the ULDM server: (1) Inserts the value into the Operation Log. (2) Creates a new version of the Object Map that includes the hashes of the new entries. (3) Inserts the new map root into the Map Root Log. Step 1 is batched (multiple values are inserted into the Operation Log together) as is Step 2 (multiple values are inserted into the

Object Map together). Step 3 is synchronous with Step 2.

5.2 Merge Promises

Inserts would ideally be performed synchronously, allowing the server to return inclusion proofs for all three trees in response to the insert. Unfortunately, this results in a severe performance penalty as the ratio of new data to overhead (internal nodes in the trees) is poor. This is the same conclusion that Certificate Transparency reaches, and we use a similar solution: batching with promises. When inserting a value, a client receives a *merge promise* (called Signed Certificate Timestamp in CT) which states that the inserted value will be present by a certain point in time. In addition to the absolute timestamp used in CT, ULDM merge promises include the version of the root log as this allows a proof of misbehavior without a trusted source of time. Uncompromised clients must check the value has been merged later. To prove misbehavior when a value is not inserted on time, a client can present a merge promise along with a signed Map Root Log head where the corresponding Object Map does not contain the value and where the version of the head is greater than that in the promise; i.e., a server would need to stop operating completely if it wishes to both avoid merging an object and revealing it is compromised.

5.3 Retrieving Values

To retrieve a value, the client sends the storage server the Map Root Log version that it received in a previous request, along with the object identifier it is retrieving (e.g., the hash of an attestation or revocation commitment). If the object exists but has not yet been merged, the merge promise will be returned. There is no guarantee that the storage server will return a value before its merge promise deadline. If the object has been merged or doesn't exist, the server responds with: (1) the object or nil, (2) a proof that the object existed or did not exist in the Object Map at the latest map root, (3) a proof that the latest map root exists in the Map Root Log at the current Map Root Log head, and (4) a consistency proof that the current Map Root Log head is an append-only extension of the version the client passed in its request. This mechanism allows the client to verify that every map satisfying their queries is contained in the Map Root Log, and that the Map Root Log is consistent. Notably, it does not allow the client to verify that the map was correctly derived from the Operation Log. This task is performed by the auditors.

5.4 Auditing

An auditor is a party that connects to a storage server and replays the Operation Log to construct replicas of the Object Map and check the Map Root Log. Each client reports the latest Map Root Log head it obtains from the server (signed by the server along with a version number) to the auditors with some frequency. As the entries in the ULDM object map are the hashes of the objects, not the objects themselves, the map constructed by the auditor is several orders of magnitude smaller than the sum of stored objects. For every entry in the Map Root Log, the auditor will read the incremental additions to the map from the Operation Log and apply them to its own copy. It then ensures the hash of the replica Object Map root matches the hash stored in the Map Root Log, proving that the map is correctly derived from the operation log (no objects were modified or removed).

The strength of the ULDM auditing scheme is that a client can report a single value to an auditor (the client's Map Root Log head) and this is sufficient to catch any dishonesty that might have occurred at any point in the client's history. Without the Map Root Log (such as in [34]), any auditing scheme would need to make the client report every Object Map root to the auditor or take the risk that some dishonesty might remain undiscovered. To see how this might occur, imagine that a storage server removes a revocation from the map, answers a query and then re-adds the revocation. Without the Map Root Log, if the client only reports the final map root to an auditor, it would conclude it is valid. In the ULDM case, the client would report the Map Root Log head which covers all prior map versions, enabling the auditor to discover that the previous query was satisfied from an invalid map.

Detecting dishonesty with a single infrequently-reported value has important scalability implications: as we expect there to be many clients, it is important that the load placed on auditors is much less than the query load generated by the clients, otherwise, only large companies could afford to be auditors. In the ULDM model, it is sufficient for a client to contact an auditor rarely (perhaps once a day) to ensure any prior equivocation is discovered.

We expect clients to periodically check in with a random auditor from a public list of auditors. This ensures that the storage server cannot maintain different states for different auditors as it will be discovered when auditor receives a Map Root Log head from a client that is inconsistent with the one received from the storage server directly.

5.5 Security Guarantee

We formalize the security guarantee of a ULDM, as follows. By honest client, we denote a client that is neither faulty nor compromised.

Guarantee 2 (ULDM)**.** *Let C be a set of honest clients and S be a ULDM server. Observe that the Merge Promises following insert requests by these clients and Map Root Log heads sent with retrieval requests by these clients define a partial ordering L over all requests received by S. Suppose that there exists a nonempty set R of requests made by clients in C, such that there exists no possible history of requests made to S that is consistent with both L and all of S's responses to requests in R.* ***If*** *there exists an auditor A such that each client in C has sent A a Map Root Log head it received from S* at least as recent *as the one it received for its latest request in R,* ***then*** *one of the following holds:*

1. *One or more clients in C will be able to detect the in-*

consistency by inspecting the responses it received to requests that it made to S.

2. *The auditor A will be able to detect the inconsistency by inspecting the Map Root Log heads it received from clients in C and from S.*

We provide a proof sketch in Appendix A.

6 Revocation and Naming

With the functionality of RDE and ULDM's, we can easily construct a revocation scheme and a PKI-replacing entity naming scheme.

6.1 Commitment-Based Revocation

When a user creates an attestation, it derives a random revocation secret s from a seed stored with the entity private keys and includes a cryptographic hash of s, $\mathsf{hash}(s)$, called the *revocation commitment*, in the attestation. The user then inserts the attestation into ULDM storage. Later on, the user can revoke the attestation by publishing the revocation secret s to the same storage. Revocation of entities works similarly. An entity must have their private key to perform revocation; mechanisms such as [53] can be used to ensure this.

When verifying a proof, the WAVE service ensures that no attestations in the proof have been revoked. To do so, it queries the storage tier for an object matching the revocation commitment $\mathsf{hash}(s)$ in the attestation. If such an object exists, the verifier knows that the attestation has been revoked. If such an object does not exist, the verifier receives a *proof of nonexistence* for that hash from the storage tier. WAVE ensures revocation only after the Merge promise deadline. The security of this procedure relies on the guarantees of our ULDM transparency log (§5). Alternatively, the entity forming the proof can include proofs of nonexistence, signed by the storage tier with a timestamp, with the attestations, so that the verifier does not have to perform this lookup.

6.2 Secure Lookup of Public Keys

To facilitate looking up entity public keys (to be used as the subject in an attestation, and for RDE), without relying on an external PKI, WAVE implements a naming scheme that extends the proposal in SDSI [49]. The base functionality (shared by WAVE and SDSI) allows an entity to name another entity by creating a signed *name declaration*. These name declarations form a web-of-trust global graph, similar to that formed by attestations. By traversing this graph, an entity can resolve hierarchical names. For example, consider when an entity representing a company `ACME` names an entity representing a department `Marketing`, which in turn names an entity held by an employee `Alice`. Then, by verifying the identity of a single entity out of band (the company), an entity can resolve the names of all employees within the company's departments, such as `Alice.Marketing.ACME`, without having to manually establish the validity of individual employee entities.

The functionality above, proposed by SDSI, does not provide a distribution mechanism for entities to discover the name declarations required to perform resolution, nor a mechanism to ensure the privacy of declarations so that only authorized parties may read them. WAVE solves both of these problems. Firstly, WAVE stores name declarations in the ULDM storage tier (§5) to ensure name declarations are discoverable without compromising on the goals of the system (especially without requiring on-line participants). Secondly, WAVE uses a variation of the encryption scheme described in §4 to encrypt the name declarations in storage. When creating a name declaration, it is associated with a resource in a namespace (for example, `acme/directory/marketing`) and an entity must be explicitly granted permission on that resource in order to gain the keys required to decrypt the name declaration. In other words, the same attestations that are used to form a proof of authorization are also used to govern which entities can read name declarations, without relying on a central directory server. Resolution of names is done from each entity's cache of decrypted name declarations, stored alongside decrypted attestations.

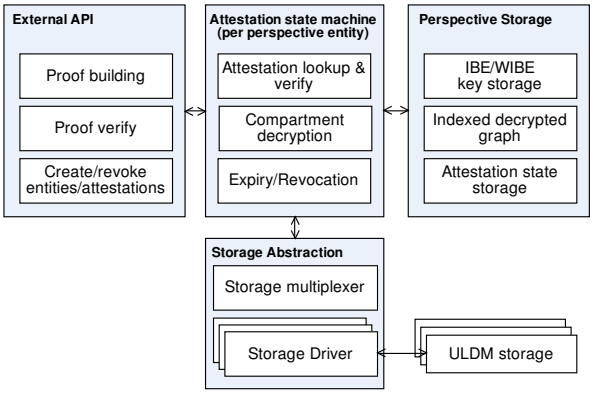


Figure 5: Overview of WAVE's implementation.

7 Implementation

WAVE is implemented in Go and released as open source [7]. It runs as a background service and applications connect via IPC. The service is composed of four logical parts (Fig. 5).

The storage abstraction permits multiple distinct storage providers operating in parallel. As long as the provider implements the API discussed in §5, WAVE can use it. Each storage *driver* is responsible for ensuring the storage is trustworthy, e.g. for a ULDM-based storage it must verify the proofs given by the remote storage server. Attestations can span storage media, i.e., an entity residing on one server can grant permissions to an entity on a different server. We implemented the ULDMs using Merkle trees in Trillian [33] backed by MySQL.

The perspective storage keeps track of the decrypted attestations that form the *perspective graph*. This is the portion of the global graph visible from the perspective of the proving entity. WAVE indexes it to allow efficient key retrieval

Operation	AMD64	ARMv8
Create attestation[1]	43.7	445
Create entity	8.9	88.5
Decrypt attestation as verifier	0.48	4.44
Decrypt attestation as subject	3.87	44.0
Decrypt delegated attestation	6.22	67.9

Table 1: Object operation times [ms].

based on a new attestation and efficient attestation retrieval based on a new key. The index also allows for efficient proof building: finding attestations granted from a given issuer that match specific permissions.

The state machine is responsible for transitioning the attestation through the states of decryption following the discovery process described in §4.7.

The external API is a GRPC [31] API that listens for connections from applications and allows them to use the application API functions given in Fig. 1b. GRPC can generate bindings for multiple languages, so we expect that applications can be written in any language.

The proof builder, when asked to build a proof, begins at the namespace authority (the entity that created the RTree namespace) for the resource that permissions are being proved on, and then performs a shortest path discovery through the perspective graph terminating at the proving entity. Note that this is the opposite direction that attestations are traversed during discovery. Only edges granting a superset of the required permissions are traversed and the maximum depth of traversal is limited by the indirections parameter in the traversed attestations. These two filters make proof building fast for common cases (see §8.1).

8 Evaluation

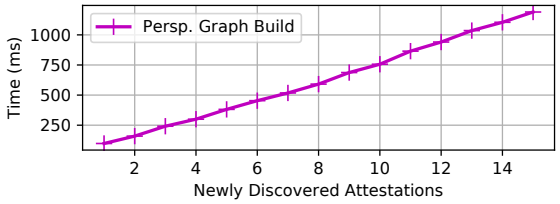
Despite relying on cryptography for its security guarantees, WAVE remains performant, competitive to traditional authentication and authorization systems.

8.1 Microbenchmarks

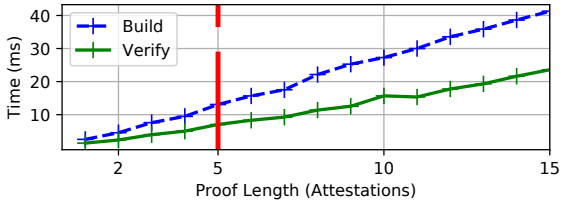
WAVE's performance is dominated by the cost of the core cryptographic operations, shown in Table 1. These are the times measured by a client using the GRPC application API. The measurement is on an Intel i7-8650U AMD64 CPU representative of a standard modern laptop, and on a Raspberry Pi 3, indicative of a low-cost IoT-class ARMv8 platform.

The verifier does not perform any WIBE decryption, as it has the AES verifier key. The subject entity (the direct recipient of the attestation) can skip the IBE decryption of the partition, but must still perform WIBE decryption. Any other entity that is interested in the attestation because it lies further up the delegation chain must perform IBE decryption, WIBE decryption, and then AES decryption. These decryption operations take place only once—when an attestation is added to the perspective graph—so are a one-off cost of receiving permissions. The verifier decryption happens once per unique proof; after that, it is cached so that subsequent verifications complete in negligible time.

[1] Create attestation uses multiple cores



(a) Perspective graph update/build time



(b) Proof build/verification time

Figure 6: Single core timings for proof operations. Vertical line in Fig. 6b is the expected maximum proof length for common applications.

System	Authentication	Authorization
LDAP+MySQL	6.3ms	0.8ms
OAuth2 JWT	0.3ms	
WAVE 1 attest.	1.2ms	
WAVE 3 attest.	3.6ms	

Table 2: Latency of LDAP+MySQL, OAuth2 vs. WAVE.

When decrypting attestations and building the perspective graph, we also need to index all the obtained keys and store them on disk. We can see the cost of decryption combined with indexing by measuring the time taken to update a perspective graph, for different sizes of changes to the graph, as shown in Fig. 6a. This includes the time taken to retrieve the encrypted ciphertexts from ULDM-based storage. The dashed vertical line is likely the maximum number of attestations that will be found in a proof as more than five delegations, although supported, is rare in all our deployments.

8.2 Traditional Authorization Flow

To compare WAVE against a traditional authorization system, we benchmark the time taken by a representative backend to turn a username and password into an authorization policy using an OpenLDAP server (which authenticates the user and yields the groups they are part of) and a MySQL database (which turns the groups into policy). We also add the time taken to verify an OAuth2 JWT token containing the authorization policy in the form of scopes.

The results are shown in Table 2. For a WAVE proof mirroring the single-delegation structure present in the LDAP/OAuth2 case, the proof verifies in a sixth of the time taken by the traditional LDAP flow. For a case where transitive delegation has been used three times and the proof con-

	PUT 2KB	GET 2KB	En-Queue	Iter-Queue
Latency [ms]	10.7	10.4	10.1	10.0

Table 3: Average storage operation time (ms/op) under 4 uniform loads ($\approx$ 100 requests per second), measured over 30 seconds ($\approx$ 3k requests per type).

sists of three attestations, the WAVE verification is about half the time of the single-delegation LDAP flow.

As in WAVE, OAuth2 offers a bearer token that can be validated without communicating with the server. In this case, validating a JSON Web Token with a 2048-bit RSA signature takes 0.3ms. WAVE is roughly 4x slower, but completely removes the centralized token-issuing server, leaving the user as the only authority in the system. In OAuth a compromised token issuing server can generate valid tokens without the user's knowledge.

Note that although OAuth2 has added a form of delegation [36], it requires the OAuth2 server to issue a new token, so is identical to the single-delegation scenario tested here.

This example shows that using WAVE as a replacement for common authorization flows will likely not reduce performance, despite providing transitive delegation and removing all central authorities.

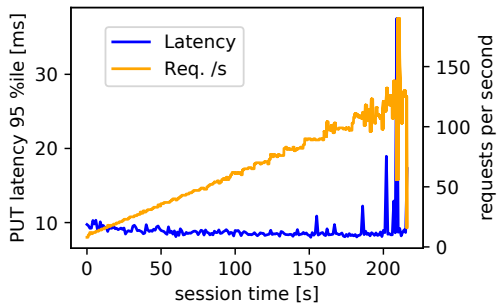
8.3 Storage Evaluation

Since an entity in WAVE does not communicate with any other entity, except via the storage, WAVE's scalability depends on the performance of the global storage. As mentioned in §5, a blockchain is a natural solution, but not scalable enough.

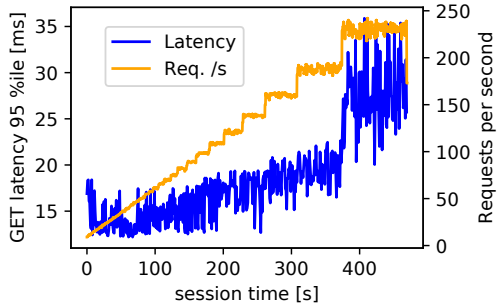
In contrast, the ULDM-based system is shared-nothing and horizontally scalable: the performance of one node does not limit the performance of the overall system. For completeness, we include single-system performance metrics here. Table 3 shows the average latency of the ULDM storage performing single operations at a time (i.e. just GETs or just PUTs). The times for the ULDM-based storage include both the generation of the proofs server-side and the verification of the proofs client-side. Every operation concerns a unique object, so there is no caching.

This ULDM storage was constructed using Trillian backed by MySQL. Fig. 7 shows the limits of a single node, where performance for PUTs degrades at approximately 110 requests per second and performance for GETs degrades at approximately 200 requests per second. We expect that performance could be increased if Trillian were deployed on Spanner [20] as the designers intended, but defer this to future work. Note that in this evaluation, every operation concerns a unique object, so as to benchmark the underlying cost of forming proofs, rather than the cache. Real workloads would likely have more cache hits.

Although our storage implementation is unoptimized and built using an off-the-shelf Merkle tree database, single nodes handle insert loads an order of magnitude higher than



(a) 95th %ile PUT



(b) 95th %ile GET

Figure 7: Latencies for ULDM PUT/GET as the throughput is ramped up to the single-node maximum.

possible on a blockchain system [22]. In addition, every added node scales the capacity of the system linearly. We envision that multiple storage providers, potentially operated by distinct parties, would operate in parallel, similar to Certificate Transparency [41].

8.4 Deployment Experiences

WAVE is a real-world artifact and is open source [7]. We operated various versions of WAVE for roughly two years in over 20 buildings, controlling more than 800 devices (thermostats, control processes, motion sensors, and others with little to no existing authorization capabilities) comprising 363 entities, 27 namespaces and 529 attestations (both valid and expired). The global authorization graph in our deployment is visualized in Fig. 8. The median number of delegations in a path is 4 (the maximum is 9). This deployment has given us the opportunity to refine and validate the performance, usability, and expressiveness of WAVE's authorization model in practice. Applying WAVE to legacy devices whose firmware cannot be modified is done by using an adaptation layer microservice and ensuring all communication with the legacy device flows through that service [8].

Performance. In the deployment, most proofs build in under 20ms and validate in under 10ms (as in Fig. 6b). The performance impact of WAVE is imperceptible during normal operation: proofs are cached after processing, accelerating subsequent generation and validation. As mentioned, we built an earlier version of WAVE on top of a blockchain instead of our current ULDM. We conducted extensive bench-

Work	Transitive delegation	Discoverability	No order constraints	Offline participants	No trusted central storage	Protected permissions
Auth. languages [12, 49, 13, 43, 45, 29, 27]	Yes	**No**	**Unknown: no mechanism given**			
Hidden credentials [58, 37, 30, 48]	Yes	**No**	**Unknown: no mechanism given**			
Centralized authorization [19, 15, 56, 28]	Yes	Yes	Yes	Yes	**No**	**No**
Distributed authorization [44, 59, 18, 57, 50]	Yes	Yes	Yes	**No**	Yes	**No**
WAVE	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**

Table 4: Related work on decentralized authorization compared to WAVE. We elaborate on these categories in §9.

marks of that version and concluded that it cannot scale past a load roughly equivalent to a city ($\approx$ 1 million buildings). It also incurs significant CPU and bandwidth costs, even when only storing permissions (not data).

Usability. In addition to our experience with the deployment, we have also held multiple tutorials with 200+ users. User feedback indicated that WAVE improved most aspects of management (especially administrators having autonomy to grant and revoke permissions). Some aspects of WAVE are harder to manage: no user can enumerate all delegations in the system, which reduces auditability. We were able to mitigate unfamiliarity with WAVE's authorization model with careful user interface design (which provides secure defaults such as short expiry times) and with teaching users through familiar analogies (e.g., comparing RTree to file paths).

Expressiveness. We found that WAVE was able to capture exactly the authorization patterns required in typical cyber-physical usage scenarios. The transitive delegation capability was invaluable in lowering the administrative overhead of deployments. Rather than requiring the building manager to be a part of every commissioning workflow (to create credentials for each new device), permission is granted to the person heading the deployment effort, who then acts with autonomy. For permanent installations, the installing entity can be removed from the permission flow afterwards by granting "around" them directly from the building manager to the devices. For temporary installations, keeping the installing entity in the flow simplifies revocation when the study is over.



Figure 8: The permission graph for the multi-building deployment. "Bolded" nodes are namespace authorities. Most nodes with a high degree are entities that administer a set of namespaces. Leaf nodes correspond to devices and services that do not perform any delegation.

9 Related Work

Table 4, compares prior authorization and trust management systems with WAVE. Here, we provide additional details.

9.1 Trust Management and Authorization

Trust Management (TM) literature over the past two decades has thoroughly researched techniques for transitively delegable authorization. Overviews of TM systems are provided in [14, 51, 11, 6].

Languages used to express authorization policies are summarized in the first row of Table 4 [12, 49, 13, 43, 10, 27]. For example Macaroons [12] provides a mechanism for expressing authorization policy with delegation and context-specific third-party caveats. The goals are quite different, e.g. the authorization is verifiable by the authority only and permissions can only be granted in-order. The system does not specify how cookies are stored and discovered or how it would work with offline participants. In general, authorization language work is complementary to WAVE, as we focus on the layers of the system that lie below the language (how the pieces of policy are stored, disseminated, and discovered). In our deployments we use RTree, based on SPKI's pkpfs [24], but mechanisms like third-party caveats could be introduced with no changes to the underlying layers.

Hidden credentials (row 2 in Table 4) [58, 37, 30, 48] address a different privacy problem: allowing a prover and verifier to hide their credentials from each other. WAVE solves an orthogonal problem: the privacy of credentials in storage and during discovery.

The remaining literature can be categorized as relying on a centralized credential store for discovery [19, 15, 56], or a distributed credential store [44, 59, 18, 57, 50]. Centralized discovery mechanisms put all credentials in *one place* which makes discovery simple but, as constructed in work thus far, requires this central storage to be trusted. Blockchain work [55, 26] avoids this problem but does not scale, and

thus far has focused on identity, not authorization. Work such as [28] decreases centralization by reducing the trust in cross-administrative-domain applications, such as IFTTT, but still places trust in the central authorization servers belonging to each vendor. In contrast, distributed discovery mechanisms store each credential with its *issuer* and/or *subject*, avoiding the need to trust a central storage system. The resulting discovery mechanisms are more complex and cannot operate if any credential holder is offline. Both the centralized and decentralized credential discovery work thus far have overlooked the privacy of credentials at rest (in the centralized case) or during discovery (in the distributed case); in both cases, there are parties who can read credentials that do not grant them permissions even indirectly.

A concurrent work, Droplet [52], presents a distributed authorization system, but it does not meet the requirements of a general purpose authorization system in §1: Droplet does not provide transitive delegation, it only handles authorization for time series data streams as opposed to the more general policies of WAVE, and it induces a blockchain transaction for every change to an ACL, which scales poorly.

WAVEs attestations and RDE can be used as the key exchange protocol for an end-to-end encryption scheme such as JEDI [38]. JEDI provides resource-oriented message encryption on a tree of resources, which interfaces well with WAVEs RTree authorization policy.

9.2 Storage

WAVE's Map Log Root is similar to the approach used by CONIKS [47] and Key Transparency (KT) [32]. There are several differences between a ULDM and the CONIKS/KT data structures. As a ULDM does not need to prevent iteration of the contents, it can be log derived, allowing an efficient verification that it is append-only. In contrast, CONIKS/KT requires every user to check every epoch of the map to ensure the values stored match expectations. This approach would not work for our use case as we expect every user to create hundreds or thousands of objects, and requiring every user to check each of these objects at every map epoch is intractable. The ULDM approach 1) reduces the amount of work as it scales with the number of *additions* to the map rather than the *size* of the map, as in CONIKS, and 2) places the majority of the burden on auditors, rather than users who may be offline.

Revocation Transparency [40] is also similar to a ULDM. It was posted as an informal short note, and to our knowledge, it was never fully developed. It lacks the Operation Log, which requires the client/auditor to request a consistency proof between two versions of the map without knowing the contents (as it cannot construct a replica). We are not aware of any Merkle tree map databases that support this operation. A ULDM is built on simpler operations and can be constructed using an off-the-shelf database, such as Trillian [33], with full auditability.

10 Conclusion

WAVE is a *decentralized* authorization framework leveraging an improved graph-based authorization model. It introduces an encryption technique, RDE, for hiding attestation contents, while still allowing efficient discovery of permissions granted out of order to offline participants. WAVE introduces a storage mechanism, the ULDM, that is efficiently auditable. This enables untrusted, horizontally scalable, servers to store the attestations without compromising on the security of the system as a whole.

We used WAVE to manage IoT deployments in 20 buildings for two years, during which we identified six requirements that are critical for IoT deployments. In meeting these requirements, WAVE (1) has no reliance on central trust, (2) provides transitive fine-grained delegation and revocation, (3) protects permissions during discovery and at rest, (4) allows for any party to verify a proof of authorization, (5) allows delegations to occur in any order with no communication between granter and receiver, and finally (6) allows for granting permissions to offline participants. No existing work meets these requirements simultaneously. Our open-source implementation of WAVE offers similar performance to traditional centralized systems while providing stronger security guarantees.

Acknowledgements

We thank our anonymous reviewers and our shepherd for their invaluable feedback. This research was supported by Intel/NSF CPS-Security #1505773 and #20153754, DoE #DE-EE000768, NSF CISE Expeditions #CCF-1730628, NSF GRFP #DGE-1752814, and gifts from the Sloan Foundation, Hellman Fellows Fund, Alibaba, Amazon, Ant Financial, Arm, Capital One, Ericsson, Facebook, Google, Intel, Microsoft, Scotiabank, Splunk and VMware.

References

[1] Tor project: Anonymity online. `https://www.torproject.org/`.

[2] Facebook permission bug. `https://money.cnn.com/2018/06/07/technology/facebook-public-post-error/index.html`, 2018.

[3] If This Then That. `https://ifttt.com/`, 2018.

[4] OAuth 2.0. `https://oauth.net/2/`, 2018.

[5] Michel Abdalla et al. Identity-based encryption gone wild. In *ICALP*, 2006.

[6] A Ahadipour and M Schanzenbach. A survey on authorization in distributed systems: Information storage, data retrieval and trust evaluation. In *Trustcom*, 2017.

[7] Michael Andersen and Sam Kumar. Source for WAVE. `https://github.com/immesys/wave`.

[8] Michael P Andersen, John Kolb, Kaifei Chen, Gabe Fierro, David E Culler, and Randy Katz. Democratizing authority in the built environment. *TOSN*, 2018.

[9] Michael P Andersen, John Kolb, Kaifei Chen, Gabriel Fierro, David E Culler, and Raluca Ada Popa. WAVE: A decentralized authorization system for IoT via blockchain smart contracts. *UC Berkeley Tech. Rep. UCB/EECS-2017-234*, 2017.

[10] Moritz Becker et al. SecPAL: Design and semantics of a decentralized authorization language. *JCS*, 2010.

[11] Elisa Bertino, Elena Ferrari, and Anna Squicciarini. Trust negotiations: concepts, systems, and languages. *Computing in science & engineering*, 6(4), 2004.

[12] Arnar Birgisson, Joe Gibbs Politz, Ulfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *NDSS*, 2014.

[13] Matt Blaze et al. Keynote: Trust management for public-key infrastructures. In *SWP*, 1998.

[14] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *IEEE S & P*, 1996.

[15] Matt Blaze, Joan Feigenbaum, and Martin Strauss. Compliance checking in the policymaker trust management system. In *FC*, 1998.

[16] D. Boneh and M. Franklin. Identity-based encryption from the weil pairing. In *SIAM J Comput*, 2003.

[17] Christian Cachin. Architecture of the hyperledger blockchain fabric. 2016.

[18] Ke Chen, Kai Hwang, and Gang Chen. Heuristic discovery of role-based trust chains in peer-to-peer networks. *IEEE TPDS*, 20(1):83–96, 2009.

[19] Dwaine Clarke et al. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 2001.

[20] James C Corbett et al. Spanner: Google's globally distributed database. *ACM TOCS*, 31(3):8, 2013.

[21] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *IEEE S&P*, 2015.

[22] Kyle Croman et al. On scaling decentralized blockchains. In *FC*, 2016.

[23] Adam Eijdenberg, Ben Laurie, and Al Cutter. Verifiable data structures. `https://github.com/google/trillian/blob/master/docs/VerifiableDataStructures.pdf`.

[24] Carl M Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M Thomas, and Tatu Ylonen. SPKI examples, 1998.

[25] Ksenia Ermoshina, Francesca Musiani, and Harry Halpin. End-to-end encrypted messaging protocols: An overview. In *INRIA*, 2017.

[26] Evernym Inc. Everynm: Self-sovereign identity with verifiable claims, 2018.

[27] A. Felkner and A. Kozakiewicz. Practical extensions of trust management credentials. In *iNetSApp*. 2017.

[28] Earlence Fernandes, Amir Rahmati, Jaeyeon Jung, and Atul Prakash. Decentralized action integrity for trigger-action IoT platforms. In *NDSS*, 2018.

[29] Philip WL Fong. Relationship-based access control: protection model and policy language. In *CODASPY*, 2011.

[30] Keith Frikken et al. Attribute-based access control with hidden policies and hidden credentials. *IEEE TC*, 2006.

[31] Google. GRPC, a high performance, open-source universal RPC framework. `https://grpc.io/`.

[32] Google. Key transparency. `https://github.com/google/keytransparency/blob/master/docs/design.md`.

[33] Google. Trillian. `https://github.com/google/trillian`.

[34] Google. VLBM implementation. `https://github.com/google/trillian/tree/master/examples/ct/ctmapper`.

[35] V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *CCS*, 2006.

[36] OAuth Working Group. Oauth 2 token exchange. `https://tools.ietf.org/html/draft-ietf-oauth-token-exchange-15`, 2018.

[37] Jason E Holt et al. Hidden credentials. In *ACM workshop on privacy in the electronic society*, 2003.

[38] Sam Kumar, Yuncong Hu, Michael P Andersen, Raluca Ada Popa, and David E. Culler. JEDI: Many-to-many end-to-end encryption and key delegation for iot. In *USENIX Security*, 2019.

[39] Selena Larson. Every single yahoo account was hacked - 3 billion in all, October 2017. Online.

[40] Ben Laurie. Revocation Transparency. `https://www.links.org/files/RevocationTransparency.pdf`, 2018.

[41] Ben Laurie, A. Langley, and E. Kasper. Certificate transparency (rfc 6992), 2013.

[42] David Lazar. Open-source IBE implementation. `https://github.com/vuvuzela/crypto`.

[43] Ninghui Li et al. Design of a role-based trust-management framework. In *IEEE S & P*, 2002.

[44] Ninghui Li et al. Distributed credential chain discovery in trust management. *J. CS, IOS Press*, 2003.

[45] Ninghui Li and John C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *PADL*, 2003.

[46] Benoît Libert and Jean-Jacques Quisquater. Identity based encryption without redundancy. In *ACNS*, 2005.

[47] Marcela S. Melara et al. CONIKS: Bringing key transparency to end users. In *USENIX Security*, 2015.

[48] Sascha Müller and Stefan Katzenbeisser. Hiding the policy in cryptographic access control. In *STM*, 2011.

[49] Ronald Rivest and Butler Lampson. SDSI-a simple distributed security infrastructure. CRYPTO, 1996.

[50] Martin Schanzenbach et al. Practical decentralized attribute-based delegation using secure name systems. *arXiv:1805.06398*, 2018.

[51] Kent E. Seamons et al. Requirements for policy languages for trust negotiation. In *POLICY*. IEEE, 2002.

[52] Hossein Shafagh, Lukas Burkhalter, Simon Duquennoy, Anwar Hithnawi, and Sylvia Ratnasamy. Droplet: Decentralized authorization for iot data streams, 2018.

[53] Adi Shamir. How to share a secret. *Comm. ACM*, 1979.

[54] Mudhakar Srivatsa and Mike Hicks. Deanonymizing mobility traces: Using social network as a side-channel. In *ACM CCS*, 2012.

[55] The Sovrin Foundation. A protocol and token for self-sovereign identity and decentralized trust, 2018.

[56] Vamsi Thummala and Jeff Chase. SAFE: A declarative trust management system with linked credentials. *arXiv preprint arXiv:1510.04629*, 2015.

[57] Daniel Trivellato et al. GEM: A distributed goal evaluation algorithm for trust management. *TPLP*, 2014.

[58] Marianne Winslett, Ting Yu, Kent E Seamons, Adam Hess, Jared Jacobson, Ryan Jarvis, Bryan Smith, and Lina Yu. Negotiating trust in the web. *IEEE IC*, 2002.

[59] Xian Zhu et al. Distributed credential chain discovery in trust-management with parameterized roles. In *CANS*, 2005.

A Proof of ULDM Security Guarantee

We provide a proof sketch for Guarantee 2.

Proof Sketch for Guarantee 2. We show that if neither clients in C nor the auditor A detect an attack, then there exists a possible history H of requests consistent with L and all responses to requests in R. Concretely, we show that the Operation Log that the storage server tells the auditor A is such a valid history H. Because A did not detect an inconsistency, we know that, for each client $c \in C$, (1) its Map Root Log head, at some point after its last request in R, is consistent with H. Because c did not detect an inconsistency, we know that (2) c's sequence of Map Root Log heads is append-only, (3) for each request, the returned object did (or did not, if no object was returned) exist in the Object Map, and (4) for each request, the Map Root Log at the time of the request contains the object map used in (3).

Together, (1) and (2) indicate that (5) the client's entire sequence of Map Root Log heads is consistent with H. Together, (3) and (4) indicate that (6) the response received for each request in R is consistent with the current Map Root Log head at the time of the request. Putting together (5) and (6), we can conclude that the response that each client receives to each request in R is consistent with H. Putting together (2) and (6), we can conclude that H is consistent with the partial ordering imposed by Map Root Log heads for each client c. Because clients make requests to the server to validate every Merge Promise, this also guarantees that H is consistent with the partial ordering imposed by Merge Promises. Thus, H fulfills all desired properties. □

B RDE Security Guarantee

Below, we develop definitions to precisely describe the global authorization graph, and then we use them to formalize RDE's security guarantee.

Definition 1 (Path)**.** *Let x and y be entities. $(A_1,\ldots,A_n)$ is a **path** from x to y if either $n > 0$ and $A_1.\mathsf{issuer} = x$, $A_n.\mathsf{subject} = y$, and $A_i.\mathsf{subject} = A_{i+1}.\mathsf{issuer}$ for all $i \in \{1,\ldots n-1\}$, or $n = 0$ and $x = y$.*

Definition 2 (Compatibility)**.** *Let A and B be attestations such that $A.\mathsf{subject} = B.\mathsf{issuer}$. We write $A \rightsquigarrow B$ and say "A is **partition-compatible** with B" if a key corresponding to one of the ID^*s in $Q(A.\mathsf{policy})$ can decrypt a WIBE ciphertext with the ID $P(B.\mathsf{policy})$. We analogously write $A \rightarrowtail B$ and say "A is **partition-label-compatible** with B" if a key corresponding to one of the ID^*s in $M(A.\mathsf{policy})$ can decrypt an IBE ciphertext with the ID $L(B.\mathsf{policy})$. We extend this to paths as follows. A path $(A_1,\ldots,A_n)$ is **partition-compatible** if either $n = 0$, or $A_i \rightsquigarrow A_{i+1}$ for all $i \in \{1,\ldots,n-1\}$. A path $(A_1,\ldots,A_n)$ is **partition-label-compatible** if either $n = 0$, or $A_1 \rightarrowtail A_2$ and $(A_2,\ldots,A_n)$ is partition-compatible.*

Based on our definitions of P, Q, L, and M in §4.3 and §4.4, we can attach semantic meaning to compatibility:

Note 1 (Compatibility Semantics for RTree)**.** *$A \rightsquigarrow B$ means that $A.\mathsf{policy}$ and $B.\mathsf{policy}$ have overlapping time ranges, URIs with the same namespace, and the same permission string. $A \rightarrowtail B$ means that $A.\mathsf{policy}$ and $B.\mathsf{policy}$ have URIs with the same namespace.*

Now, we formally define the states attached to an attestations during the discovery process (§4.7) so we can later express the leakage of an attestation in each state.

Definition 3 (Attestation State Machine)**.** *Let A be an attestation. If there exists a partition-compatible path $p = (A,P_1,\ldots,P_n)$ to an entity compromised by Adv, then we say that A is **useful** with respect to Adv.*

*Otherwise, if there exists a partition-label-compatible path $p = (A,P_1,\ldots,P_n)$ to an entity compromised by Adv, then we say that A is **partition-known** with respect to Adv.*

*Otherwise, if there exists a partition-compatible path from $A.\mathsf{subject}$ to an entity compromised by Adv, then we say that A is **interesting** with respect to Adv.*

*Otherwise, we say that A is **unknown** with respect to Adv.*

From `D`'s perspective in Fig. 2, for example, `#1`, `#4`, and `#3` are useful, `#5` is partition-known, and `#2` is unknown. The components of an RTree policy are described in §2.4.

Based on Definition 3, we can now *informally* state the security guarantee of RDE. Let A be an attestation such that there does not exist a partition-compatible path from A.subject to a partition-compatible cycle in the global authorization graph. If A is **unknown** or **interesting** with respect to Adv, then Adv learns nothing about A except A.subject and A's revocation commitment. If A is **partition-known** with respect to Adv, then Adv learns nothing about A except (1) A.subject, and (2) $P(A.\text{policy})$. If A is useful with respect to Adv, then Adv can decrypt A and see all of its fields.

We now formalize the security guarantee of RDE as a game played by a challenger Chl and an adversary Adv.

Guarantee 3 (RDE)**.** *Let λ denote the security parameter. Consider any list of entities in the system, represented as names in $\{0,1\}^*$, any subset of these entities compromised by* Adv*, and any two authorization graphs G_0 and G_1 each described as a list of attestations in terms of the entity names, subject to the constraints below:*

1. *$|G_0| = |G_1|$ and attestations at position i in the lists of G_0 and G_1 must have the same length. We say that these two attestations **correspond**.*
2. *Corresponding attestations must have the same state **unknown/interesting/partition-known/useful** w.r.t.* Adv.
3. *If corresponding attestations are **useful** to* Adv*, or if either has a partition-compatible path from its subject to a partition-compatible cycle, then they must be identical.*
4. *If corresponding attestations A_0 and A_1 are **partition-known** to* Adv*, or if there exists a partition-label-compatible path from A_0.*subject *(or A_1.*subject*) to a partition-compatible cycle in G_0 (or G_1), they must have the same subject and revocation commitment and satisfy $P(A_0) = P(A_1)$, but may otherwise differ arbitrarily.*
5. *If corresponding attestations are **unknown** or **interesting** to* Adv *(and if there is no partition-label-compatible path from the subject to a partition-compatible cycle) then they must have the same subject and revocation commitment, but may otherwise differ arbitrarily.*

Each attestation in the graph is described in terms of the information in §2.1, not RDE ciphertexts. RDE guarantees that Adv*'s advantage in the following game is negligible in the security parameter λ:*

Initialization. Chl *generates each entity's keypairs. It sends to* Adv *the public keys (verification key and WIBE/IBE public parameters) corresponding to each entity. For entities corresponding to malicious users,* Chl *also provides the secret keys (signing key and WIBE/IBE master keys). Furthermore,* Chl *chooses a random bit $b \in \{0,1\}$, computes the RDE ciphertext for each attestation in G_b, and gives them to* Adv.

Guess. Adv *outputs a bit $b' \in \{0,1\}$. The adversary's advantage in the game is defined as $\left|\Pr[b = b'] - \frac{1}{2}\right|$.*

The constraints on cycles in Conditions #3, #4, and #5 are due to the lack of KDM-security for the WIBE and IBE used. It may be possible to remove these constraints with KDM-secure variants.

Proof Sketch for Guarantee 3. We define a new game in which Adv has no advantage and prove via a hybrid argument that Adv's advantage in the real game differs from its advantage in this new game by at most a negligible amount.

In the hybrid argument, each hybrid represents a game. In the sequence of hybrids, the encrypted graph provided by the challenger if $b = 0$ is identical to the encrypted graph in the previous hybrid, except that either (1) one of the WIBE or IBE ciphertexts generated by Chl in the Challenge phase is replaced with an encryption of a different string of correct length, or (2) the ID used for IBE encryption is changed to a different ID. Adv cannot distinguish between adjacent hybrids due to CPA-security of WIBE and IBE in case (1), and due to the *anonymity* of IBE in case (2). Because adjacent hybrids are indistinguishable to Adv, the difference in its advantage in adjacent hybrids is negligible. The first game is the real game (Guarantee 3). In the final game, Adv's advantage is 0. By the hybrid argument, we can conclude that Adv's advantage in the real game is negligible.

The order in which ciphertexts are replaced must be chosen carefully. This is because a ciphertext cannot be replaced with an encryption of zero if a secret key to decrypt the ciphertext exists in the graph. We now describe the hybrids.

We identify attestations in the graph in Conditions #4 and #5. Observe that the "partition-compatible" relation defines a directed graph over these attestations in each G_0 and G_1, where each attestation is a vertex and edges indicate partition-compatibility. We denote these new graphs S_0 and S_1. Both S_0 and S_1 are directed acyclic graphs, due to the stipulations in Conditions #4 and #5 regarding cycles. Thus, S_0 and S_1 can be linearized. Via a sequence of hybrids, we first replace ciphertexts provided by Chl when it chooses $b = 0$ with encryptions of a dummy "zero string," following the reverse order of S_0's linearization. For attestations in Condition #4, we replace the WIBE ciphertexts in the attestations with encryptions of zero, in a single hybrid game for each attestation. For each attestation in Condition #5, we make two hybrid games; the first replaces its IBE ciphertext with an encryption of zeros, and the second replaces the ID used to encrypt with IBE for that ciphertext with a dummy ID. At the end of this hybrid sequence, the challenger provides a graph containing encryptions of zero in non-useful attestations if $b = 0$, and a proper encryption of G_1 if $b = 1$.

This is followed by another sequence of hybrids where we similarly transform the encryptions of zero provided by the challenger if $b = 0$ to proper encryptions of the attestations in G_1. This is done by transforming attestations in the forward order of S_1's linearization. In the final game, the challenger provides a graph containing a proper encryption of G_1, regardless of the chosen bit b, so Adv's advantage is 0. This completes the proof sketch. □

in-toto: Providing farm-to-table guarantees for bits and bytes

Santiago Torres-Arias[†], santiago@nyu.edu
Hammad Afzali[‡], ha285@njit.edu
Trishank Karthik Kuppusamy[*], trishank@datadog.com
Reza Curtmola[‡], crix@njit.edu
Justin Cappos[†], jcappos@nyu.edu

[†]New York University, Tandon School of Engineering
[*]Datadog
[‡]Department of Computer Science, New Jersey Institute of Technology

Abstract

The software development process is quite complex and involves a number of independent actors. Developers check source code into a version control system, the code is compiled into software at a build farm, and CI/CD systems run multiple tests to ensure the software's quality among a myriad of other operations. Finally, the software is packaged for distribution into a delivered product, to be consumed by end users. An attacker that is able to compromise any single step in the process can maliciously modify the software and harm any of the software's users.

To address these issues, we designed `in-toto`, a framework that cryptographically ensures the integrity of the software supply chain. `in-toto` grants the end user the ability to verify the software's supply chain from the project's inception to its deployment. We demonstrate `in-toto`'s effectiveness on 30 software supply chain compromises that affected hundreds of million of users and showcase `in-toto`'s usage over cloud-native, hybrid-cloud and cloud-agnostic applications. `in-toto` is integrated into products and open source projects that are used by millions of people daily. The project website is available at: `https://in-toto.io`.

1 Introduction

Modern software is built through a complex series of steps called a *software supply chain*. These steps are performed as the software is written, tested, built, packaged, localized, obfuscated, optimized, and distributed. In a typical software supply chain, these steps are "chained" together to transform (e.g., compilation) or verify the state (e.g., the code quality) of the project in order to drive it into a *delivered product*, i.e., the finished software that will be installed on a device. Usually, the software supply chain starts with the inclusion of code and other assets (icons, documentation, etc.) in a version control system. The software supply chain ends with the creation, testing and distribution of a delivered product.

Securing the supply chain is crucial to the overall security of a software product. An attacker who is able to control any step in this chain may be able to modify its output for malicious reasons that can range from introducing backdoors in the source code to including vulnerable libraries in the delivered product. Hence, attacks on the software supply chain are an impactful mechanism for an attacker to affect many users at once. Moreover, attacks against steps of the software supply chain are difficult to identify, as they misuse processes that are normally trusted.

Unfortunately, such attacks are common occurrences, have high impact, and have experienced a spike in recent years [60, 129]. Attackers have been able to infiltrate version control systems, including getting commit access to the Linux kernel [58] and Gentoo Linux [76], stealing Google's search engine code [22], and putting a backdoor in Juniper routers [48, 96]. Popular build systems, such as Fedora, have been breached when attackers were able to sign backdoored versions of security packages on two different occasions [75, 123]. In another prominent example, attackers infiltrated the build environment of the free computer-cleanup tool CCleaner, and inserted a backdoor into a build that was downloaded over 2 million times [126]. Furthermore, attackers have used software updaters to launch attacks, with Microsoft [108], Adobe [95], Google [50, 74, 140], and Linux distributions [46, 143] all showing significant vulnerabilities. Perhaps most troubling are several attacks in which nation states have used software supply chain compromises to target their own citizens and political enemies [35,55,82,92,93,108, 127,128,138]. There are dozens of other publicly disclosed instances of such attacks [8,33,38,39,41,52,53,65,70,76,79,80, 83,95,107,113,115,118,119,122,130–132,134,139,141,146].

Currently, supply chain security strategies are limited to securing each individual step within it. For example, Git commit signing controls which developers can modify a repository [78], reproducible builds enables multiple parties to build software from source and verify they received the same result [25], and there are a myriad of security systems that protect software delivery [2, 20, 28, 100, 102]. These building blocks help to secure an individual step in the process.

Although the security of each individual step is critical, such efforts can be undone if attackers can modify the output of a step before it is fed to the next one in the chain [22, 47]. These piecemeal measures by themselves can not stop malicious actors because there is no mechanism to verify that: 1) the correct steps were followed and 2) that tampering did not occur in between steps. For example a web server compromise was enough to allow hackers to redirect user downloads to a modified Linux Mint disk image, even though every single package in the image was signed and the image checksums on the site did not match. Though this was a trivial compromise, it allowed attackers to build a hundred-host botnet in a couple of hours [146] due to the lack of verification on the tampered image.

In this paper we introduce `in-toto`, Latin for "as a whole," the first framework that holistically enforces the integrity of a software supply chain by gathering cryptographically verifiable information about the chain itself. To achieve this, `in-toto` requires a project owner to declare and sign a

layout of how the supply chain's steps need to be carried out, and by whom. When these steps are performed, the involved parties will record their actions and create a cryptographically signed statement — called *link metadata* — for the step they performed. The link metadata recorded from each step can be verified to ensure that all steps were carried out appropriately and by the correct party in the manner specified by the layout.

The layout and collection of link metadata tightly connect the inputs and outputs of the steps in such a chain, which ensures that tampering can not occur between steps. The layout file also defines requirements (e.g., Twistlock [30] must not indicate that any included libraries have high severity CVEs) that will be enforced to ensure the quality of the end product. These additions can take the form of either distinct commands that must be executed, or limitations on which files can be altered during that step (e.g., a step that localizes the software's documentation for Mexican Spanish must not alter the source code). Collectively, these requirements can minimize the impact of a malicious actor, drastically limiting the scope and range of actions such an attacker can perform, even if steps in the chain are compromised.

We have built a series of production-ready implementations of `in-toto` that have now been integrated across several vendors. This includes integration into cloud vendors such as Datadog and Control Plane, to protect more than 8,000 cloud deployments. Outside of the cloud, `in-toto` is used in Debian to verify packages were not tampered with as part of the reproducible builds project [25]. These deployments have helped us to refine and validate the flexibility and effectiveness of `in-toto`.

Finally, as shown by our security analysis of three `in-toto` deployments, `in-toto` is not a "lose-one, lose-all" solution, in that its security properties only partially degrade with a key compromise. Depending on which key the attacker has accessed, `in-toto`'s security properties will vary. Our `in-toto` deployments could be used to address most (between 83% - 100%) historical supply chain attacks.

2 Definitions and Threat Model

This section defines the terms we use to discuss the software supply chain and details the specific threat model `in-toto` was designed to defend against.

2.1 Definitions

The software supply chain refers to the series of *steps* performed in order to create and distribute a *delivered product*. A *step* is an operation within this chain that takes in *materials* (e.g., source code, icons, documentation, binaries, etc.) and and creates one or more *products* (e.g., libraries, software packages, file system images, installers, etc.). We refer to both materials and products generically as *artifacts*.

It is common to have the products of one step be used as materials in another step, but this does not mean that a supply chain is a sequential series of operations in practice. Depending on the specifics of a supply chain's workflow, steps may be executed in sequence, in parallel, or as a combination of both. Furthermore, steps may be carried out by any number of hosts, and many hosts can perform the same step (e.g., to test a step's reproducibility).

In addition to the materials and products, a step in the supply chain produces another key piece of information, *byproducts*. The step's byproducts are things like the `STDOUT`, `STDERR`, and return value that indicate whether a step was successful or had any problems. For example, a step that runs unit tests may return a non-zero code if one of the unit tests fails. Validating byproducts is key to ensuring that steps of the supply chain indicate that the software is ready to use.

As each step executes, information called *link* metadata that describes what occured, is generated. This contains the materials, products, and byproducts for the step. This information is signed by a key used by the party who performs the action, which we call a *functionary*. Regardless of whether the functionary commits code, builds software, performs QA, localizes documentation, etc., the same link metadata structure is followed. Sometimes a functionary's participation involves repeated human action, such as a developer making a signed git commit for their latest code changes. In other cases, a functionary may participate in the supply chain in a nearly autonomous manner after setup, such as a CI/CD system. Further, many functionaries can be tasked to perform the same step for the sake of redundancy and a minimum threshold of them may be required to agree on the result of a step they all carried out.

To tie all of the pieces together, the *project owner* sets up the rules for the steps that should be performed in a software supply chain. In essence, the project owner serves as the foundation of trust, stating which steps should be performed by which functionaries, along with specifying rules for products, byproducts, and materials in a file called the *layout*. The layout enables a *client* that retrieves the software to cryptographically validate that all actions were performed correctly. In order to make this validation possible, a client is given the *delivered product*, which contains the software, layout, and link metadata. The layout also contains any additional actions besides the standard verification of the artifact rules to be performed by the client. These actions, called *inspections*, are used to validate software by further performing operations on the artifacts inside the delivered product (e.g., verifying no extraneous files are inside a zip file). This way, through standard verification and inspections, a client can assure that the software went through the appropriate software supply chain processes.

2.2 Threat Model

The goal of `in-toto` is to minimize the impact of a party that attempts to tamper with the software supply chain. More specifically, the goal is to retain the maximum amount of security that is practical, in any of the following scenarios:

- Interpose between two existing elements of the supply chain to change the input of a step. For example, an attacker may ask a hardware security module to sign a malicious copy of a package before it is added to the repository and signed repository metadata is created to index it [27, 44, 51, 76, 107, 120, 120, 147].

- Act as a step (e.g., compilation), perhaps by compromising or coercing the party that usually performs that step [27, 57, 62, 64, 76, 81, 99, 112, 125]. For example, a hacked compiler could insert malicious code into binaries it produces [126, 136].
- Provide a delivered product for which not all steps have been performed. Note that this can also be a result of an honest mistake [37, 49, 56, 68, 73, 97, 142].
- Include outdated or vulnerable elements in the supply chain [59, 61, 91, 94, 117]. For example, an attacker could bundle an outdated compression library that has many known exploits.
- Provide a counterfeit version of the delivered product to users [8, 35, 66, 70, 71, 95, 118, 134, 135, 146]. This software product can come from any source and be signed by any keys. While `in-toto` will not mandate how trust is bootstrapped, Section 6 will show how other protocols such as TUF [28], as well as popular package managers [2] can be used to bootstrap project owner keys.

Key Compromise. We assume that the public keys of project owners are known to the verifiers and that the attacker is not able to compromise the corresponding secret key. In addition, private keys of developers, CI systems and other infrastructure public keys are known to a project owner and their corresponding secret keys are not known to the attacker. In section 5.2, we explore additional threat models that result from different degrees of attacker access to the supply chain, including access to infrastructure and keys (both online and offline).

2.3 Security Goals

To build a secure software supply chain that can combat the aforementioned threats, we envision that the following security goals would need to be achieved:

- **supply chain layout integrity:** All of the steps defined in a supply chain are performed in the specified order. This means that no steps can be added or removed, and no steps can be reordered.
- **artifact flow integrity:** All of the artifacts created, transformed, and used by steps must not be altered in-between steps. This means that if step A creates a file `foo.txt` and step B uses it as a material, step B must use the exact file `foo.txt` created by step A. It must not use, for example, an earlier version of the file created in a prior run.
- **step authentication:** Steps can only be performed by the intended parties. No party can perform a step unless it is given explicit permission to do so. Further, no delivered products can be released unless all steps have been performed by the right party (e.g., no releases can be made without a signoff by a release engineer, which would stop accidental development releases [68]).
- **implementation transparency:** `in-toto` should not require existing supply chains to change their practices in order to secure them. However, `in-toto` can be used to represent the existing supply chain configuration and reason about its security practices.
- **graceful degradation of security properties:** `in-toto` should not lose all security properties in the event of key compromise. That is, even if certain supply chain steps are compromised, the security of the system is not completely undermined.

In addition to these security goals, `in-toto` is also geared towards practicality and, as such, it should maintain minimal operational, storage and network overheads.

3 System overview

The current landscape of software supply chain security is focused on point-solutions that ensure that an individual step's actions have not been tampered with. This limitation usually leads to attackers compromising a weaker step in the chain (e.g., breaking into a buildfarm [115]), removing steps from the chain [68] or tampering with artifacts while in transit (i.e., adding steps to the chain [66]). As such, we identify two fundamental limitations of current approaches to secure the software supply chain:

1. Point solutions designed to secure individual supply chain steps cannot guarantee the security of the entire chain as a whole.
2. Despite the widespread use of unit testing tools and analysis tools, like fuzzers and static analyzers, software rarely (if ever) includes information about what tools were run or their results. So point solutions, even if used, provide limited protection because information about these tools is not appropriately utilized or even shown to clients who can make decisions about the state of the product they are about to utilize.

We designed `in-toto` to address these limitations by ensuring that all individual measures are applied, and by the right party in a cryptographically verifiable fashion.

In concrete terms, `in-toto` is a framework to gather and verify metadata about different stages of the supply chain, from the first step (e.g., checking-in code on a version control system) to delivered product (e.g., a .deb installable package). If used within a software supply chain, `in-toto` ensures that the aforementioned security goals are achieved.

3.1 `in-toto` parties and their roles

Similar to other modern security systems [101, 102, 121], `in-toto` uses security concepts like delegations and roles to limit the scope of key compromise and provide a graceful degradation of its security properties.

In the context of `in-toto`, a role is a set of duties and actions that an actor must perform. The use of delegations and roles not only provides an important security function (limiting the impact of compromise and providing separation of privilege), but it also helps the system remain flexible and usable so that behaviors like key sharing are not needed. Given that every project uses a very specific set of tools and practices, flexibility is a necessary requirement for `in-toto`. There are three roles in the framework:

- Project Owner: The project owner is the party in charge of defining the software supply chain layout (i.e., define

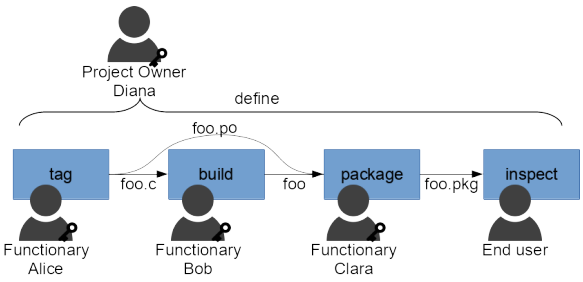


Figure 1: Graphical depiction of the software supply chain with `in-toto` elements added. The project owner creates a layout with three steps, each of which will be performed by a functionary. Notice how the tag step creates `foo.c` and a localization file `foo.po`, which are fed to different steps down the chain.

which steps must be performed and by who). In practice, this would be the maintainer of an open-source project or the dev-ops engineers of a project.

- Functionaries: Functionaries are the parties that perform the steps within the supply chain, and provide an authenticated record of the artifacts used as materials and the resulting products. Functionaries can be humans carrying out a step (e.g., signing off a security audit) or an automated system (e.g., a build farm).
- Client: (e.g., end user): The client is the party that will inspect and afterwards utilize a delivered product.

We will now elaborate on how these three parties interact with the components of `in-toto`.

3.2 `in-toto` components

`in-toto` secures the software supply chain by using three different types of information: the software supply chain layout (or layout, for short), link metadata, and the delivered product. Each of these has a unique function within `in-toto`.

3.2.1 The supply chain layout

Laying out the structure of the supply chain allows the developers and maintainers of a project to define requirements for steps involved in source code writing, testing, and distribution within a software product's lifecycle. In the abstract sense, this supply chain layout is a recipe that identifies which steps will be performed, by whom, and in what order.

The supply chain layout defines a series of *steps* in the supply chain. These definitions are used to enforce measures on what artifacts should be used as *materials*. To ensure that only the intended parties execute the right steps, a public key is associated with each step. In order to ensure that the layout was created by the project owner, it is cryptographically signed with the project owner's private key.

The project owner will define this supply chain layout by setting different requirements for the project's steps. These requirements take the form of types of artifacts that can be produced (e.g., a localization step can only produce `.po` files), the expected return values, the type of host that can carry out this step and so forth. When consuming the delivered product, the client (end user) verifies that these requirements are satisfied.

In addition to defining supply chain steps, the layout will also specify a series of *inspection steps* (or inspections). These inspections will be performed by the verifier on the delivered product to draw further insight about its correctness. This is useful for complex supply chains in which the basic semantics of `in-toto` cannot describe their specific requirements. For example, an inspection step can be used to namespace restrict certain VCS-specific operations to specific functionaries such as making sure that only a QA team member merges code into the develop branch and that all commits are signed.

For example, as seen in Figure 1, a project owner can define a supply chain consisting of three steps: a tag, a build and a package step. With these definitions, the project owner also defines how the artifacts will flow through the supply chain (e.g., foo.c is used by build, yet foo.po is packaged directly from tag). Afterwards, the project owner can assign functionaries to carry out each of these steps and define an inspection so the end user can verify that foo was indeed created during build and that foo.po came from the tagged release.

Layout creation tool. We provide a web-based layout creation tool [12] to help project owners create `in-toto` layouts. The tool uses an intuitive, graphical interface to define: (1) the steps of the software supply chain (i.e., how is the source code managed? how is the software's quality verified? how is the software built? how is the software packaged?), (2) the actors (functionaries) who are allowed to perform different steps of the software supply chain. An `in-toto` layout is generated based on this information. In addition, the `in-toto` website [13, 15] provides several examples of layouts, which can serve as starting templates for project owners seeking to integrate `in-toto`.

3.2.2 Link metadata

Verifying the actions carried out in the supply chain, requires information about all steps performed in creating the delivered product. Like a chain in real life, an `in-toto` supply chain consists of conjoined *links*, with each link serving as a statement that a given step was carried out.

Functionaries in charge of executing a step within the supply chain must share information about these links. Sharing such information as what *materials* were fed to the step, and what *product(s)* were created, can ensure no artifacts are altered in transit. To ensure that only the right functionaries performed this step, the piece of link metadata must be signed with the private key that corresponds to this functionary's key (as defined in the supply chain layout).

There is a one-to-one relationship between the step definitions in the supply chain layout and the link metadata. That is, each piece of link metadata gathered during each step within the supply chain must match what the requirements prescribe for that step. In order to ensure that the link metadata is generated by the intended entity, it must be cryptographically signed with one (or more, if there is a threshold higher than one defined) of the keys indicated in the requirements for that link.

When all the link metadata has been collected, and the supply chain has been properly defined, the supply chain layout and all the links can be shipped, along with the delivered product, to the end user for verification. We show

a minimal software supply chain, along with a graphical representation of an in-toto layout in Figure 1.

3.2.3 The delivered product

The delivered product is the piece of software that the end user wants to install. In order to verify the delivered product, the end user (or client) will utilize the supply chain layout and its corresponding pieces of link metadata. The end user will use the link metadata to verify that the software provided has not been tampered with, and that all the steps were performed as the project owner intended. In Figure 1 the delivered product consists of the foo.pkg file.

3.3 in-toto usage lifecycle

The in-toto usage lifecycle encompasses the following overarching operations:

1. The project owner defines a supply-chain layout.
2. Each step is carried out as specified, and functionaries gather and sign link metadata.
3. A delivered product is shipped to the client, who verifies it upon installation by:
   - ensuring the layout provided was signed by the project owner and is not expired.
   - checking that all the steps defined have enough pieces of link metadata; that such links were signed by the indicated functionaries; and that all artifacts recorded flowed properly between the steps as indicated in the layout.
   - carrying out any inspection steps contained in the layout and making sure that all artifacts recorded match the flow described in the layout.

As seen in Figure 1 a project owner creates the layout to describe an overarching structure of the supply chain that the client can use to verify. Later, functionaries carry out their operations as usual, and submit link metadata to attest for the result of their operation. Finally, a client uses a delivered product, metadata links and a layout to verify the integrity of the delivered product and of the entire chain.

By following the chain of attestations in the link metadata, the client can reconstruct the operations described in Figure 1. Afterwards, the client can verify these attestations against the layout and execute any inspections to make sure everything is in order before consuming the delivered product.

4 in-toto internals

In order to avoid tampered, incomplete or counterfeit software, in-toto ensures the integrity and accuracy of all software supply chain operations. in-toto ensures supply chain integrity by the verifying the collected link metadata against a software supply chain layout file. This ensures that all operations were carried out, by the intended party and as the legitimate project owner intended.

Understanding how the system's metadata helps to ensure the integrity of the supply chain is critical to a deeper appreciation of how in-toto works. In this section, we will explore the specifics of the link metadata and the layout file to understand how in-toto operates.

For the context of this section, we will demonstrate the different features of in-toto using Figure 1 as an example. The project owner Diana will create a layout that describes three steps and three functionaries for each step. The first step, tag, will produce a file foo.c to be input into the build step, as well as a foo.po localization file. The second step, build, will use the foo.c file from the tag step and produce a foo binary. Finally, the package step will take the foo.po and foo files and produce a package installable by the end user.

For a more complete and thorough description of all the fields, signature schemes, implementations, a layout editing tool and more, refer to the resources on the project website: https://in-toto.io.

4.1 The supply chain layout

The supply chain layout explicitly defines the expected layout of the software supply chain. This way, end users can ensure that its integrity is not violated upon verification. To do this, the layout contains the following fields:

```
{ "_type" : "layout",
  "expires" : "<EXPIRES>",
  "readme": "<README>",
  "keys" : {"<KEYID>":"<PUBKEY_OBJECT>"...},
  "steps" : ["<STEP>", "..." ],
  "inspections" : ["<INSPECTION>","..." ]
}
```

Listing 1: The supply chain layout structure

The overarching architecture of the layout definition includes the following relevant fields:

- An expiration date: this will ensure that the supply chain information is still fresh, and that old delivered products can not be replayed to users.
- A readme field: this is intended to provide a human-readable description of the supply chain.
- A list of public keys: these keys belong to each functionary in the supply chain and will be assigned to different steps to ensure that only the right functionary performs a particular step in the supply chain.
- A list of steps: these are the steps to be performed in the supply chain and by who. Step definitions, described in depth in Section 4.1.1, will contain a series of requirements that limit the types of changes that can be done in the pipeline and what functionary can sign link metadata to attest for its existence.
- A list of inspections: these are the inspections to be performed in the supply chain. As described in depth in section 4.1.2, inspections are verification steps to be performed on the delivered product by the client to further probe into its completeness and accuracy.

Though its structure is quite simple, the layout actually provides a detailed description of the supply chain topology. It characterizes each of the steps, and defines any possible requirements for every step. Likewise, it contains instructions for local inspection routines (e.g., verify that every file in a tar archive was created by the right party in the supply chain), which further ensure the delivered product has not been

tampered with. As such the layout allows the project owner to construct the necessary framework for a secure supply chain.

For our example supply chain, Diana would have to list the public keys as described on Listing 2, as well as all the steps.

```
{ "_type" : "layout",
  "expires" : "<EXPIRES>",
  "readme": "foo.pkg supply chain",
  "keys" : {"<BOBS_KEYID>":"<PUBKEY>",
            "<ALICES_KEYID":"<PUBKEY>",
            "<CLARAS_KEYID":"<PUBKEY>"},
  "steps" : [{"name": "tag", "..."},
             {"name": "build", "..."},
             {"name": "package", "..."} ],
  "inspections" : ["{"name": "inspect", "..."}]
}
```

Listing 2: The supply chain for our example

As described, the layout file already limits all actions to trusted parties (by means of their public keys), defines the steps that are carried out (to limit the scope of any step) and specifies verification routines that are used to dive into the specifics of a particular supply chain. We will describe the latter two fields in depth now.

4.1.1 Step definition

```
{ "_name": "<NAME>",
  "threshold": "<THRESHOLD>",
  "expected_materials": [["<ARTIFACT_RULE>"], "..."],
  "expected_products": [["<ARTIFACT_RULE>"], "..."],
  "pubkeys": ["<KEYID>", "..." ],
  "expected_command": "<COMMAND>"
}
```

Listing 3: A supply chain step in the supply chain layout

Every step of the supply chain contains the following fields:

- **name**: A unique identifier that describes a step. This identifier will be used to match this definition with the corresponding pieces of link metadata.
- **expected_materials**: The materials expected as input `ARTIFACT_RULES` as described in Section 4.1.3. It serves as a master reference for all the artifacts used in a step.
- **expected_products**: Given the step's output information, or *evidence*, what should be expected from it? The expected products also contains a list of `ARTIFACT_RULES` as described in section 4.1.3.
- **expected_command**: The command to execute and any flags that may be passed to it.
- **threshold**: The minimum number of pieces of signed link metadata that must be provided to verify this step. This field is intended for steps that require a higher degree of trust, so multiple functionaries must perform the operation and report the same results. For example, if the threshold is set to k, then at least k pieces of signed link metadata need to be present during verification.
- **a list of public keys id's**: The id's of the keys that can be used to sign the link metadata for this step.

The fields within this definition list will indicate requirements for the step identified with that name. To verify these requirements, these fields will be matched against the link metadata associated with the step. The `expected_materials` and `expected_products` fields will be used to compare against the materials and products reported in the link metadata. This ensures that no disallowed artifacts are included, that no required artifacts are missing, and the artifacts used are from allowed steps who created them as products. Listing 4 contains the step definition for the build step for our example Layout above.

```
{ "_name": "build",
  "threshold": "1",
  "expected_materials": [
    ["MATCH", "foo.c", "WITH",
     "PRODUCTS", "FROM", "tag"]
  ],
  "expected_products": [["CREATE", "foo"]],
  "pubkeys": ["<BOBS_PUBKEY>"],
  "expected_command": "gcc foo.c -o foo"
}
```

Listing 4: The build step in our example layout

4.1.2 Inspection definition

Inspection definitions are nearly identical to step definitions. However, since an inspection causes the verifier on the client device to run a command (which can also generate artifacts), there cannot be a threshold of actions. The other fields are identical to the link metadata generated by a step.

4.1.3 Artifact rules

Artifact rules are central to describing the topology of the supply chain by means of its artifacts. These rules behave like firewall rules and describe whether an artifact should be consumed down the chain, or if an artifact can be created or modified at a specific step. As such, they serve two primary roles: to limit the types of artifacts that a step can create and consume; and to describe the flow of artifacts between steps.

For the former, a series of rules describes the operation within the step. A rule, such as `CREATE`, indicates that a material must not exist before the step is carried out and must be reported as a product. Other rules, such as `MODIFY`, `DELETE`, `ALLOW` and `DISALLOW` are used to further limit what a step can register as artifacts within the supply chain. These rules are described in Grammar 5 (full definition in Appendix A). An example of a simple `CREATE` rule can be seen on the step definition in Listing 4.

```
[CREATE|DELETE|MODIFY|ALLOW|DISALLOW] artifact_pattern
```

Grammar 5: Grammar for operations within a step. artifact_pattern is a regular expression for the paths to artifacts.

For the latter, the `MATCH` rule is used by project owners to describe the flow of artifacts *between steps*. With it, a project owner can mandate that, e.g., a buildfarm must only use the sources that were created during a tag-release step or that only the right localization files are included during a localization step. Compared to the rules above, the `MATCH` rule has a richer syntax, as it needs to account for artifacts relocated during steps (e.g,. a packaging step moving .py files to `/usr/lib/pythonX.X/site-packages/` or a build step moving artifacts to a build directory) using the `IN` clause. Grammar 6 describes this rule and the Match function

describes the algorithm for processing it during verification. An example of a `MATCH` rule used to match the `foo.c` source from tag into the build step is shown in Listing 4.

```
MATCH source_pattern [IN prefix]
  WITH <MATERIALS|PRODUCTS> [IN prefix] FROM step_name
```

Grammar 6: The match rule grammar. The IN clauses are optional and source_pattern is a regular expression

function MATCH
Input: source_artifacts; destination_artifacts, rule
Output: *result: (SUCCESS/FAIL)*

```
1:  // Filter source and destination materials using the rule's patterns
2:  source_artifacts_filtered = filter(rule.source_prefix + rule.source_pattern,
    source_artifacts)
3:  destination_artifacts_filtered = filter(rule.destination_prefix +
    rule.destination_pattern, destination_artifacts)
4:  // Apply the IN clauses, to the paths, if any
5:  for artifact in source_artifacts_filtered do
6:    artifact.path -= rule.source_in_clause
7:  for artifact in destination_artifacts_filtered do
8:    artifact.path -= rule.destination_in_clause
9:  // compare both sets
10: for artifact in source_artifacts_filtered do
11:   destination_artifact = find_artifact_by_path(destination_artifacts,
    artifact.path)
12:   // the artifact with this path does not exist?
13:   if destination_artifact == NULL then
14:     return FAIL
15:   // are the files not the same?
16:   if destination_artifact.hash != artifact.hash then
17:     return FAIL
18: // all of the files filtered by the source materials exist
19: return SUCCESS
```

4.2 Link metadata files

Link metadata serves as a record that the steps prescribed in the layout actually took place. Its fields show *evidence* that is used for verification by the client. For example, the *materials* and *products* fields of the metadata are used to ensure that no intermediate products were altered in transit before being used in a step.

In order to determine if the executed step complies with its corresponding metadata, several types of information need to be gathered as evidence. A link includes the following fields:

```
{ "_type" : "link",
  "_name" :  "<NAME>",
  "command" : "<COMMAND>",
  "materials": {"<PATH>": "<HASH>", "..." : "..." },
  "products": {"<PATH>": "<HASH>", "..." : "..." },
  "byproducts": { "stdin": "", "stdout": "",
        "return-value": "" },
  "environment": {"variables": "<ENV>",
        "filesystem": "<FS>",  ...}
}
```

Listing 7: Link metadata format

- Name: This will be used to identify the step and to match it with its corresponding definition inside the layout.
- Material(s): Input file(s) that were used in this step, along with their cryptographic hashes to verify their integrity.
- Command: The command run, along with its arguments.
- Product(s): The output(s) produced and its corresponding cryptographic hash.
- Byproduct(s): Reported information about the step. Elements like the standard error buffer and standard output buffer will be used.
- Signature: A cryptographic signature over the metadata.

Of these fields, the `name`, `materials`, and `products` fields are the counterpart of the fields within the layout definition. This, along with a cryptographic signature used to authenticate the functionary who carried out the step can be used to provide a baseline verification of the supply chain topology (i.e., the steps performed and how do they interrelate via their materials and products). For example, the build step metadata described in Listing 8 shows the file `foo.c` used as a material and the product `foo` as created in the build step.

The *byproducts* field is used to include other meaningful information about a step's execution to further introspect into the specifics of the step that was carried out. Common fields included as byproducts are the standard output, standard error buffers and a return value. For example, if a command exited with non-zero status, then the byproduct field be populated with a value such as `return-value: "126"`. In this case, inspections can be set up to ensure that the return value of this specific command must be 0.

```
{ "_type": "link",
  "name": "build",
  "command": ["gcc", "foo.c", "-o", "foo" ],
  "materials": {"foo.c": { "sha256": "bff95e..."}},
  "products": {"foo": {"sha256": "25c696..."}}
  "byproducts": { "return-value": 0,
        "stderr": "", "stdout": ""},
  "environment": {},
}
```

Listing 8: The link for the build step

Having a software supply chain layout along with the matching pieces of link metadata and the delivered product are all the parts needed to perform verification. We will describe verification next.

4.3 Verifying the delivered product

Verification occurs when the link metadata and the layout are received by the client and upon installing the delivered product. A standalone or operating-system tool will perform the verification, as described in the function Verify_Final_Product. To do this, the user will need an initial public key that corresponds to the supply chain layout, as distributed by a trusted channel or as part of the operating system's installation [106].

The end user starts the verification by ensuring that the supply chain layout provided was signed with a trusted key (lines 2-3) and by checking the layout expiration date to make sure the layout is still valid (lines 5-6). If these checks pass, the public keys of all the functionaries are loaded from the layout (line 8). With the keys loaded, the verification routine will start iterating over all the steps defined in the layout and make sure there are enough pieces of link metadata signed by the right functionaries to match the threshold specified for that role (lines 10-20). If enough pieces of link metadata

```
function VERIFY_FINAL_PRODUCT
Input: layout; links; project_owner_key
Output: result: (SUCCESS/FAIL)
 1: // verify that the supply chain layout was properly signed
 2: if not verify_signature(layout, project_owner_key) then
 3:     return FAIL
 4: // Check that the layout has not expired
 5: if layout.expiration < TODAY then
 6:     return FAIL
 7: // Load the functionary public keys from the layout
 8: functionary_pubkeys = layout.keys
 9: // verify link metadata
10: for step in layout.steps do
11:     // Obtain the functionary keys relevant to this step and its corresponding
        metadata
12:     step_links = get_links_for_step(step, links)
13:     step_keys = get_keys_for_step(step, functionary_pubkeys)
14:     // Remove all links with invalid signatures
15:     for link in step_links do
16:         if not verify_signature(link, step_keys) then
17:             step_links.remove(link)
18:     // Check there are enough properly-signed links to meet the threshold
19:     if length(step_links) < step.threshold then
20:         return error("Link metadata is missing!")
21: // Apply artifact rules between all steps
22: if apply_artifact_rules(steps, links) == FAIL then
23:     return FAIL
24: // Execute inspections
25: for inspection in layout.inspections do
26:     inspections.add(Run(inspection))
27: // Verify inspections
28: if apply_artifact_rules(steps + inspections, links) == FAIL then
29:     return FAIL
30: return SUCCESS
```

could be loaded for each of the steps and their signatures pass verification, then the verification routine will apply the artifact rules and build a graph of the supply chain using the artifacts recorded in the link metadata (lines 22-23). If no extraneous artifacts were found and all the `MATCH` rules pass, then inspections will be executed[1] (line 25-26). Finally, once all inspections were executed successfully, artifact rules are re-applied to the resulting graph to check that rules on inspection steps match after inspections are executed, because inspections may produce new artifacts or re-create existing artifacts (lines 28-29). If all verifications pass, the function will return `SUCCESS`.

With this verification in place, the user is sure that the integrity of the supply chain is not violated, and that all requirements made by the project's maintainers were met.

4.4 Layout and Key Management

A layout can be revoked in one of two ways, the choice being up to the project owner. One is by revoking the key that was used to sign the layout, the other is by superseding/updating the layout with a newer one. To update a layout, the project owner needs to replace an existing layout with a newer layout. This can be used to deal with situations when a public key of a misbehaving functionary needs to be changed/revoked, because when the project owner publishes a newer layout without that public key, any metadata from such misbehaving functionary is automatically revoked. Updating a layout can also be used to address an improperly designed initial layout. The right expiration date for a layout depends on the operational security practices of the integrator.

[1] Inspections are executed only after all the steps are verified to avoid executing an inspection on an artifact that a functionary did not create.

5 Security Analysis

`in-toto` was designed to protect the software supply chain as a whole by leveraging existing security mechanisms, ensuring that they are properly set up and relaying this information to a client upon verification. This allows the client to make sure that all the operations were properly performed and that no malicious actors tampered with the delivered product.

To analyze the security properties of `in-toto`, we need to revisit the goals described in Section 2. Of these, the relevant goals to consider are *supply chain layout integrity*, *artifact flow integrity*, and *step authentication*. In this section, we explore how these properties hold, and how during partial key compromise the security properties of `in-toto` degrade gracefully.

`in-toto`'s security properties are strictly dependent on an attacker's level of access to a threshold of signing keys for any role. These security properties degrade depending on the type of key compromise and the supply chain configuration.

5.1 Security properties with no key compromise

When an attacker is able to compromise infrastructure or communication channels but not functionary keys, `in-toto`'s security properties ensure that the integrity of the supply chain is not violated. Considering our threat model in Section 2, and contrasting it to `in-toto`'s design which stipulates that the supply chain layout and link metadata are signed, we can assert the following:

- An attacker cannot interpose between two consecutive steps of the supply chain because, during verification, the hash on the products field of the link for the first step will not match the hash on the materials field of the link for the subsequent step. Further, a completely counterfeit version of the delivered product will fail validation because its hash will not match the one contained in the corresponding link metadata. Thus, **artifact flow integrity holds**.
- An attacker cannot provide a product that is missing steps or has its steps reordered because the corresponding links will be missing or will not be in the correct order. The user knows exactly which steps and in what order they need to be performed to receive the delivered product. As such, **supply chain layout integrity holds**.
- Finally, an attacker cannot provide link metadata for which he does not have permission to provide (i.e., their key is not listed as one that can sign link metadata for a certain step). Thus, **step authentication holds**.

However, it is important to underline that this threat model requires that the developer's host systems are not compromised. Likewise, it assumes that there are no rogue developers wishing to subvert the supply chain. For practical purposes, we consider a rogue functionary to be equivalent

to a functionary key compromise. Hence this section frames attacks from the standpoint of a key compromise, even when the issue may be executed as a confused deputy problem or a similar issue with equivalent impact.

5.2 Security properties if there is a key compromise

`in-toto` is not a "lose-one, lose-all" solution, in that its security properties only partially degrade with a key compromise. Depending on which key the attacker has accessed, `in-toto`'s security properties will vary. To further explore the consequences of key compromise, we outline the following types of attacks in the supply chain:

- Fake-check: a malicious party can provide evidence of a step taking place, but that step generates no products (it can still, however, generate byproducts). For example, an attacker could forge the results of a test suite being executed in order to trick other functionaries into releasing a delivered product with failing tests.
- Product Modification: a malicious party is able to provide a tampered artifact in a step to be used as material in subsequent steps. For example, an attacker could take over a buildfarm and create a backdoored binary that will be packaged into the delivered product.
- Unintended Retention: a malicious party does not destroy artifacts that were intended to be destroyed in a step. For example, an attacker that compromises a cleanup step before packaging can retain exploitable libraries that will be shipped along with the delivered product.
- Arbitrary Supply Chain Control: a malicious party is able to provide a tampered or counterfeit delivered product, effectively creating an alternate supply chain.

5.2.1 Functionary compromise

A compromise on a threshold of keys held for any functionary role will only affect a specific step in the supply chain to which that functionary is assigned to. When this happens, the **artifact flow integrity** and **step authentication** security properties may be violated. In this case, the attacker can arbitrarily forge link metadata that corresponds to that step.

The impact of this may vary depending on the specific link compromised. For example, an attacker can fabricate an attestation for a step that does not produce artifacts (i.e., a fake-check), or create malicious products (i.e., a product modification), or pass along artifacts that should have been deleted (i.e., an unintended retention). When an attacker creates malicious products or fails to remove artifacts, the impact is limited by the usage of such products in subsequent steps of the chain. Table 1 describes the impact of these in detail from rows 2 to 5 (row 1 captures the case when the attacker does not compromise enough keys to meet the threshold defined for a step). As a recommended best practice, we assume there is a "`DISALLOW *`" rule at the end of the rule list for each step.

It is of note from Table 1 that an attacker who is able to compromise crucial steps (e.g., a build step) will have a greater impact on the client than one which, for example, can only alter localization files. Further, a compromise in functionary keys that do not create a product is restricted to a fake check attack (row two). To trigger an unintended retention, the first step must also have rules that allow for some artifacts before the `DELETE` rule (e.g., the `ALLOW` rule with a similar artifact pattern). This is because rules behave like artifact rules, and the attacker can leverage the ambiguity of the wildcard patterns to register an artifact that was meant to be deleted. Lastly, note that the effect of product modification and unintended retention is limited by the namespace on such rules (i.e., the `artifact_pattern`).

Type of Key Compromise	Compromised Step Rule	Subsequent Step Rule	Impact
Under threshold	Regardless of rule	Regardless of rule	None
Step	None	Regardless of rule	Fake-check
Step	`ALLOW pattern1` `DELETE pattern2`	`MATCH pattern*`	Unintended Retention
Step	`[ALLOW | CREATE | MODIFY] pattern`	`MATCH pattern`	Product Modification
Layout	N/A	N/A	Arbitrary Supply Chain Control

Table 1: Key compromise and impact based on the layout characteristics.

Mitigating risk. As discussed earlier, the bar can be raised against an attacker if a role is required to have a higher threshold. For example, two parties could be in charge of signing the tag for a release, which would require the attacker to compromise two keys to successfully subvert the step.

Finally, further steps and inspections can be added to the supply chain with the intention of limiting the possible transformations on any step. For example, as shown in Section 6, an inspection can be used to dive into a Python's wheel and ensure that only Python sources in the tag release are contained in the package.

5.2.2 Project owner compromise

A compromise of a threshold of keys belonging to the project owner role allows the attacker to redefine the layout, and thereby subvert the supply chain completely. However, like with step-level compromises, an increased threshold setting can be used to ensure an attacker needs to compromise many keys at once. Further, given the way `in-toto` is designed, the layout key is designed to be used rarely, and thus it should be kept offline.

5.3 User actions in response to `in-toto` failures

Detecting a failure to validate `in-toto` metadata involves making a decision about whether verification succeeded or whether it failed and, if so, why. The user's device and the reason for failure are likely to be paramount in the user's decision about how to respond. If the user is installing in an ephemeral environment on a testing VM, they may choose to ignore the warning and install the package regardless. If the user is installing in a production environment processing PCI data, the failure to validate `in-toto` metadata will be a serious concern. So, we expect users of `in-toto` will respond in much the same way as administrators do today for a package that is not properly signed.

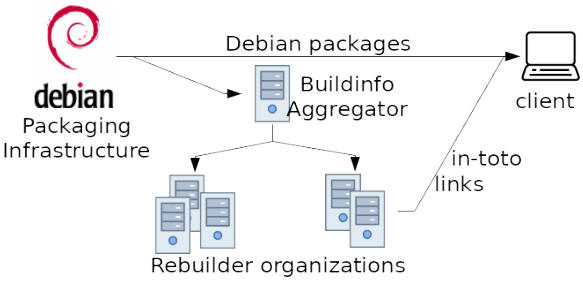


Figure 2: The rebuilder setup.

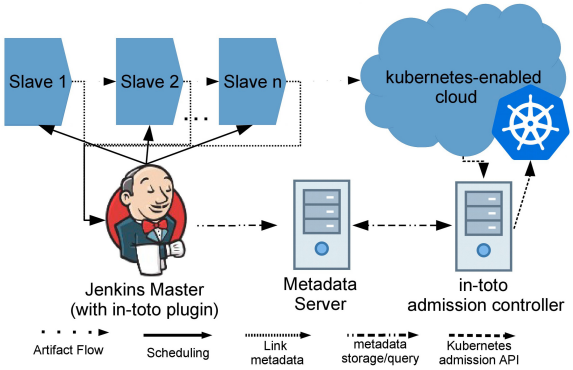


Figure 3: The kubesec supply chain.

6 Deployment

`in-toto` has about a dozen different integrations that protect software supply chains for millions of end users. This section uses three such integrations to examine how threshold signing, metadata generation, and end-to-end verification function in practical deployments of `in-toto`.

6.1 Debian rebuilder constellation

Debian is one of the biggest stakeholders in the reproducible builds project [26], an initiative to ensure bit-by-bit deterministic builds of a source package. One of the main motivations behind reproducible builds is to avoid backdooring compilers [136] or compromised toolchains in the Debian build infrastructure. `in-toto` helps Debian achieve this goal via its step-thresholding mechanism.

The apt-transport [16] for `in-toto` verifies the trusted rebuilder metadata upon installing any Debian package. Meanwhile, various institutions (that range from private to non-profit and educational) run rebuilder infrastructure to rebuild Debian packages independently and produce attestations of the resulting builds using `in-toto` link metadata. This way, it is possible to cryptographically assert that a Debian package has been reproducibly built by a set of k out of n rebuilders. Figure 2 shows a graphical description of this setup.

By using the `in-toto` verifiable transport, users can make sure that no package was tampered with unless an attacker is also able to compromise at least k rebuilders and the Debian buildfarm. Throughout this deployment, we were able to test the thresholding mechanism, as well as practical ways to bootstrap project owner signatures through the existing package manager trust infrastructure [32, 34].

Deployment insight. Through our interaction with reproducible builds, we were able to better understand how the thresholding mechanism can be used to represent concepts such as a build's reproducibility and how to build `in-toto` into operating-system tools to facilitate adoption.

6.2 Cloud native builds with Jenkins and Kubernetes

"Cloud native" is used to refer to container-based environments [3]. Cloud native ecosystems are characterized by rapid changes and constant re-deployment of the internal components. They are generally distributed systems, and often managed by container orchestration systems such as Kubernetes [23] or Docker Swarm [6]. Thus, their pipelines are mostly automated using pipeline managers such as Jenkins [18] or Travis [137]. In this type of ecosystems, a host- and infrastructure-agnostic, automated way to collect supply-chain metadata is necessary not only for security, but also for auditing and analyzing the execution of build processes that led to the creation of the delivered product.

In the context of cloud native applications, `in-toto` is used by Control Plane to track the build and quality-assurance steps on kubesec [19], a Kubernetes resource and configuration static analyzer. In order to secure the kubesec supply chain, we developed two `in-toto` components: a Jenkins plugin [11] and a Kubernetes admission controller [7, 17]. These two components allow us to track all operations within a distributed system, both of containers and aggregate `in-toto` link metadata, to verify any container image before it is provisioned. Figure 3 shows a (simplified) graphical depiction of their supply chain.

This deployment exemplifies an architecture for the supply chains of cloud native applications, in which new container images, serverless functions and many types of deployments are quickly updated using highly-automated pipelines. In this case, a pipeline serves as a coordinator, scheduling steps to worker nodes that serve as functionaries. These functionaries then submit their metadata to an `in-toto` metadata store. Once a new artifact is ready to be promoted to a cloud environment, a container orchestration system queries an `in-toto` admission controller. This admission controller ensures that every operation on this delivered product has been performed by allowed nodes and that all the artifacts were acted on according to the specification in the `in-toto` layout.

Deployment insight. Our interaction with kubesec forced us to investigate other artifact identifiers such as container images (in addition to files). While `in-toto` can be used today to track container images, the ability to point to an OCIv2 [21] image manifest can provide a more succinct link metadata representation and will be part of future work.

6.3 Datadog: E2E verification of Python packages

Datadog is a monitoring service for cloud-scale applications, providing monitoring of servers, databases, tools, and services, through a software-as-a-service-based data analytics platform [5]. It supports multiple cloud service providers, including Amazon Web Services (AWS), Microsoft Azure,

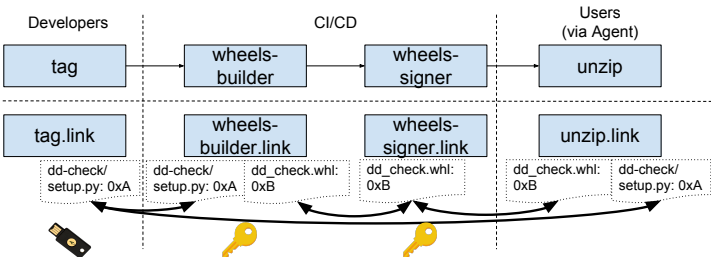


Figure 4: The simplified Datadog agent integrations supply chain. There are three steps (`tag` step, `wheels-builder` step, `wheels-signer` step), and one inspection. Arrows denote `MATCH` rules, the tag step is signed using a hardware dongle whereas the CI system uses online keys.

Google Cloud Platform, and Red Hat OpenShift. At the time of writing, it has over 8,000 customers, and collects trillions of monitoring record points per day.

The Datadog *agent* is software that runs on hosts. It collects events and metrics from hosts and sends them to Datadog, where customers can analyze their monitoring and performance data. The agent *integrations* are plug-ins that collect metrics from services running on customer infrastructure. Presently, there are more than one hundred integrations that come installed out-of-the-box with the Agent.

Datadog developers wanted an ability to automatically build and publish new or updated integrations independently of agent releases. This is so interested users can try new or updated integrations as they become available, and test whether they are applicable to their needs.

This section will cover how Datadog built the first tamper-evident pipeline using `in-toto` and how it leveraged TUF to safely bootstrap key distribution and provide replay-protection and freshness guarantees to in-toto metadata.

End-to-end verification with `in-toto`. The Datadog agent integrations supply chain, shown in Figure 4, has three steps:

1. The first `tag` step outputs Python source code as products. Every integration consists of Python source code and several YAML [133] configuration files. The link for this step is signed using a Yubikey hardware dongle [29]
2. In the second `wheels-builder` step, the pipeline must receive the same source code from the `tag` step and produce a Python wheel [24], as well as its updated Python metadata. Each wheel is a ZIP file and its metadata is an HTML file that points to all the available *versions* of an integration.
3. In the third `wheels-signer` step, the pipeline must receive, as materials, the same products as the `wheels-builder` step. This steps signs for all wheels using the system described in the next subsection. It can be dangerous packaging Python source code, because arbitrary code can be executed during the packaging process, which can be inserted by compromising the GitHub repository. Therefore, this step is separate from the `wheels-builder` step, so that a compromise of the former does not yield the signing keys of this step.

Finally, there is one inspection, which first ensures that a given wheel matches the materials of the `wheels-signer` step. It then extracts files from the wheel and checks that they correspond to exactly the same Python source code and YAML configuration files as the products of the `tag` step. Thus, this layout provides end-to-end verification: it prevents a compromised pipeline from causing users to trust wheels with source code that was never released by Datadog developers.

Transport with The Update Framework (TUF). Whereas `in-toto` provides end-to-end verification of the Datadog pipeline, it does not solve a crucial problem that arises in practice: How to securely distribute, revoke, and replace the public keys used to verify the `in-toto` layout. This mechanism must be *compromise-resilient* [100–102, 121], and resistant to a compromise of the software repository or server used to serve files. While SSL / TLS protects users from man-in-the-middle (MitM) attacks, it is not compromise-resilient, because attackers who compromise the repository can simply switch the public keys used to verify `in-toto` layout undetected, and thus defeat end-to-end verification. Likewise, other solutions, such as X509 certificates do not support necessary features such as in-band key revocation and key rotation.

The Update Framework (TUF) [100–102, 121] provides precisely this type of compromise-resilient mechanism, as well as in-band key revocation and key rotation. To do so, TUF adds a higher layer of signed metadata to the repository following two design principles that inspired the `in-toto` design. The first is the use of *roles* in a similar fashion to `in-toto`, so that a key compromise does not necessarily affect all *targets* (i.e., any Python wheels, or even `in-toto` metadata). The second principle is minimizing the risk of a key compromise using *offline* keys, or signing keys that are kept off the repository and pipeline in a cold storage mechanism, such as safe deposit boxes, so that attackers who compromise the infrastructure are unable to find these keys.

TUF is used within the Datadog integrations downloader to distribute, in a compromise-resilient manner, the: (1) root of trust for all wheels, TUF and `in-toto` metadata, (2) `in-toto` layout, and (3) public keys used to verify this layout. TUF also guarantees that MitM attackers cannot tamper with the consistency, authenticity, and integrity of these files, nor rollback or indefinitely replay `in-toto` metadata. This security model is simplified because it ignores some considerations that are out of the scope of this paper.

In summary, the Datadog pipeline uses TUF to appropriately bootstrap the root of the trust for the entire system, and `in-toto` to guarantee that the pipeline packaged exactly the source code signed by one of the Datadog developers inside universal Python wheels. By tightly integrating TUF and `in-toto`, Datadog's users obtain the compromise resilience of both systems combined.

Deployment insight. Through the Datadog deployment, we learned how to use other last-mile systems like TUF to provide not only compromise-resilience, but also replay-protection, freshness guarantees, and mix-and-match protection for in-toto metadata.

7 Evaluation

We evaluated in-toto's ability to guarantee software supply chain integrity on two fronts: efficiency and security. We set off to answer the following questions:

- Does in-toto incur reasonable overheads in terms of bandwidth, storage overhead and verification time?
- Can in-toto be used to protect systems against real-life supply chain compromises?

In order to answer the first question, we explored in-toto as used in the context of Datadog for two reasons: Datadog offers more than 111 integration packages to verify with in-toto, and its data and source code is publicly available. Furthermore, it is a production-ready integration that can be used by Datadog's more than 8,000 clients today [31]. Their clients include major companies like Twitter, NASDAQ and The Washington Post [4].

Then, we surveyed historical supply chain compromises and catalogued them. We evaluated these compromises against the in-toto deployments described in Section 6, accounting for their supply chain configuration, and including the actors involved and their possible key assignments. By studying the nature of each compromise, we were able to estimate what degree of key compromise could hypothetically happen and, with it, the consequences of such a compromise on these supply chains when in-toto is in place.

7.1 in-toto's overhead in the Datadog deployment

In the three major costs that in-toto presents are the storage, transfer and verification cost. In order to explore these costs, we set out to use the publicly available Datadog agent integration client and software repository. From this data, we can derive the cost of storing in-toto metadata in the repository, the cost of transferring the in-toto metadata for any given package and the verification times when installing any package.

Storage overhead. In order to understand the storage overhead, we mirrored the existing agent integrations Python package repository. Then, we inspected the package payloads and the repository metadata to show the cost breakdown of the repository as a whole. Table 2 depicts the cost breakdown of the Datadog repository, as mirrored on February 8 of 2019.

Type	total package	Python metadata	TUF	in-toto links	in-toto Layout
RSA 4096	74.02%	0.83%	5.51%	16.75%	2.89%
DSA 1024 & ed25519	74.48%	0.84%	5.54%	16.35%	2.79%
optimized	79.56%	0.90%	5.92%	10.65%	2.97%

Table 2: Storage overhead breakdown for a in-toto enabled package repository. All metadata is compressed using zlib.

Table 2 shows that in-toto takes up about 19% of the total repository size, and thus makes it a feasible solution in terms of storage overhead. In addition, compared to its co-located security system TUF, the cost of using in-toto is higher, with almost four times the metadata storage cost. Further, the breakdown also indicates that the governing factor for this storage overhead are the in-toto links, rather than the layout file, with a layout being approximately 6 to 3 times smaller than the links (42 KB in comparison of the 148KB for all the link metadata).

There are two main reasons that drive this cost. First and foremost, is the engineering decision to track all the files within the pipeline (including Python metadata). Although these are not required to be tracked with in-toto, for the sake of security (as this type of metadata is being protected by TUF), it eases the implementation at a manageable cost. The second is that of signatures: the signatures used within the Datadog deployment come from either 4096-bit openpgp keys on a Yubikey, or 4096-bit PEM keys. These alone account for almost half of the in-toto metadata size.

For this reason, it is possible to further reduce the size of the metadata to 13% of the total repository size. Rows two and three of Table 2 represent the repository overhead when limiting the amount of files tracked to only Python sources and packages and using a DSA1024 as the signing algorithm.

Network overhead. Similar to storage overhead, the network overhead incurred by clients when installing any integration is of utmost importance. To explore this cost, we investigate the raw package sizes and contrast it with the size of the package-specific metadata size. It is of note though, that the size of in-toto metadata does not scale with the size of the package, but rather the number of files inside of it. This is because most of the metadata cost is taken by pieces of link metadata, of which the biggest three fields are the signature, expected_materials and expected_products. Figure 5 shows both the distribution of package sizes and the distribution of metadata sizes in increasing order.

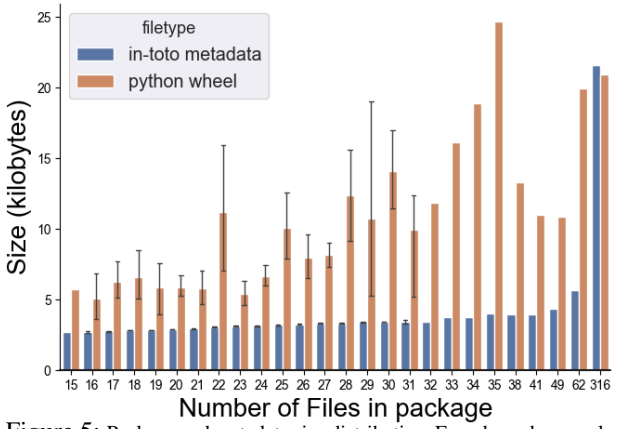


Figure 5: Package and metadata size distribution. Error bars show packages with the same number of files but different sizes.

In Figure 5 we can see that, for most packages, the metadata size cost is below 44% of the package size. In fact for the 90th percentile, the metadata cost approaches a costly 64%, to a worst case of 103%. However, upon inspecting these cases, we found that the issue is that these are cases in which *link metadata is tracking files not contained in the delivered product*. Indeed, in-toto is tracking files, such as test suites, fixtures and even iconography that does not get packaged on the integrations Python wheel. The worst case scenario is in fact cisco_aci, which only packages 12 files out of 316 contained in the tag step metadata.

Verification overhead. Finally, to draw insight from the computation time required to verify each package, we ran a series of micro-benchmarks on a laptop with an Intel i7-6500U processor and 8GB of RAM. In this case, we ran an iterated verification routine with the packages already fetched and instrumented the Datadog agent installer to measure the installation time with and without `in-toto` verification.

From this experiment, we conclude that `in-toto` verification adds less than 0.6 seconds on all cases. This is mostly dominated by the signature verification, and is thus bounded by the number of links to verify (i.e., the number of steps times the threshold).

7.2 Supply chain data breaches

We surveyed 30 major different supply chain breaches and incidents occurring from January 2010 to January 2019 (this list of historical attacks is included in Appendix B). These historical incidents cover a variety of software products and platforms, such as Apple's Xcode [113], Android GTK [8], MeDoc financial software [35], Adobe updater [95], PHP PEAR repository [33], and South Korean organizations [138].

Studying these historical attacks identified the type of access that the attacker had (or was speculated to have) and identified three categories: the attacker had control of infrastructure (but not functionary keys), the attacker had control over part of the infrastructure or keys of a specific functionary, and the attacker was able to control the entire supply chain by compromising a project owner's infrastructure (including their signing key).

For the historical attacks in Appendix B, we determined whether an attack used a compromised key, and then labeled those attacks with "Key Compromise". We also determined the degree of access in the attack (all the way to the possible step) and labeled each attack with an "Access Level" that indicates the step in the chain where the attack took place.

We now analyze how these compromises could affect the three supply chains where `in-toto` was deployed (as described in Section 6). Our analysis indicates that the majority of attacks (23 out of 30) took place without any key compromise. In those cases, none of the three `in-toto` deployments would have been affected since the client inspection (as described in Sec. 4.3) could detect extraneous artifacts or malicious delivered products.

Out of the 30 studied incidents, 7 involved a key compromise. We summarize our analysis of these attacks in Table 3. One attack, Keydnap [71], used a stolen Apple developer certificate to sign the malicious software package. Therefore, this attack would not have affected any `in-toto` deployments, because `in-toto` would detect that an unauthorized functionary signed the link metadata. Another attack used the developer's ssh key to upload a malicious Python package on PyPI [52]. All `in-toto` deployments could have detected this attack since files extracted from the malicious package would not exactly match the source code as the products of the first step in the supply chain.

The remaining five attacks involving a key compromise were recent sophisticated incidents that affected many clients and companies. The CCleaner [126] and RedHat [125] attacks are not effective against the Reproducible Builds deployment (RB) and Datadog (DD), as the former implements a threshold mechanism in the build step and the latter does not build binaries in their infrastructure. In a similar flavor, three attacks (Operation Red [138], NotPetya [35], and KingSlayer [118]) would not affect the Datadog deployment, as it implements a threshold mechanism in the packaging step. The Cloud Native deployment, on the other hand, would detect none of these five attacks, as it does not employ thresholds. To conclude, the `in-toto` deployments would detect most of the historical attacks based on the general `in-toto` design principles. For those attacks that involve key compromises, our analysis shows that `in-toto`'s use of thresholds is an effective mechanism.

Attack Name	DD	RB	CN
Keydnap [71]	✓	✓	✓
backdoored-pypi [52]	✓	✓	✓
CCleaner Atatck [126]	✓	✓	✗
RedHat breach [125]	✓	✓	✗
*NotPetya [35]	✓	✗	✗
Operation Red [138]	✓	✗	✗
KingSlayer [118]	✓	✗	✗

Table 3: The impact of the historical attacks on the three `in-toto` deployments: Datadog (DD), Reproducible Builds (RB), Cloud Native (CN). Out of the 30 historical attacks, 23 did not involve a key compromise, so none of the deployments would have been affected. This table shows the remaining attacks which involved a key compromise. In one attack, marked with a star (*), it is unknown if a key compromise took place. We assumed that was the case. A ✓ indicates that the deployment could have detected the attack.

Key Takeaway. Cloud Native (83%) and reproducible builds (90%) integrations of `in-toto` would prevent most historical supply chain attacks. However, integration into a secure update system as was done by Datadog (100%) provides further protection.

8 Related Work

To the best of our knowledge, work that attempts to use an automated tool to secure the supply chain is scarce. However, there has been a general push to increase the security of different aspects within the supply chain, as well as to tighten the binding between neighboring processes within that chain. In this section, we mention work relevant to supply chain security, as some of it is crucial for the success of `in-toto` as a framework. We also list work that can further increase the security guarantees offered by `in-toto`.

Automated supply chain administration systems. Configuring and automating processes of the supply chain has been widely studied. Works by Bégin et al. [45], Banzai et al., [43] and Andreetto et al. [36] focus on designing supply chains that automatically assign resources and designate parties to take part in different processes to create a product. This work is similar to `in-toto` in that it requires a supply chain topology to carry out the work. However, none of these projects were focused on security. Instead, they deal with adaptability of resources and supply chain automation.

Perhaps most closely related to in-toto is the Grafeas API [9] released by Google. However, Grafeas's focus is on tracking and storing supply chain metadata rather than security. Grafeas provides a centralized store for supply chain metadata, which can be later queried by verification tools such as Google's Binary Authorization [84]. Grafeas does not provide a mechanism to describe what steps should be performed, validate performed steps, or even support cryptographic signatures [1]. Finally, in-toto is architecture agnostic, while Grafeas is mostly cloud-native; in-toto was geared to represent supply chains whether they are cloud-native, off-cloud or hybrid-cloud. We are collaborating with the Grafeas team to natively support in-toto link metadata within Grafeas [10].

Software supply chain security. In addition, many software engineering practices have been introduced to increase the security of the software development lifecycle [42, 104, 105, 111, 116]. Additional work by Devanbu et al. [67] has explored different techniques to "construct safe software that inspires trust in hosts." These techniques are similar to in-toto in that they suggest releasing supply chain information to the end users for verification.

Though none of these proposals suggest an automated tool to ensure the integrity of the supply chain, they do serve as a helpful first step in designing in-toto. As such, their practices could be used as templates for safe supply chain layouts.

Finally, there have been hints by the industry to support features that could be provided by in-toto [90, 114, 145]. This includes providing certificates noting the presence of a process within the supply chain and providing software transparency through mechanisms such as reproducible builds.

Source control security. The source code repository is usually the first link in the supply chain. Early work in this field has explored the different security properties that must be included in software configuration management tools [63]. Version control systems, such as Git, incorporate protection mechanisms to ensure the integrity of the source code repository, which include commit hash chaining and signed commits [77, 78].

Buildsystem and verification security. The field of automated testing and continuous integration has also received attention from researchers. Recently, self-hosted and public automated testing and continuous integration systems have become popular [54, 72, 137]. Work by Gruhn et al. [85] has explored the security implications of malicious code running on CI systems, showing that it is possible for attackers to affect other projects being tested in the same server, or the server itself. This work, and others [69] serve as a motivation for in-toto's threat model.

Further work by Hanawa et al. [87] explores different techniques for automated testing in distributed systems. The work is similar to in-toto in that it allocates hosts in the cloud to automatically run tests for different environments and platforms. However, in-toto requires such systems to provide certification (in the form of link metadata) that the tests were run and the system was successful.

Subverting the development environment, including subverting the compiler, can have a serious impact on the software supply chain [135]. Techniques such as Wheeler's diverse double-compiling (DDC) [144] can be used to mitigate such "trusting trust" attacks. In the context of reproducible builds project, DDC can also be used for multi-party verification of compiler executables.

Verifying compilers, applications and kernels. Ongoing work on verifying compilers, applications and kernels will provide a robust framework for applications that fully comply with their specification [88, 98]. Such work is similar to in-toto in that a specification is provided for the compiler to ensure that their products meet stated requirements. However, in contrast to our work, most of this work is not intended to secure the origin of such specification, or to provide any proof of the compilation's results to steps further down the supply chain. Needless to say, verifying compilers could be part of a supply chain protected with in-toto.

Furthermore, work by Necula et al. introduces proof-carrying code [109, 110], a concept that relies on the compiler to accompany machine code with proof for verification at runtime. Adding to this, industry standards have included machine code signing [40] to be verified at runtime. This work is similar to in-toto in that compilers generate information that will be verified by the end user upon runtime. Although these techniques are more granular than in-toto's (runtime verification vs verification upon installation), they do not aim to secure the totality of the supply chain.

Package management and software distribution security. Work by Cappos et al. has been foundational to the design of in-toto's security mechanisms [46, 102, 121]. The mechanisms used to secure package managers are similar to in-toto in that they rely on key distribution and role separation to provide security guarantees that degrade with partial key compromise. However, unlike in-toto, these systems are restricted to software updates, which limit their scope. Concepts from this line of work could be overlaid on in-toto to provide additional "last mile" guarantees for the resulting product, such as package freshness or protection against dependencies that are not packaged with the delivered product.

9 Conclusions and future work

In this paper, we have described many aspects of in-toto, including its security properties, workflow and metadata. We also explored and described several extensions and implications of using in-toto in a number of real-world applications. With this we have shown that protecting the entirety of the supply chain is possible, and that it can be done automatically by in-toto. Further, we showed that, in a number of practical applications, in-toto is a practical solution to many contemporary supply chain compromises.

Although plenty of work needs to be done in the context of the in-toto framework (e.g., decreasing its storage cost), tackling the first major limitations of supply chain security will increase the quality of software products. We expect that, through continued interaction with the industry and

elaborating on the framework, we can provide strong security guarantees for future software users.

Acknowledgments

We would like to thank the USENIX reviewers and Luke Valenta for reviewing this paper. We would also like to thank Lukas Pühringer and Lois DeLong from the `in-toto` team; Holger Levsen, Chris Lamb, kpcyrd, and Morten Linderud from Reproducible Builds; the Datadog Agent Integrations (especially Ofek Lev) and Product Security teams; as well as Andrew Martin and Luke Bond from Control Plane for their valuable work towards integrating `in-toto` in all these communities. This research was supported by the NSF under Grants No. CNS 1801430 and DGE 1565478.

References

[1] Add Signature message to v1beta common.proto. #253. https://github.com/grafeas/grafeas/pull/253.

[2] Apt. https://wiki.debian.org/Apt.

[3] Cloud native computing foundation. https://www.cncf.io/.

[4] Customers | Datadog. https://www.datadoghq.com/customers/.

[5] Datadog: Modern monitoring & analytics. https://www.datadoghq.com/.

[6] Docker Swarm overview. https://docs.docker.com/swarm/overview/.

[7] Dynamic admission control. https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/.

[8] ExpensiveWall: A Dangerous Packed Malware On Google Play. https://blog.checkpoint.com/2017/09/14/expensivewall-dangerous-packed-malware-google-play-will-hit-wallet/.

[9] Grafeas. https://grafeas.io/.

[10] Grafeas + in-toto. https://github.com/in-toto/totoify-grafeas.

[11] in-toto Jenkins plugin. https://plugins.jenkins.io/in-toto.

[12] in-toto layout creation tool. https://in-toto.engineering.nyu.edu.

[13] in-toto Metadata Examples. https://in-toto.github.io/metadata-examples.html.

[14] in-toto Specification: Version 0.9. https://github.com/in-toto/docs/blob/v0.9/in-toto-spec.md.

[15] in-toto Specifications. https://in-toto.github.io/specs.html.

[16] in-toto transport for apt. https://github.com/in-toto/apt-transport-in-toto.

[17] in-toto-webhook. https://github.com/SantiagoTorres/in-toto-webhook.

[18] Jenkins: Build great things at any scale. https://jenkins.io/.

[19] Kubesec.io: Quantify risk for kubernetes resources. https://kubesec.io/.

[20] Notary. https://docs.docker.com/samples/library/notary/.

[21] Oci image format specification. https://github.com/opencontainers/image-spec.

[22] Operation Aurora. https://en.wikipedia.org/wiki/Operation_Aurora.

[23] Production-Grade Container Orchestration. https://kubernetes.io/.

[24] Python Wheels. https://pythonwheels.com/.

[25] Reproducible builds. https://reproducible-builds.org/.

[26] Reproducible builds: Who is involved? https://reproducible-builds.org/who/.

[27] Some Debian Project machines compromised. https://www.debian.org/News/2003/20031121.

[28] The Update Framework (TUF). https://theupdateframework.github.io/.

[29] The YubiKey. https://www.yubico.com/products/yubikey-hardware/.

[30] Twistlock: Cloud Native Security for Docker, Kubernetes and Beyond. https://www.twistlock.com/.

[31] Forbes Cloud 100: #19 Datadog, 2018. https://www.forbes.com/companies/datadog/?list=cloud100#3cad45279e03.

[32] in-toto at the reproducible builds summit-paris 2018, 2019. https://ssl.engineering.nyu.edu/blog/2019-01-18-in-toto-paris.

[33] *PHP PEAR Software Supply Chain Attack*, 2019. https://blog.dcso.de/php-pear-software-supply-chain-attack/.

[34] Reproducible builds: Weekly report #196, 2019. https://reproducible-builds.org/blog/posts/196/.

[35] A. Cherepanov. *Analysis of TeleBots' cunning backdoor*. https://www.welivesecurity.com/2017/07/04/analysis-of-telebots-cunning-backdoor.

[36] P. Andreetto, S. Andreozzi, G. Avellino, S. Beco, A. Cavallini, M. Cecchi, V. Ciaschini, A. Dorise, F. Giacomini, A. Gianelle, et al. The glite workload management system. In *J. of Physics: Conf. Series*, volume 119, page 062007. IOP Publishing, 2008.

[37] Andy Greenberg. *MacOS Update Accidentally Undoes Apple's "Root" Bug Patch*. https://www.wired.com/story/macos-update-undoes-apple-root-bug-patch/.

[38] Apache Infrastructure Team. apache.org incident report for 8/28/2009. https://blogs.apache.org/infra/entry/apache_org_downtime_report, 2009.

[39] Apache Infrastructure Team. apache.org incident report for 04/09/2010. https://blogs.apache.org/infra/entry/apache_org_04_09_2010, 2010.

[40] Apple Computers. iOS Security Guide, 2016. https://www.apple.com/business/docs/iOS_Security_Guide.pdf.

[41] B. Arkin. Adobe to Revoke Code Signing Certificate. https://blogs.adobe.com/conversations/2012/09/adobe-to-revoke-code-signing-certificate.html, 2012.

[42] R. Bachmann and A. D. Brucker. Developing secure software. *Datenschutz und Datensicherheit*, 38(4):257–261, 2014.

[43] T. Banzai, H. Koizumi, R. Kanbayashi, T. Imada, T. Hanawa, and M. Sato. D-cloud: Design of a software testing environment for reliable distributed systems using cloud computing technology. In *Proc. of the 10th IEEE/ACM CCGrid*, 2010.

[44] Barb Darrow. *Adobe source code breach; it's bad, real bad*. https://gigaom.com/2013/10/04/adobe-source-code-breech-its-bad-real-bad.

[45] M.-E. Bégin, G. D.-A. Sancho, A. Di Meglio, E. Ferro, E. Ronchieri, M. Selmi, and M. Żurek. Build, configuration, integration and testing tools for large software projects: Etics. In *Rapid Integration of Software Engineering Techniques*, pages 81–97. Springer, 2006.

[46] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman. A look in the mirror: Attacks on package managers. In *Proc. of the 15th ACM CCS*, pages 565–574, 2008.

[47] S. Checkoway, S. Cohney, C. Garman, M. Green, N. Heninger, J. Maskiewicz, E. Rescorla, H. Shacham, and R.-P. Weinmann. A systematic analysis of the juniper dual ec incident. Cryptology ePrint Archive, Report 2016/376, 2016. http://eprint.iacr.org/.

[48] S. Checkoway, J. Maskiewicz, C. Garman, J. Fried, S. Cohney, M. Green, N. Heninger, R. P. Weinmann, E. Rescorla, and H. Shacham. A Systematic Analysis of the Juniper Dual EC Incident. In *Proc. of ACM CCS '16*, 2016.

[49] R. Chirgwin. *Microsoft deletes deleterious file deletion bug from Windows 10*. https://www.theregister.co.uk/2018/10/10/microsoft_windows_deletion_bug/.

[50] A. Chitu. The Android Bug 8219321. https://googlesystem.blogspot.com/2013/07/the-8219321-android-bug.html#gsc.tab=0, 2013.

[51] Christian Nutt. *Cloud source host Code Spaces hacked, developers lose code*. http://www.gamasutra.com/view/news/219462/Cloud_source_host_Code_Spaces_hacked_developers_lose_code.php.

[52] C. Cimpanu. *Backdoored Python Library Caught Stealing SSH Credentials*, 2018. https://www.bleepingcomputer.com/news/security/backdoored-python-library-caught-stealing-ssh-credentials/.

[53] C. Cimpanu. *Microsoft Discovers Supply Chain Attack at Unnamed Maker of PDF Software*, 2018. https://www.bleepingcomputer.com/news/security/microsoft-discovers-supply-chain-attack-at-unnamed-maker-of-pdf-software/.

[54] Codeship. Continuous Delivery with Codeship: Fast, Secure, and fully customizable. https://codeship.com/.

[55] Context Threat Intelligence. *Threat Advisory: The Monju Incident*, 2014. https://paper.seebug.org/papers/APT/APT_CyberCriminal_Campagin/2014/The_Monju_Incident.pdf.

[56] M. Coppock. *Windows Update not working after October 2018 patch? Here's how to fix it* . https://www.digitaltrends.com/computing/windows-update-not-working/.

[57] J. Corbet. An attempt to backdoor the kernel. http://lwn.net/Articles/57135/, 2003.

[58] J. Corbet. The cracking of kernel.org. http://www.linuxfoundation.org/news-media/blogs/browse/2011/08/cracking-kernelorg, 2011.

[59] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti. A large-scale analysis of the security of embedded firmwares. In *Proc. of the 23rd USENIX Security Symposium*, pages 95–110, 2014.

[60] CrowdStrike. *Securing the supply chain*. https://www.crowdstrike.com/resources/wp-content/brochures/pr/CrowdStrike-Security-Supply-Chain.pdf.

[61] A. Cui, M. Costello, and S. J. Stolfo. When firmware modifications attack: A case study of embedded exploitation. In *NDSS*, 2013.

[62] Dan Goodin. Kernel.org Linux repository rooted in hack attack. http://www.theregister.co.uk/2011/08/31/linux_kernel_security_breach/.

[63] David A. Wheeler. Software Configuration Management Security. http://www.dwheeler.com/essays/scm-security.html.

[64] Debian. Debian Investigation Report after Server Compromises. https://www.debian.org/News/2003/20031202.

[65] Debian. Security breach on the Debian wiki 2012-07-25. https://wiki.debian.org/DebianWiki/SecurityIncident2012, 2012.

[66] Dennis Fisher. *Researcher Finds Tor Exit Node Adding Malware to Binaries*. https://threatpost.com/researcher-finds-tor-exit-node-adding-malware-to-binaries/109008/.

[67] P. T. Devanbu, P. W. Fong, and S. G. Stubblebine. Techniques for trusted software engineering. In *Proceedings of the 20th international conference on Software engineering*, pages 126–135. IEEE Computer Society, 1998.

[68] Dona Sarkar. *A note about the unintentional release of builds today*. https://blogs.windows.com/windowsexperience/2017/06/01/note-unintentional-release-builds-today/.

[69] P. M. Duvall, S. Matyas, and A. Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[70] Edward Iskra. *Vulnerable Wallets and the Suspicious File*, 2017. https://bitcoingold.org/vulnerable-wallets/.

[71] ESET Research. *OSX/Keydnap spreads via signed Transmission application*. https://www.welivesecurity.com/2016/08/30/osxkeydnap-spreads-via-signed-transmission-application/.

[72] B. Fitzgerald and K.-J. Stol. Continuous software engineering and beyond: trends and challenges. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, pages 1–9. ACM, 2014.

[73] A. Forums. Numix gnome 3.20. https://bbs.archlinux.org/viewtopic.php?id=211164.

[74] J. Freeman. Yet Another Android Master Key Bug. http://www.saurik.com/id/19, 2014.

[75] P. W. Frields. Infrastructure report, 2008-08-22 UTC 1200. https://www.redhat.com/archives/fedora-announce-list/2008-August/msg00012.html, 2008.

[76] Gentoo Linux. *Incident Reports/2018-06-28 Github*. https://wiki.gentoo.org/wiki/Project:Infrastructure/Incident_Reports/2018-06-28_Github.

[77] M. Gerwitz. A Git Horror Story: Repository Integrity With Signed Commits. http://mikegerwitz.com/papers/git-horror-story.

[78] Git SCM. Signing your work. https://git-scm.com/book/en/v2/Git-Tools-Signing-Your-Work.

[79] GitHub, Inc. Public Key Security Vulnerability and Mitigation. https://github.com/blog/1068-public-key-security-vulnerability-and-mitigation, 2012.

[80] GNU Savannah. Compromise2010. https://savannah.gnu.org/maintenance/Compromise2010/.

[81] D. Goodin. Fedora servers breached after external compromise. http://www.theregister.co.uk/2011/01/25/fedora_server_compromised/.

[82] D. Goodin. *Meet "Great Cannon", the man-in-the-middle weapon China used on GitHub*. https://arstechnica.com/security/2015/04/meet-great-cannon-the-man-in-the-middle-weapon-china-used-on-github/.

[83] D. Goodin. Attackers sign malware using crypto certificate stolen from Opera Software. http://arstechnica.com/security/2013/06/attackers-sign-malware-using-crypto-certificate-stolen-from-opera-software/, 2013.

[84] Google. Binary Authorization. https://cloud.google.com/binary-authorization/.

[85] V. Gruhn, C. Hannebauer, and C. John. Security of public continuous integration services. In *Proc. of the 9th ACM International Symposium on Open Collaboration*, page 15, 2013.

[86] Hackread. *Proton malware*. https://www.hackread.com/hackers-infect-mac-users-proton-malware-using-elmedia-player.

[87] T. Hanawa, T. Banzai, H. Koizumi, R. Kanbayashi, T. Imada, and M. Sato. Large-scale software testing environment using cloud computing technology for dependable parallel and distributed systems. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 428–433. IEEE, 2010.

[88] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: End-to-end security via automated full-system verification. In *Proc. of the 11th USENIX OSDI*, pages 165–181, 2014.

[89] Idrees Patel. *Janus Vulnerability*. https://www.xda-developers.com/janus-vulnerability-android-apps.

[90] ISO/IEC JTC 1/SC 27 IT Security techniques. *ISO/IEC 27034:2011 Information technology – Security techniques – Application security*. https://www.iso.org/standard/44378.html?browse=tc.

[91] Jane Silber. Notice of Ubuntu Forums breach. https://blog.ubuntu.com/2016/07/15/notice-of-security-breach-on-ubuntu-forums.

[92] Jared Newman. Gauss Malware: What You Need to Know. https://www.pcworld.com/article/260735/gauss_malware_what_you_need_to_know.html.

[93] Jeff Erickson. Inside OilRig – Tracking Iran's Busiest Hacker Crew On Its Global Rampage. https://www.forbes.com/sites/thomasbrewster/2017/02/15/oilrig-iran-hackers-cyberespionage-us-turkey-saudi-arabia/#5415a493468a.

[94] Jensen Beeler. Millions of Motorcyclists Hacked in VerticalScope Breach. https://www.asphaltandrubber.com/news/verticalscope-hack/.

[95] Jeremy Kirk. *New malware overwrites software updaters*, 2010. https://www.itworld.com/article/2755831/security/new-malware-overwrites-software-updaters.html.

[96] Juniper. 2015-12 Out of Cycle Security Bulletin: ScreenOS: Multiple Security issues with ScreenOS (CVE-2015-7755, CVE-2015-7756). https://kb.juniper.net/InfoCenter/index?page=content&id=JSA10713, Dec. 15.

[97] G. Kelly. *Apple iOS 12.1.4 Fails To Fix Cellular, WiFi Problems*. https://www.forbes.com/sites/gordonkelly/2019/02/10/apple-ios-12-1-4-problem-iphone-cellular-data-wifi-upgrade-ipad/.

[98] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. sel4: Formal verification of an os kernel. In *Proc. of the 22nd ACM SOSP*, pages 207–220, 2009.

[99] B. Kuhn. News: IMPORTANT: Information Regarding Savannah Restoration for All Users. https://savannah.gnu.org/forum/forum.php?forum_id=2752, 2003.

[100] T. K. Kuppusamy, A. Brown, S. Awwad, D. McCoy, R. Bielawski, C. Mott, S. Lauzon, A. Weimerskirch, and J. Cappos. Uptane: Securing software updates for automobiles. *14th ESCAR Europe*, 2016.

[101] T. K. Kuppusamy, V. Diaz, and J. Cappos. Mercury: Bandwidth-effective prevention of rollback attacks against community repositories. In *Proc. of the 2017 USENIX Annual Technical Conference*, 2017.

[102] T. K. Kuppusamy, S. Torres-Arias, V. Diaz, and J. Cappos. Diplomat: using delegations to protect community repositories. In *proc. of the 13th USENIX NSDI*, pages 567–581, 2016.

[103] Martin Brinkmann. *Attention: Some Fosshub downloads compromised*. https://www.ghacks.net/2016/08/03/attention-fosshub-downloads-compromised/.

[104] M. S. Merkow and L. Raghavan. Secure and resilient software: Requirements, test cases, and testing methods. 2011.

[105] Microsoft. Microsoft secure development lifecycle. https://www.microsoft.com/en-us/sdl/default.aspx.

[106] Microsoft. Microsoft Trusted Publishers Certificate Store. https://msdn.microsoft.com/en-us/library/windows/hardware/ff553504(v=vs.85).aspx.

[107] M. Mullenweg. Passwords Reset. https://wordpress.org/news/2011/06/passwords-reset/, 2011.

[108] Naked Security. Flame malware used man-in-the-middle attack against Windows Update. https://nakedsecurity.sophos.com/2012/06/04/flame-malware-used-man-in-the-middle-attack-against-windows-update/.

[109] G. C. Necula. Proof-carrying code. In *Proceedings of the ACM SIGPLAN*, 1997.

[110] G. C. Necula. *Proof-carrying code. design and implementation*. Springer, 2002.

[111] I. S. Organization. Information technology – security techniques – application security – part 1: Overview and concepts. http://www.iso.org/iso/catalogue_detail.htm?csnumber=44378.

[112] Patrick Gray. Gentoo Linux server compromised. https://www.zdnet.com/article/gentoo-linux-server-compromised/, 2003.

[113] D. Pauli. icloud phishing attack hooks 39 ios apps and wechat. theregister, 2015. https://www.theregister.co.uk/2015/09/21/icloud_phishing_attack_hooks_39_ios_apps_most_popular_message_client/.

[114] S. Quirolgico, J. Voas, T. Karygiannis, C. Michael, and K. Scarfone. *Vetting the Security of Mobile Applications*. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-163.pdf.

[115] Red Hat, Inc. Infrastructure report, 2008-08-22 UTC 1200. https://rhn.redhat.com/errata/RHSA-2008-0855.html, 2008.

[116] J. Robbins. Adopting open source software engineering (OSSE) practices by adopting OSSE tools. *Perspectives on free and open source software*, pages 245–264, 2005.

[117] RODRIGO ARANGUA. The security flaws at the heart of the Panama Papers. https://www.wired.co.uk/article/panama-papers-mossack-fonseca-website-security-problems.

[118] RSA Research. *Kingslayer-A Supply Chain Attack*. https://www.rsa.com/content/dam/premium/en/white-paper/kingslayer-a-supply-chain-attack.pdf.

[119] RubyGems.org. Data Verification. http://blog.rubygems.org/2013/01/31/data-verification.html, 2013.

[120] Ryan Naraine. *Open-source ProFTPD hacked, backdoor planted in source code*. http://www.zdnet.com/article/open-source-proftpd-hacked-backdoor-planted-in-source-code.

[121] J. Samuel, N. Mathewson, J. Cappos, and R. Dingledine. Survivable key compromise in software update systems. In *Proc. of the 17th ACM CCS*, pages 61–72. ACM, 2010.

[122] Slashdot Media. phpMyAdmin corrupted copy on Korean mirror server. https://sourceforge.net/blog/phpmyadmin-back-door/, 2012.

[123] J. K. Smith. Security incident on Fedora infrastructure on 23 Jan 2011. https://lists.fedoraproject.org/pipermail/announce/2011-January/002911.html, 2011.

[124] Steve Klabnik. *Security advisory for crates.io, 2017-09-19*. https://users.rust-lang.org/t/security-advisory-for-crates-io-2017-09-19/12960.

[125] Steven J. Vaughan-Nichols. *Red Hat's Ceph and Inktank code repositories were cracked*. http://www.zdnet.com/article/red-hats-ceph-and-inktank-code-repositories-were-cracked.

[126] Swati Khandelwal. CCleaner Attack Timeline–Here's How Hackers Infected 2.3 Million PCs. https://thehackernews.com/2018/04/ccleaner-malware-attack.html, 2018.

[127] Symantec. W32.Duqu: The precursor to the next Stuxnet. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_duqu_the_precursor_to_the_next_stuxnet.pdf.

[128] Symantec. W32.Stuxnet Dossier. https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_

dossier.pdf.
[129] Symantec Corporation. Internet threat security report, 2018. https://www.symantec.com/content/dam/symantec/docs/reports/istr-23-2018-en.pdf.
[130] The FreeBSD Project. FreeBSD.org intrusion announced November 17th 2012. http://www.freebsd.org/news/2012-compromise.html, 2012.
[131] The PHP Group. php.net security notice. http://www.php.net/archive/2011.php#id2011-03-19-1, 2011.
[132] The PHP Group. A further update on php.net. http://php.net/archive/2013.php#id2013-10-24-2, 2013.
[133] The YAML Project. The Official YAML Web Site. https://yaml.org/, 2019.
[134] Thomas Reed. *HandBrake hacked to drop new variant of Proton malware*. https://blog.malwarebytes.com/threat-analysis/mac-threat-analysis/2017/05/handbrake-hacked-to-drop-new-variant-of-proton-malware/.
[135] Thomas Reed. XcodeGhost malware infiltrates App Store. https://blog.malwarebytes.com/cybercrime/2015/09/xcodeghost-malware-infiltrates-app-store/.
[136] K. Thompson. Reflections on Trusting Trust. http://cm.bell-labs.com/who/ken/trust.html.
[137] Travis CI. Travis CI – test and deploy your code with confidence. https://travis-ci.org/.
[138] Trend Micro Cyber Safety Solutions Team. *Supply Chain Attack Operation Red Signature Targets South Korean Organizations*, 2018. https://blog.trendmicro.com/trendlabs-security-intelligence/supply-chain-attack-operation-red-signature-targets-south-korean-organizations/.
[139] Trend Micro Cyber Safety Solutions Team. *New Magecart Attack Delivered Through Compromised Advertising Supply Chain*, 2019. https://blog.trendmicro.com/trendlabs-security-intelligence/new-magecart-attack-delivered-through-compromised-advertising-supply-chain/.
[140] W. Verduzu. Xposed Patch for Master Key and Bug 9695860 Vulnerabilities. https://www.xda-developers.com/xposed-patch-for-master-key-and-bug-9695860-vulnerabilities/, 2013.
[141] L. Voss. Newly Paranoid Maintainers. http://blog.npmjs.org/post/80277229932/newly-paranoid-maintainers, 2014.
[142] T. Warren. *Major new iOS bug can crash iPhones*. https://www.theverge.com/2018/2/15/17015654/apple-iphone-crash-ios-11-bug-imessage.
[143] F. Weimer. CVE-2013-6435. https://access.redhat.com/security/cve/CVE-2013-6435, 2013.
[144] D. A. Wheeler. Fully countering trusting trust through diverse double-compiling. *arXiv preprint arXiv:1004.5534*, 2010.
[145] Yan/Bcrypt. Software transparency: Part 1. https://zyan.scripts.mit.edu/blog/software-transparency/.
[146] Zack Whittaker. Hacker explains how he put 'backdoor' in hundreds of linux mint downloads. http://www.zdnet.com/article/hacker-hundreds-were-tricked-into-installing-linux-mint-backdoor.
[147] K. Zetter. *'Google' Hackers had ability to alter source code'*. https://www.wired.com/2010/03/source-code-hacks.

A `in-toto` artifact rule definition

The following artifact rule definition is taken from the `in-toto` specification v0.9 [14].

- `ALLOW`: indicates that artifacts matched by the pattern are allowed as materials or products of this step.
- `DISALLOW`: indicates that artifacts matched by the pattern are not allowed as materials or products of this step.
- `REQUIRE`: indicates that a pattern must appear as a material or product of this step.
- `CREATE`: indicates that products matched by the pattern must not appear as materials of this step.
- `DELETE`: indicates that materials matched by the pattern must not appear as products of this step.
- `MODIFY`: indicates that products matched by the pattern must appear as materials of this step, and their hashes must not by the same.
- `MATCH`: indicates that the artifacts filtered in using `source-path-prefix/pattern` must be matched to a "MATERIAL" or "PRODUCT" from a destination step with the "`destination-path-prefix/pattern`".

B Surveyed Attacks

Attack Name	Key Compromise	Access Level
*NotPetya [35]	✓	PI
CCleaner Attack [126]	✓	BS, PI
Operation Red [138]	✓	PI
KingSlayer [118]	✓	PI
RedHat breach [125]	✓	BS
keydnap [71]	✓	PI
backdoored-pypi [52]	✓	PI
PEAR breach [33]	✗	PI
Monju Incident [55]	✗	PI
Janus Vulnerability [89]	✗	PI
Rust flaw [124]	✗	PI
XcodeGhost [113]	✗	BS
Expensive Wall [8]	✗	BS
WordPress breach [107]	✗	CR
HandBrake breach [134]	✗	PI
Proton malware [86]	✗	PI
FOSSHub breach [103]	✗	PI
BadExit Tor [66]	✗	PI
Fake updater [95]	✗	PI
Bitcoin Gold breach [70]	✗	PI
Adobe breach [44]	✗	CR
Google Breach [147]	✗	CR
ProFTPD breach [120]	✗	CR
Kernel.org breach [62]	✗	CR
Hacked Linux Mint [146]	✗	PI
Code Spaces breach [51]	✗	CR
Unnamed Maker [53]	✗	PI
Gentoo backdoor [76]	✗	CR
Buggy Windows [68]	✗	PI
Buggy Mac [37]	✗	PI

Table 4: Summary of surveyed supply chain attacks. CR, BS and PI stand for Code Repository, Build System and Publishing Infrastructure, respectively. A ✓indicates that the attack involved a key compromise. In one attack, marked with a star (*), it was unknown if a compromised key was involved. We assumed that was the case.

IODINE: Verifying Constant-Time Execution of Hardware

Klaus v. Gleissenthall
University of California, San Diego

Rami Gökhan Kıcı
University of California, San Diego

Deian Stefan
University of California, San Diego

Ranjit Jhala
University of California, San Diego

Abstract. To be secure, cryptographic algorithms crucially rely on the underlying hardware to avoid inadvertent leakage of secrets through timing side channels. Unfortunately, such timing channels are ubiquitous in modern hardware, due to its labyrinthine fast-paths and optimizations. A promising way to avoid timing vulnerabilities is to devise—and verify—conditions under which a hardware design is free of timing variability, *i.e.*, executes in *constant-time*. In this paper, we present IODINE: a clock-precise, constant-time approach to eliminating timing side channels in hardware. IODINE succeeds in verifying various open source hardware designs in seconds and with little developer effort. IODINE also discovered two constant-time violations: one in a floating-point unit and another one in an RSA encryption module.

1 Introduction

Trust in software systems is always rooted in the underlying hardware. This trust is apparent when using hardware security features like enclaves (*e.g.*, SGX and TrustZone), crypto units (*e.g.*, AES-NI and the TPM), or MMUs. But our trust goes deeper. Even for simple `ADD` or `MUL` instructions, we expect the processor to avoid leaking any of the operands via *timing side channels*, *e.g.*, by varying the execution time of the operation according to the data. Indeed, even algorithms specifically designed to be resilient to such timing side-channel attacks crucially rely on these assumptions [23–25]. Alas, recently discovered vulnerabilities have shown that the labyrinthine fast-paths and optimizations ubiquitous in modern hardware expose a plethora of side channels that undermine many of our deeply held beliefs [34, 36, 42].

A promising way to ensure that trust in hardware is properly earned is to formally specify our expectations, and then, to *verify*—through mathematical proof—that the units used in security critical contexts do not exhibit any timing variability, *i.e.*, are *constant-time*. For instance, by verifying that certain parts of an arithmetic logic unit (ALU) are constant-time, we can provide a foundation for implementing secure crypto algorithms in software [16, 20, 22]. Dually, if timing variability is unavoidable, *e.g.*, in SIMD or floating-point units, making this variability *explicit* can better inform mechanisms that attempt to mitigate timing channels at the software level [18, 46, 54] in order to avoid vulnerabilities due to gaps in the hardware-software contract [17, 18].

In this paper, we introduce IODINE: a clock-precise, constant-time approach to eliminating timing side channels in hardware. Given a hardware circuit described in Verilog, a *specification* comprising a set of sources and sinks (*e.g.*, an FPU pipeline start and end) and a set of usage assumptions (*e.g.*, no division is performed), IODINE allows developers to automatically synthesize *proofs* which ensure that the hardware runs in constant-time, *i.e.*, under the given usage assumptions, the time taken to flow from source to sink, is independent of operands, processor flags and interference by concurrent computations.

Using IODINE, a crypto hardware designer can be certain that their encryption core does not leak secret keys or messages by taking a different number of cycles depending on the secret values. Similarly, a CPU designer can guarantee that programs (*e.g.*, cryptographic algorithms, SVG filters) will run in constant-time when properly structured (*e.g.*, when they do not branch or access memory depending on secrets [20]).

IODINE is *clock-precise* in that it enforces constant-time execution directly as a semantic property of the circuit rather than through indirect means like information flow control [55]. As a result, IODINE neither requires the constant-time property to hold unconditionally nor

demands the circuit be partitioned between different security levels (*e.g.*, as in SecVerilog [55]). This makes IODINE particularly suited for verifying existing hardware designs. For example, we envision IODINE to be useful in verifying ARM's recent set of *data independent timing (DIT)* instructions which should execute in constant-time, if the PSTATE.DIT processor state flag is set [2, 41].

While there have been significant strides in verifying the constant-time execution of software [14–16, 18, 20–22, 53], IODINE unfortunately cannot directly reuse these efforts. Constant time methods for software focus on straight-line, sequential—often cryptographic—code.

Hardware designs, however, are inherently *concurrent* and *long-lived*: circuits can be viewed as collections of processes that run forever, performing parallel computations that update registers and memory in every clock cycle. As a result, in hardware, even the definition of constant-time execution becomes problematic: how can we measure the timing of a hardware design that never stops and performs multiple concurrent computations that mutually influence each other?

In IODINE, we address these challenges through the following contributions.

1. Definition. First, we define a notion of constant-time execution for concurrent, long-lived computations. In order to reason about the timing of values flowing between sources and sinks, we introduce the notion of *influence set*. The influence set of a value contains all cycles t, such that an input (*i.e.*, a source value) at t was used in its computation. We say that a hardware design is constant time, if all its computation paths (that satisfy usage assumptions) produce the same sequence of influence sets for sinks.

2. Verification. To enable its efficient verification, we show how to reduce the problem of checking constant-time execution—as defined through influence sets—to the standard problem of checking assertion validity. For this, we first eschew the complexity of reasoning about several concurrent computations at once, by focusing on a *single* computation starting (*i.e.*, inputs issued) at some cycle t. We say that a value is *live* for cycle t (t-live), if it was influenced by the computation started at t, *i.e.*, t is in the value's influence set. This allows us to reduce the problem of checking equality of influence sets, to checking the equivalence of membership, for their elements. We say that a hardware design is *liveness equivalent*, if, for any two executions (that satisfy usage assumptions), and any t, t-live values are assigned to sinks in the same way, *i.e.*, whenever a t-live value is assigned to a sink in one execution, a t-live value must also be assigned to a sink in the other.

To check a hardware design for liveness equivalence, we *mark* source data as live in some *arbitrarily chosen* start cycle t, and track the flow of t-live values through the circuit using a simple standard taint tracking monitor [44]; the problem of checking liveness equivalence then reduces to checking a simple assertion stating that sinks are always tainted in the same way. Reducing constant-time execution to the standard problem of checking assertion validity allows us to rely on off-the-shelf, mature verification technology, which explains IODINE's effectiveness.

3. Evaluation. Our final contribution is an implementation and evaluation of IODINE on seven open source VERILOG projects—CPU cores, an ALU, crypto-cores, and floating-point units (FPUs). We find that IODINE succeeds in verifying different kinds of hardware designs in a matter of seconds, with modest developer effort (§ 6). Many of our benchmarks are constant-time for intricate reasons (§ 6.3), *e.g.*, whether or not a circuit is constant-time depends on its execution history, circuits are constant-time despite triggering different control flow paths depending on secrets, and require a carefully chosen set of assumptions to be shown constant-time. In our experience, these characteristics—combined with the circuit size—make determining whether a hardware design is constant-time by code inspection near impossible.

IODINE also revealed two constant-time violations: one in the division unit of an FPU designs, another in the modular exponentiation module of an RSA encryption module. The second violation—a classical timing side channel—can be abused to leak secret keys [27, 35].

In summary, this paper makes the following contributions.

- First, we give a definition for constant-time execution of hardware, based on the notion of *influence sets* (§ 2). We formalize the semantics of VERILOG programs with influence sets (§ 3), and use this formalization to define constant-time execution with respect to usage assumptions (§ 4).
- Our second contribution is a reduction of constant-time execution to the easy-to-verify problem of liveness equivalence. We formalize this property (§ 4), prove its equivalence to our original notion of constant-time execution (§ 4.3), and show how to verify it using standard methods (§ 5).

```
1  // source(x); source(y); sink(out);
2  // assume(ct = 1);
3
4  reg flp_res, x, y, ct, out, out_ready, ...;
5  wire iszero, isNaN, ...;
6
7  assign iszero = (x == 0) || (y == 0);
8
9  always @(posedge clk) begin
10   ...
11   flp_res <= ... // (2) compute x * y
12 end
13
14 always @(posedge clk) begin
15   if (ct)
16     ...; out <= flp_res; // (4)
17   else
18     if (iszero)
19       out <= 0; // (1)
20     else if (isNaN)
21       ...
22     else out <= flp_res; // (3)
23     end
24   end
25 end
```

Figure 1: Floating point multiplier (EX1).

- Our final contribution is an implementation and evaluation of IODINE on several challenging open source hardware designs (§ 6). Our evaluation shows that IODINE can be used to verify constant-time execution of existing hardware designs, rapidly, and with modest user effort.

2 Overview

In this section, we give an overview of IODINE and show how our tool can be used to verify that a piece of VERILOG code executes in constant-time. As a running example, we consider a simple implementation of a floating-point multiplication unit.

Floating Point Multiplier. Our running example, like most FPUs, is generally *not* constant-time—common operations (*e.g.*, multiplication by zero) are dramatically faster than rare ones (*e.g.*, multiplication by denormal numbers [17,36]). But, like the ARM's recent support for data independent timing instructions, our FPU contains a processor flag that can be set to ensure that all multiplications are constant-time, at the cost of performance. Fig. 1 gives a simplified fragment of VERILOG code that implements this FPU multiplier. While our benchmarks consist of hundreds of threads with shared variables, pipelining, and contain a myriad of branches and flags which cause dependencies on the execution history (see § 6.3), we have kept our running example as simple as possible: our multiplier takes two floating-point values—input registers x and y—and stores the computation result in output register out. Recall that VERILOG programs operate on two kinds of data-structures: *registers* which are assigned in always-blocks and store values across clock cycles and *wires* which are assigned in assign-blocks and hold values only within a cycle. Control register ct is used to configure the FPU to run in constant-time (or not). For simplicity, we omit most other control logic (*e.g.*, reset or output-ready bits and processing of inputs). Internally, the multiplier consists of several *fast* paths and a single *slow* path. For example, to implement multiplication by zero, one, NAN, and other special values we, inspect the input registers for these values and produce a result in a single cycle (see (1)). Multiplication by other numbers is more complex, however, and generally takes more than a single cycle. As shown in Fig. 1, this *slow* path consists of multiple intermediate steps, the final result of which is assigned to a temporary register flp_res (see (2)) before out (see (3)).[1] Importantly, when the constant-time configuration register ct is set, only this slow path is taken (see (4)).

Outline. In the rest of this section, we show how IODINE verifies that this hardware design runs in constant-time when the ct flag is set and violates the constant-time property otherwise. We present our definition of constant-time based on of influence sets in § 2.1, liveness equivalence in § 2.2, and finally show how IODINE formally verifies liveness equivalence by reducing it to a simple safety property that can be handled by standard verification methods § 2.3.

2.1 Constant-Time For Hardware

We start by defining *constant-time* execution for hardware.

Assumptions and Attacker Model. Like SecVerilog [55], we scope our work to synchronous circuits with a single, fixed-rate clock. We further assume an *external attacker* that can measure the execution time of a piece of hardware (given as influence sets) using a cycle-precise timer. In particular, an attacker can observe the timing of *all* inputs that influenced a given output. These assumptions afford us many benefits. (Though, as we describe in § 7, they are not for free.) For example, assuming a single

[1] For simplicity, we omit the intermediate steps and assume that they implement floating-point multiplication in constant-time. In practice, FPUs may also take different amounts of time depending on such values.

P	::=	**Program**
	$\mid [s]_{id}$	*process*
	$\mid P \parallel P$	*parallel composition*
	$\mid$ repeat P	*sync. iteration*
	$\mid \rightarrow$	*empty process*
s	::=	**Command**
	$\mid$ skip	*no-op*
	$\mid v = e$	*blocking*
	$\mid v \Leftarrow e$	*non-blocking*
	$\mid v := e$	*continuous*
	$\mid$ ite(e,s,s)	*conditional*
	$\mid s_1 ; \ldots ; s_k$	*sequence*
	$\mid a$	*annotation*
e	::=	**Expression**
	$\mid v$	*variables*
	$\mid n$	*constants*
	$\mid f(e_1,\ldots,e_k)$	*function literal*

Figure 2: Syntax for intermediate language VINTER.

```
    repeat   [iszero := (x == 0 || y == 0)]
||  repeat     [... ; flp_res ⇐ ...]
||  repeat   [ ite(ct,
                 out ⇐ flp_res,
                 ite(iszero,
                   out ⇐ 0,
                   out ⇐ flp_res)) ]
```

Figure 3: EX1 written in VINTER

fixed-rate clock, allows us to translate VERILOG programs, such as our FPU multiplier to a more concise representation shown in Figure 3.

Intermediate Language. In this language—called VINTER—VERILOG always- and assign-blocks are represented as concurrent *processes*, wrapped inside an infinite repeat-loop. As Fig. 2 shows, each process sequentially executes a series of VERILOG-like statements. (Each process also has a unique identifier $id \in PIDs$, which we sometimes omit, for brevity.) Most of these are standard; we only note that VINTER—like VERILOG—supports three types of assignment statements: blocking ($v = e$), non-blocking ($v \Leftarrow e$) and continuous ($v := e$). Blocking assignments take effect immediately, within the current cycle; non-blocking assignments are deferred until the next cycle. Finally, continuous assignments enforce directed equalities between registers or wires: whenever the right-hand side of an equality is changed, the left-hand side is updated by re-running the assignment. Note that VINTER focuses only on the synthesizable fragment of VERILOG, *i.e.*, does not model delays, etc., which are only relevant for simulation.

VINTER processes are composed in parallel using the ($\parallel$) operator. Unlike concurrent software processes, they are, however, synchronized using a single (implicit) fixed-rate clock: each process waits for all other (parallel) processes to finish executing before moving on to the next iteration, *i.e.*, next clock cycle. Moreover, unlike software, these programs are usually data-race free, in order to be synthesizable to hardware.

VINTER processes run forever; they perform computations and update registers (*e.g.*, out in our multiplier) on every clock cycle. For example, pipelined hardware units execute multiple, different computations simultaneously.

From Software to Hardware. This execution model, together with the fact that software operates at a higher level of abstraction than hardware, makes it difficult for us to use existing verification tools for constant-time software (*e.g.*, [16, 20]).

First, constant-time verification for software only considers straight-line, sequential code. This makes it ill-suited for the concurrent, long-lived execution model of hardware.

Second, software constant-time models are necessarily conservative. They deliberately abstract over hardware details—*i.e.*, they don't rely on a precise hardware models (*e.g.*, of caches or branch predictors)—and instead use *leakage models* that make control flow and memory access patterns observable to the attacker. This makes constant-time software portable across hardware. But, it also makes the programming model restrictive: the model disallows any branching to protect against hidden microarchitectural state (*e.g.*, the branch predictor).

Since we operate on VERILOG, where all state is explicit and visible, we can instead directly track the influence of secret values on the timing of attacker-observable outputs. This allows us to be more permissive than software constant-time models. For instance, if we can show that the execution of two branches of a hardware design takes the same amount of time, independent of secret inputs, we can safely allow branches on secrets. However, this still leaves the problem of pipelining: hardware ingests inputs and produce outputs at every clock cycle: how then do we know (if and) which secret inputs influenced a particular output?

Influence Sets. This motivates our definition for *influence sets*. In order to define a notion of constant-time execution that is suitable for hardware, we first add annotations marking inputs (*i.e.*, x and y in our example) as *sources* and outputs (*i.e.*, out) as *sinks*. For a given cycle, we then associate with each register x its *influence-*

Cycle #	x	y	ct	fr	out	θ(x)	θ(y)	θ(ct)	θ(fr)	θ(out)
0	0	1	**F**	**X**	**X**	{0}	{0}	∅	∅	∅
1	0	1	F	X	0	{1}	{1}	∅	∅	{0}
					⋮					
k-1	0	1	F	0	0	{k−1}	{k−1}	∅	{0}	{k−2}
k	0	1	F	0	0	{k}	{k}	∅	{1}	{**k−1**}

Figure 4: Execution of Ex1, where x = 0 and y = 1, and ct is unset. For each variable and cycle, we show its current value and influence set. We assume that it takes k cycles to compute the output along the slow path, and abbreviate flp_res as fr. **X** denotes an unknown/irrelevant value. Register out is only influenced by values from the last cycle. Highlighted cells are the difference with Figure 5. Values that stayed the same in the next cycle are shaded.

Cycle #	x	y	ct	fr	out	θ(x)	θ(y)	θ(ct)	θ(fr)	θ(out)
0	1	1	**F**	**X**	**X**	{0}	{0}	∅	∅	∅
1	1	1	F	X	**X**	{1}	{1}	∅	∅	{0}
					⋮					
k-1	1	1	F	1	X	{k−1}	{k−1}	∅	{0}	{k−2}
k	1	1	F	1	1	{k}	{k}	∅	{1}	{**0**,**k−1**}

Figure 5: Execution of Ex1, where both x = 1 and y = 1, and ct is unset. The execution produces the same influence sets as the execution in Fig. 4, except for cycle k, where out's influence set contains the additional value 0, thereby violating our definition of constant-time execution.

set θ(x). The influence set of a register x contains all cycles t, such that an input at t was used in the computation of x's current value. This allows us to define constant-time execution for hardware: we say that a hardware design is constant-time, if any two executions (that satisfy usage assumptions) produce the same sequence of influence sets for their sinks.

Example. We now illustrate this definition using our running example Ex1 by showing that Ex1 violates our definition of constant-time, if the ct flag is unset. For this, consider Fig. 4 and Fig. 5, which show the state of registers and wires as well as their respective influence sets, for two executions. In both executions, we let y = 1, but vary the value of the x register: in Fig. 4, we set x to 0 to trigger the fast path in Fig. 5 we set it to 1. In both executions, sources x and y are only influenced by the current cycle, constant-time flag ct is set independently of inputs, and temporary register flp_res is influenced by the inputs that were issued k − 1 cycles ago, as it takes k − 1 cycles to compute flp_res along the slow path.

The two executions differ in the influence sets of out. In Fig. 4, out is only influenced by the input issued in the last cycle, through a control dependency on iszero. In the execution in Fig. 5, its value at cycle k is however also influenced by the input at 0. This reflects the propagation of the computation result through the slow path. Crucially, it also shows that the multiplier is not constant-time—the sets θ(out) differing between two runs reflects the influence of data on the duration of the computation.

2.2 Liveness Equivalence

We now show how to reduce verifying whether a given hardware is constant-time to an easy-to-check, yet equivalent problem called liveness equivalence. Intuitively, liveness equivalence reduces the problem of checking equality of influence sets, to checking the equivalence of membership, for arbitrary elements.

Liveness Equivalence. Our reduction focuses on a single computation started at some cycle t. We say that register x is *live* for cycle t (t-live), if its current value is influenced by an input issued in cycle t, *i.e.*, if t ∈ θ(x). Two executions are t-liveness equivalent, if whenever a t-live value is assigned to a sink in one execution, a t-live value must also be assigned in the other. Finally, a hardware design is liveness equivalent, if any two executions that satisfy usage assumptions are t-liveness equivalent, for any t.

Live Value Propagation. To track t-liveness for a fixed t, IODINE internally transforms VINTER programs as follows. For each register or wire (*e.g.*, x in our multiplier), we introduce a new shadow variable (*e.g.*, x•) that represents its liveness; a shadow variable x• is set to L if x is live and D (dead) otherwise.[2] We then propagate live-

[2] For liveness-bits x• and y•, we define a join operator ∨, such that

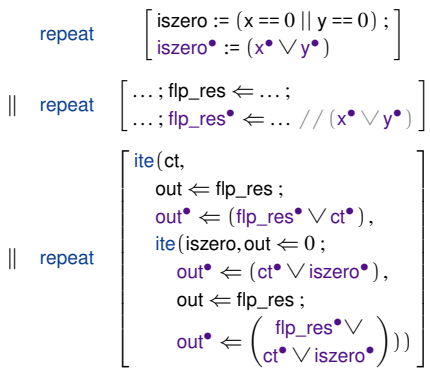


Figure 6: EX1, after we propagate liveness using a standard taint-tracking inline monitor.

	x	y	ct	fr	out	x•	y•	ct•	fr•	out•
0	0	1	**F**	**X**	**X**	L	L	D	D	D
1	0	1	F	X	0	D	D	D	D	L
					⋮					
k-1	0	1	F	0	0	D	D	D	L	D
k	0	1	F	0	0	D	D	D	D	D

Figure 7: Execution of EX1•, where x = 0 and y = 1. We show current value and liveness bit for each register and cycle. Register out is live in cycle one, due to the fast path and dead, otherwise. Highlights are the differences with Figure 8. Values that stayed the same in the next cycle are shaded.

ness using a standard taint-tracking inline monitor [44] shown in Figure 6. Intuitively, our monitor ensures that registers and wires that depend on a live value—directly or indirectly, via control flow—are marked live.

Example. By tracking liveness, we can again see that our floating-point multiplier is not constant-time when the ct flag is unset. To this end, we "inject" live values at sources (x and y) at time t = 0 for two runs; as before, we set y = 1, and vary the value of x: in one execution, we set x to 0 to trigger the fast path, in the other execution, we set it to 1. Fig. 7 and 8 show the state of the different registers and wires for these runs. In both runs, out is live at cycle 1—due to a control dependency in Fig. 7, due a direct assignment in Fig. 8. But, in the latter, out is *also* live at the kth cycle. This reflects the fact that the influence sets of out at cycle k differ in the membership of 0, and therefore witnesses the constant-time violation.

2.3 Verifying Liveness Equivalence

Using our reduction to liveness equivalence, we can *verify* that a VERILOG program executes in constant-time using standard methods. For this, we *mark* source data as

	x	y	ct	fr	out	x•	y•	ct•	fr•	out•
0	1	1	**F**	**X**	**X**	L	L	D	D	D
1	1	1	F	X	**X**	D	D	D	D	L
					⋮					
k-1	1	1	F	1	X	D	D	D	L	D
k	1	1	F	1	1	D	D	D	D	L

Figure 8: Execution of EX1•, where both x = 1 and y = 1. The liveness bits are the same as in 7, except for cycle k, where out is now live. This reflects the propagation of the output value through the slow path and shows the constant-time violation.

live in some *arbitrarily chosen* start cycle t. We then verify that any *two* executions that satisfy usage assumptions assign t-live values to sinks, in the same way.

Product Programs. Like previous work on verifying constant-time software [16], IODINE reduced the problem of verifying properties of *two* executions of some program P by proving a property about a *single* execution of a new program Q. This program—the so-called *product program* [22] – consists of two disjoint copies of the original program.

Race-Freedom. Our product construction exploits the fact that VERILOG programs are *race-free*, *i.e.*, the order in which always-blocks are scheduled within a cycle does not matter. While races in software often serve a purpose (*e.g.*, a task distribution service may allow races between equivalent worker threads to increase throughput), races in VERILOG are always artifacts of poorly designed code: any synthesized circuit is, by its nature, race-free, *i.e.*, the scheduling of processes *within* a cycle does not affect the computation outcome. Indeed, races in VERILOG represent an under-specification of the intended design.

Per-Process Product. We leverage this insight to compose the two copies of a program in *lock-step*. Specifically, we merge each process of the two program copies and execute the "left" (L) and "right" (R) copies together. For example, IODINE transforms the VINTER multiplier code from Figure 6 into the *per-process product program* shown in Figure 9.

Merging two copies of a program as such is sound: since the program is race-free—any ordering of process transitions *within* a cycle yields the same results—we are free to pick an arbitrary schedule.[3] Hence, IODINE takes a simple ordering approach and schedules the left and right copy of same process at the same time.

Constant-Time Assertion. Given such a product program, we can now frame the constant-time verification

x• ∨ y• is L, if x• or y• is L and D, otherwise.

[3] To ensure that hardware designs are indeed race-free, our implementation performs a light-weight static analysis to check for races.

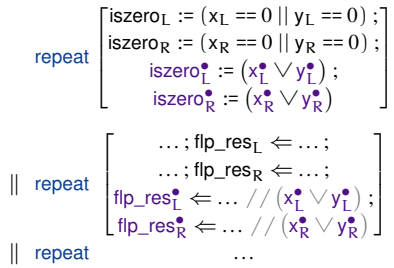


Figure 9: Per-process product form of EX1.

challenge as a simple *assertion*: the liveness of the left and right program sink-variables must be the same (regardless of when the computation started). In our example, this assertion is simply $\mathsf{out}^\bullet{}_L = \mathsf{out}^\bullet{}_R$. This assertion can be verified using standard methods. In particular, IODINE synthesize process-modular invariants [45] that imply the constant-time assertion (§ 5).

The following two sections formalize the material presented in this overview.

3 Syntax and Semantics

Since VERILOG's execution model can be subtle [12], we formally define syntax and semantics of the VERILOG fragment considered in this paper.

3.1 Preliminaries

For a function f, we write *dom* f to denote f's domain and *ran* f for its co-domain. For a set $S \subseteq \mathit{dom}\ f$, we let $f[S \leftarrow b]$ denote the function that behaves the same as f except S, where it returns b, *i.e.*, $f[S \leftarrow b](x)$ evaluates to b if $x \in S$ and $f(x)$, otherwise. We use $f[a \leftarrow b]$ as a short hand for $f[\{a\} \leftarrow b]$. Sometimes, we want to update a function by setting the function values of some subset S of its domain to a non-deterministically chosen value. For $S \subseteq \mathit{dom}\ f$, we write $f[S \leftarrow *](x)$ to denote the function that evaluates to some y with $y \in \mathit{ran}\ f$, if $x \in S$ and $f(x)$ otherwise.

3.2 Syntax

We restrict ourselves to the *synthesizable* fragment of VERILOG, *i.e.*, we do not include commands like initial blocks that only affect simulation and implement a *normalization step* [32] in which the program is "flattened" by removing module instantiation through in-lining. We provide VERILOG syntax and a translation to VINTER in Appendix A.2, but define semantics in VINTER (Fig. 2).

Annotations. We define annotations in Figure 10. Let *Regs* denote the set of registers and *Wires* the set of wires and let VARS denote their disjoint union, *i.e.*,

$a ::=$	**In/Out**		**Assump.**
$\mid \mathsf{source}(v)$	*source*	$\mid \mathsf{init}(\varphi)$	*initially* φ
$\mid \mathsf{sink}(v)$	*sink*	$\mid \Box(\varphi)$	*always* φ

Figure 10: Annotation syntax.

Config	**Meaning**	**Trace**	**Meaning**
σ	*store*	Σ	*configuration*
τ	*liveness map*	l	*label*
θ	*influence map*	b	*liveness bit*
μ	*assign. buffer*	π	*trace*
ev	*event set*	$\mathit{store}(\pi, i)$	σ_i
P	*current program*	$\mathit{live}(\pi, i)$	τ_i
I	*initial program*	$\mathit{inf}(\pi, i)$	θ_i
c	*clock cycle*	$\mathit{clk}(\pi, i)$	c_i
		$\mathit{reset}(\pi, i)$	b_i

Figure 11: Configuration and trace syntax.

$\mathrm{VARS} \triangleq \mathit{Regs} \uplus \mathit{Wires}$. For a register $v \in \mathit{Regs}$, annotations $\mathsf{source}(v)$ and $\mathsf{sink}(v)$ designate v as source or sink, respectively.[4] We let $\mathrm{IO} \triangleq (\mathit{Src}, \mathit{Sink})$ denote the set of input/output assumptions, where *Src* denotes the set of all sources and *Sink* denote the set of all sinks. Let φ be a first-order formula over some background theory that refers to two disjoint sets of variables Vars_L and VARS_R. Then, annotations $\mathsf{init}(\varphi)$ and $\Box(\varphi)$ indicate that formula φ holds initially or throughout the execution. The assumptions are collected in $A \triangleq (\mathrm{INIT}, \mathrm{ALL})$, such that INIT contains all formulas under init and ALL all formulas under $\Box$.

3.3 Semantics

Values. The set of values $\mathrm{VALS} \triangleq \mathbb{Z} \uplus \{\mathbf{X}\}$ consists of the disjoint union of the integers and special value **X** which represents an irrelevant value. A function application that contains **X** as an argument evaluates to **X**.

Configurations. The program state is represented by a *configuration* $\Sigma \in \mathit{Configs}$. Figure 11 shows the components of a configuration. A store $\sigma \in \mathrm{STORES} \triangleq (\mathrm{VARS} \mapsto \mathrm{VALS})$ is a map from registers and wires to values. A *liveness map* $\tau \in \mathrm{LIVEMAP} \triangleq (\mathrm{VARS} \mapsto \{L, D\})$ is a map from registers and wires to liveness bits. A *influence map* $\theta \in \mathrm{INFMAPS} \triangleq (\mathrm{VARS} \mapsto \mathcal{P}(\mathbb{Z}))$ is a map from registers and wires to influence sets. *Assignment buffers* serve to model non-blocking assignments. Let *PIDs* denote a set of process identifiers. An assignment buffer $\mu \in \mathit{PIDs} \mapsto (\mathrm{VARS} \times \mathrm{VALS} \times \{L, D\} \times \mathcal{P}(\mathbb{Z}))^*$ is a map from pro-

[4] To use wires as source/sink, one has to define an auxiliary register.

$$\frac{}{\nu,\sigma,\tau,\theta \dashrightarrow \sigma(\nu),\tau(\nu),\theta(\nu)}\ [\textsc{Var}] \qquad \frac{}{n,\sigma,\tau,\theta \dashrightarrow n,\mathsf{D},\emptyset}\ [\textsc{Const}]$$

$$\frac{e_1,\sigma,\tau,\theta \dashrightarrow \nu_1,t_1,i_1 \quad \ldots \quad e_k,\sigma,\tau,\theta \dashrightarrow \nu_k,t_k,i_k \qquad t = (t_1 \vee \cdots \vee t_k) \qquad i = (i_1 \cup \cdots \cup i_k)}{f(e_1,\ldots,e_k),\sigma,\tau,\theta \dashrightarrow f(\nu_1,\ldots,\nu_k),t,i}\ [\textsc{Fun}]$$

Figure 12: Expression evaluation.

cess identifier to a sequence of variable/value/liveness-bit/influence set tuples. An *event set* $ev \in \mathcal{P}(\textsc{Vars})$ is a set of variables, where we use $\nu \in ev$ to indicate that variable ν has been changed in the current cycle. Finally, $I \in \mathit{Progs}$ contains the initial program. Intuitively, the initial program is used to activate all processes when a new clock cycle begins.

Evaluating Expressions. We define an evaluation relation $\dashrightarrow \in (\textsc{Expr} \times \textsc{Stores} \times \textsc{LiveMap} \times \textsc{InfMaps}) \mapsto (\textsc{Vals} \times \{\mathsf{L},\mathsf{D}\} \times \mathcal{P}(\mathbb{Z}))$ that computes value, liveness-bit, and influence map for an expression. We define the relation through the inference rules shown in Fig. 12. An evaluation step (below the line) can be taken, if the preconditions (above the line) are met. Rule [Var] evaluates a variable to its current value under the store, its current liveness-bit and influence set. A numerical constant evaluates to itself, is dead and not influenced by any cycle. To evaluate a function literal, we evaluate its arguments and apply the function on the resulting values. A function value is live if any of its arguments are, and its influence set is the union of its influences.

Transition Relations. We define our semantics in terms of four separate transition relations of type $(\mathit{Configs} \times \mathit{Labels} \times \mathit{Configs})$. We now discuss the individual relations and then describe how to combine them into an overall transition relation $\rightsquigarrow$.

Per-process transition $\rightsquigarrow_{\mathsf{P}}$. The per-process transition relation $\rightsquigarrow_{\mathsf{P}}$ describes how to step along individual processes. It is defined in Fig. 13. Rules [Seq-Step] and [Par-Step] are standard and describe sequential and parallel composition. Rule [B-Asn] reduces a blocking update $x = e$ to skip, by first evaluating e to yield a value ν, liveness bit t and influence set i, updating store σ, liveness map τ and influence map θ, and finally adding x to the set of modified variables. Rule [NB-Asn] defers a non-blocking assignment. In order to reduce an assignment $(x \Leftarrow e)_{id}$ for process *id* to skip, the rule evaluates expression e to value ν, liveness bit t and influence set i, and defers the assignment by appending the tuple (x,ν,t,i) to the back of *id*'s buffer. We omit rules for conditionals and structural equivalence. Structural equivalence allows transitions between trivially equivalent programs such as $P \parallel Q$ and $Q \parallel P$.

Non-blocking Transition $\rightsquigarrow_{\mathsf{N}}$. Transition relation $\rightsquigarrow_{\mathsf{N}}$ applies deferred non-blocking assignments. It is defined by a single rule [NB-App] shown in Fig. 13. The rule first picks a tuple (x,ν,t,i) from the front of the buffer of some process *id*, and, like [B-Asn], updates store σ, liveness map τ and influence map θ, and finally adds x to the set of updated variables.

Continuous Transition $\rightsquigarrow_{\mathsf{C}}$. Relation $\rightsquigarrow_{\mathsf{C}}$ specifies how to execute continuous assignments. It is described by rule [C-Asn] in Fig. 13, which reduces a continuous assignment $x := e$ to skip under the condition that some variable y occurring in e has changed, *i.e.*, $y \in ev$. To apply the assignment, it evaluates e to value, liveness bit and influence set, and updates store and liveness map and influence map. Importantly, variable y is not removed from the set of events, *i.e.*, a single assignment can enable several continuous assignments.

Global Transition $\rightsquigarrow_{\mathsf{G}}$. Finally, global transition relation $\rightsquigarrow_{\mathsf{G}}$ is defined by rules [NewCycle] and [NewCycle-Issue] shown in Fig. 13. [NewCycle] starts a new clock cycle by discarding the current program and event set, emptying the assignment buffer, resetting the wires to some non-deterministically chosen state (as wires only hold their value *within* a cycle), and rescheduling and activating a new set of processes, extracted from initial program I. For a program P, let $\textsc{Repeat}(P) \in \mathcal{P}(\mathit{Progs})$ denote the set of processes that occur under repeat. For a set of programs S, we let $\sqcap S$ denote their parallel composition. [NewCycle] uses these constructs to reschedule all processes that appear under repeat in I. Both sources and wires are set to D. The influence map is updated by mapping all wires to the empty set, and each source to the set containing only the current cycle.

[NewCycle-Issue] performs the same step, but additionally updates the liveness map by issuing new live bits for the source variables. Both rules increment the cycle counter c. The rules issue a *label* $l \in \mathit{Labels} \triangleq ((\textsc{Stores} \times \textsc{LiveMap} \times \textsc{InfMaps} \times \mathbb{N} \times \{\mathsf{L},\mathsf{D}\}) \uplus \epsilon)$ which is written above the arrow (all previous rules issue the empty label ϵ). The label contains the current store, liveness map, influence map, clock cycle, and a bit indicating whether new live-bits have been issued. Labels are used to construct the *trace* of an execution, as

we will discuss later.

Overall Transition $\rightsquigarrow$**.** We define the overall transition relation $\rightsquigarrow \in \mathit{Configs} \times \mathit{Labels} \times \mathit{Configs}$ by fixing an order in which to apply the relations. Whenever a *continuous assignment* step (relation $\rightsquigarrow_C$) can be applied, that step is taken. Whenever no continuous assignment step can be applied, however, a *per-process* step (relation $\rightsquigarrow_P$) can be applied, a $\rightsquigarrow_P$ step is taken. If no continuous assignment and process local steps can be applied, however, an *non-blocking assignment* step (relation $\rightsquigarrow_N$) is applicable, a $\rightsquigarrow_N$ step is taken. Finally, if neither continuous assignment, per-process, or non-blocking steps can be applied, the program moves to a new clock cycle by applying a *global step* (relation $\rightsquigarrow_G$). Our overall transition relation closely follows the Verilog simulation reference model from Section 11.4 of the standard [12].

Executions and Traces. An *execution* is a finite sequence of configurations and transition labels $r \triangleq \Sigma_0 l_0 \Sigma_1 \ldots \Sigma_{m-1} l_{m-1} \Sigma_m$ such that $\Sigma_i \overset{l_i}{\rightsquigarrow} \Sigma_{i+1}$ for $i \in \{1,\ldots,m-1\}$. We call Σ_0 *initial state* and require that all taint bits are set to D, the influence map maps each variable to the empty set, the assignment buffer is empty, the current program is the empty program $\rangle$, and the clock is set to 0. The *trace* of an execution is the sequence of its (non-empty) labels. For a trace $\pi \triangleq (\sigma_0,\tau_0,\theta_0,c_0,b_0)\ldots(\sigma_{n-1},\tau_{n-1},\theta_{n-1}c_{n-1},b_{n-1}) \in \mathit{Labels}^*$ and for $i \in \{0,\ldots,n-1\}$ we let $\mathit{store}(\pi,i) \triangleq \sigma_i$, $\mathit{live}(\pi,i) \triangleq \tau_i$, $\mathit{inf}(\pi,i) \triangleq \theta_i$, $\mathit{clk}(\pi,i) \triangleq c_i$ and $\mathit{reset}(\pi,i) = b_i$, and say the trace has length n. For a program P we use $\textsc{Traces}(P) \in \mathcal{P}(\mathit{Labels}^*)$ to denote the set of its traces, *i.e.*, all traces with initial program P.

4 Constant-Time Execution

We now first define constant-time execution with respect to a set of assumptions. We then define liveness equivalence and show that the two notions are equivalent.

4.1 Constant-Time Execution

Assumptions. For a formula φ that ranges over two disjoint sets of variables $\textsc{Vars}_L$ and $\textsc{Vars}_R$ and stores σ_L and σ_R such that $\mathit{dom}\ \sigma_L = \textsc{Vars}_L$ and $\mathit{dom}\ \sigma_R = \textsc{Vars}_R$, we write $\sigma_L,\sigma_R \models \varphi$ to denote that formula φ holds when evaluated on σ_L and σ_R. For some program P and a set of assumptions $A \triangleq (\textsc{Init},\textsc{All})$, we say that two traces $\pi_L,\pi_R \in \textsc{Traces}(P)$ of length n *satisfy* A if *i)* for each formula $\varphi_I \in \textsc{Init}$, φ_I holds initially, and *ii)* for each formula $\varphi_A \in \textsc{All}$, φ_A hold throughout, *i.e.*, $\mathit{store}(\pi_L,0),\mathit{store}(\pi_R,0) \models \varphi_I$ and $\mathit{store}(\pi_L,i),\mathit{store}(\pi_R,i) \models \varphi_A$, for $0 \leqslant i \leqslant n-1$. Intuitively, pairs of traces that satisfy the assumptions are "low" or "input" equivalent.

Constant Time Execution. For a program P, assumptions A and traces $\pi_L,\pi_R \in \textsc{Traces}(P)$ of length n that satisfy A, π_L and π_R are *constant time* with respect to A, if they produce the same influence sets for all sinks, *i.e.*, $\mathit{inf}(\pi_L,i)(v) = \mathit{inf}(\pi_R,i)(v)$, for $0 \leqslant i \leqslant n-1$ and all $v \in \mathit{Sink}$, and where two sets are equal if they contain the same elements. A program is constant time with respect to A, if all pairs of its traces that satisfy A are constant time.

4.2 Liveness Equivalence

t-Trace. For a trace π, we say that π is a t-trace, if $\mathit{reset}(\pi,t) = L$ and $\mathit{reset}(\pi,i) = D$, for $i \neq t$.

Liveness Equivalence. For a program P, let $\pi_L,\pi_R \in \textsc{Traces}(P)$, such that both π_L and π_R are of length n. We say that π_L and π_R are t-*liveness equivalent*, if both are t-traces, and $\mathit{live}(\pi_L,i)(v) = \mathit{live}(\pi_R,i)(v)$, for $0 \leqslant i \leqslant n-1$ and all $v \in \mathit{Sink}$. A program is t-liveness equivalent, with respect to a set of assumptions A, if all pairs of t-traces that satisfy A are t−liveness equivalent. Finally, a program is liveness equivalent with respect to A, if it is t-liveness equivalent with respect to A, for all t.

4.3 Equivalence

We can now state our equivalence theorem.

Theorem 1. *For all programs* P *and assumptions* A*,* P *executes in constant-time with respect to* A *if and only if it is liveness equivalent with respect to* A*.*

We first give a lemma which states that, if a register is t-live, then t is in its influence set.

Lemma 1. *For any* t*-trace* π *of length* n*, index* $0 \leqslant i \leqslant n-1$*, and variable* v*, if* v *is* t*−live,* i.e.*,* $\mathit{live}(\pi,i)(v) = L$*, then* t *is in* v*'s influence map,* i.e.*,* $t \in \mathit{inf}(\pi,i)(v)$*.*

We can now state our proof for Theorem 1.

Proof Theorem 1. The interesting direction is "right-to-left", *i.e.*, we want to show that a liveness equivalent program is also constant-time. We prove the contrapositive, *i.e.*, if a program violates constant-time, it must also violate liveness equivalence. For a proof by contradiction, we assume that P violates constant time execution, but satisfies liveness equivalence. If P violates constant-time execution, then there must be a sink v^*, two trace $\pi_L^*,\pi_R^* \in \textsc{Traces}(P)$ that satisfy A, and some

$$\text{[SEQ-STEP]}\quad \frac{\langle \sigma,\mu,\theta,ev,\tau,s_1,I,c\rangle \rightsquigarrow_P \langle \sigma',\mu',\theta',ev',\tau',s_1',I,c\rangle}{\langle \sigma,\mu,\theta,ev,\tau,[s_1;s_2],I,c\rangle \rightsquigarrow_P \langle \sigma',\mu',\theta',ev',\tau',[s_1';s_2],I,c\rangle}$$

$$\text{[PAR-STEP]}\quad \frac{\langle \sigma,\mu,\theta,ev,\tau,P,I,c\rangle \rightsquigarrow_P \langle \sigma',\mu',\theta',ev',\tau',P',I,c\rangle}{\langle \sigma,\mu,\theta,ev,\tau,P \parallel Q,I,c\rangle \rightsquigarrow_P \langle \sigma',\mu',\theta',ev',\tau',P' \parallel Q,I,c\rangle}$$

$$\text{[B-ASN]}\quad \frac{e,\sigma,\tau,\theta \dashrightarrow v,t,i \qquad \sigma' = \sigma[x \leftarrow v] \qquad \tau' = \tau[x \leftarrow t] \qquad \theta' = \theta[x \leftarrow i]}{\langle \sigma,\mu,\theta,ev,\tau,x = e,I,c\rangle \rightsquigarrow_P \langle \sigma',\mu,\theta',ev \cup \{x\},\tau',\mathsf{skip},I,c\rangle}$$

$$\text{[NB-ASN]}\quad \frac{e,\sigma,\tau,\theta \dashrightarrow v,t,i \qquad \mu' = \mu[id \leftarrow (x,v,t,i)\cdot q]}{\langle \sigma,\mu[id \leftarrow q],\theta,ev,\tau,(x \Leftarrow e)_{id},I,c\rangle \rightsquigarrow_P \langle \sigma,\mu',\theta,ev,\tau,\mathsf{skip},I,c\rangle}$$

$$\text{[NB-APP]}\quad \frac{\sigma' = \sigma[x \leftarrow v] \qquad \mu' = \mu[id \leftarrow q] \qquad \theta' = \theta[x \leftarrow i] \qquad \tau' = \tau[x \leftarrow t] \qquad ev' = ev \cup \{x\}}{\langle \sigma,\mu[id \leftarrow q\cdot(x,v,t,i)],\theta,ev,\tau,P,I,c\rangle \rightsquigarrow_N \langle \sigma',\mu',\theta',ev',\tau',P,I,c\rangle}$$

$$\text{[C-ASN]}\quad \frac{e,\sigma,\tau,i \dashrightarrow v,t,i \qquad y \in \textsc{Vars}(e) \qquad \sigma' = \sigma[x \leftarrow v] \qquad \tau' = \tau[x \leftarrow t] \qquad \theta' = \theta[x \leftarrow i]}{\langle \sigma,\mu,\theta,ev \cup \{y\},\tau,x := e,I,c\rangle \rightsquigarrow_C \langle \sigma',\mu,\theta',ev \cup \{x,y\},\tau',\mathsf{skip},I,c\rangle}$$

$$\text{[NEWCYCLE]}\quad \frac{\sigma' \triangleq \sigma[Wires \leftarrow *] \qquad \tau' \triangleq \tau[Src \leftarrow D][Wires \leftarrow D] \qquad \theta' \triangleq \theta[Wires \leftarrow \emptyset][Src \leftarrow \{c+1\}] \qquad \mu' \triangleq \mu[PIDs \leftarrow \epsilon]}{\langle \sigma,\mu,\theta,ev,\tau,P,I,c\rangle \overset{(\sigma,\tau,\theta,c,D)}{\rightsquigarrow_G} \langle \sigma',\mu',\theta',\emptyset,\tau,\sqcap \textsc{Repeat}(I),I,c+1\rangle}$$

$$\text{[NEWCYCLE-ISSUE]}\quad \frac{\sigma' \triangleq \sigma[Wires \leftarrow *] \qquad \tau' \triangleq \tau[Src \leftarrow L][(\textsc{Vars} - Src) \leftarrow D] \qquad \theta' \triangleq \theta[Wires \leftarrow \emptyset][Src \leftarrow \{c+1\}] \qquad \mu' \triangleq \mu[PIDs \leftarrow \epsilon]}{\langle \sigma,\mu,\theta,ev,\tau,P,I,c\rangle \overset{(\sigma,\tau,\theta,c,L)}{\rightsquigarrow_G} \langle \sigma',\mu',\theta',\emptyset,\tau',\sqcap \textsc{Repeat}(I),I,c+1\rangle}$$

Figure 13: Per-thread transition relation $\rightsquigarrow_P$, non-blocking transition relation $\rightsquigarrow_N$, continuous transition relation $\rightsquigarrow_C$, and global restart relation $\rightsquigarrow_G$.

index i^* such that $inf(\pi_L^*, i^*)(v^*) \neq inf(\pi_R^*, i^*)(v^*)$, and therefore without loss of generality, there is a cycle t^*, such that $t^* \in inf(\pi_L^*, i^*)(v^*)$ and $t^* \notin inf(\pi_R^*, i^*)(v^*)$. We can find two traces t^*-traces $\hat{\pi}_L$ and $\hat{\pi}_R$ that only differ from π_L^* and π_R^* in their liveness maps. But then, since the traces are t^*-liveness equivalent, by definition, at index i^* both $\hat{\pi}_L$ and $\hat{\pi}_R$ are t^*−live, *i.e.*, $live(\hat{\pi}_L, i^*)(v^*) = live(\hat{\pi}_R, i^*)(v^*) = L$ and, by lemma 1, $t^* \in inf(\hat{\pi}_R, i^*)(v^*)$. Since $\hat{\pi}_R$ and π_R^* only differ in their liveness map, this implies $t^* \in inf(\pi_R^*, i^*)(v^*)$, from which the contradiction follows. □

5 Verifying Constant Time Execution

In this section, we describe how IODINE verifies liveness equivalence by using standard techniques.

***Algorithm* IODINE.** Given a VINTER program P, a set of input/output specifications IO and a set of assumptions A, IODINE checks that P executes in constant time with respect to A. For this, IODINE first checks for race-freedom. If a race is detected, IODINE returns a witness describing the violation. If no race is detected, IODINE takes the following four steps: **(1)** It builds a set of Horn clause constraints hs [26, 33] whose solution characterizes the set of all configurations that are reachable by the per-process product and satisfy A. **(2)** Next, it builds a set of constraints cs whose solutions characterize the set of liveness equivalent states. **(3)** It then computes a solution Sol to hs and checks whether the solution satisfies cs. To find a more precise solution, the user can supply additional hints in the form of a set of predicates which we describe later. **(4)** If the check succeeds, P executes in constant time with respect to A, otherwise, P can potentially exhibit timing variations.

Constraint Solving. IODINE solves the reachability constraints by using Liquid Fixpoint [10], which computes the *strongest solution* that can be expressed as a conjunction of elements of a set of logical formulas. These formulas are composed of a set of *base predicates*. We use base predicates that track equalities between the liveness bits and values of each variable between the two runs. In addition to these base predicates, we use hints that are defined by the user. We discuss in § 6 which predicates were used in our benchmarks.

6 Implementation and Evaluation

In this section, we describe our implementation and evaluate IODINE on several open source VERILOG projects, spanning from RISC processors, to floating-point units and crypto cores. We find that IODINE is able to show that a piece of code is not constant-time and otherwise verify that the hardware is constant-time in a matter of seconds. Except our processor use cases, we found the annotation burden to be light weight—often less than 10 lines of code. All the source code and data are available on GitHub, under an open source license.[5]

6.1 Implementation

IODINE consists of a front-end pass, which takes annotated hardware descriptions and compiles them to VINTER, and a back-end that verifies the constant-time execution of these VINTER programs. We think this modular designs will make it easy for IODINE to be extended to support different hardware description languages beyond VERILOG (*e.g.*, VHDL or Chisel [19]).

Our front-end extends the Icarus Verilog parser [9] and consists of 2000 lines of C++. Since VINTER shares many similarities with VERILOG, this pass is relatively straightforward, however, IODINE does not distinguish between clock edges (positive or negative) and, thus, removes them during compilation. Moreover, our prototype does not support the whole VERILOG language (*e.g.*, we do not support assignments to multiple variables).

IODINE's back-end takes a VINTER program and, following § 5, generates and checks a set of verification conditions. We implement the back-end in 4000 lines of Haskell. Internally, this Haskell back-end generates Horn clauses and solves them using the liquid-fixpoint library that wraps the Z3 [29] SMT solver. Our back-end outputs the generated invariants, which **(1)** serve as the proof of correctness when the verification succeeds, or **(2)** helps pinpoint why verification fails.

Tool Correctness. The IODINE implementation and Z3 SMT solver [29] are part of our trusted computing base. This is similar to other constant-time and information flow tools (*e.g.*, SecVerilog [55] and ct-verif [16]). As such, the formal guarantees of IODINE can be undermined by implementation bugs. We perform several tests to catch such bugs early—in particular, we validate: (1) our translation into VINTER against the original VERILOG code; (2) our translation from VINTER into Horn clauses against our semantics; and, (3) the generated invariants against both the VINTER and VERILOG code.

[5] https://iodine.programming.systems

6.2 Evaluation

Our evaluation seeks to answer three questions: (Q1) Can IODINE be easily applied to existing hardware designs? (Q2) How efficient is IODINE? (Q3) What is the annotation burden on developers?

(Q1) Applicability. To evaluate its applicability, we run IODINE on several open source hardware modules from GitHub and OpenCores. We chose VERILOG programs that fit into three categories—processors, crypto-cores, and floating-point units (FPUs)—these have previously been shown to expose timing side channels. In particular, our benchmarks consist of:

- MIPS- and RISCV-32I-based pipe-lined CPU cores with a single level memory hierarchy.
- Crypto cores implementing the SHA 256 hash function and RSA 4096-bit encryption.
- Two FPUs that implement core operations $(+, -, \times, \div)$ according to the IEEE-754 standard.
- An ALU [1] that implements $(+, -, \times, \ll, \ldots)$.

In our benchmarks, following our attacker model from § 2.1, we annotated all the inputs to the computation. For example, this includes the sequence of instructions for the benchmarks with a pipeline (*i.e.*, MIPS, RISC-V, FPU and FPU2) in addition to other control inputs, and all the top level VERILOG inputs for the rest (*i.e.*, SHA-256, ALU and RSA). Similarly, we annotated as sinks, all the outputs of the computation. In the case of benchmarks with a pipeline, this includes the output from the last stage and other results (*e.g.*, whether the result is NaN in FPU), and all the top level VERILOG outputs for the rest. The modifications we had to perform to run IODINE on these benchmarks were minimal and due to parser restrictions (*e.g.*, desugaring assignments to multiple variables into individual assignments, unrolling the code generated by the loop inside the generate blocks).

(Q2) Efficiency. To evaluate its efficiency, we run IODINE on the annotated programs. As highlighted in Table 1, IODINE can successfully verify different VERILOG programs of modest size (up to 1.1K lines of code) relatively quickly (<20s). All but the constant-time FPU finished in under 3 seconds. Verifying the constant-time FPU took 12 seconds, despite the complexity of IEEE-754 standard which manifests as a series of case splits in VERILOG. We find these measurements encouraging, especially relative to the time it takes to synthesize VERILOG—verification is orders of magnitude smaller.

Name	#LOC	#Assum		CT	Check (s)
		#flush	#always		
MIPS [5]	434	31	2	✓	1.329
RISC-V [7]	745	50	19	✓	1.787
SHA-256 [8]	651	5	3	✓	2.739
FPU [6]	1182	0	0	✓	12.013
ALU [1]	913	1	5	✓	1.595
FPU2 [3]	272	3	4	✗	0.705
RSA [4]	870	4	0	✗	1.061
Total	5067	94	33	-	21.163

Table 1: #LOC is the number of lines of Verilog code, **#Assum** is the number of assumptions (excluding source and sink); **flush** and **always** are annotations of the form init and $\square$ respectively, **CT** shows if the program is constant-time, and **Check** is the time IODINE took to check the program. All experiments were run on a Intel Core i7 processor with 16 GB RAM.

Discovered Timing Variability. Running IODINE revealed that two of our use cases are not constant-time: one of the FPU implementations and the RSA crypto-core. The division module of the FPU exhibits timing variability depending on the value of the operands. In particular, similar to the example from § 2, the module triggers a fast path if the operands are special values.

The RSA encryption core similarly exhibited time variability. In particular, the internal modular exponentiation algorithm performs a Montgomery multiplication depending on the value of a source bit e_i: if $e_i = 1$ then $\overline{c} := \mathsf{ModPro}(\overline{c}, \overline{m})$. Since e is a secret, this timing variability can be exploited to reveal the secret key [27, 35].

(Q3) Annotation burden. While IODINE automatically discovers proofs, the user has to provide a set of assumptions A under which the hardware design executes in constant time. To evaluate the burden this places on developers, we count the number and kinds of assumptions we had to add to each of our use cases. Table 1 summarizes our results: except for the CPU cores, most of our other benchmarks required only a handful of assumptions. Beyond declaring sinks and sources, we rely on two other kinds of annotations. First, we find it useful to specify that the initial state of an input variable x is equal in any pair of runs, *i.e.*, $\mathsf{init}(x_L = x_R)$. This assumption essentially specifies that register x is flushed, *i.e.*, is set to a constant value, to remove any effects of a previous execution from our initial state. Second, we find it useful to specify that the state of an input variable x is equal, throughout any pair of runs, *i.e.*, $\square(x_L = x_R)$. This assumption is important when certain behavior is expected to be the same in both runs. We now describe these assumptions for our benchmarks.

- **MIPS:** We specify that the values of the fetched instructions, and the reset bit are the same.
- **RISC-V:** In addition to the assumptions required by the MIPS core, we also specify that both runs take the same conditional branch, and that the type of memory access (read or write) is the same in both runs (however, the actual values remain unrestricted). This corresponds to the assumption that programs running on the CPU do not branch or access memory based on secret values. Finally, CSR registers must not be accessed illegally (see § 6.3).
- **ALU:** Both runs execute the same type of operations (*e.g.*, bitwise, arithmetic), operands have the same bit width, instructions are valid, reset pins are the same.
- **SHA-256 and FPU (division):** We specify that the reset and input-ready bits are the same.

In all cases, we start with no assumptions and add the assumptions incrementally by manually investigating the constant-time "violation" flagged by IODINE.

Identifying Assumptions. From our experience, the assumptions that a user needs to specify fall into three categories. The first are straightforward assumptions—*e.g.*, that any two runs execute the same code. The second class of assumptions specify that certain registers need to be flushed, *i.e.*, they need to initially be the same (flushed) for any two runs. To identify these, we first flush large parts of circuits, and then, in a minimization step, we remove all unnecessary assumptions. The last, and most challenging, are implicit invariants on data and control—*e.g.*, the constraints on CSR registers. IODINE performs delta debugging to help pinpoint violations but, ultimately, these assumptions require user intervention to be resolved. Indeed, specifying these assumptions require a deep understanding of the circuit and its intended usage. In our experience, though, only a small fraction of assumptions fall into this third category.

User Hints. For one of our benchmarks (FPU), we needed to supply a small number of user hints (<5) to the solver. These hints come in the form of predicates that track additional equalities between liveness bits of the *same* run. This is required, when the two executions can take different control paths, yet execute in constant time. We hope to remove those hints in the future.

6.3 Case Studies

We now illustrate how IODINE verifies benchmarks with challenging features and helps explicate conditions under which a hardware design is constant-time, using exam-

```
1  always @(*) begin
2    if (...)
3      Stall = 1; else Stall = 0;
4  end
5  always @(posedge clk) begin
6    if (Stall)
7      ID_instr <= ID_instr;
8    else
9      ID_instr <= IF_instr;
10 end
```

Figure 14: Stalling in MIPS [5].

ples from our benchmarks.

History Dependencies. In hardware, the result of a computation often depends on inputs from previous cycles, *i.e.*, the computation depends on execution history. For example, when a hardware unit is in use by a previous instruction, the CPU stalls until the unit becomes free.

The code snippet in Fig. 14 contains a simplified version of the stalling logic from our MIPS processor benchmark. On line 3, register Stall is set to 1 if instructions in the *execute* and *instruction decode* stages conflict. Its value is then used to update the state of each pipeline stage. In this example, if the pipeline is stalled, the value of the register ID_instr, which corresponds to the instruction currently executing in the *instruction decode* stage, stays the same. Otherwise, it is updated with IF_instr—the value coming from the *instruction fetch* stage.

Without further assumptions, IODINE flags this behavior as non-constant time, as an instruction can take different times to process, depending on which other instructions are before it in the pipeline. However, after adding the assumption that any two runs execute the *same sequence of instructions*, IODINE is able to prove that Stall has the same value in any pair of traces, from which the constant time behavior follows. Importantly, however, we have no assumption on the state of the registers and memory elements that the instructions use.

Diverging Control Flow. Methods for enforcing constant time execution of software often require that any two executions take the same control flow path [16]. In hardware, this assumption is too restrictive. Consider the code snippet in Fig. 15 taken from our constant time FPU benchmark (the full logic is shown in Fig. 20 of the Appendix). The first always block calculates the sign bit of the multiplication result (`sign_mul_r`), using inputs opa and opb. The FPU uses this bit in line 17 (through `sign_mul_final`), to calculate output `out` in line 12. Even though we cannot assume that all exe-

```
1  always @(*)
2     case({opa[31], opb[31]})
3          2'b0_0: sign_mul_r <= 0;
4          2'b0_1: sign_mul_r <= 1;
5          ...
6     endcase
7  ...
8  assign sign_mul_final = (sign_exe_r & ...) ?
9              !sign_mul_r : sign_mul_r;
10 ...
11 always @(posedge clk)
12 out <= { ( ... ?
13             (f2i_out_sign &
14             !(qnan_d | snan_d) ) :
15             (((fpu_op_r3 == 3'b010)
16               & ... ?
17              sign_mul_final : ...))) };
```

Figure 15: Diverging control flow in FPU [6].

cutions select the same branches, IODINE can infer that every branch produces the same *influence sets* for the variables assigned under them. Using this information, IODINE can prove that the FPU operates in constant-time, despite diverging control flow paths.

Assumptions. IODINE can be used to inform software mechanisms for mitigating timing side-channels by explicating—and verifying—conditions under which a circuit executes in constant time. Consider Figure 16, which shows the logic for updating *Control and Status Registers (CSR)* in our RISC-V benchmark. The wire `de_illegal_csr_access`, defined on line 1 is set by checking whether a CSR instruction is executed in non-privileged mode. For this, the circuit compares the machine status register `csr_mstatus` to the instructions status bit. When `de_illegal_csr_access` is set, the branch instruction on line 8 traps the error and jumps to a predefined handler code. In order to prove that the cycle executes in constant-time, we add an assumption stating that CSR registers are not accessed illegally. This assumption translates into an obligation for software mitigation mechanisms to ensure proper use of CSR registers.

7 Limitations and Future Work

We discuss some of IODINE's limitations.

Clocks and Assumptions. For example, IODINE presupposes a single fixed-cycle clock and thus does not allow for checking arbitrary VERILOG programs. We leave an extension to multiple clocks as future work. Similarly, IODINE requires users to add assumptions by hand in somewhat ad-hoc trial-and-error fashion. For large circuits this could prove extremely difficult and poten-

```
wire de_illegal_csr_access =
     de_valid &&
     de_inst‘opcode == ‘SYSTEM &&
     de_inst‘funct3 != ‘PRIV &&
     ( csr_mstatus‘PRV < de_inst[29:28] ||
       ... );
always @(posedge clk) begin
   if (de_illegal_csr_access) begin
      ex_restart <= 1;
      ex_next_pc <= ...;
   end
end
```

Figure 16: Update of CSRs in RISC-V [7].

tially lead to errors where erroneous assumptions may lead IODINE to falsely mark a variable time circuit as constant-time. We leave the inference and validation of assumptions to future work.

Scale. We evaluate IODINE on relatively small sized (500-1000 lines) hardware designs. We did not (yet) evaluate the tool on larger circuits, such as modern processors with advanced features like a memory hierarchy, and out-of-order and transient-execution. In principle, these features boil down to the same primitives (always blocks and assignments) that IODINE already handles. But, we anticipate that scaling will require further changes to IODINE, for instance, finding per-module invariants rather than the naive in-lining currently performed by IODINE. We leave the evaluation to larger systems to future work.

8 Related Work

Constant-Time Software. Almeida et al. [16] verify constant-time execution of cryptographic libraries for LLVM. Their notion of constant-time execution is based on a *leakage model*. This choice allows them to be flexible enough to capture various properties like (timing) variability in memory access patterns and improper use of timing sensitive instructions like `DIV`. Unfortunately, their notion of constant-time is too restrictive for our setting, as it requires the control flow path of any two runs to be the same. This would, for example, incorrectly flag our FPU multiplier as variable-time. Like IODINE, their tool ct-verif employs a product construction that use the fact that loops can often be completely unrolled in cryptographic code, whereas we rely on race freedom.

Barthe et al. [20] build on the CompCert compiler [39] to enforce constant time execution through an information flow type system.

Reparaz et al. [47] present a method for discovering timing variability in existing systems through a black-box approach, based on statistical measurements.

All of these approaches address constant-time execution in software and do not translate to the hardware setting (see § 2).

Self-Composition and Product Programs. Barthe et al. [22] introduce the notion of self composition to verify information flow. Terauchi and Aiken generalize this construction to arbitrary 2-safety properties [50], *i.e.*, properties that relate two runs, and Clarkson and Schneider [28] generalize to multiple runs. Barthe et al. [21] introduce product programs that, instead of conjoining copies sequentially, compose copies in lock-step; this was later used in other tools like ct-verif. This technique is further developed in [49], which presents an extension of Hoare logic to hyper-properties that computes lock-step compositions on demand, per Hoare-triple.

Information Flow Safety and Side Channels. There are many techniques for proving information flow safety (*e.g.*, non-interference) in both hardware and software. Kwon et al. [37] prove information flow safety of hardware for policies that allow explicit declassification and are expressed over streams of input data. They construct relational invariants by using propositional interpolation and implicitly build a full self-composition; by contrast, we leverage race-freedom to create a per-thread product which contains only a subset of behaviors.

SecVerilog [55] proves timing-sensitive non-interference for circuits implemented in an extension of VERILOG that uses value-dependent information flow types. Caisson [40] is a hardware description language that uses information flow types to ensure that generated circuits are secure. GLIF [51, 52] tracks the flow of information at the gate level to eliminate explicit and covert channels. All these approaches have been used to implement information flow secure hardware that do not suffer from (timing) side-channels.

IODINE focuses on clock-precise constant-time execution, not information flow. The two properties are related, but information flow safety does not imply constant-time execution nor the converse (see Appendix A.1 for details). Moreover, SecVerilog, Caisson, and GLIF take a language-design approach whereas we take an analysis-centric view that is more suitable for verifying *existing* hardware designs. Thus, we see our work as largely complementary. Indeed, it may be useful to use IODINE alongside these HDLs to verify constant-time execution for parts of the hardware that handle secret data only, and are thus not checked for timing variability, thereby extending their attacker model.

Combining Hardware & Software Mitigations. HyperFlow [31] and GhostRider [43], take hardware/software co-design approach to eliminating timing channels. Zhang et al. [54] present a method for mitigating timing side-channels in software and give conditions on hardware that ensure the validity of mitigations is preserved. Instead of eliminating timing flows all together, they specify quantitative bounds on leakage and offers primitives to mitigate timing leaks through padding. Many other tools [11, 13, 30, 38, 48] automatically quantify leakage through timing and cache side-channels. Our approach is complementary and focuses on clock-precise analysis of existing hardware. However, the explicit assumptions that IODINE needs to verify constant-time behavior can be used to inform software mitigation techniques.

References

[1] https://github.com/scarv/xcrypto-ref.

[2] ARM A64 instruction set architecture. https://static.docs.arm.com.

[3] https://github.com/dawsonjon/fpu.

[4] https://github.com/fatestudio/RSA4096.

[5] https://github.com/gokhankici/iodine.

[6] https://github.com/monajalal/fpga_-mc/tree/master/fpu.

[7] https://github.com/tommythorn/yarvi.

[8] https://opencores.org/project/sha_core.

[9] Icarus verilog. http://iverilog.icarus.com/.

[10] Liquid fixpoint. https://github.com/ucsd-progsys.

[11] TIS-CT. http://trust-in-soft.com/tis-ct/.

[12] *IEEE Standard for Verilog Hardware Description Language.* IEEE Std 1364-2005, 2005.

[13] J Bacelar Almeida, Manuel Barbosa, Jorge S Pinto, and Bárbara Vieira. Formal verification of side-channel countermeasures using self-composition. In *Science of Computer Programming*, 2013.

[14] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *CCS*, 2017.

[15] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. Verifiable side-channel security of cryptographic implementations: Constant-time mee-cbc. In *FSE*, 2016.

[16] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *USENIX Security*, 2016.

[17] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *S&P*, 2015.

[18] Marc Andrysco, Andres Noetzli, Fraser Brown, Ranjit Jhala, and Deian Stefan. Towards verified, constant-time floating point operations. In *CCS*, 2018.

[19] Jonathan Bachrach, Huy Vo, Brian C. Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: constructing hardware in a scala embedded language. In *DAC*, 2012.

[20] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In *CCS*, 2014.

[21] Gilles Barthe, Juan Manuel Crespo, and Cesar Kunz. Relational verification using product programs. In *FM*, 2011.

[22] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *CSF*, 2004.

[23] Daniel J. Bernstein. The poly1305-aes message-authentication code. In *Fast Software Encryption*, 2005.

[24] Daniel J. Bernstein. Curve25519: New diffie-hellman speed records. In *Public Key Cryptography*, 2006.

[25] Daniel J Bernstein. The salsa20 family of stream ciphers. In *New stream cipher designs*. Springer, 2008.

[26] Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation*. 2015.

[27] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 2005.

[28] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 2010.

[29] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[30] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. Cacheaudit: A tool for the static analysis of cache side channels. In *USENIX Security*, 2013.

[31] Andrew Ferraiuolo, Mark Zhao, Andrew C Myers, and G Edward Suh. Hyperflow: A processor architecture for nonmalleable, timing-safe information flow security. In *SIGSAC*, 2018.

[32] Michael J. C. Gordon. The semantic challenge of verilog hdl. In *LICS*, 1995.

[33] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, 2012.

[34] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *CoRR*, 2018.

[35] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO*, 1996.

[36] David Kohlbrenner and Hovav Shacham. On the effectiveness of mitigations against floating-point timing channels. In *USENIX Security*, 2017.

[37] Hyoukjun Kwon, William Harris, and Hadi Esameilzadeh. Proving flow security of sequential logic via automatically-synthesized relational invariants. In *CSF*, 2017.

[38] Adam Langley. ctgrind: Checking that functions are constant time with valgrind. https://github.com/agl/ctgrind/.

[39] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL*, 2006.

[40] Xun Li, Mohit Tiwari, Jason K Oberg, Vineeth Kashyap, Frederic T Chong, Timothy Sherwood, and Ben Hardekopf. Caisson: a hardware description language for secure information flow. In *PLDI*, 2011.

[41] Linux on ARM. ARM64 prepping ARM v8.4 features, KPTI improvements for Linux 4.17. https://www.linux-arm.info/.

[42] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security*, 2018.

[43] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. *SIGPLAN Notices*, 2015.

[44] Jonas Magazinius, Alejandro Russo, and Andrei Sabelfeld. On-the-fly inlining of dynamic security monitors. In *IFIP*, 2010.

[45] Susan Owicki and David Gries. Verifying properties of parallel programs: an axiomatic approach. *Communicationsof the ACM*, 1976.

[46] Ashay Rane, Calvin Lin, and Mohit Tiwari. Secure, precise, and fast floating-point operations on x86 processors. In *USENIX Security*, 2016.

[47] Oscar Reparaz, Joseph Balasch, and Ingrid Verbauwhede. Dude, is my code constant time? In *DATE*, 2017.

[48] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F Aranha. Sparse representation of implicit flows with applications to side-channel detection. In *CCC*, 2016.

[49] Marcelo Sousa and Isil Dillig. Cartesian hoare logic for verifying k-safety properties. In *PLDI*, 2016.

[50] Tachio Terauchi and Alex Aiken. Secure information flow as a safety problem. In *SAS*, 2005.

[51] Mohit Tiwari, Jason K Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security. In *ISCA*, 2011.

[52] Mohit Tiwari, Hassan MG Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. In *Sigplan Notices*, 2009.

[53] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. Ct-wasm: Type-driven secure cryptography for the web ecosystem. 2019.

[54] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. In *PLDI*, 2012.

[55] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A hardware design language for timing-sensitive information-flow security. In *ASPLOS*, 2015.

A Appendix

A.1 Comparison to Information Flow

In this section, we discuss the relationship between constant time execution and information flow checking. Information flow safety (IFS) and constant time execution (CTE) are *incomparable*, *i.e.*, IFS does not imply CTE, and vice versa. We illustrate this using two examples: one is information flow safe but does not execute in constant time and one executes in constant time but is not information flow safe.

Figure 17 contains example program EX2 which is information flow safe but not constant time. The example contains three registers that are typed high as indicated by the annotation H, and one register that is typed low as indicated by the annotation L. The program is information flow safe, as there are no flows from high to low. Indeed, SecVerilog [55] type checks this program.

This program, however, is not constant time when $\mathsf{slow}_L \neq \mathsf{slow}_R$. This does not mean that the program leaks high data to low sinks—indeed it does not. Instead, what this means is that the high computation takes a variable amount of time dependent on the secret input values. In cases like crypto cores where the attacker has a stop watch and can measure the duration of the sensitive computation, it's not enough to be information flow safe: we must ensure the core is constant-time.

Next, consider Figure 18 that contains program EX3 which executes in constant time but is not information flow safe. EX3 violates information flow safety by assigning high input sec to low output out. The example however executes constant time with source in and sink out under the assumption that + does not contain asynchronous assignments.

```
// source(in_low); source(in_high);
// sink(out_low); sink(out_high);
module test(input  {L} clk,
            input  {L} in_low,
            input  {H} in_high,
            output {L} out_low,
            output {H} out_high);
  reg {H} flp_res;
  reg {H} slow;
  reg {L} out_low;
  reg {H} out_high;
  always @(posedge clk) begin
    out_low <= in_low;
    flp_res <= in_hi;
    if (slow)
      out_hi <= flp_res;
    else
      out_hi <= in_hi;
  end
endmodule
```

Figure 17: EX2: Non-constant time but info-flow safe.

```
// source(in); sink(out);
// □ (slow_L = slow_R);
reg {L} in;
reg {L} out;
reg {H} sec;
always @(posedge clk) begin
  out <= in + sec;
end
```

Figure 18: EX3: Constant time but not info-flow safe.

A.2 Translation

In Figure 19, we define a relation $\Rightarrow$ that translates VERILOG programs into VINTER programs. The relation is given in terms of inference rules where a transition step in the rule's conclusion (below the line) is applicable only if all its preconditions (above the line) are met. Both always- and assign-blocks are translated into threads that are executed at every clock tick using withclock. Each process is given a unique id. Our translation does not distinguish between posedge and negedge events thereby relaxing the semantics by allowing them to occur in any order. assign blocks are transformed into threads executing a continuous assignment. Blocking and non-blocking assignments remain unchanged.

$$\frac{P \Rightarrow P' \qquad Q \Rightarrow Q'}{P \cdot Q \Rightarrow P' \parallel Q'} \qquad \frac{s_1 \Rightarrow s_1' \qquad \ldots \qquad s_n \Rightarrow s_n'}{\textsf{begin } s_1;\ldots;s_n;\textsf{ end} \Rightarrow s_1';\ldots;s_n'}$$

$$\frac{s \Rightarrow s' \qquad id \text{ fresh}}{\textsf{always @(_) } s \Rightarrow \textsf{repeat } [s']_{id}}$$

$$\frac{id \text{ fresh}}{\textsf{assign } v = e \Rightarrow \textsf{repeat } [v := e]_{id}}$$

$$\frac{s_1 \Rightarrow s_1' \qquad s_2 \Rightarrow s_2'}{\textsf{if } (e)\ s_1 \textsf{ else } s_2 \textsf{ end} \ \Rightarrow\ \textsf{ite}(e, s_1', s_2')}$$

Figure 19: Translation from VERILOG to VINTER.

```
always @(*)
   case({opa[31], opb[31]})
        2'b0_0:  sign_mul_r <= 0;
        2'b0_1:  sign_mul_r <= 1;
        ...
   endcase
assign sign_mul_final =
  (sign_exe_r &
   ((opa_00 & opb_inf) |
    (opb_00 & opa_inf))) ?
   !sign_mul_r : sign_mul_r;
always @(posedge clk)
  out <= {
   (((fpu_op_r3 == 3'b101) & out_d_00) ?
    (f2i_out_sign & !(qnan_d | snan_d)) :
    (((fpu_op_r3 == 3'b010) &
      !(snan_d | qnan_d)) ?
     sign_mul_final :
     (((fpu_op_r3 == 3'b011) &
       !(snan_d | qnan_d)) ? sign_div_final :
      ((snan_d | qnan_d | ind_d) ?
       nan_sign_d :
       (output_zero_fasu ?
        result_zero_sign_d :
        sign_fasu_r))))) ,
   ((mul_inf | div_inf |
     (inf_d & (fpu_op_r3 != 3'b011) &
      (fpu_op_r3 != 3'b101)) |
     snan_d | qnan_d) &
    fpu_op_r3 != 3'b100 ? out_fixed :
out_d) };
```

Figure 20: Example diverging computation in [6]

VRASED: A Verified Hardware/Software Co-Design for Remote Attestation

Ivan De Oliveira Nunes
University of California, Irvine
ivanoliv@uci.edu

Karim Eldefrawy
SRI International
karim.eldefrawy@sri.com

Norrathep Rattanavipanon
University of California, Irvine
nrattana@uci.edu

Michael Steiner
Intel
michael.steiner@intel.com

Gene Tsudik
University of California, Irvine
gene.tsudik@uci.edu

Abstract

Remote Attestation (RA) is a distinct security service that allows a trusted verifier ($\mathcal{V}$rf) to measure the software state of an untrusted remote prover ($\mathcal{P}$rv). If correctly implemented, RA allows $\mathcal{V}$rf to remotely detect if $\mathcal{P}$rv is in an illegal or compromised state. Although several RA approaches have been explored (including hardware-based, software-based, and hybrid) and many concrete methods have been proposed, comparatively little attention has been devoted to formal verification. In particular, thus far, no RA designs and no implementations have been formally verified with respect to claimed security properties.

In this work, we take the first step towards formal verification of RA by designing and verifying an architecture called *VRASED*: **V**erifiable **R**emote **A**ttestation for **S**imple **E**mbedded **D**evices. *VRASED* instantiates a hybrid (HW/SW) RA co-design aimed at low-end embedded systems, e.g., simple IoT devices. *VRASED* provides a level of security comparable to HW-based approaches, while relying on SW to minimize additional HW costs. Since security properties must be jointly guaranteed by HW and SW, verification is a challenging task, which has never been attempted before in the context of RA. We believe that *VRASED* is the first formally verified RA scheme. To the best of our knowledge, it is also the first formal verification of a HW/SW co-design implementation of any security service. To demonstrate *VRASED*'s practicality and low overhead, we instantiate and evaluate it on a commodity platform (TI MSP430). *VRASED* was deployed using the Basys3 Artix-7 FPGA and its implementation is publicly available.

1 Introduction

The number and variety of special-purpose computing devices is increasing dramatically. This includes all kinds of embedded devices, cyber-physical systems (CPS) and Internet-of-Things (IoT) gadgets, that are utilized in various "smart" settings, such as homes, offices, factories, automotive systems and public venues. As society becomes increasingly accustomed to being surrounded by, and dependent on, such devices, their security becomes extremely important. For actuation-capable devices, malware can impact both security and safety, e.g., as demonstrated by Stuxnet [49]. Whereas, for sensing devices, malware can undermine privacy by obtaining ambient information. Furthermore, clever malware can turn vulnerable IoT devices into zombies that can become sources for DDoS attacks. For example, in 2016, a multitude of compromised "smart" cameras and DVRs formed the Mirai Botnet [2] which was used to mount a massive-scale DDoS attack (the largest in history).

Unfortunately, security is typically not a key priority for low-end device manufacturers, due to cost, size or power constraints. It is thus unrealistic to expect such devices to have the means to prevent current and future malware attacks. The next best thing is detection of malware presence. This typically requires some form of **Remote Attestation** (RA) – a distinct security service for detecting malware on CPS, embedded and IoT devices. RA is especially applicable to low-end embedded devices that are incapable of defending themselves against malware infection. This is in contrast to more powerful devices (both embedded and general-purpose) that can avail themselves of sophisticated anti-malware protection. RA involves verification of current internal state (i.e., RAM and/or flash) of an untrusted remote hardware platform (prover or $\mathcal{P}$rv) by a trusted entity (verifier or $\mathcal{V}$rf). If $\mathcal{V}$rf detects malware presence, $\mathcal{P}$rv's software can be re-set or rolled back and out-of-band measures can be taken to prevent similar infections. In general, RA can help $\mathcal{V}$rf establish a static or dynamic root of trust in $\mathcal{P}$rv and can also be used to construct other security services, such as software updates [43] and secure deletion [40]. Hybrid RA (implemented as a HW/SW co-design) is a particularly promising approach for low-end embedded devices. It aims to provide the same security guarantees as (more expensive) hardware-based approaches, while minimizing modifications to the underlying hardware.

Even though numerous RA techniques with different assumptions, security guarantees, and designs, have been proposed [9, 10, 14–16, 20, 21, 25, 30, 35, 38, 38–40, 43], a major missing aspect of RA is the high-assurance and rigor derivable from utilizing computer-aided formal verification to guarantee security of the design and implementation of RA techniques. Because all aforementioned architectures and their implementations are not systematically designed from abstract models, their soundness and security can not be formally argued. In fact, our RA verification efforts revealed that a previous hybrid RA design – SMART [21] – assumed that disabling interrupts is an atomic operation and hence opened the door to compromise of $\mathcal{P}$rv's secret key in the window between the time of

the invocation of disable interrupts functionality and the time when interrupts are actually disabled. Another low/medium-end architecture – Trustlite [30] – does not achieve our formal definition of RA soundness. As a consequence, this architecture is vulnerable to self-relocating malware (See [13] for details). Formal specification of RA properties and their verification significantly increases our confidence that such subtle issues are not overlooked.

In this paper we take a "verifiable-by-design" approach and develop, from scratch, an architecture for **V**erifiable **R**emote **A**ttestation for **S**imple **E**mbedded **D**evices (*VRASED*). *VRASED* is the first formally specified and verified RA architecture accompanied by a formally verified implementation. Verification is carried out for all trusted components, including hardware, software, and the composition of both, all the way up to end-to-end notions for RA soundness and security. The resulting verified implementation – along with its computer proofs – is publicly available [1]. Formally reasoning about, and verifying, *VRASED* involves overcoming major challenges that have not been attempted in the context of RA and, to the best of our knowledge, not attempted for any security service implemented as a HW/SW co-design. These challenges include:

1 – Formal definitions of: `(i)` end-to-end notions for RA soundness and security; `(ii)` a realistic machine model for low-end embedded systems; and `(iii)` *VRASED*'s guarantees. These definitions must be made in single formal system that is powerful enough to provide a common ground for reasoning about their interplay. In particular, our end goal is to prove that the definitions for RA soundness and security are implied by *VRASED*'s guarantees when applied to our machine model. Our formal system of choice is Linear Temporal Logic (LTL). A background on LTL and our reasons for choosing it are discussed in Section 2.

2 – Automatic end-to-end verification of complex systems such as *VRASED* is challenging from the computability perspective, as the space of possible states is extremely large. To cope with this challenge, we take a "divide-to-conquer" approach. We start by dividing the end-to-end goal of RA soundness and security into smaller sub-properties that are also defined in LTL. Each HW sub-module, responsible for enforcing a given sub-property, is specified as a Finite State Machine (FSM), and verified using a Model Checker. *VRASED*'s SW relies on an F* verified implementation (see Section 4.3) which is also specified in LTL. This modular approach allows us to efficiently prove sub-properties enforced by individual building blocks in *VRASED*.

3 – All proven sub-properties must be composed together in order to reason about RA security and soundness of *VRASED* as one whole system. To this end, we use a theorem prover to show (by using LTL equivalences) that the sub-properties that were proved for each of *VRASED*'s sub-modules, when composed, imply the end-to-end definitions of RA soundness and security. This modular approach enables efficient system-wide formal verification.

1.1 The Scope of Low-End Devices

This work focuses on low-end devices based on low-power single core microcontrollers with a few KBytes of program and data memory. A representative of this class of devices is the Texas Instrument's MSP430 microcontroller (MCU) family [26]. It has a 16-bit word size, resulting in $\approx$ 64 KBytes of addressable memory. SRAM is used as data memory and its size ranges between 4 and 16KBytes (depending on the specific MSP430 model), while the rest of the address space can be used for program memory, e.g., `ROM` and Flash. MSP430 is a Von Neumann architecture processor with common data and code address spaces. It can perform multiple memory accesses within a single instruction; its instruction execution time varies from 1 to 6 clock cycles, and instruction length varies from 16 to 48 bits. MSP430 was designed for low-power and low-cost. It is widely used in many application domains, e.g., automotive industry, utility meters, as well as consumer devices and computer peripherals. Our choice is also motivated by availability of a well-maintained open-source MSP430 hardware design from Open Cores [22]. Nevertheless, our machine model is applicable to other low-end MCUs in the same class as MSP430 (e.g., Atmel AVR ATMega).

1.2 Organization

Section 2 provides relevant background on RA and formal verification. Section 3 contains the details of the *VRASED* architecture and an overview of the verification approach. Section 4 contains the formal definitions of end-to-end RA soundness and security and the formalization of the necessary sub-properties along with the implementation of verified components to realize such sub-properties. Due to space limitation, the proofs for end-to-end soundness and security derived from the sub-properties are discussed in Appendix A. Section 5 discusses alternative designs to guarantee the same required properties and their trade-offs with the standard design. Section 6 presents experimental results demonstrating the minimal overhead of the formally verified and synthesized components. Section 7 discusses related work. Section 8 concludes with a summary of our results. End-to-end proofs of soundness and security, optional parts of the design, *VRASED*'s API, and discussion on *VRASED*'s prototype can be found in Appendices A to C.

2 Background

This section overviews RA and provides some background on computer-aided verification.

2.1 RA for Low-end Devices

As mentioned earlier, RA is a security service that facilitates detection of malware presence on a remote device. Specifically, it allows a trusted verifier ($\mathcal{V}$rf) to remotely measure the software state of an untrusted remote device ($\mathcal{P}$rv). As shown in Figure 1, RA is typically obtained via a simple challenge-response protocol:

1. $\mathcal{V}$rf sends an attestation request containing a challenge ($\mathcal{C}$hal) to $\mathcal{P}$rv. This request might also contain a token derived from a secret that allows $\mathcal{P}$rv to authenticate $\mathcal{V}$rf.
2. $\mathcal{P}$rv receives the attestation request and computes an *authenticated integrity check* over its memory and $\mathcal{C}$hal. The memory region might be either pre-defined, or explicitly specified in the request. In the latter case, authentication of $\mathcal{V}$rf in step (1) is paramount to the overall security/privacy of $\mathcal{P}$rv, as the request can specify arbitrary memory regions.
3. $\mathcal{P}$rv returns the result to $\mathcal{V}$rf.
4. $\mathcal{V}$rf receives the result from $\mathcal{P}$rv, and checks whether it corresponds to a valid memory state.

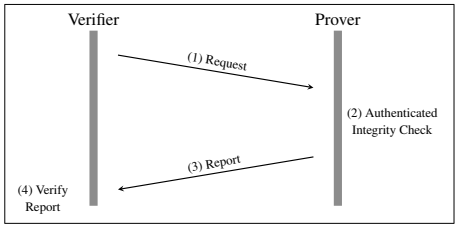


Figure 1: Remote attestation (RA) protocol

The *authenticated integrity check* can be realized as a Message Authentication Code (MAC) over $\mathcal{P}$rv's memory. However, computing a MAC requires $\mathcal{P}$rv to have a unique secret key (denoted by $\mathcal{K}$) shared with $\mathcal{V}$rf. This $\mathcal{K}$ must reside in secure storage, where it is **not** accessible to any software running on $\mathcal{P}$rv, except for attestation code. Since most RA threat models assume a fully compromised software state on $\mathcal{P}$rv, secure storage implies some level of hardware support.

Prior RA approaches can be divided into three groups: software-based, hardware-based, and hybrid. Software-based (or timing-based) RA is the only viable approach for legacy devices with no hardware security features. Without hardware support, it is (currently) impossible to guarantee that $\mathcal{K}$ is not accessible by malware. Therefore, security of software-based approaches [35, 44] is attained by setting threshold communication delays between $\mathcal{V}$rf and $\mathcal{P}$rv. Thus, software-based RA is unsuitable for multi-hop and jitter-prone communication, or settings where a compromised $\mathcal{P}$rv is aided (during attestation) by a more powerful accomplice device. It also requires strong constraints and assumptions on the hardware platform and attestation usage [31, 34]. On the other extreme, hardware-based approaches require either i) $\mathcal{P}$rv's attestation functionality to be housed entirely within dedicated hardware, e.g., Trusted Platform Modules (TPMs) [47]; or ii) modifications to the CPU semantics or instruction sets to support the execution of trusted software, e.g., SGX [27] or TrustZone [3]. Such hardware features are too expensive (in terms of physical area, energy consumption, and actual cost) for low-end devices.

While neither hardware- nor software-based approaches are well-suited for settings where low-end devices communicate over the Internet (which is often the case in the IoT), hybrid RA (based on HW/SW co-design) is a more promising approach. Hybrid RA aims at providing the same security guarantees as hardware-based techniques with minimal hardware support. SMART [21] is the first hybrid RA architecture targeting low-end MCUs. In SMART, attestation's integrity check is implemented in software. SMART's small hardware footprint guarantees that the attestation code runs safely and that the attestation key is not leaked. HYDRA [20] is a hybrid RA scheme that relies on a secure boot hardware feature and on a secure micro-kernel. Trustlite [30] modifies Memory Protection Unit (MPU) and CPU exception engine hardware to implement RA. Tytan [9] is built on top of Trustlite, extending its capabilities for applications with real-time requirements.

Despite much progress, a major missing aspect in RA research is high-assurance and rigor obtained by using formal methods to guarantee security of a concrete RA design and its implementation We believe that verifiability and formal security guarantees are particularly important for hybrid RA designs aimed at low-end embedded and IoT devices, as their proliferation keeps growing. This serves as the main motivation for our efforts to develop the first formally verified RA architecture.

2.2 Formal Verification, Model Checking & Linear Temporal Logic

Computer-aided formal verification typically involves three basic steps. First, the system of interest (e.g., hardware, software, communication protocol) must be described using a formal model, e.g., a Finite State Machine (FSM). Second, properties that the model should satisfy must be formally specified. Third, the system model must be checked against formally specified properties to guarantee that the system retains such properties. This checking can be achieved via either Theorem Proving or Model Checking.

In Model Checking, properties are specified as *formulae* using Temporal Logic and system models are represented as FSMs. Hence, a system is represented by a triple (S, S_0, T), where S is a finite set of states, $S_0 \subseteq S$ is the set of possible initial states, and $T \subseteq S \times S$ is the transition relation set, i.e., it describes the set of states that can be reached in a single step from each state. The use of Temporal Logic to specify properties allows representation of expected system behavior over time.

We apply the model checker NuSMV [17], which can be

used to verify generic HW or SW models. For digital hardware described at Register Transfer Level (RTL) – which is the case in this work – conversion from Hardware Description Language (HDL) to NuSMV model specification is simple. Furthermore, it can be automated [28]. This is because the standard RTL design already relies on describing hardware as an FSM.

In NuSMV, properties are specified in Linear Temporal Logic (LTL), which is particularly useful for verifying sequential systems. This is because it extends common logic statements with temporal clauses. In addition to propositional connectives, such as conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), and implication ($\rightarrow$), LTL includes temporal connectives, thus enabling sequential reasoning. We are interested in the following temporal connectives:

- $\mathbf{X}\phi$ – ne$\underline{\text{X}}$t ϕ: holds if ϕ is true at the next system state.
- $\mathbf{F}\phi$ – $\underline{\text{F}}$uture ϕ: holds if there exists a future state where ϕ is true.
- $\mathbf{G}\phi$ – $\underline{\text{G}}$lobally ϕ: holds if for all future states ϕ is true.
- ϕ **U** ψ – ϕ $\underline{\text{U}}$ntil ψ: holds if there is a future state where ψ holds and ϕ holds for all states prior to that.

This set of temporal connectives combined with propositional connectives (with their usual meanings) allows us to specify powerful rules. NuSMV works by checking LTL specifications against the system FSM for all reachable states in such FSM. In particular, all *VRASED*'s desired security sub-properties are specified using LTL and verified by NuSMV. Finally, a theorem prover [19] is used to show (via LTL equivalences) that the verified sub-properties imply end-to-end definitions of RA soundness and security.

3 Overview of VRASED

VRASED is composed of a HW module (`HW-Mod`) and a SW implementation (`SW-Att`) of $\mathcal{P}$rv's behavior according to the RA protocol. `HW-Mod` enforces access control to $\mathcal{K}$ in addition to secure and atomic execution of `SW-Att` (these properties are discussed in detail below). `HW-Mod` is designed with minimality in mind. The verified FSMs contain a minimal state space, which keeps hardware cost low. `SW-Att` is responsible for computing an attestation report. As *VRASED*'s security properties are jointly enforced by `HW-Mod` and `SW-Att`, both must be verified to ensure that the overall design conforms to the system specification.

3.1 Adversarial Capabilities & Verification Axioms

We consider an adversary, $\mathcal{A}$, that can control the entire software state, code, and data of $\mathcal{P}$rv. $\mathcal{A}$ can modify any writable memory and read any memory that is not explicitly protected by access control rules, i.e., it can read anything (including secrets) that is not explicitly protected by `HW-Mod`. It can also re-locate malware from one memory segment to another, in order to hide it from being detected. $\mathcal{A}$ may also have full control over all Direct Memory Access (DMA) controllers on $\mathcal{P}$rv. DMA allows a hardware controller to directly access main memory (e.g., `RAM`, flash or `ROM`) without going through the CPU.

We focus on attestation functionality of $\mathcal{P}$rv; verification of the entire MCU architecture is beyond the scope of this paper. Therefore, we assume the MCU architecture strictly adheres to, and correctly implements, its specifications. In particular, our verification approach relies on the following simple axioms:

- **A1 - Program Counter:** The program counter (*PC*) always contains the address of the instruction being executed in a given cycle.
- **A2 - Memory Address:** Whenever memory is read or written, a data-address signal (D_{addr}) contains the address of the corresponding memory location. For a read access, a data read-enable bit (R_{en}) must be set, and for a write access, a data write-enable bit (W_{en}) must be set.
- **A3 - DMA:** Whenever a DMA controller attempts to access main system memory, a DMA-address signal (DMA_{addr}) reflects the address of the memory location being accessed and a DMA-enable bit (DMA_{en}) must be set. DMA can not access memory when DMA_{en} is off (logical zero).
- **A4 - MCU reset:** At the end of a successful *reset* routine, all registers (including *PC*) are set to zero before resuming normal software execution flow. Resets are handled by the MCU in hardware; thus, reset handling routine can not be modified.
- **A5 - Interrupts:** When interrupts happen, the corresponding *irq* signal is set.

Remark: *Note that Axioms **A1** to **A5** are satisfied by the OpenMSP430 design.*

`SW-Att` uses the HACL* [52] HMAC-SHA256 function which is implemented and verified in F*[1]. F* can be automatically translated to C and the proof of correctness for the translation is provided in [41]. However, even though efforts have been made to build formally verified C compilers (CompCert [33] is the most prominent example), there are currently no verified compilers targeting lower-end MCUs, such as MSP430. Hence, we assume that the standard compiler can be trusted to semantically preserve its expected behavior, especially with respect to the following:

- **A6 - Callee-Saves-Register:** Any register touched in a function is cleaned by default when the function returns.
- **A7 - Semantic Preservation:** Functional correctness of the verified HMAC implementation in C, when converted to assembly, is semantically preserved.

Remark: *Axioms **A6** and **A7** reflect the corresponding compiler specification (e.g., msp430-gcc).*

Physical hardware attacks are out of scope in this paper.

[1] `https://www.fstar-lang.org/`

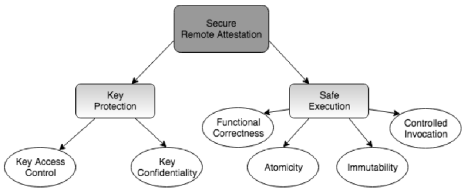


Figure 2: Properties of secure RA.

Specifically, $\mathcal{A}$ can not modify code stored in ROM, induce hardware faults, or retrieve $\mathcal{P}$rv secrets via physical presence side-channels. Protection against physical attacks is considered orthogonal and could be supported via standard tamper-resistance techniques [42].

3.2 High-Level Properties of Secure Attestation

We now describe, in high level, the sub-properties required for RA. In section 4, we formalize these sub-properties in LTL and provide single end-to-end definitions for RA soundness and security. Then we prove that *VRASED*'s design satisfies the aforementioned sub-properties and that the end-to-end definitions for soundness and security are implied by them. The properties, shown in Figure 2, fall into two groups: *key protection* and *safe execution*.

Key Protection:

As mentioned earlier, $\mathcal{K}$ must not be accessible by regular software running on $\mathcal{P}$rv. To guarantee this, the following features must be correctly implemented:

- **P1- Access Control:** $\mathcal{K}$ can only be accessed by SW-Att.
- **P2- No Leakage:** Neither $\mathcal{K}$ (nor any function of $\mathcal{K}$ other than the correctly computed HMAC) can remain in unprotected memory or registers after execution of SW-Att.
- **P3- Secure Reset:** Any memory tainted by $\mathcal{K}$ and all registers (including PC) must be erased (or be inaccessible to regular software) after MCU reset. Since a reset might be triggered during SW-Att execution, lack of this property could result in leakage of privileged information about the system state or $\mathcal{K}$. Erasure of registers as part of the reset ensures that no state from a previous execution persists. Therefore, the system must return to the default initialization state.

Safe Execution:

Safe execution ensures that $\mathcal{K}$ is properly and securely used by SW-Att for its intended purpose in the RA protocol. Safe execution can be divided into four sub-properties:

- **P4- Functional Correctness:** SW-Att must implement expected behavior of $\mathcal{P}$rv's role in the RA protocol. For instance, if $\mathcal{V}$rf expects a response containing an HMAC of memory in address range $[A,B]$, SW-Att implementation should always reply accordingly. Moreover, SW-Att must always finish in finite time, regardless of input size and other parameters.
- **P5- Immutability:** SW-Att executable must be immutable. Otherwise, malware residing in $\mathcal{P}$rv could modify SW-Att, e.g., to always generate valid RA measurements or to leak $\mathcal{K}$.
- **P6- Atomicity:** SW-Att execution can not be interrupted. The first reason for atomicity is to prevent leakage of intermediate values in registers and SW-Att's data memory (including locations that could leak functions of $\mathcal{K}$) during SW-Att execution. This relates to **P2** above. The second reason is to prevent roving malware from relocating itself to escape being measured by SW-Att.
- **P7- Controlled Invocation:** SW-Att must always start from the first instruction and execute until the last instruction. Even though correct implementation of SW-Att is guaranteed by **P4**, isolated execution of chunks of a correctly implemented code could lead to catastrophic results. Potential ROP attacks could be constructed using gadgets of SW-Att (which, based on **P1**, have access to $\mathcal{K}$) to compute valid attestation results.

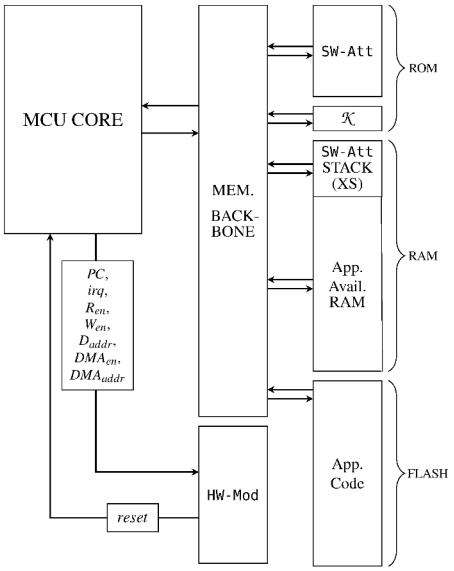


Figure 3: VRASED system architecture

Beyond aforementioned core security properties, in some settings, $\mathcal{P}$rv might need to authenticate $\mathcal{V}$rf's attestation requests in order to mitigate potential DoS attacks on $\mathcal{P}$rv. This functionality is also provided (and verified) as an optional feature in the design of *VRASED*. The differences between the standard design and the one with support for $\mathcal{V}$rf authentication are discussed in Appendix B.

3.3 System Architecture

VRASED architecture is depicted in Figure 3. *VRASED* is implemented by adding HW-Mod to the MCU architecture, e.g., MSP430. MCU memory layout is extended to include Read-Only Memory (ROM) that houses SW-Att code and $\mathcal{K}$ used in the HMAC computation. Because $\mathcal{K}$ and SW-Att code are stored in ROM, we have guaranteed immutability, i.e., **P5**. *VRASED* also reserves a fixed part of the memory address space for SW-Att stack. This amounts to $\approx 3\%$ of the address space, as discussed in Section 6 [2]. Access control to dedicated memory regions, as well as SW-Att atomic execution are enforced by HW-Mod. The memory backbone is extended to support multiplexing of the new memory regions. HW-Mod takes 7 input signals from the MCU core: PC, irq, D_{addr}, R_{en}, W_{en}, DMA_{addr} and DMA_{en}. These inputs are used to determine a one-bit *reset* signal output, that, when set to 1, resets the MCU core immediately, i.e., before execution of the next instruction. The *reset* output is triggered when HW-Mod detects any violation of security properties [3].

3.4 Verification Approach

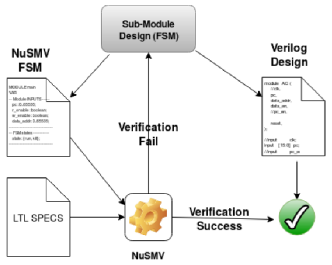


Figure 4: *VRASED*'s submodule verification

An overview of HW-Mod verification is shown in Figures 4 and 5. We start by formalizing RA sub-properties discussed in this section using Linear Temporal Logic (LTL) to define invariants that must hold throughout the entire system execution. HW-Mod is implemented as a composition of sub-modules written in the Verilog hardware description language (HDL). Each sub-module implements the hardware responsible for ensuring a given subset of the LTL specifications. Each sub-module is described as an FSM in: (1) Verilog at Register Transfer Level (RTL); and (2) the Model-Checking language SMV [17]. We then use the NuSMV model checker to verify that the FSM complies with the LTL specifications. If verification fails, the sub-module is re-designed.

Once each sub-module is verified, they are combined into a single Verilog design. The composition is converted to SMV using the automatic translation tool Verilog2SMV [28]. The resulting SMV is simultaneously verified against all LTL specifications to prove that the final Verilog design for HW-Mod complies with all secure RA properties.

We clarify that the individual SMV sub-modules' design and verification steps are not strictly required in the verification pipeline. This is because verifying SMV that is automatically translated from the composition of HW-Mod would suffice. Nevertheless, we design FSMs in SMV first so as to facilitate sub-modules' development and reasoning with an early additional check before going into their actual implementation and composition in Verilog.

Remark: *Automatic conversion of the composition of HW-Mod from Verilog to SMV rules out the possibility of human mistakes in representing Verilog FSMs as SMV.*

For the SW-Att part of *VRASED*, we use the HMAC-SHA-256 from the HACL* library [52] to compute an authenticated integrity check of attested memory and $\mathcal{C}$hal received from $\mathcal{V}$rf. This function is formally verified with respect to memory safety, functional correctness, and cryptographic security. However, key secrecy properties (such as clean-up of memory tainted by the key) are not formally verified in HACL* and thus must be ensured by HW-Mod.

As the last step, we prove that the conjunction of the LTL properties guaranteed by HW-Mod and SW-Att implies soundness and security of the RA architecture. These are formally specified in Section 4.2.

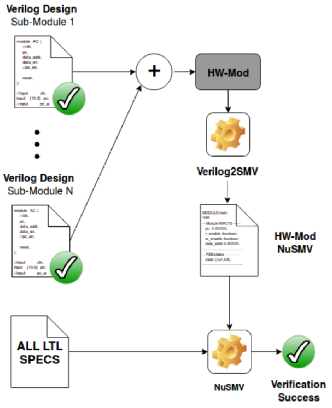


Figure 5: Verification framework for the composition of sub-modules (HW-Mod).

4 Verifying VRASED

In this section we formalize RA sub-properties. For each sub-property, we represent it as a set of LTL specifications and construct an FSM that is verified to conform to such specifications. Finally, the conjunction of these FSMs is implemented in Verilog HDL and translated to SMV using Verilog2SMV. The

[2] A separate region in RAM is not strictly required. Alternatives and trade-offs are discussed in Section 5

[3] Resets due to *VRASED* violations do not give malware advantages as malware can always trigger resets on the unmodified MCU by inducing software faults.

generated SMV description for the conjunction is proved to simultaneously hold for all specifications. We also define end-to-end soundness and security goals which are derived from the verified sub-properties (See Appendix A for the proof).

4.1 Notation

To facilitate generic LTL specifications that represent *VRASED*'s architecture (see Figure 3) we use the following:

- AR_{min} and AR_{max}: first and last physical addresses of the memory region to be attested;
- CR_{min} and CR_{max}: physical addresses of first and last instructions of `SW-Att` in `ROM`;
- K_{min} and K_{max}: first and last physical addresses of the `ROM` region where $\mathcal{K}$ is stored;
- XS_{min} and XS_{max}: first and last physical addresses of the `RAM` region reserved for `SW-Att` computation;
- MAC_{addr}: fixed address that stores the result of `SW-Att` computation (HMAC);
- MAC_{size}: size of HMAC result;

Table 1 uses the above definitions and summarizes the notation used in our LTL specifications throughout the rest of this paper.

To simplify specification of defined security properties, we use $[A,B]$ to denote a contiguous memory region between A and B. Therefore, the following equivalence holds:

$$C \in [A,B] \Leftrightarrow (C \leq B \wedge C \geq A) \tag{1}$$

For example, expression $PC \in CR$ holds when the current value of PC signal is within CR_{min} and CR_{max}, meaning that the MCU is currently executing an instruction in CR, i.e, a `SW-Att` instruction. This is because in the notation introduced above: $PC \in CR \Leftrightarrow PC \in [CR_{min}, CR_{max}] \Leftrightarrow (PC \leq CR_{max} \wedge PC \geq CR_{min})$.

FSM Representation. As discussed in Section 3, `HW-Mod` sub-modules are represented as FSMs that are verified to hold for LTL specifications. These FSMs correspond to the Verilog hardware design of `HW-Mod` sub-modules. The FSMs are implemented as Mealy machines, where output changes at any time as a function of both the current state and current input values[4]. Each FSM has as inputs a subset of the following signals and wires: $\{PC, irq, R_{en}, W_{en}, D_{addr}, DMA_{en}, DMA_{addr}\}$.

Each FSM has only one output, *reset*, that indicates whether any security property was violated. For the sake of presentation, we do not explicitly represent the value of the *reset* output for each state. Instead, we define the following implicit representation:

1. *reset* output is 1 whenever an FSM transitions to the *Reset* state;
2. *reset* output remains 1 until a transition leaving the *Reset* state is triggered;
3. *reset* output is 0 in all other states.

[4]This is in contrast with Moore machines where the output is defined solely based on the current state.

Table 1: Notation summary

Notation	Description
PC	Current Program Counter value
R_{en}	Signal that indicates if the MCU is reading from memory (1-bit)
W_{en}	Signal that indicates if the MCU is writing to memory (1-bit)
D_{addr}	Address for an MCU memory access
DMA_{en}	Signal that indicates if DMA is currently enabled (1-bit)
DMA_{addr}	Memory address being accessed by DMA, if any
irq	Signal that indicates if and interrupt is occurring (1-bit)
CR	(Code `ROM`) Memory region where `SW-Att` is stored: $CR = [CR_{min}, CR_{max}]$
KR	($\mathcal{K}$ `ROM`) Memory region where $\mathcal{K}$ is stored: $KR = [K_{min}, K_{max}]$
XS	(eXclusive Stack) secure `RAM` region reserved for `SW-Att` computations: $XS = [XS_{min}, XS_{max}]$
MR	(MAC `RAM`) `RAM` region in which `SW-Att` computation result is written: $MR = [MAC_{addr}, MAC_{addr} + MAC_{size} - 1]$. The same region is also used to pass the attestation challenge as input to `SW-Att`
AR	(Attested Region) Memory region to be attested. Can be fixed/predefined or specified in an authenticated request from $\mathcal{V}$rf: $AR = [AR_{min}, AR_{max}]$
$reset$	A 1-bit signal that reboots the MCU when set to logic 1
A1, A2, ..., A7	Verification axioms (outlined in section 3.1)
P1, P2, ..., P7	Properties required for secure RA (outlined in section 3.2)

4.2 Formalizing RA Soundness and Security

We now define the notions of soundness and security. Intuitively, RA soundness corresponds to computing an integrity ensuring function over memory at time t. Our integrity ensuring function is an HMAC computed on memory AR with a one-time key derived from $\mathcal{K}$ and $\mathcal{C}$hal. Since `SW-Att` computation is not instantaneous, RA soundness must ensure that attested memory does not change during computation of the HMAC. This is the notion of temporal consistency in remote attestation [14]. In other words, the result of `SW-Att` call must reflect the entire state of the attested memory at the time when `SW-Att` is called. This notion is captured in LTL by Definition 1.

Definition 1. *End-to-end definition for soundness of* RA *computation*

$$\boldsymbol{G}: \{\, PC = CR_{min} \wedge AR = M \wedge MR = \mathcal{C}\mathsf{hal} \wedge [(\neg reset)\ \boldsymbol{U}\ (PC = CR_{max})] \rightarrow$$
$$\boldsymbol{F}: [PC = CR_{max} \wedge MR = HMAC(KDF(\mathcal{K}, \mathcal{C}\mathsf{hal}), M)] \,\}$$

where M is any AR value and KDF is a secure key derivation function.

In Definition 1, $PC = CR_{min}$ captures the time when `SW-Att` is called (execution of its first instruction). M and $\mathcal{C}$hal are the values of AR and MR. From this pre-condition, Definition 1 asserts that there is a time in the future when `SW-Att` computation finishes and, at that time, MR stores the result of $HMAC(KDF(\mathcal{K}, \mathcal{C}\mathsf{hal}), M)$. Note that, to satisfy Definition 1, $\mathcal{C}$hal and M in the resulting HMAC must correspond to the values in AR and MR, respectively, when `SW-Att` was called.

RA security is defined using the security game in Figure 6.

It models an adversary $\mathcal{A}$ (that is a probabilistic polynomial time, ppt, machine) that has full control of the software state of $\mathcal{P}$rv (as the one described in Section 3.1). It can modify *AR* at will and call SW-Att a polynomial number of times in the security parameter ($\mathcal{K}$ and $\mathcal{C}$hal bit-lengths). However, $\mathcal{A}$ can not modify SW-Att code, which is stored in immutable memory. The game assumes that $\mathcal{A}$ does not have direct access to $\mathcal{K}$, and only learns $\mathcal{C}$hal after it receives from $\mathcal{V}$rf as part of the attestation request.

Definition 2.
2.1 RA ***Security Game*** (RA-***game*****):**
Assumptions:
- SW-Att is immutable, and $\mathcal{K}$ is not known to $\mathcal{A}$
- l is the security parameter and $|\mathcal{K}| = |\mathcal{C}\text{hal}| = |MR| = l$
- $AR(t)$ denotes the content in AR at time t
- $\mathcal{A}$ can modify AR and MR at will; however, it loses its ability to modify them while SW-Att is running

RA-*game:*
1. *Setup: $\mathcal{A}$ is given oracle access to SW-Att.*
2. *Challenge: A random challenge $\mathcal{C}\text{hal} \leftarrow \${0,1}^l$ is generated and given to $\mathcal{A}$. $\mathcal{A}$ continues to have oracle access to SW-Att.*
3. *Response: Eventually, $\mathcal{A}$ responds with a pair (M, σ), where σ is either forged by $\mathcal{A}$, or the result of calling SW-Att at some arbitrary time t.*
4. *$\mathcal{A}$ wins if and only if $\sigma = HMAC(KDF(\mathcal{K}, \mathcal{C}\text{hal}), M)$ and $M \neq AR(t)$.*

2.2 RA ***Security Definition:***
An RA *protocol is considered secure if there is no ppt $\mathcal{A}$, polynomial in l, capable of winning the game defined in 2.1 with $Pr[\mathcal{A}, \text{RA-}game] > negl(l)$*

Figure 6: RA security definition for *VRASED*

In the following sections, we define SW-Att functional correctness, LTL specifications 2-10 and formally verify that *VRASED*'s design guarantees such LTL specifications. We define LTL specifications from the intuitive properties discussed in Section 3.2 and depicted in Figure 2. In Appendix A we prove that the conjunction of such properties achieves soundness (Definition 1) and security (Definition 2). For the security proof, we first show that *VRASED* guarantees that $\mathcal{A}$ can never learn $\mathcal{K}$ with more than negligible probability, thus satisfying the assumption in the security game. We then complete the proof of security via reduction, i.e., show that existence of an adversary that wins the game in Definition 2 implies the existence of an adversary that breaks the conjectured existential unforgeability of HMAC.

Remark: *The rest of this section focuses on conveying the intuition behind the specification of LTL sub-properties. Therefore, our references to the MCU machine model are via Axioms* ***A1 - A7*** *which were described in high level. The interested reader can find an LTL machine model formalizing these notions in Appendix A, where we describe how such machine model is used construct computer proofs for Definitions 1 and 2.*

4.3 *VRASED* SW-Att

To minimize required hardware features, hybrid RA approaches implement integrity ensuring functions (e.g., HMAC) in software. *VRASED*'s SW-Att implementation is built on top of

```
1  void Hacl_HMAC_SHA2_256_hmac_entry() {
2      uint8_t key[64] = {0};
3      memcpy(key, (uint8_t*) KEY_ADDR, 64);
4      hacl_hmac((uint8_t*) key, (uint8_t*) key, (uint32_t) 64, (uint8_t*)
           CHALL_ADDR, (uint32_t) 32);
5      hacl_hmac((uint8_t*) MAC_ADDR, (uint8_t*) key, (uint32_t) 32, (uint8_t*)
           ATTEST_DATA_ADDR, (uint32_t) ATTEST_SIZE);
6      return();
7  }
```

Figure 7: SW-Att C Implementation

HACL*'s HMAC implementation [52]. HACL* code is verified to be functionally correct, memory safe and secret independent. In addition, all memory is allocated on the stack making it predictable and deterministic.

SW-Att is simple, as depicted in Figure 7. It first derives a new unique context-specific key (*key*) from the master key ($\mathcal{K}$) by computing an HMAC-based key derivation function, HKDF [32], on $\mathcal{C}$hal. This key derivation can be extended to incorporate attested memory boundaries if $\mathcal{V}$rf specifies the range (see Appendix B). Finally, it calls HACL*'s HMAC, using *key* as the HMAC key. *ATTEST_DATA_ADDR* and *ATTEST_SIZE* specify the memory range to be attested (*AR* in our notation). We emphasize that SW-Att resides in ROM, which guarantees **P5** under the assumption of no hardware attacks. Moreover, as discussed below, HW-Mod enforces that no other software running on $\mathcal{P}$rv can access memory allocated by SW-Att code, e.g., *key*[64] buffer allocated in line 2 of Figure 7.

HACL*'s verified HMAC is the core for guaranteeing **P4** (Functional Correctness) in *VRASED*'s design. SW-Att functional correctness means that, as long as the memory regions storing values used in SW-Att computation (*CR*, *AR*, and *KR*) do not change during its computation, the result of such computation is the correct HMAC. This guarantee can be formally expressed in LTL as in Definition 3. We note that since HACL*'s HMAC functional correctness is specified in F*, instead of LTL, we manually convert its guarantees to the LTL expressed by Definition 3. By this definition, the value in *MR* does not need to remain the same, as it will eventually be overwritten by the result of SW-Att computation.

Definition 3. *SW-Att functional correctness*

$$\boldsymbol{G}: \{ PC = CR_{min} \wedge MR = \mathcal{C}\text{hal} \wedge [(\neg reset \wedge \neg irq \wedge CR = \text{SW-Att} \wedge KR = \mathcal{K} \wedge AR = M) \; \boldsymbol{U} \; PC = CR_{max}]$$
$$\rightarrow \boldsymbol{F}: [PC = CR_{max} \wedge MR = HMAC(KDF(\mathcal{K}, \mathcal{C}\text{hal}), M)] \}$$

where M is any arbitrary value for AR.

In addition, some HACL* properties, such as stack-based and deterministic memory allocation, are used in alternative designs of *VRASED* to ensure **P2** – see Section 5.

Functional correctness implies that the HMAC implementation conforms to its published standard specification on all possible inputs, retaining the specification's cryptographic security. It also implies that HMAC executes in finite time. Secret

independence ensures that there are no branches taken as a function of secrets, i.e., $\mathcal{K}$ and *key* in Figure 7. This mitigates $\mathcal{K}$ leakage via timing side-channel attacks. Memory safety guarantees that implemented code is type safe, meaning that it never reads from, or writes to: invalid memory locations, out-of-bounds memory, or unallocated memory. This is particularly important for preventing ROP attacks, as long as **P7** (controlled invocation) is also preserved[5].

Having all memory allocated on the stack allows us to either: (1) confine SW-Att execution to a fixed size protected memory region inaccessible to regular software (including malware) running on $\mathcal{P}$rv; or (2) ensure that SW-Att stack is erased before the end of execution. Note that HACL* does not provide stack erasure, in order to improve performance. Therefore, **P2** does not follow from HACL* implementation. This practice is common because inter-process memory isolation is usually provided by the Operating System (OS). However, erasure before SW-Att terminates must be guaranteed. Recall that *VRASED* targets low-end MCUs that might run applications on bare-metal and thus can not rely on any OS features.

As discussed above, even though HACL* implementation guarantees **P4** and storage in ROM guarantees **P5**, these must be combined with **P6** and **P7** to provide safe execution. **P6** and **P7** – along with the key protection properties (**P1**, **P2**, and **P3**) – are ensured by HW-Mod and are described next.

4.4 Key Access Control (HW-Mod)

If malware manages to read $\mathcal{K}$ from ROM, it can reply to $\mathcal{V}$rf with a forged result. HW-Mod access control (AC) sub-module enforces that $\mathcal{K}$ can only be accessed by SW-Att (**P1**).

4.4.1 LTL Specification

The invariant for key access control (AC) is defined in LTL Specification (2). It stipulates that system must transition to the *Reset* state whenever code from outside *CR* tries to read from D_{addr} within the key space.

$$\mathbf{G}: \{\neg(PC \in CR) \wedge R_{en} \wedge (D_{addr} \in KR) \rightarrow reset\} \quad (2)$$

4.4.2 Verified Model

Figure 8 shows the FSM implemented by the AC sub-module which is verified to hold for LTL Specification 2. This FSM has two states: *Run* and *Reset*. It outputs $reset = 1$ when the AC sub-module transitions to state *Reset*. This implies a hard-reset of the MCU. Once the reset process completes, the system leaves the *Reset* state.

[5]Otherwise, even though the implementation is memory-safe and correct as a whole, chunks of a memory-safe code could still be used in ROP attacks.

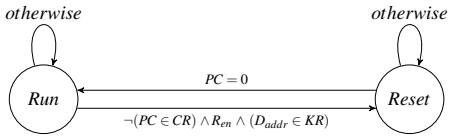


Figure 8: Verified FSM for Key AC

4.5 Atomicity and Controlled Invocation (HW-Mod)

In addition to functional correctness, safe execution of attestation code requires immutability (**P5**), atomicity (**P6**), and controlled invocation (**P7**). **P5** is achieved directly by placing SW-Att in ROM. Therefore, we only need to formalize invariants for the other two properties: atomicity and controlled execution.

4.5.1 LTL Specification

To guarantee atomic execution and controlled invocation, LTL Specifications (3), (4) and (5) must hold:

$$\mathbf{G}: \{[\neg reset \wedge (PC \in CR) \wedge \neg(\mathbf{X}(PC) \in CR)] \rightarrow [PC = CR_{max} \vee \mathbf{X}(reset)]\} \quad (3)$$

$$\mathbf{G}: \{[\neg reset \wedge \neg(PC \in CR) \wedge (\mathbf{X}(PC) \in CR)] \rightarrow [\mathbf{X}(PC) = CR_{min} \vee \mathbf{X}(reset)]\} \quad (4)$$

$$\mathbf{G}: \{irq \wedge (PC \in CR) \rightarrow reset\} \quad (5)$$

LTL Specification (3) enforces that the only way for SW-Att execution to terminate is through its last instruction: $PC = CR_{max}$. This is specified by checking current and next *PC* values using LTL ne**X**t operator. In particular, if current *PC* value is within SW-Att region, and next *PC* value is out of SW-Att region, then either current *PC* value is the address of the last instruction in SW-Att (CR_{max}), or *reset* is triggered in the next cycle. Also, LTL Specification (4) enforces that the only way for *PC* to enter SW-Att region is through the very first instruction: CR_{min}. Together, these two invariants imply **P7**: it is impossible to jump into the middle of SW-Att, or to leave SW-Att before reaching the last instruction.

P6 is satisfied through LTL Specification (5). Atomicity could be violated by interrupts. However, LTL Specification (5) prevents an interrupt to happen while SW-Att is executing. Therefore, if interrupts are not disabled by software running on $\mathcal{P}$rv before calling SW-Att, any interrupt that could violate SW-Att atomicity will necessarily cause an MCU *reset*.

4.5.2 Verified Model

Figure 9 presents a verified model for atomicity and controlled invocation enforcement. The FSM has five states. Two basic states *notCR* and *midCR* represent moments when *PC* points to an address: (1) outside *CR*, and (2) within *CR*, respectively, not including the first and last instructions of SW-Att. Another

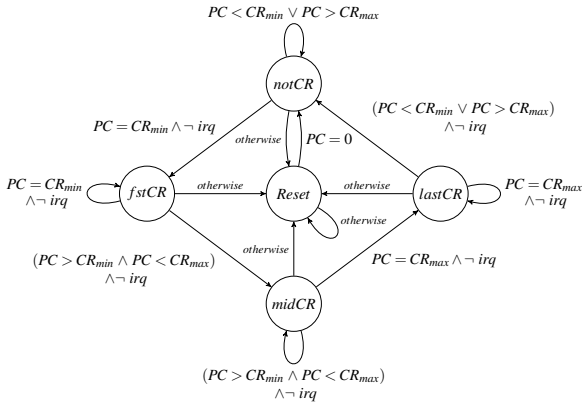


Figure 9: Verified FSM for atomicity and controlled invocation.

two: *fstCR* and *lstCR* represent states when *PC* points to the first and last instructions of SW-Att, respectively. Note that the only possible path from *notCR* to *midCR* is through *fstCR*. Similarly, the only path from *midCR* to *notCR* is through *lstCR*. The FSM transitions to the *Reset* state whenever: (1) any sequence of values for *PC* does not obey the aforementioned conditions; or (2) *irq* is logical 1 while executing SW-Att.

4.6 Key Confidentiality (HW-Mod)

To guarantee secrecy of $\mathcal{K}$ and thus satisfy **P2**, *VRASED* must enforce the following:

1. No leaks after attestation: any registers and memory accessible to applications must be erased at the end of each attestation instance, i.e., before application execution resumes.
2. No leaks on reset: since a reset can be triggered during attestation execution, any registers and memory accessible to regular applications must be erased upon reset.

Per Axiom **A4**, all registers are zeroed out upon reset and at boot time. Therefore, the only time when register clean-up is necessary is at the end of SW-Att. Such clean-up is guaranteed by the Callee-Saves-Register convention: Axiom **A6**.

Nonetheless, the leakage problem remains because of RAM allocated by SW-Att. Thus, we must guarantee that $\mathcal{K}$ is not leaked through "dead" memory, which could be accessed by application (possibly, malware) after SW-Att terminates. A simple and effective way of addressing this issue is by reserving a separate secure stack in RAM that is only accessible (i.e., readable and writable) by attestation code. All memory allocations by SW-Att must be done on this stack, and access control to the stack must be enforced by HW-Mod. As discussed in Section 6, the size of this stack is constant – 2.3KBytes. This corresponds to $\approx 3\%$ of MSP430 16-bit address space. We discuss *VRASED* variants that do not require a reserved stack and trade-offs between them in Section 5.

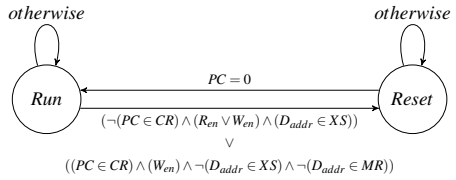


Figure 10: Verified FSM for Key Confidentiality

4.6.1 LTL Specification

Recall that XS denote a contiguous secure memory region reserved for exclusive access by SW-Att. LTL Specification for the secure stack sub-module is as follows:

$$\mathbf{G}: \{\neg(PC \in CR) \wedge (R_{en} \vee W_{en}) \wedge (D_{addr} \in XS) \rightarrow reset\} \quad (6)$$

We also want to prevent attestation code from writing into application memory. Therefore, it is only allowed to write to the designated fixed region for the HMAC result (MR).

$$\mathbf{G}: \{(PC \in CR) \wedge (W_{en}) \wedge \neg(D_{addr} \in XS) \wedge \neg(D_{addr} \in MR) \rightarrow reset\} \quad (7)$$

In summary, invariants (6) and (7) enforce that only attestation code can read from/write to the secure reserved stack and that attestation code can only write to regular memory within the space reserved for the HMAC result. If any of these conditions is violated, the system resets.

4.6.2 Verified Model

Figure 10 shows the FSM verified to comply with invariants (6) and (7).

4.7 DMA Support

So far, we presented a formalization of HW-Mod sub-modules under the assumption that DMA is either not present or disabled on $\mathcal{P}$rv. However, when present, a DMA controller can access arbitrary memory regions. Such memory access is performed concurrently in the memory backbone and without MCU intervention, while the MCU executes regular instructions.

DMA data transfer is performed using dedicated memory buses, e.g., DMA_{en} and DMA_{addr}. Hence, regular memory access control (based on monitoring D_{addr}) does not apply to memory access by DMA controller. Thus, if DMA controller is compromised, it may lead to violation of **P1** and **P2** by directly reading $\mathcal{K}$ and values in the attestation stack, respectively. In addition, it can assist $\mathcal{P}$rv-resident malware to escape detection by either copying it out of the measurement range or deleting it, which results in a violation of **P6**.

4.7.1 LTL Specification

We introduce three additional LTL Specifications to protect against aforementioned attacks. First, we enforce that DMA

cannot access $\mathcal{K}$.

$$\mathbf{G}: \{DMA_{en} \wedge (DMA_{addr} \in KR) \rightarrow reset\} \quad (8)$$

Similarly, LTL Specification for preventing DMA access to the attestation stack is defined as:

$$\mathbf{G}: \{DMA_{en} \wedge (DMA_{addr} \in XS) \rightarrow reset\} \quad (9)$$

Finally, invariant (10) specifies that DMA must be always disabled while *PC* is in `SW-Att` region. This prevents DMA controller from helping malware escape during attestation.

$$\mathbf{G}: \{(PC \in CR) \wedge DMA_{en} \rightarrow reset\} \quad (10)$$

4.7.2 Verified Model

Figure 11 shows the FSM verified to comply with invariants (8) to (10).

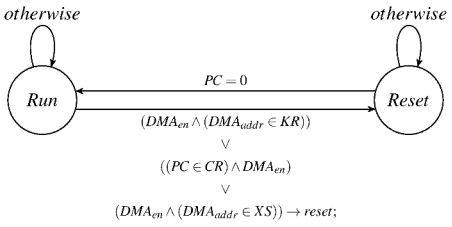


Figure 11: Verified FSM for DMA protection

4.8 `HW-Mod` Composition

Thus far, we designed and verified individual `HW-Mod` sub-modules according to the methodology in Section 3.4 and illustrated in Figure 4. We now follow the workflow of Figure 5 to combine the sub-modules into a single Verilog module. Since each sub-module individually guarantees a subset of properties **P1–P7**, the composition is simple: the system must reset whenever any sub-module reset is triggered. This is implemented by a logical OR of sub-modules reset signals. The composition is shown in Figure 12.

To verify that all LTL specifications still hold for the composition, we use Verilog2SMV [28] to translate `HW-Mod` to SMV and verify SMV for all of these specifications simultaneously.

4.9 Secure Reset (`HW-Mod`)

Finally, we define an LTL Specification for secure reset (**P3**). According to Axiom **A4**, all registers (including *PC*) are set to 0 on reset. However, the reset routine implemented by the MCU might take several clock cycles. Ensuring that $reset = 1$ until the point when registers are wiped is important in order to guarantee that $\mathcal{K}$ is not leaked through registers after a reset. That is because some part of $\mathcal{K}$ might remain in some of the registers if a reset happens during `SW-Att` execution.

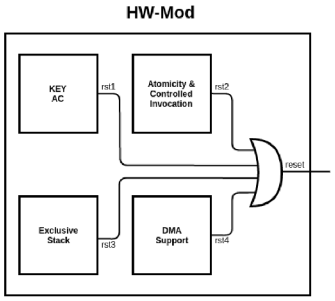


Figure 12: `HW-Mod` composition from sub-modules

4.9.1 LTL Specification

To guarantee that the reset signal is active for long enough so that the MCU reset finishes and all registers are cleaned-up, it must hold that:

$$\mathbf{G}: \{reset \rightarrow [(reset \quad \mathbf{U} \quad PC = 0) \quad \vee \quad \mathbf{G}(reset)]\} \quad (11)$$

Invariant (11) states: when reset signal is triggered, it can only be released after $PC = 0$. Transition from *Reset* state in all sub-modules presented in this section already takes this invariant into account. Thus, `HW-Mod` composition also verifies LTL Specification (11).

5 Alternative Designs

We now discuss alternative designs for *VRASED* that guarantee verified properties without requiring a separate secure stack region for `SW-Att` operations. Recall that `HW-Mod` enforces that only `SW-Att` can access this stack. Since memory usage in HACL* HMAC is deterministic, the size of the separate stack can be pre-determined – 2,332bytes. Even though resulting in overall (HW and SW) design simplicity, dedicating 3% of addressable memory to secure `RA` might not be desirable. Therefore, we consider several alternatives. In Section 6 the costs involved with these alternatives are quantified and compared to the standard design of *VRASED*.

5.1 Erasure on `SW-Att`

The most intuitive alternative to a reserved secure stack (which prevents accidental key leakage by `SW-Att`) is to encode corresponding properties into the HACL* implementation and proof. Specifically, it would require extending the HACL* implementation to zero out all allocated memory before every function return. In addition, to retain verification of **P2** (in Section 3.2) and ensure no leakage, HACL*-verified properties must be extended to incorporate memory erasure. This is not yet supported in HACL* and doing so would incur a slight performance overhead. However, the trade-off between performance and `RAM` savings might be worthwhile.

At the same time, we note that, even with verified erasure as a part of SW-Att, **P2** is still not guaranteed if the MCU does not guarantee erasure of the entire RAM upon boot. This is necessary in order to consider the case when $\mathcal{P}$rv re-boots in the middle of SW-Att execution. Without a reserved stack, $\mathcal{K}$ might persist in RAM. Since the memory range for SW-Att execution is not fixed, hardware support is required to bootstrap secure RAM erasure before starting any software execution. In fact, such support is necessary for all approaches without a separate secure stack.

5.2 Compiler-Based Clean-Up

While stack erasure in HACL* would integrate nicely with the overall proof of SW-Att, the assurance would be at the language abstraction level, and not necessarily at the machine level. The latter would require additional assumptions about the compilation tool chain. We could also consider performing stack erasure directly in the compiler. In fact, a recent proposal to do exactly that was made in zerostack [45], an extension to Clang/LLVM. In case of *VRASED*, this feature could be used on unmodified HACL* (at compilation time), to add instructions to erase the stack before the return of each function enabling **P2**, assuming the existence of a verified RAM erasure routine upon boot. We emphasize that this approach may increase the compiler's trusted code base. Ideally, it should be implemented and formally verified as part of a verified compiler suite, such as CompCert [33].

5.3 Double-HMAC Call

Finally, complete stack erasure could also be achieved directly using currently verified HACL* properties, without any further modifications. This approach involves invoking HACL* HMAC function a second time, after the computation of the actual HMAC. The second "dummy" call would use the same input data, however, instead of using $\mathcal{K}$, an independent constant, such as $\{0\}^{512}$, would be used as the HMAC key.

Recall that HACL* is verified to only allocate memory on the stack in a deterministic manner. Also, due to HACL*'s verified properties that mitigate side-channels, software flow does not change based on the secret key. Therefore, this deterministic allocation implies that, for inputs of the same size, any variable allocated by the first "real" HMAC call (tainted by $\mathcal{K}$), would be overwritten by the corresponding variable in the second "dummy" call. Note that the same guarantee discussed in Section 5.1 is provided here and secure RAM erasure at boot would still be needed for the same reasons. Admittedly, this double-HMAC approach would consume twice as many CPU cycles. Still, it might be a worthwhile trade-off, especially, if there is memory shortage and lack of previously discussed HACL* or compiler extension.

6 Evaluation

We now discuss implementation details and evaluate *VRASED*'s overhead and performance. Section 6.2 reports on verification complexity. Section 6.3 discusses performance in terms of time and space complexity as well as its hardware overhead. We also provide a comparison between *VRASED* and other RA architectures targeting low-end devices, namely SANCUS [38] and SMART [21], in Section 6.4.

6.1 Implementation

As mentioned earlier, we use OpenMSP430 [22] as an open core implementation of the MSP430 architecture. OpenMSP430 is written in the Verilog hardware description language (HDL) and can execute software generated by any MSP430 toolchain with near cycle accuracy. We modified the standard OpenMSP430 to implement the hardware architecture presented in Section 3.3, as shown in Figure 3. This includes adding ROM to store $\mathcal{K}$ and SW-Att, adding HW-Mod, and adapting the memory backbone accordingly. We use Xilinx Vivado [50] – a popular logic synthesis tool – to synthesize an RTL description of HW-Mod into hardware in FPGA. FPGA synthesized hardware consists of a number of logic cells. Each consists of Look-Up Tables (LUTs) and registers; LUTs are used to implement combinatorial boolean logic while registers are used for sequential logic elements, i.e., FSM states and data storage. We compiled SW-Att using the native msp430-gcc [46] and used Linker scripts to generate software images compatible with the memory layout of Figure 3. Finally, we evaluated *VRASED* on the FPGA platform targeting Artix-7 [51] class of devices.

6.2 Verification Results

As discussed in Section 3.2, *VRASED*'s verification consists of properties **P1**–**P7**. **P5** is achieved directly by executing SW-Att from ROM. Meanwhile, HACL* HMAC verification implies **P4**. All other properties are automatically verified using NuSMV model checker. Table 2 shows the verification results of *VRASED*'s HW-Mod composition as well as results for individual sub-modules. It shows that *VRASED* successfully achieves all the required security properties. These results also demonstrate feasibility of our verification approach, since the verification process – running on a commodity desktop computer – consumes only small amount of memory and time: $<$ 14MB and 0.3sec, respectively, for all properties.

Table 3: Evaluation of cost, overhead, and performance of RA

Method	RAM Erasure Required Upon Boot?	FPGA Hardware LUT	Reg	Cell	Verilog LoC	Memory (byte) ROM	Sec. RAM	Time to attest 4KB CPU cycles	ms (at 8MHz)
Core (Baseline)	N/A	1842	684	3044	4034	0	0	N/A	N/A
Secure Stack (Section 4)	No	1964	721	3237	4621	4500	2332	3601216	450.15
Erasure on SW-Att (Section 5.1)	Yes	1954	717	3220	4516	4522	0	3613283	451.66
Compiler-based Clean-up (Section 5.2) [6]	Yes	1954	717	3220	4516	4522	0	3613283	451.66
Double-HMAC Call (Section 5.3)	Yes	1954	717	3220	4516	4570	0	7201605	900.20

Table 2: Verification results running on a desktop @ 3.40 GHz.

HW Submod.	LTL Spec.	Mem. (MB)	Time (s)	Verified
Key AC	2,11	7.5	.02	✓
Atomicity	3,4,5,11	8.5	.05	✓
Exclusive Stack	6,7,11	8.1	.03	✓
DMA Support	8-11	8.2	.04	✓
HW-Mod	2-11	13.6	.28	✓

Table 4: Qualitative comparison between RA architectures targeting low-end devices

	VRASED	SMART	SANCUS
Design Type	Hybrid (HW/SW)	Hybrid (HW/SW)	Pure HW
RA function	HMAC-SHA256	HMAC-SHA1	SPONGENT-128/128/8
ROM for RA code	Yes	Yes	No
DMA Support	Yes	No	No
Formally Verified	Yes	No	No

6.3 Performance and Hardware Cost

We now report on *VRASED*'s performance considering the standard design (described in Section 4) and alternatives discussed in Section 5. We evaluate the hardware footprint, memory (ROM and secure RAM), and run-time. Table 3 summarizes the results.

Hardware Footprint. The secure stack approach adds around 587 lines of code in Verilog HDL. This corresponds to around 15% of the code in the original OpenMSP430 core. In terms of synthesized hardware, it requires 122 (6.6%) and 37 (5.4%) additional LUTs and registers respectively. Overall, *VRASED* contains 193 logic cells more than the unmodified OpenMSP430 core, corresponding to a 6.3% increase.

Memory. *VRASED* requires ∼4.5KB of ROM; most of which (96%) is for storing HACL* HMAC-SHA256 code. The secure stack approach has the smallest ROM size, as it does not need to perform a memory clean-up in software. However, this advantage is attained at the price of requiring 2.3KBytes of reserved RAM. This overhead corresponds to 3.5% of MSP430 16-bit address space.

Attestation Run-time. Attestation run-time is dominated by the time it takes to compute the HMAC of $\mathcal{P}$rv's memory. The secure stack, erasure on SW-Att and compiler-based clean-up approaches take roughly .45s to attest 4KB of RAM on an MSP430 device with a clock frequency at 8MHz. Whereas, the double MAC approach requires invoking the HMAC function twice, leading its run-time to be roughly two times slower.

Discussion. We consider *VRASED*'s overhead to be affordable. The additional hardware, including registers, logic gates and exclusive memory, resulted in only a 3-6% increase. The number of cycles required by SW-Att exhibits a linear increase with the size of attested memory. As MSP430 typically runs at 8-25MHz, attestation of the entire RAM on a typical MSP430 can be computed in less than a second. *VRASED*'s RA is relatively cheap to the $\mathcal{P}$rv. As a point of comparison we can consider a common cryptographic primitive such as the Curve25519 Elliptic-Curve Diffie-Hellman (ECDH) key exchange. A single execution of an optimized version of such protocol on MSP430 has been reported to take ≈ 9 million cycles [24]. As Table 3 shows, attestation of 4KBytes (typical size of RAM in some MSP430 models) can be computed three times faster.

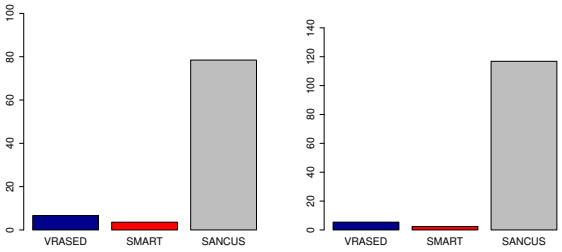
6.4 Comparison with Other Low-End RA Architectures

We here compare *VRASED*'s overhead with two widely known RA architectures targeting low-end embedded systems: SMART [21] and SANCUS [38]. We emphasize, however, that both SMART and SANCUS were designed in an ad hoc manner. Thus, they can not be formally verified and do not provide any guarantees offered by *VRASED*'s verified architecture. Nevertheless, it is considered important to contrast *VRASED*'s cost with such architectures to demonstrate its affordability.

Table 4 presents a comparison between features offered and required by aforementioned architectures. SANCUS is, to the best of our knowledge, the cheapest pure HW-based architecture, while SMART is a minimal HW/SW RA co-design. Since SANCUS's RA routine is implemented entirely in HW, it does not require ROM to store the SW implementation of the integrity ensuring function. *VRASED* implements a MAC with digest sizes of 256-bits. SMART and SANCUS, on the other hand, use SHA1-based MAC and SPONGENT-128/128/8 [7], respectively. Such MACs do not offer strong collision resistance due to the small digest sizes (and known collisions). Of the three architectures, *VRASED* is the only one secure in the presence of DMA and the only one to be rigorously specified and formally verified.

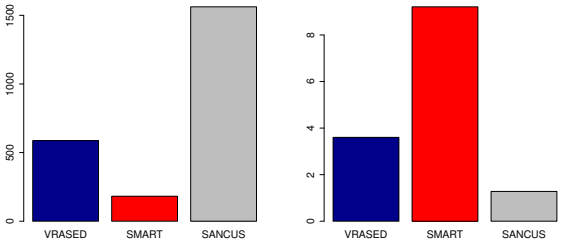
Figure 13 presents a quantitative comparison between the RA architectures. It considers additional overhead in relation to the latest version of the unmodified OpenMSP430 (Available

[6] As mentioned in Section 5.2, there is no formally verified msp430 compiler capable of performing stack erasure. Thus, we estimate overhead of this approach by manually inserting code required for erasing the stack in SW-Att.



(a) Additional HW overhead (%) in Number of Look-Up Tables

(b) Additional HW overhead (%) in Number of Registers



(c) Additional Verilog Lines of Code

(d) Time to attest 4KB (in millions of CPU cycles)

Figure 13: Comparison between RA architectures targeting low-end devices

at [22]). Compared to *VRASED*, SANCUS requires 12× more Look-Up Tables, 22× more registers, and its (unverified) TCB is 2.5 times larger in lines of Verilog code. This comparison demonstrates the cost of relying on a HW-only approach even when designed for minimality. SMART's overhead is slightly smaller than that of *VRASED* due to lack of DMA support. In terms of attestation execution time, SMART is the slowest, requiring 9.2M clock cycles to attest 4KB of memory. SANCUS achieves the fastest attestation time (1.3M cycles) due to the HW implementation of SPONGENT-128/128/8. *VRASED* sits in between the two with a total attestation time of 3.6M cycles.

7 Related Work

We are unaware of any previous work that yielded a formally verified RA design (RA architectures are overviewed in Section 2.1). To the best of our knowledge, *VRASED* is the first verification of a security service implemented as HW/SW co-design. Nevertheless, formal verification has been widely used as the *de facto* means to guarantee that a system is free of implementation errors and bugs. In recent years, several efforts focused on verifying security-critical systems.

In terms of cryptographic primitives, Hawblitzel et al. [23] verified new implementations of SHA, HMAC, and RSA. Beringer et al. [4] verified the Open-SSL SHA-256 implementation. Bond et al. [8] verified an assembly implementation of SHA-256, Poly1305, AES and ECDSA. More recently, Zinzindohoué, et al. [52] developed HACL*, a verified cryptographic library containing the entire cryptographic API of NaCl [5]. As discussed earlier, HACL*'s verified HMAC forms the core of *VRASED*'s software component.

Larger security-critical systems have also been successfully verified. For example, Bhargavan [6] implemented the TLS protocol with verified cryptographic security. CompCert [33] is a C compiler that is formally verified to preserve C code semantics in generated assembly code. Klein et al. [29] designed and proved functional correctness of `seL4` – the first verified general-purpose microkernel. More recently, Tuncay et al. verified a design for Android OS App permissions model [48].

The importance of verifying RA has been recently acknowledged by Lugou et al. [36], which discussed methodologies for specifically verifying HW/SW RA co-designs. A follow-on result proposed the SMASH-UP tool [37]. By modeling a hardware abstraction, SMASH-UP allows automatic conversion of assembly instructions to the effects on hardware representation. Similarly, Cabodi et al. [11, 12] discussed the first steps towards formalizing hybrid RA properties. However, none of these results yielded a fully verified (and publicly available) RA architecture, such as *VRASED*.

8 Conclusion

This paper presents *VRASED* – the first formally verified RA method that uses a verified cryptographic software implementation and combines it with a verified hardware design to guarantee correct implementation of RA security properties. *VRASED* is also the first verified security service implemented as a HW/SW co-design. *VRASED* was designed with simplicity and minimality in mind. It results in efficient computation and low hardware cost, realistic even for low-end embedded systems. *VRASED*'s practicality is demonstrated via publicly available implementation using the low-end MSP430 platform. The design and verification methodology presented in this paper can be extended to other MCU architectures. We believe that this work represents an important and timely advance in embedded systems security, especially, with the rise of heterogeneous ecosystems of (inter-)connected IoT devices.

The most natural direction for future work is to adapt *VRASED* to other MCU architectures. Such an effort could follow the same verification methodology presented in this paper. It would involve: (1) mapping MCUs specifications to a set of axioms (as we did for MSP430 in Section 3), and (2) adapting the proofs by modifying the LTL Specifications and hardware design (as in Section 4) accordingly. A second direction is to extend *VRASED*'s capabilities to include and verify other trusted computing services such as secure updates, secure deletion, and remote code execution. It would also be interesting to verify and implement other RA designs with different requirements and trade-offs, such as software- and hardware-based techniques. In the same vein, one promising

direction would be to verify HYDRA RA architecture [20], which builds on top of the formally verified seL4 [29] microkernel. Finally, the optimization of *VRASED*'s HMAC, with respect to computation and memory allocation, while retaining its verified properties, is an interesting open problem.

Acknowledgments: UC Irvine authors' work was supported in part by DHS, under subcontract from HRL Laboratories, and ARO under contract: W911NF-16-1-0536, as well as NSF WiFiUS Program Award #: 1702911. The authors thank the paper's shepherd, Stephen McCamant, and the anonymous reviewers for their valuable comments.

References

[1] VRASED source code. https://github.com/sprout-uci/vrased, 2019.

[2] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, et al. Understanding the mirai botnet. In *USENIX Security*, 2017.

[3] Arm Ltd. Arm TrustZone. https://www.arm.com/products/security-on-arm/trustzone, 2018.

[4] L. Beringer, A. Petcher, Q. Y. Katherine, and A. W. Appel. Verified correctness and security of OpenSSL HMAC. In *USENIX Security*, 2015.

[5] D. J. Bernstein, T. Lange, and P. Schwabe. The security impact of a new cryptographic library. In *International Conference on Cryptology and Information Security in Latin America*, 2012.

[6] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub. Implementing TLS with verified cryptographic security. In *IEEE S&P*, 2013.

[7] A. Bogdanov, M. Knezevic, G. Leander, D. Toz, K. Varici, and I. Verbauwhede. Spongent: The design space of lightweight cryptographic hashing. *IEEE Transactions on Computers*, 62, 2013.

[8] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson. Vale: Verifying high-performance cryptographic assembly code. In *USENIX Security*, 2017.

[9] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl. TyTAN: tiny trust anchor for tiny devices. In *DAC*, 2015.

[10] F. Brasser, A.-R. Sadeghi, and G. Tsudik. Remote attestation for low-end embedded devices: the prover's perspective. In *DAC*, 2016.

[11] G. Cabodi, P. Camurati, S. F. Finocchiaro, C. Loiacono, F. Savarese, and D. Vendraminetto. Secure embedded architectures: Taint properties verification. In *DAS*, 2016.

[12] G. Cabodi, P. Camurati, C. Loiacono, G. Pipitone, F. Savarese, and D. Vendraminetto. Formal verification of embedded systems for remote attestation. *WSEAS Transactions on Computers*, 14, 2015.

[13] X. Carpent, K. Eldefrawy, N. Rattanavipanon, A.-R. Sadeghi, and G. Tsudik. Reconciling remote attestation and safety-critical operation on simple iot devices. In *DAC*, 2018.

[14] X. Carpent, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik. Temporal consistency of integrity-ensuring computations and applications to embedded systems security. In *ASIACCS*, 2018.

[15] X. Carpent, N. Rattanavipanon, and G. Tsudik. ERASMUS: Efficient remote attestation via self-measurement for unattended settings. In *DATE*, 2018.

[16] X. Carpent, N. Rattanavipanon, and G. Tsudik. Remote attestation of iot devices via SMARM: Shuffled measurements against roving malware. In *IEEE HOST*, 2018.

[17] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *CAV*, 2002.

[18] I. De Oliveira Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik. Formally verified hardware/software co-design for remote attestation. *arXiv preprint arXiv:1811.00175*, 2018.

[19] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu. Spot 2.0—a framework for ltl and ω-automata manipulation. In *ATVA*, 2016.

[20] K. Eldefrawy, N. Rattanavipanon, and G. Tsudik. HYDRA: hybrid design for remote attestation (using a formally verified microkernel). In *WiSec*, 2017.

[21] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito. SMART: Secure and minimal architecture for (establishing dynamic) root of trust. In *NDSS*, 2012.

[22] O. Girard. openMSP430, 2009.

[23] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: End-to-end security via automated full-system verification. In *USENIX OSDI*, 2014.

[24] G. Hinterwälder, A. Moradi, M. Hutter, P. Schwabe, and C. Paar. Full-size high-security ECC implementation on MSP430 microcontrollers. In *International Conference on Cryptology and Information Security in Latin America*, pages 31–47. Springer, 2014.

[25] A. Ibrahim, A.-R. Sadeghi, and S. Zeitouni. SeED: secure non-interactive attestation for embedded devices. In *ACM WiSec*, 2017.

[26] T. Instruments. Msp430 ultra-low-power sensing & measurement mcus. http://www.ti.com/microcontrollers/msp430-ultra-low-power-mcus/overview.html.

[27] Intel. Intel Software Guard Extensions (Intel SGX). https://software.intel.com/en-us/sgx.

[28] A. Irfan, A. Cimatti, A. Griggio, M. Roveri, and R. Sebastiani. Verilog2SMV: A tool for word-level verification. In *DATE*, 2016.

[29] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *SOSP*, 2009.

[30] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. TrustLite: A security architecture for tiny embedded devices. In *EuroSys*, 2014.

[31] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth. New results for timing-based attestation. In *IEEE S&P*, 2012.

[32] H. Krawczyk and P. Eronen. HMAC-based extract-and-expand key derivation function (HKDF). Internet Request for Comment RFC 5869, Internet Engineering Task Force, May 2010.

[33] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[34] Y. Li, Y. Cheng, V. Gligor, and A. Perrig. Establishing software-only root of trust on embedded systems: Facts and fiction. In *Security Protocols—22nd International Workshop*, 2015.

[35] Y. Li, J. M. McCune, and A. Perrig. VIPER: verifying the integrity of peripherals' firmware. In *CCS*, 2011.

[36] F. Lugou, L. Apvrille, and A. Francillon. Toward a methodology for unified verification of hardware/software co-designs. *Journal of Cryptographic Engineering*, 2016.

[37] F. Lugou, L. Apvrille, and A. Francillon. Smashup: a toolchain for unified verification of hardware/software co-designs. *Journal of Cryptographic Engineering*, 7(1):63–74, 2017.

[38] J. Noorman, J. V. Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling. Sancus 2.0: A low-cost security architecture for iot devices. *ACM Trans. Priv. Secur.*, 20(3):7:1–7:33, July 2017.

[39] I. D. O. Nunes, G. Dessouky, A. Ibrahim, N. Rattanavipanon, A.-R. Sadeghi, and G. Tsudik. Towards systematic design of collective remote attestation protocols. In *ICDCS*, 2019.

[40] D. Perito and G. Tsudik. Secure code update for embedded devices via proofs of secure erasure. In *ESORICS*, 2010.

[41] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hriţcu, K. Bhargavan, C. Fournet, et al. Verified low-level programming embedded in F*. *Proceedings of the ACM on Programming Languages*, 1, 2017.

[42] S. Ravi, A. Raghunathan, and S. Chakradhar. Tamper resistance mechanisms for secure embedded systems. In *VLSI Design*, 2004.

[43] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla. Scuba: Secure code update by attestation in sensor networks. In *ACM workshop on Wireless security*, 2006.

[44] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. *ACM SIGOPS Operating Systems Review*, December 2005.

[45] L. Simon, D. Chisnall, and R. Anderson. What you get is what you c: Controlling side effects in mainstream c compilers. In *IEEE EuroS&P*, 2018.

[46] Texas Instruments. MSP430 GCC user's guide, 2016.

[47] Trusted Computing Group. Trusted platform module (tpm), 2017.

[48] G. S. Tuncay, S. Demetriou, K. Ganju, and C. A. Gunter. Resolving the predicament of Android custom permissions. In *NDSS*, 2018.

[49] J. Vijayan. Stuxnet renews power grid security concerns. `http://www.computerworld.com/article/2519574/security0/stuxnet-renews-power-grid-security-concerns.html`, june 2010.

[50] Xilinx. Vivado design suite user guide, 2017.

[51] Xilinx Inc. Artix-7 FPGA family. `https://www.xilinx.com/products/silicon-devices/fpga/artix-7.html`, 2018.

[52] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL*: A verified modern cryptographic library. In *CCS*, 2017.

APPENDIX

A RA Soundness and Security Proofs

A.1 Proof Strategy

We present the proofs for RA soundness (Definition 1) and RA security (Definition 2). Soundness is proved entirely via LTL equivalences. In the proof of security we first show, via LTL equivalences, that *VRASED* guarantees that adversary $\mathcal{A}$ can never learn $\mathcal{K}$ with more than negligible probability. We then prove security by showing a reduction of HMAC's existential unforgeability to *VRASED*'s security. In other words, we show that existence of $\mathcal{A}$ that breaks *VRASED* implies existence of HMAC-$\mathcal{A}$ able to break conjectured existential unforgeability of HMAC. The full machine-checked proofs for the LTL equivalences (using Spot 2.0 [19] proof assistant) discussed in the remainder of this section are available in [1].

A.2 Machine Model

To prove that *VRASED*'s design satisfies end-to-end definitions of soundness and security for RA, we start by formally defining (in LTL) memory and execution models corresponding to the architecture introduced in Section 3.

Definition 4 (Memory model)**.**

1. *$\mathcal{K}$ is stored in* `ROM` $\leftrightarrow G:\{KR=\mathcal{K}\}$
2. `SW-Att` *is stored in* `ROM` $\leftrightarrow G:\{CR=$ `SW-Att`$\}$
3. *MR, CR, AR, KR, and XS are non-overlapping memory regions*

The memory model in Definition 4 captures that *KR* and *CR* are `ROM` regions, and are thus immutable. Hence, the values stored in those regions always correspond to $\mathcal{K}$ and `SW-Att` code, respectively. Finally, the memory model states that *MR*, *CR*, *AR*, *KR*, and *XS* are disjoint regions in the memory layout, corresponding to the architecture in Figure 3.

Definition 5 (Execution model)**.**

1. $Modify_Mem(i) \rightarrow (W_{en} \wedge D_{addr} = i) \vee (DMA_{en} \wedge DMA_{addr} = i)$
2. $Read_Mem(i) \rightarrow (R_{en} \wedge D_{addr} = i) \vee (DMA_{en} \wedge DMA_{addr} = i)$
3. $Interrupt \rightarrow irq$

Our execution model, in Definition 5, translates MSP430 behavior by capturing the effects on the processor signals when reading and writing from/to memory. We do not model the effects of instructions that only modify register values (e.g., ALU operations, such as *add* and *mul*) because they are not necessary in our proofs.

The execution model defines that a given memory address can be modified in two cases: by a CPU instruction or by DMA. In the first case, the W_{en} signal must be on and D_{addr} must contain the memory address being accessed. In the second case, DMA_{en} signal must be on and DMA_{addr} must contain the address being modified by DMA. The requirements for reading from a given address are similar, except that instead of W_{en}, R_{en} must be on. Finally, the execution model also captures the fact that an interrupt implies setting the *irq* signal to 1.

A.3 RA Soundness Proof

The proof follows from `SW-Att` functional correctness (expressed by Definition 3) and LTL specifications 3, 5, 7, and 10

Theorem 1. *VRASED is sound according to Definition 1.*

Proof.

$$Definition\ 3 \wedge LTL_3 \wedge LTL_5 \wedge LTL_7 \wedge LTL_{10} \rightarrow Theorem\ 1$$

□

The formal computer proof for Theorem 1 can be found in [1]. Due to space limitations, we only provide some intuition, by splitting the proof into two parts. First, SW-Att functional correctness (Definition 3) would imply Theorem 1 if *AR*, *CR*, *KR* never change and an interrupt does not happen during SW-Att computation. However, memory model Definitions 4.1 and 4.2 already guarantee that *CR* and *KR* never change. Also, LTL 5 states that an interrupt cannot happen during SW-Att computation, otherwise the device resets. Therefore, it remains for us to show that *AR* does not change during SW-Att computation. This is stated in Lemma 1.

Lemma 1. *Temporal Consistency – Attested memory does not change during* SW-Att *computation*

$$\boldsymbol{G}: \{ \\ PC = CR_{min} \wedge AR = M \wedge \neg reset \; \boldsymbol{U} \; (PC = CR_{max}) \rightarrow \\ (AR = M) \; \boldsymbol{U} \; (PC = CR_{max}) \; \}$$

In turn, Lemma 1 can be proved by:

$$LTL_3 \wedge LTL_7 \wedge LTL_{10} \rightarrow Lemma\ 1 \tag{12}$$

The reasoning for Equation 12 is as follows:

- LTL_3 prevents the CPU from stopping execution of SW-Att before its last instruction.
- LTL_7 guarantees that the only memory regions written by the CPU during SW-Att execution are XS and MR, which do not overlap with AR.
- LTL_{10} prevents DMA from writing to memory during SW-Att execution.

Therefore, there are no means for modifying AR during SW-Att execution, implying Lemma 1. As discussed above, it is easy to see that:

$$Lemma\ 1 \wedge LTL_5 \wedge Definition\ 3 \rightarrow Theorem\ 1 \tag{13}$$

A.4 RA Security Proof

Recall the definition of RA security in the game in Figure 6. The game makes two key assumptions:

1. SW-Att call results in a temporally consistent HMAC of AR using a key derived from $\mathcal{K}$ and $\mathcal{C}$hal. This is already proved by *VRASED*'s soundness.
2. $\mathcal{A}$ never learns $\mathcal{K}$ with more than negligible probability.

By proving that *VRASED*'s design satisfies assumptions 1 and 2, we show that the capabilities of untrusted software (any DMA or CPU software other than SW-Att) on $\mathcal{P}$rv are equivalent to the capabilities of $\mathcal{A}$ in RA-game. Therefore, we still need to prove item 2 before we can use such game to prove *VRASED*'s security. The proof of $\mathcal{A}$'s inability to learn $\mathcal{K}$ with more than negligible probability is facilitated by *A6 - Callee-Saves-Register* convention stated in Section 3. A6 directly implies no leakage of information through registers on the return of SW-Att. This is because, before the return of a function, registers must be restored to their state prior to the function call. Thus, untrusted software can only learn $\mathcal{K}$ (or any function of $\mathcal{K}$) through memory. However, if untrusted software can never read memory written by SW-Att, it never learns anything about $\mathcal{K}$ (the secret-independence of SW-Att at the HACL* level even implies a lack of timing side-channels, subject to our assumption that this property is preserved by msp430-gcc and the MCU implementation). Now, it suffices to prove that untrusted software can not access $\mathcal{K}$ directly and that it can never read memory written by SW-Att. These conditions are stated in LTL in Lemma 2. We prove that *VRASED* satisfies Lemma 2 by writing a computer proof (available in [1]) for Equation 14. The reasoning for this proof is similar to that of RA soundness and omitted due to space constraints.

Lemma 2. *Key confidentiality – $\mathcal{K}$ can not be accessed directly by untrusted software ($\neg(PC \in CR)$) and any memory written to by* SW-Att *can never be read by untrusted software.*

$$\boldsymbol{G}: \{ \\ (\neg(PC \in CR) \wedge Read_Mem(i) \wedge i \in KR \rightarrow reset) \wedge \\ (DMA_{en} \wedge DMA_{addr} = i \wedge i \in KR \rightarrow reset) \wedge \\ [\neg reset \wedge PC \in CR \wedge Modify_Mem(i) \wedge \neg(i \in MR) \rightarrow \\ \boldsymbol{G}: \{(\neg(PC \in CR) \wedge Read_Mem(i) \vee DMA_{en} \wedge DMA_{addr} = i) \\ \rightarrow reset\}] \\ \}$$

$$LTL_2 \wedge LTL_6 \wedge LTL_7 \wedge LTL_8 \wedge LTL_9 \wedge LTL_{10} \rightarrow \text{Lemma 2} \tag{14}$$

We emphasize that Lemma 2 does not restrict reads and writes to MR, since this memory is used for inputting $\mathcal{C}$hal and receiving SW-Att result. Nonetheless, the already proved RA soundness and LTL 4 (which makes it impossible to execute fractions of SW-Att) guarantee that MR will not leak anything, because at the end of SW-Att computation it will always contain an HMAC result, which does not leak information about $\mathcal{K}$. After proving Lemma 2, the capabilities of untrusted software on $\mathcal{P}$rv are equivalent to those of adversary $\mathcal{A}$ in RA-game of Definition 2. Therefore, in order to prove *VRASED*'s security, it remains to show a reduction from HMAC security according to the game in Definition 2. *VRASED*'s security is stated and proved in Theorem 2.

Theorem 2. *VRASED is secure according to Definition 2 as long as HMAC is a secure* MAC.

Proof. A MAC *is defined as tuple of algorithms* $\{\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrf}\}$. *For the reduction we construct a slightly modified HMAC', which has the same* Mac *and* Vrf *algorithms as standard HMAC but* $\mathsf{Gen} \leftarrow KDF(\mathcal{K}, \mathcal{C}\mathsf{hal})$ *where* $\mathcal{C}\mathsf{hal} \leftarrow \$\{0,1\}^l$. *Since KDF function itself is implemented as a* Mac *call, it is easy to see that the outputs of*

Gen are indistinguishable from random. In other words, the security of this slightly modified construction follows from the security of HMAC itself. Assuming that there exists $\mathcal{A}$ such that $Pr[\mathcal{A}, \mathsf{RA}_{game}] > negl(l)$, we show that such adversary can be used to construct HMAC-$\mathcal{A}$ that breaks existential unforgeability of HMAC' with probability Pr[HMAC-$\mathcal{A}$,MAC-game] $> negl(l)$. To that purpose HMAC-$\mathcal{A}$ behaves as follows:

1. *HMAC-$\mathcal{A}$ selects msg to be the same $M \neq AR$ as in* RA-*game and asks $\mathcal{A}$ to produce the same output used to win* RA-*game.*
2. *HMAC-$\mathcal{A}$ outputs the pair (msg,σ) as a response for the challenge in the standard existential unforgeability game, where σ is the output produced by $\mathcal{A}$ in step 1.*

By construction, (msg,σ) is a valid response to a challenge in the existential unforgeability MAC *game considering HMAC' as defined above. Therefore, HMAC-$\mathcal{A}$ is able to win the existential unforgeability game with the same $> negl(l)$ probability that $\mathcal{A}$ has of winning* RA-*game in Definition 2.* □

B Optional Verifier Authentication

```
1  void Hacl_HMAC_SHA2_256_hmac_entry() {
2      uint8_t key[64] = {0};
3      uint8_t verification[32] = {0};
4      if (memcmp(CHALL_ADDR, CTR_ADDR, 32) > 0)
5      {
6          memcpy(key, KEY_ADDR, 64);
7
8          hacl_hmac((uint8_t*) verification, (uint8_t*) key,
9                    (uint32_t) 64, *((uint8_t*)CHALL_ADDR),
10                   (uint32_t) 32);
11
12         if (!memcmp(VRF_AUTH, verification, 32)
13         {
14             hacl_hmac((uint8_t*) key, (uint8_t*) key,
15                 (uint32_t) 64, (uint8_t*) verification,
16                 (uint32_t) 32);
17             hacl_hmac((uint8_t*) MAC_ADDR, (uint8_t*) key,
18                 (uint32_t) 32, (uint8_t*) ATTEST_DATA_ADDR,
19                 (uint32_t) ATTEST_SIZE);
20             memcpy(CTR_ADDR, CHALL_ADDR, 32);
21         }
22     }
23
24     return();
25 }
```

Figure 14: SW-Att Implementation with $\mathcal{V}$rf authentication

Depending on the setting where $\mathcal{P}$rv is deployed, authenticating the attestation request before executing SW-Att may be required. For example, if $\mathcal{P}$rv is in a public network, the adversary may try to communicate with it. In particular, the adversary can impersonate $\mathcal{V}$rf and send fake attestation requests to $\mathcal{P}$rv, attempting to cause denial-of-service. This is particularly relevant if $\mathcal{P}$rv is a safety-critical device. If $\mathcal{P}$rv receives too many attestation requests, regular (and likely honest) software running on $\mathcal{P}$rv would not execute because SW-Att would run all the time. Thus, we now discuss an optional part of *VRASED*'s design suitable for such settings. It supports authentication of $\mathcal{V}$rf as part of SW-Att execution. Our implementation is based on the protocol in [10].

Figure 14 presents an implementation of SW-Att that includes $\mathcal{V}$rf authentication. It also builds upon HACL* verified HMAC to authenticate $\mathcal{V}$rf, in addition to computing the authenticated integrity check. In this case, $\mathcal{V}$rf's request additionally contains an HMAC of the challenge computed using $\mathcal{K}$. Before calling SW-Att, software running on $\mathcal{P}$rv is expected to store the received challenge on a fixed address *CHALL_ADDR* and the corresponding received HMAC on *VRF_AUTH*. SW-Att discards the attestation request if (1) the received challenge is less than or equal to the latest challenge, or (2) HMAC of the received challenge is mismatched. After that, it derives a new unique key using HKDF [32] from $\mathcal{K}$ and the received HMAC and uses it as the attestation key.

HW-Mod must also be slightly modified to ensure security of $\mathcal{V}$rf's authentication. In particular, regular software must not be able modify the memory region that stores $\mathcal{P}$rv's counter. Notably, the counter requires persistent and writable storage, because SW-Att needs to modify it at the end of each attestation execution. Therefore, *CTR* region resides on FLASH. We denote this region as:

- $CTR = [CTR_{min}, CTR_{max}]$;

LTL Specifications (15) and (16) must hold (in addition to the ones discussed in Section 4).

$$\mathbf{G}: \{\neg(PC \in CR) \wedge W_{en} \wedge (D_{addr} \in CTR) \rightarrow reset\} \quad (15)$$

$$\mathbf{G}: \{DMA_{en} \wedge (DMA_{addr} \in CTR) \rightarrow reset\} \quad (16)$$

LTL Specification (15) ensures that regular software does not modify $\mathcal{P}$rv's counter, while (16) ensures that the same is not possible via the DMA controller. FSMs in Figures 8 and 11, corresponding to HW-Mod access control and DMA sub-modules, must be modified to transition into *Reset* state according to these new conditions. In addition, LTL Specification (7) must be relaxed to allow SW-Att to write to *CTR*. Implementation and verification of the modified version of these sub-modules are publicly available at *VRASED*'s repository [1] as an optional part of the design.

C API & Sample Application

VRASED ensures that any violation of secure RA properties is detected and causes the system to reset. However, benign applications running on the MCU must also comply with *VRASED* rules to execute successfully. To ease the process of setting up the system for a call to SW-Att, *VRASED* provides an API that takes care of necessary configuration on the application's behalf. This API and a sample application deployed using FPGAs are described in the extended version of this paper, available at [18].

Mobile Private Contact Discovery at Scale

Daniel Kales
Graz University of Technology

Christian Rechberger
Graz University of Technology

Thomas Schneider
TU Darmstadt

Matthias Senker
TU Darmstadt

Christian Weinert
TU Darmstadt

Abstract

Mobile messengers like WhatsApp perform contact discovery by uploading the user's entire address book to the service provider. This allows the service provider to determine which of the user's contacts are registered to the messaging service. However, such a procedure poses significant privacy risks and legal challenges. As we find, even messengers with privacy in mind currently do not deploy proper mechanisms to perform contact discovery privately.

The most promising approaches addressing this problem revolve around private set intersection (PSI) protocols. Unfortunately, even in a weak security model where clients are assumed to follow the protocol honestly, previous protocols and implementations turned out to be far from practical when used at scale. This is due to their high computation and/or communication complexity as well as lacking optimization for mobile devices. In our work, we remove most obstacles for large-scale global deployment by significantly improving two promising protocols by Kiss et al. (PoPETS'17) while also allowing for malicious clients.

Concretely, we present novel precomputation techniques for correlated oblivious transfers (reducing the online communication by factor 2x), Cuckoo filter compression (with a compression ratio of $\approx 70\,\%$), as well as 4.3x smaller Cuckoo filter updates. In a protocol performing oblivious PRF evaluations via garbled circuits, we replace AES as the evaluated PRF with a variant of LowMC (Albrecht et al., EUROCRYPT'15) for which we determine optimal parameters, thereby reducing the communication by factor 8.2x. Furthermore, we implement both protocols with security against malicious clients in C/C++ and utilize the ARM Cryptography Extensions available in most recent smartphones. Compared to previous smartphone implementations, this yields a performance improvement of factor 1,000x for circuit evaluations. The online phase of our fastest protocol takes only 2.92 s measured on a real WiFi connection (6.53 s on LTE) to check 1,024 client contacts against a large-scale database with 2^{28} entries. As a proof-of-concept, we integrate our protocols in the client application of the open-source messenger Signal.

1 Introduction

After installation, mobile messaging applications first perform a so-called *contact discovery*. This allows new users to automatically connect with all other users of the messaging service whose phone numbers are stored in their address book. There exist various ways to perform contact discovery. For example, WhatsApp simply uploads the user's entire address book on a regular basis to match contacts [1].

However, revealing all personal contacts to a service provider poses significant privacy risks: from the social graph of users a variety of personal information can be inferred and journalists, for example, may need to cover the identity of some of their informants to protect whistleblowers from potential consequences. When installing a mobile messaging application, users also jeopardize the privacy of people who are not even connected to the particular service by transmitting their contact information without consent. An illustrative example of a severe breach of privacy can be seen in the case of WhatsApp, which was acquired by Facebook in 2014 and shared its database with the parent company: Facebook users received friend recommendations of strangers who happened to see the same psychiatrists [33].

Unfortunately, applying simple protection mechanisms like hashing the phone numbers of contacts locally before the upload to the service provider is not helpful since these hashes are vulnerable to brute-force and dictionary attacks due to the relatively small range of possible pre-images. Furthermore, the service provider can still tell whether two users share a contact even a long time after running the discovery routine by storing the received hash values. Custom wrappers[1] for messaging applications can somewhat circumvent these problems by allowing users to manually select contacts to expose to the messaging application. However, this approach only protects the contacts of users actually using such custom wrappers. Furthermore, manually selecting the contacts to match is a usability problem.

[1] e.g., https://www.backes-srt.com/en/solutions-2/whatsbox

One possible solution to this dilemma is to apply a particular form of secure two-party computation. In general, secure two-party computation allows parties P_1 and P_2 to jointly compute a publicly known function f on their respective inputs X_1 and X_2 s.t. the parties learn no information from the protocol execution but the result. The research area of *private set intersection* (PSI) focuses on optimized protocols for the case where X_1 and X_2 are sets of elements, and f is the intersection function. PSI has been studied in great depth in the past years, yielding very efficient protocols (e.g., [41, 51]) based on oblivious transfer extensions (OTe, cf. [4, 36, 39]). However, while these protocols are very efficient in many scenarios, they turn out to be impractical for use-cases like private contact discovery on mobile devices, where the input set of the service provider is much larger (sometimes by a factor of a few million) than the input set of the user. This is because the online phase of these protocols (which depends on the actual inputs) has a computation and communication complexity that is linear in the size of the larger set.

Therefore, other PSI protocols for the case of *unbalanced* set sizes were developed (e.g., [19, 21, 40, 59]). However, only [40] actually provides an implementation on real mobile smartphone clients. The experiments performed by the authors of [40] show a rather large discrepancy between protocol execution on x86-based PC hardware and Android smartphones where performance-critical cryptographic operations are implemented in Java. In fact, their performance results do not encourage real-world deployment. For example, their fastest protocol that can easily be made secure against malicious clients requires more than 52 s on a smartphone with WiFi connection to check a single client contact against a database with only 2^{20} entries.

The developers of Signal, a mobile messaging service similar to WhatsApp but with focus on privacy, considered the use of PSI protocols for contact discovery. However, they refrained from actually implementing PSI since the academic research in PSI and the related private information retrieval (PIR) protocols "is quite a disappointment" [44]. Instead, they presented a technology preview that protects the contact discovery task on the server side with Intel Software Guard Extensions (SGX), a trusted execution environment that can be attested by remote users [45]. In theory, this yields a secure contact discovery service with negligible performance overhead compared to plain computation. However, Intel SGX is a proprietary engineering-driven solution with no cryptographic security guarantees and vulnerable to severe attacks, e.g., the recent Foreshadow attack [16] managed to reliably extract confidential data from enclaves. Moreover, some fixes for hardware security designs such as Intel SGX require hardware changes that can take years to enter the market and result in repeated acquisition costs. In contrast, fixes for flawed implementations of provably secure cryptographic protocols can be deployed quickly via software updates.

Thus, we revisit state-of-the-art unbalanced PSI protocols which provide cryptographic security and show that using new optimizations and native implementations they turn out to be practical on modern smartphones. Furthermore, we achieve security against malicious clients: since every user could run a manipulated version of the messaging application, deviations from the protocol may lead to revealing information about the server's database. On the other hand, we assume that the server behaves semi-honestly, i.e., it follows the protocol but tries to learn as much information as possible. This is a reasonable assumption since there are legal requirements and financial incentives to behave correctly: once misconduct gets known publicly, users will abandon the misbehaving service and switch to a more trustworthy alternative.

1.1 Our Contributions

As a motivation, we investigate how contact discovery is handled in widely used mobile messaging applications. For this, we conduct a survey where we analyze privacy policies, source code, and network traffic. Our results show that in practice none of these applications protect the users' privacy during contact discovery.

We optimize two protocols for unbalanced PSI that can easily be made secure against malicious clients and are suitable for private contact discovery: one that uses oblivious evaluations of the Naor-Reingold PRF (NR-PSI, cf. [31, 40, 47]) and one that uses Yao's garbled circuits (GC-PSI, cf. [40, 52, 56]) to run oblivious AES evaluations. For both protocols we apply new forms of correlated random OT precomputation (reducing the online communication by factor 2x, which is of independent interest) and introduce a method for Cuckoo filter compression (with a compression ratio of $\approx 70\,\%$ and negligible computational overhead) as well as 4.3x smaller Cuckoo filter updates to reduce the required network communication. Moreover, we improve the GC-PSI protocol by instantiating the PRF with LowMC [2], a cipher specifically designed for efficient evaluation in secure protocols, instead of the default choice AES. While this was already proposed in [40], we find optimal parameter sets for LowMC and provide implementations. Compared to AES, we thereby reduce the communication by factor 8.2x.

We provide C/C++ implementations for both protocols with security against malicious clients that make use of the Cryptography Extensions (CE) in the ARMv8 architecture available in most recent smartphones for hardware-accelerated execution. Thereby, we improve the runtime of the online phase of the GC-PSI protocol by more than a factor of 1,000x compared to the previous work of [40] that only implements security against semi-honest clients. We overcome further shortcomings of previous works w.r.t security and scalability by evaluating the implementations using recommended security parameters, reasonable false positive probabilities, and considering large-scale set sizes on the server side.

Our fastest protocol takes only 2.92 s measured on a real WiFi connection (6.53 s on LTE) and 6.07 MiB of communication in the online phase to check 1,024 client contacts against a database with 2^{28} entries (more than the number of monthly active users for popular messengers like Telegram [61]). For the setup phase it is required to transfer a compressed Cuckoo filter once whose size is linear in the number of the database entries (≈ 1 GiB for 2^{28} entries); since the filter is identical for all clients, service providers can handle the resulting traffic efficiently via CDNs. To remain practical for even larger set sizes (the market leader WhatsApp currently has more than 1.6 billion users [61]), we suggest multiple extensions, e.g., combining our protocols with multi-server PIR s.t. the overall client-server communication complexity becomes logarithmic in the size of the server database.

As a proof-of-concept, we integrate both of our protocols in the Signal Android client, thereby positioning our secure cryptographic approach as a practical alternative to vulnerable trusted execution environments like Intel SGX.

1.2 Motivating Survey

To determine how contact discovery is currently being done in practice, we conducted a survey on a comprehensive selection of mobile messengers that are "secure" in the sense that they offer end-to-end encryption. Each application was analyzed by evaluating the mandatory privacy policy, which is supposed to state exactly which data the application transmits to its server and how the server processes and stores that data. Unfortunately, these policies are not always precise enough to determine the employed contact discovery method. In these cases, we inspected the source code (if publicly available) or the network communication by means of the man-in-the-middle proxy *mitmproxy*[2]. We circumvented certificate pinning by using the *Xposed*[3] framework together with the *JustTrustMe*[4] plugin that can disable certificate checking routines in several commonly used security libraries.

Our results are summarized in Tab. 1. All surveyed messengers upload contact information (at least the contact's phone number) either in the clear or in hashed form. While this form of contact discovery is very efficient (requiring only a few bytes of communication per element), it threatens the privacy of users directly or indirectly via brute-force or dictionary attacks. Furthermore, even if the server cannot determine the actual contact data, it can still tell whether two users share a contact by comparing uploaded hash values.

This can be somewhat mitigated by using salted hashing s.t. the hashes received by the server are different whenever a client triggers contact discovery. However, only one of the surveyed messengers employs this approach as it requires to hash the entire server database for each fresh salt received by a client. Furthermore, brute-force attacks are still feasible.

Messenger	Hashed	Salted	Analysis Technique
Confide*	✓	✗	Privacy policy
Dust*	✗	✗	Network traffic
Eleet*	✗	✗	Privacy policy
G DATA Secure Chat	✓	✗	Network traffic
Signal (legacy)	✓	✗	Source code
SIMSme	✓	✓	Network traffic
Telegram	✗	✗	Privacy policy
Threema	✓	✗	Privacy policy
Viber	✗	✗	Privacy policy
WhatsApp	✗	✗	Privacy policy
Wickr Me	✓	✗	Privacy policy
Wire	✓	✗	Privacy policy

Table 1: Results of our contact discovery survey on secure mobile messengers. All applications upload contact information either in the clear or hashed (with salt). Messengers marked with * denote that contact discovery is optional.

2 Related Work

In this section, we discuss existing unbalanced PSI protocols and other works that focus on PSI in the smartphone setting.

Unbalanced PSI. Kiss et al. [40] discuss multiple unbalanced PSI protocols with precomputation (cf. §3.5) and security against semi-honest adversaries. Their NR-PSI and GC-PSI protocols (based on [31] and [52], respectively) are the foundation of our work. We augment these protocols with new OT precomputation techniques, efficient Cuckoo filters [27, 59], a specialized cipher [2] for the GC-PSI protocol, and security against malicious clients. The authors of [40] also evaluate their protocols on smartphones, but based on less efficient Java implementations. In our work, we present C/C++ implementations that make use of the hardware-accelerated cryptography available in most recent smartphones.

Resende and de Freitas Aranha [59] use techniques similar to [40], but replace Bloom filters [12] with the more efficient and versatile Cuckoo filters [27] to efficiently represent the encrypted server database (cf. §3.4) in a Diffie-Hellman style PSI protocol [7] with security against semi-honest adversaries. In our work, we optimize communication by proposing methods for Cuckoo filter compression and updates, and perform evaluations with reasonable parameters: while in [59] the authors settle with an error probability of $\approx 2^{-13}$, which results, on average, in one false positive when 10 clients match 2^{10} contacts each, we propose realistic Cuckoo filter parameters for error probabilities $\approx 2^{-29}$ and $\approx 2^{-39}$.

Demmler et al. [21] present a different approach assuming multiple non-colluding servers. Their idea is to first perform a variant of private information retrieval (PIR) to reduce the

[2] https://mitmproxy.org
[3] https://repo.xposed.info
[4] https://github.com/Fuzion24/JustTrustMe

server's input set and then perform a traditional PSI protocol on the reduced sets. While this approach is very performant, the requirement of non-colluding servers presents challenges for the data-owners: they not only need to guarantee that these servers do not collude, but also need to ensure that their client data is not leaked to other parties. This leads to the difficult situation where the server party needs to trust a second server but simultaneously is assumed to not collude with it. However, even if servers are malicious and/or collude, they cannot learn more about client inputs than in currently deployed naive hashing-based contact discovery methods.

Chen et al. [19] give a PSI protocol based on fully homomorphic encryption. The authors present multiple optimizations that make the protocol practically viable. Their work was improved and extended to the special use case of *labeled* PSI [18], where for intersecting items an associated label is transferred and security is not only guaranteed in case of malicious clients but also malicious servers (with some controlled leakage). The advantage of the protocols of [18, 19] is that their communication complexity is sublinear instead of linear in the size of the server set. However, this comes at the cost of repeated high computational overhead, whereas the online phase of our protocols is very efficient and requires no cryptographic operations on the server side.

Mobile PSI. Huang et al. [34] provided first performance results for secure computation on smartphones with security against semi-honest adversaries. They implemented a circuit-based PSI protocol on Android. Their implementation managed to evaluate ≈ 100 AND gates per second, taking about 10 min to intersect two sets of 256 items each.

Asokan et al. [6] implemented an RSA-based PSI protocol with security against semi-honest adversaries on smartphones for secure mobile resource sharing.

Carter et al. [17] presented a maliciously secure system for secure outsourced garbled circuit evaluation on mobile devices. Subsequently, Mood et al. [46] showed how to further optimize outsourced evaluation. They also point out how their framework can be used to implement a secure friend finder.

"PROUD" [49] is a decentralized approach for private contact discovery based on the DNS system. It enables users to privately discover the current network addresses of friends, which differs from the scenario of a centralized messaging service we consider. Moreover, friendship bootstrapping requires an out-of-band communication channel between users.

Compared to these works, we optimize protocols for unbalanced PSI with a central service provider and provide native implementations for maximum performance on smartphones.

3 Background

In the following, we introduce cryptographic building blocks that are required for the remainder of this work.

3.1 Oblivious Transfer (Extensions)

Oblivious transfer (OT) [57] is a cryptographic protocol that in its most basic form allows a sender P_1 to obliviously transfer one out of two messages (m_0, m_1) to a receiver P_2 based on a selection bit b chosen by P_2 s.t. P_1 learns nothing about b and P_2 learns only m_b but nothing about m_{1-b}.

It was shown in [35] that performing OTs always requires some form of public key cryptography. However, with OT extension (OTe) protocols [9, 36], a small number (e.g., 128) of "base OTs" can be extended to a large number of OTs using only efficient symmetric cryptographic operations.

There exist flavors of OTe with reduced communication complexity [5]: In random OT (R-OT), neither party inputs any values, but the inputs of sender and receiver are randomly chosen by the protocol. In correlated OT (C-OT), m_0 is chosen at random, whereas m_1 is computed as a function f of m_0: $m_1 = f(m_0)$, where f is privately known to P_1 only.

It is possible to precompute OTs s.t. all computationally expensive operations are performed via R-OTs in advance [8]. Later, the random values obtained via R-OTs are used to mask the actual inputs, requiring only cheap XOR operations in the style of one-time-pad encryption.

3.2 Garbled Circuits

Yao's garbled circuits (GC) [62] is one of the most prominent techniques for secure two-party computation. (In the following the two parties are called *garbler* and *evaluator*.) The idea is to represent the function that is evaluated as a Boolean circuit and to replace each logical two-input gate by a *garbled gate*. Each wire of the garbled gate is given two random wire labels, representing 0 and 1. To garble a gate, the garbler uses all four combinations of the gate's two input wire labels to encrypt the corresponding output wire label, based on the truth table of the original gate, and sends the resulting ciphertexts, the so-called *garbled table*, to the evaluator. The evaluator can then use the two input wire labels it possesses to decrypt one of the four ciphertexts and receive the output wire label, which is then used as input for subsequent gates.

We now describe how the evaluator obtains the wire labels corresponding to the inputs of the two parties: Since the garbler knows all wire labels, it can send the wire labels corresponding to its input bits to the evaluator. However, to ensure input privacy for the evaluator, the wire labels corresponding to the evalutor's input bits are retrieved via OTs. The garbler also sends information that allows the evaluator to decode the final output wire labels to 0 or 1.

Several optimizations for Yao's original scheme have been presented s.t. today it is most efficient to combine the following techniques: Point-and-Permute [10], Free-XOR [42], fixed-key AES garbling [11], and Half-Gates [63].

3.3 OPRF Evaluation

An oblivious pseudorandom function (OPRF) is a protocol between two parties: sender P_1 holding key k and receiver P_2 holding input x. After the invocation of the protocol, P_2 learns the output $f_k(x)$ of a keyed pseudorandom function (PRF) f. Additionally, it is guaranteed that P_1 does not learn anything about x and P_2 does not learn anything about k.

OPRF evaluations can be used to build PSI protocols as proposed in [28, 30, 40, 52]: The server samples a key k uniformly at random, evaluates the PRF $f_k(x_i)$ on each of its items $x_i \in X$, and sends the results to the client. Server and client now engage in the OPRF protocol, where the server inputs key k and the client inputs elements $y_j \in Y$. After this step, the client obtains $f_k(y_j)$ for each item $y_j \in Y$ and can perform a plain intersection between the items $f_k(x_i)$ and $f_k(y_j)$. The client then outputs the elements y_j corresponding to the values in the intersection.

In this work, we instantiate the PRF either using the Naor-Reingold PRF [47] (NR-PSI) or a garbled circuit-based evaluation of a block cipher (GC-PSI). In [37], the authors describe an alternative algebraic OPRF construction based on a PRF by Dodis-Yampolskiy [25]. However, due to the use of Paillier encryption, this construction is likely slower than the Naor-Reingold PRF and their follow-up work [38], the basis for [59] (cf. §6.2). Moreover, it requires a common reference string in the form of an RSA modulus with unknown factorization.

3.4 Cuckoo Filters

Cuckoo filters [27] are an alternative to the more popular Bloom filters [12]. Like Bloom filters, they are a data structure for compact set representation that allows for fast membership testing with controllable *false positive probability* (FPP). Cuckoo filters employ a hashing technique similar to Cuckoo hashing [48], which has been used in the past as a building block in PSI protocols (e.g., [41, 51, 53–56]).

Resende and de Freitas Aranha [59] first used Cuckoo filters in a PSI protocol. This is due to several advantages over Bloom filters when representing the server's database, namely they (i) support inserting and deleting items subsequently, whereas standard Bloom filters only support inserting items, and variants that do support deletion such as counting Bloom filters have much higher storage costs; (ii) have better lookup performance; and (iii) use less space in many scenarios while having the same false positive probability.

Cuckoo filters consist of a table of buckets with fixed bucket size b. Inside the buckets, so-called tags are stored. Tags are small bitstrings obtained by hashing items. More precisely, to represent an item x in a Cuckoo filter, we first calculate its tag $t_x = H_t(x)$, where H_t is a hash function with output bitlength v. This tag is stored in one out of two possible buckets. The position of the first possible bucket is calculated as $p_1 = H(x)$, where H is another hash function that maps the input to a position in the table of buckets. In case this bucket is already full, the tag is stored in the second possible bucket at position $p_2 = p_1 \oplus H(t_x)$. Note that it is always possible to determine the other candidate bucket p_j just from knowing its tag t_x and the current position p_i: $p_j = p_i \oplus H(t_x)$. If both buckets are full, one tag in one of the buckets is chosen at random, removed from that bucket, and moved to its other possible bucket. This procedure is repeated recursively until no more relocations are necessary.

To check whether an item is contained in the Cuckoo filter, one computes its tag and both possible bucket locations and compares the tags stored there for equality. For deleting the item, the matching tag is removed from the filter.

Due to hash collisions, two items may produce equal tags. As a consequence, lookups can lead to false positives. The false positive probability ε_{max} is mainly dependent on the tagsize v and also slightly on the bucket size b since larger buckets result in more possible collisions within each bucket.

3.5 Unbalanced PSI with Precomputation

For private contact discovery, the following properties are desired: (i) the server performs the computationally expensive tasks; (ii) all computationally expensive and communication intensive tasks are performed only once; and (iii) the actual intersection computation is very fast and also allows for efficient updates. Therefore, [40] suggest to use PSI protocols with precomputation, where most time consuming tasks are performed ahead of the actual intersection.

Our PSI protocols for unbalanced set sizes share a common structure. Following the precomputation approach of [40], they are divided into the following four phases: (i) The *base phase* is completely independent of any input data and consists, e.g., of OT precomputation. Its complexity is linear in the maximum number of contacts a client expects to match in future protocol executions before the base phase is re-run. (ii) The complexity of the *setup phase* is linear in the size of the large set held by the server. It involves encrypting all elements in the server database via PRF evaluations as described in §3.3 and inserting them into a Cuckoo filter for compact representation, which is transferred to the client. (iii) During the *online phase* client and server jointly perform OPRF evaluations on all elements of the client. The client then looks up all received encryptions in the Cuckoo filter to determine the intersection. Thus, the complexity of the online phase is only linear in the size of the small client set. (iv) Changes in the server database trigger the *update phase*, where the Cuckoo filter on the client side is updated by sending a small delta for each inserted or deleted database entry.

4 Optimizing OPRF-based PSI Protocols

We propose more efficient database representations and PRFs, give the full descriptions for our optimized NR- and GC-

PSI protocols, enable security against malicious clients, and suggest multiple extensions to further increase practicality.

4.1 More Efficient Database Representations

Realistic Cuckoo Filter Parameters. Resende and de Freitas Aranha [59] propose using Cuckoo filters as an extension to the DH-based PSI protocol of [7] and they perform experiments to find optimal Cuckoo filter parameters based on the number of server items and the desired error probability. While their findings are directly applicable to our use case, they set very aggressive Cuckoo filter parameters (tagsize $v = 16$, bucket size $b = 3$) and settle for a maximum *false positive probability* (FPP) of $\varepsilon_{max} \approx 2^{-13}$. We find this FPP not practical since it implies that about one in 10 clients performing PSI for 2^{10} elements receives a false positive.

Instead, we propose to use tagsize $v = 32$ to reach a FPP of $\varepsilon_{max} \approx 2^{-29}$ or tagsize $v = 42$ to reach a FPP of $\varepsilon_{max} \approx 2^{-39}$ while still maintaining a bucket size of $b = 3$. For our experiments, we choose the parameter set $v = 32, b = 3$, and choose the size of the Cuckoo filter to have a load factor of $\approx 66\%$, leading to a Cuckoo filter size of 6 MiB per 2^{20} items.

Novel Cuckoo Filter Compression. The size of Cuckoo filters can be reduced by applying a simple but effective compression technique that to the best of our knowledge was not considered before: For each entry of a Cuckoo filter, an additional bit is transmitted that indicates whether this entry is empty or holds a tag. The entry itself is only transmitted if it is not empty. This way, the filter is represented as a bit map and a list of tags. For a Cuckoo filter storing n items with tagsize v, bucket size b, and load factor l, this reduces the size from $\frac{n}{l} \cdot v$ bits to $\frac{n}{l} + n \cdot v$ bits. In the example above, the size of the Cuckoo filter is reduced from 6 MiB to 4.19 MiB, i.e., by $\approx 30\%$. An advanced version of the compression technique presented above encodes the number of tags (0 to b) in each bucket with $\log_2(b+1)$ bits instead of sending b bits per bucket. This is possible since the actual position of each tag within a bucket is not important.

This compression technique is especially useful for very sparse Cuckoo filters, which appear in use cases where the set of items is expected to grow fast (e.g., during the release phase of a new messaging application). For example, if only 10 % of a Cuckoo filter storing a maximum of 2^{20} items is occupied, it can be compressed by a factor of 8.3x.

In concurrent and independent work, Breslow and Jayasena [15] proposed *Morton filters*, which combine these compression techniques with cache-optimized layouts and further optimizations. Morton filters provide higher insertion, lookup, and deletion throughput than traditional Cuckoo filters, while usually having equal or slightly lower storage costs. We leave the evaluation and usage of Morton filters in our protocols for future work.

Better Cuckoo Filter Updates. In [59], when performing an update after new elements are inserted into or deleted from the server's set, each encrypted element to be updated is sent to the client where it is inserted into the existing Cuckoo filter. However, for Cuckoo filters, all information required to insert a new item is its tag and the index of one of its candidate buckets. From this information, it is possible to calculate the second candidate bucket in case relocations are necessary. The same information is also sufficient to delete an item. For example, the bucket index in a Cuckoo filter storing $n = 2^{28}$ items with bucket size $b = 3$ and load factor $\approx 66\%$ can be represented with 27 bits. This results in sending 59 bits per updated element for tagsize $v = 32$. In comparison, in [59] an encrypted element is represented by one point on the GLS-254 binary elliptic curve, which results in 256 bits of communication when using point compression with two trace bits, which needs 4.3x more communication than our approach.

4.2 More Efficient PRF for GC-PSI

During the online phase of the GC-PSI protocol, both parties interactively evaluate an OPRF on the client's items using garbled circuits. For each of the client's items, the server prepares a garbled circuit $\widetilde{PRF}_k$ that evaluates the chosen PRF under the server's key k. The choice of this PRF has a significant impact on both the runtime and the communication complexity of the overall protocol. Several improvements for Yao's GC protocol [62] have appeared in recent years that changed the desired properties of the functions to be evaluated. Most notably is the Free-XOR [42] optimization, which allows XOR gates to be evaluated securely "for free", meaning all necessary operations can be performed locally without any communication between the parties. This optimization has lead to research in the area of ciphers with a low number of AND and instead many free XOR gates.

In previous GC-PSI implementations, the choice of the PRF was AES-128. Using the optimized S-Box implementation of [13], an AES-128 circuit (without key schedule) has 5,120 AND gates [32], serving as a baseline for comparison.

In this section, we focus on variants of LowMC [2], a highly parameterizable block cipher designed for use cases in multiparty computation (MPC) and fully-homomorphic encryption (FHE). [40] mentioned the possibility of using LowMC instead of AES for GC-PSI. We look at several instantiations of LowMC and present optimized parameter sets specifically for the use case of GC-PSI and mobile contact discovery. In the following, we give a short description of LowMC and highlight the different parameter choices.

LowMC [2] is a block cipher where block size n, key size k, number of S-Boxes per substitution layer m, and allowed data complexity d can be chosen freely up to some sanity constraints. The required number of rounds r to reach the security claims is then derived from these parameters.

Data Complexity. The data complexity of a cipher is the number of plaintext-ciphertext pairs allowed to be released before the security claims no longer hold. In the GC-PSI protocol, we can exactly control the maximum number of published plaintext-ciphertext pairs by limiting the number of client queries, and therefore can reduce the number of LowMC rounds required for security. We set the allowed data complexity to be $d = 2^{64}$, allowing for 2^{20} contact discoveries of 2^{10} items for each of the 2^{28} clients, while still being below the security margin by a factor of over 100x. For smaller-scale applications, we also give a parameter set for 2^{32} total data complexity, which suffices to run 2^{20} queries of 2^{10} items each. While we could also use this parameter set for larger-scale applications, the system needs to be re-keyed after the data complexity has been reached.

Key Schedule. In many MPC applications using OPRF evaluations, one party knows the entire secret key and can, therefore, perform any key-scheduling algorithm (e.g., for AES or LowMC) offline. The circuit is then modified to take the expanded key as an input. In many cases, this can be a performance improvement since the key-schedule algorithm does not have to be computed using the MPC protocol. However, when performing OPRF evaluations using garbled circuits, the party holding the secret key needs to send wire labels for each input bit, increasing the communication. While for AES-128, only 11x more wire labels need to be transferred for the expanded key, some instantiations of LowMC require several hundreds of rounds. Sending labels for the expanded key essentially removes the advantage of the lower AND count that comes with such a large number of rounds. However, we observe that in the GC-PSI protocol the OPRF evaluation is always performed with the same key. Thus, we can bundle all of the client's circuits together into one large circuit and evaluate the key-schedule only once. This means that we only need to send the wire labels corresponding to the non-expanded key once, and therefore save ≈ 2 KiB for each subsequent client item when using a 128-bit key. It is also possible to only evaluate parts of the garbled circuit if the number of client items is lower than the number of precomputed circuits.

LowMC Instances. For use in our GC-PSI protocol, we highlight several LowMC instances, exploring different parameter choices. In Tab. 2, we give the parameters and compare the number of AND gates to AES-128. The number of rounds is calculated according to the LowMCv3 round formula[5], which was updated by the LowMC team to take new cryptanalysis of LowMC (cf. [23, 24, 58]) into consideration. We can observe some interesting properties: LowMC instances (1) and (2) require the same number of rounds to be secure, but instance (1) has the maximum number of possible S-Boxes, while (2) does not. Since instance (2) provides the same security as (1) while requiring fewer S-Boxes, and therefore a lower amount of AND gates, it should always be preferred. LowMC instance (3) has the smallest possible S-Box layer with only one S-Box per round and also the lowest number of AND gates. While its 208 rounds can be a drawback in some protocols, Yao's GC protocol [62] has a constant number of communication rounds and therefore the large number of LowMC rounds does not decrease performance in high-latency networks. Additionally, using the optimizations presented by [22], the large number of linear layer computations can be reduced, bringing the evaluation time of (3) close to (1) and (2). For these reasons, we recommend the use of instance (3) for GC-PSI, which requires 8.2x fewer AND gates than standard AES-128 (6). Thus, we perform all performance evaluations using instance (3). For use cases with small data complexity requirements, we recommend LowMC instance (4), which is a small improvement of 8.3 % in runtime and communication compared to (3). For completeness and direct comparison to AES-128, we also give a variant of LowMC with data complexity of 2^{128} in (5).

[5] https://github.com/LowMC/lowmc/blob/master/determine_rounds.py

	PRF	n	k	m	d	r	#ANDs
(1)	LowMC	128	128	42	2^{64}	13	1,638
(2)	LowMC	128	128	31	2^{64}	13	1,209
(3)	**LowMC**	**128**	**128**	**1**	$\mathbf{2^{64}}$	**208**	**624**
(4)	LowMC	128	128	1	2^{32}	192	576
(5)	LowMC	128	128	1	2^{128}	287	861
(6)	AES-128	128	128	16	2^{128}	10	5,120

Table 2: Comparison of PRF instances for use in the GC-PSI protocol. The recommended instance is highlighted in bold.

4.3 Optimized GC-PSI Protocol

The idea of using Yao's GC protocol for OPRF evaluations was first proposed in [52] and used to construct a PSI protocol in the precomputation setting in [40].

The full protocol description is given in Fig. 1.We propose an optimization that halves the online communication for the OTs (which is the only communication in the online phase). This optimization is of independent interest as it improves the practicality of Yao's GC protocol in arbitrary use cases with precomputation. It is based on the observation that with the Free-XOR technique [42] for Yao's GC protocol [62], the client receives one of the two labels l^0 and $l^1 = l^0 \oplus \Delta$ via OT depending on its input bit, where l^0 is chosen at random and Δ is a random global constant only known by the garbler. A natural consideration would be to replace the real OTs, as used in [40], with correlated OTs (C-OTs) (cf. §3.1). Unfortunately, since the client input is unknown in the base phase, this prevents either the precomputation of the garbled circuits or the OTs. This is because in the online phase when using OT precomputation [8], the random messages r^0 and r^1 obtained by the sender in the base phase need to be swapped

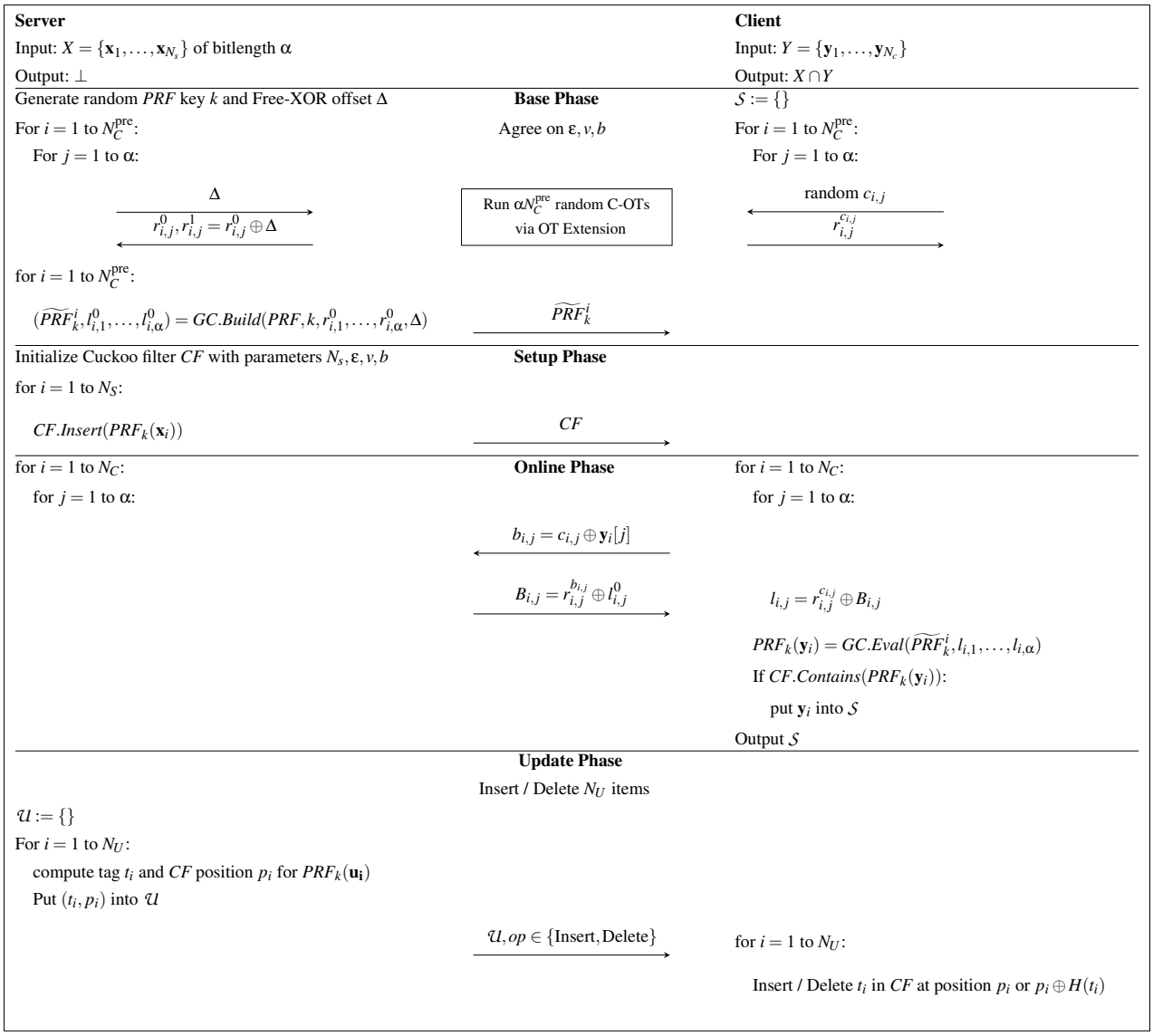


Figure 1: Our optimized GC-PSI protocol (based on [40, 52, 59]). Wire labels are computed as $l_{i,j}^0 = r_{i,j}^0 \oplus \delta_{i,j}$ and $l_{i,j}^1 = l_{i,j}^0 \oplus \Delta$, where the values $\delta_{i,j}$ are chosen at random while building the garbled circuit. $N_C^{\text{pre}} \geq N_C$ denotes the number of precomputed OTs and garbled circuits; the base phase must be repeated before further online phase executions once N_C^{pre} queries are exceeded.

in case the random choice made by the receiver differs from its actual input. Thus, it would be necessary to swap input wire labels in the garbled circuits, which requires recomputing and resending at least the first layer of those circuits.

Our novel precomputation method circumvents this dilemma: In the base phase we run C-OTs via OT extension s.t. the garbler on input Δ learns the random but correlated values r^0 and $r^1 = r^0 \oplus \Delta$, whereas the evaluator upon random choice c learns r^c. For garbling we choose the labels for the input wires of the circuit as $l^0 = r^0 \oplus \delta$ and $l^1 = l^0 \oplus \Delta$. Here, δ is a newly introduced random value that in contrast to Δ is not global but chosen individually for each label pair. In the online phase of the protocol, the evaluator sends a correction bit $b = c \oplus y$ stating whether its random choice c differs from the actual input y. The garbler responds with $B = r^b \oplus l^0$. This way, the evaluator learns either δ or $\delta \oplus \Delta$. It then sets the label for its input to $l = r^c \oplus B$. As one can easily verify for the four possible combinations of random choices c and correction bits b, the evaluator always retrieves the correct label.

The security of the C-OT precomputation is based on the same arguments as standard OT precomputation [8] and since we use a fresh uniformly random δ for each wire label, the resulting wire label is also uniformly random. In other words, we resolve the problem by fixing the wire labels but if necessary swapping the masks required to retrieve the correct label from the initial C-OT result.

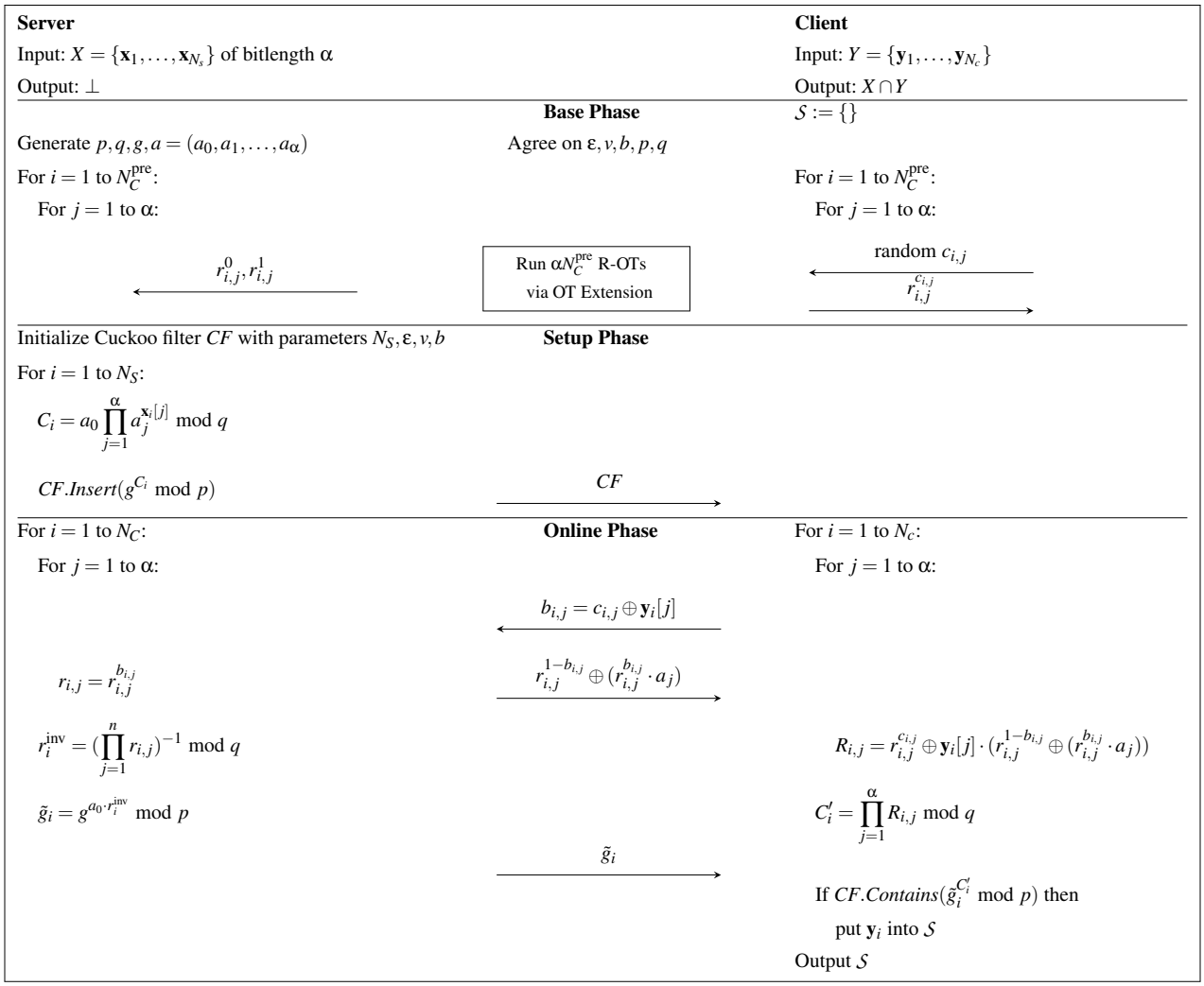


Figure 2: Our optimized NR-PSI protocol (based on [31, 40, 59]). When using a plain finite field, the modulus p is prime, q is a prime divisor of $p-1$, $g \in \mathbb{Z}_p^*$ is of order q, and $a_0, a_1, \ldots, a_\alpha$ as well as $r_{i,j}^0, r_{i,j}^1$ are random numbers in $\mathbb{Z}_q^*$. The update phase is omitted since it is similar to the GC-PSI protocol (cf. Fig. 1), except using the NR-PRF to compute tag t_i and CF position p_i.

4.4 Optimized NR-PSI Protocol

The usage of the Naor-Reingold PRF (NR-PRF) [47] for PSI was first proposed in [31] and the resulting PSI protocol transformed into the precomputation setting in [40]. The NR-PRF for key k and element x_i is defined as

$$f_k(x_i) = g^{a_0 \cdot \prod_{j=1}^{\alpha} a_i^{x_{i,j}}} \mod p, \tag{1}$$

where, when using a plain finite field, p is a prime, q is a prime divisor of $p-1$, $g \in \mathbb{Z}_p^*$ is a generator of order q, $a_0, a_1, \ldots, a_\alpha$ are random numbers in $\mathbb{Z}_q^*$ forming key k, and α is the bitlength of element x_i.

Among all protocols for mobile contact discovery evaluated in [40], NR-PSI is the only protocol besides GC-PSI that can easily be made secure against malicious clients by employing malicious secure OT extensions (cf. §4.5). Furthermore, according to the empirical performance comparison in [40], the NR-PSI protocol causes $\approx$ 30x less communication overhead than GC-PSI without our optimizations. This is why we also consider the NR-PSI protocol in this work and compare it to our optimized GC-PSI implementation in §6.

The full protocol description is given in Fig. 2. We propose an optimization that improves the online communication for OTs by factor 2x. The optimization is based on the observation that in the definitions of [31] the client chooses between a random r and $r \cdot a$ depending on the current bit of its input element. This implies that C-OTs (cf. §3.1) can be used instead of real OTs, thereby sending only one message in the size of the symmetric security parameter instead of the two messages when using the OTe protocols of [3].

Since we use the precomputation form of [40], we propose a novel combination of OT precomputation [8] and C-OT [3]. As in OT precomputation, the client sends a correction bit b stating whether its random choice c in the precomputation

phase equals its real input. Depending on b, the server then decides which of the two random messages obtained during OT precomputation is chosen as r and which is used to mask the correlated message $r \cdot a$ that is sent to the client. Likewise, the client either proceeds with the message obtained during OT precomputation as r or uses this message to unmask the received correlated message.

4.5 Malicious Security

As observed already in [40], the only messages sent by the client in the GC-PSI and NR-PSI protocols are those in the base OT and OT extension protocols as well as the correction bits during the online phase when applying OT precomputation [8]. Therefore, both protocols can easily be made secure against a malicious client by using a maliciously secure OTe protocol such as [4] or [39], together with maliciously secure base OTs such as [50]. As the OT extension contributes only a small percentage to the total runtime of the PSI protocols and today's maliciously secure OTe protocols are only slightly less efficient than the passively secure OT extension of [3], the total runtime of the PSI protocols does not increase by a noticeable amount when replacing the OTe protocols. Please note that enumeration attacks (i.e., querying the server repeatedly with different inputs) are still possible when using our protocols. However, even an ideal functionality for PSI (e.g., a trusted third party) and currently deployed non-private contact discovery methods cannot prevent this. We recommend to employ well-established measures like rate limiting to mitigate such attacks.

The case of a malicious server is different: it could, for example, send wrong wire labels, use wrong circuit descriptions, or send a wrong server set. In general, the client does not reveal the intersection result to the server, so a malicious server can only influence the correctness of the client's computation, but cannot learn any information about the client's items when using maliciously secure OTs. Unfortunately, in most mobile messaging applications, the client sends information about the intersection (most likely even the entire intersection) to the server. This allows a malicious server to learn information about the client's items that are not part of the intersection of the two actual input sets. Therefore, we need to assume a semi-honest server in such scenarios. Preventing malicious behavior on the server side could be done by combining our protocols with a trusted execution environment for hardware-enforced code and remote attestation capabilities s.t. the server's protocol deviation possibilities are restricted to wrong inputs for the Cuckoo filter construction. However, assuming a semi-honest server is reasonable since there are legal requirements and financial incentives for a service provider to behave correctly: once misconduct gets known publicly, users will abandon the malicious service and switch to a more trustworthy alternative.

4.6 Further Extensions

The bottleneck for very large server sets is the communication required to send the Cuckoo filter to the client. For example, a compressed Cuckoo filter for 2^{28} server items with false positive probability $\varepsilon_{max} \approx 2^{-29}$ has a size of ≈ 1 GiB, which is prohibitively large for transmission on mobile network speeds and data plans. For even larger server databases, the protocols eventually become impractical. For example, for a server database with 2^{31} entries, it would be necessary to download a Cuckoo filter of size ≈ 8 GiB. Therefore, we describe how to reduce the overall client-server communication to be logarithmic in the size of the server database. We propose further extensions to increase practicality in App. A.

Combination with Private Information Retrieval (PIR). In their PIR-PSI protocol, Demmler et al. [21] propose the use of multiple non-colluding servers together with a multi-server PIR protocol. Applied to our PSI protocols, the extension works as follows: After the server prepared the Cuckoo filter, it is not transmitted to the client, but to a second non-colluding server instead. Since the Cuckoo filter only contains the results of PRF evaluations, the second server does not learn anything about the items in the main server's set. The client then performs the OPRF evaluation for each of its items with the first server and then runs a multi-server PIR protocol to retrieve the fingerprints stored in the Cuckoo filter.

The communication complexity for the multi-server PIR lookup is $O(\kappa \log n)$, where κ is the symmetric security parameter and n the size of the server database [14, 21]. Since the overall client-server communication therefore is logarithmic and not linear in the size of the server database, our protocols are expected to remain practical even for server databases with more than a billion items. In practice, the remaining challenge for messaging services is to find a trustworthy partner operating the second PIR server while at the same time making it credible to users that no collusion is happening.

5 Android Implementation

To demonstrate the feasibility of our optimized PSI protocols for performing private contact discovery on mobile devices, we provide implementations for smartphones running on Android.[6] Previous works [34, 40] presented experiments on dedicated mobile devices, but the performance of these implementations was not sufficient for real-world usage. For example, the Java implementation of [40], which is based on the ObliVM framework [43], takes more than a second to evaluate a single garbled AES-128 circuit. In our implementation, we make use of native C/C++ code support in Android and also use hardware acceleration for cryptographic operations available in modern smartphones. More precisely, native

[6] https://contact-discovery.github.io

AES-128 instructions are used both as a PRNG and during the creation and evaluation of the garbled circuit. These features allow our implementation to reach truly practical performance. Compared to the Java-based implementation of [40], we evaluate a garbled AES-128 circuit more than 1,000x faster.

5.1 Base OTs and OT Extension

For performing base OTs, we use the OT protocol of Chou and Orlandi [20] with the additional verification step proposed by Doerner et al. [26]. Together with the (C-)OT extension protocol of Keller, Orsini, and Scholl [39], this results in a maliciously secure protocol (cf. [26]).

Our OT implementation is based on `libOTe` by Rindal [60], which is heavily optimized for the x86 architecture. Thus, we ported large parts of the library to the ARMv8 architecture to achieve high performance on mobile devices. At the same time, we kept the library compatible with its x86 counterpart to facilitate natural development of client-server applications.

5.2 GC-PSI Implementation

For the GC-PSI protocol, we implement Yao's GC protocol (cf. §3.2) with Free-XOR [42] and Half-Gates [63], resulting in no communication for XOR-gates and two wire labels of κ bits each per AND gate, where $\kappa = 128$ is the symmetric security parameter.

For creating and evaluating the garbled tables, the most efficient choice today is fixed-key AES [11], mainly due to the hardware support for AES that is widespread in modern x86 CPUs. The ARM Cryptography Extensions (CE) introduced in the ARMv8 architecture similarly provide hardware instructions for AES, SHA-1, and SHA-2 variants, resulting in AES speedups of factor 35x compared to a standard AES software implementation. This allows us to also use fixed-key AES [11] for garbling in our implementation.[7] Additionally, the ARMv8 architecture provides instructions for vector operations on 128-bit registers (the so-called NEON instruction set), which we use to efficiently work with 128-bit wire labels. In Tab. 7 in App. B, we demonstrate the wide availability of ARM CE in most recent smartphone processors.

5.3 NR-PSI Implementation

For implementing the NR-PSI protocol, we use the modified `libOTe` version described in §5.1 for C-OT precomputation as well as the `GNU GMP`[8] library for modular arithmetic operations and the `MIRACL`[9] library for instantiating the protocol with elliptic curve P-256. The advantage of instantiating the NR-PSI protocol with ECC instead of using a plain finite field with comparable security parameters is that the size of the values $\tilde{g}_i$ transferred during the online phase (cf. Fig. 2) is reduced by factor 8x. Also, computationally expensive modular exponentiations are replaced with point multiplications. We refer to this variant as ECC-NR-PSI in the following. All libraries are compiled specifically for the ARMv8 architecture.

[7] As recently reported by [29], many secure computation implementations use fixed-key AES incorrectly. However, according to [29], our instantiation for garbling following the definitions of [63] is not affected. In contrast, `libOTe` [60] is currently vulnerable. The suggested fixes however are not expected to result in a significant negative performance impact [29].

[8] https://gmplib.org

[9] https://github.com/miracl/MIRACL

6 Performance Evaluation

We empirically evaluate the performance of our optimized GC-PSI and NR-PSI protocols and compare them to other unbalanced PSI protocols from the literature.

Benchmark Settings. For easy comparison to related work, we choose similar sizes for the server's and the client's set: $N_s \in \{2^{20}, 2^{24}, 2^{26}, 2^{28}\}$ and $N_c \in \{1, 2^8, 2^{10}\}$. Here, $N_c = 1$ represents the case where a client wants to check a new contact. All items have a bitlength of $\alpha = 128$. We instantiate all primitives and protocols with 128-bit security.

In all of our experiments, the sever is equipped with an Intel Core™ i7-4600U CPU @ 2.6 GHz and 16 GiB of RAM. The client is a Google Pixel XL 2 smartphone with a Snapdragon 835 CPU @ 2.45 GHz and 4 GiB of RAM. We consider two network settings: (i) an IEEE 802.11ac WiFi connection with $\approx$ 230 Mbit/s down-/upload and 70 ms RTT and (ii) a mobile LTE connection with 42 Mbit/s download ($S \rightarrow C$), 4 Mbit/s upload ($S \leftarrow C$), and 80 ms RTT.

Note that the LTE network speeds are real-world parameters and exhibit a significant difference in the down- and upload rates. This is common in commercially available data plans and often not taken into account in previous evaluations.

6.1 GC-PSI and NR-PSI Protocol

The runtime and communication costs for the base, setup, and online phase of our protocols are shown in Tab. 3, Tab. 4, and Tab. 5, respectively, and are averaged over 100 executions (except for the setup phase, where we chose 10 or less executions due to the larger runtime). We use LowMC instance (3) from Tab. 2 for the evaluation. In all tests, only a single thread was used for both the server and the client. Since all phases of our protocols can be parallelized trivially, we expect a near-linear speedup when using multiple threads, except in situations where the bottleneck is network bandwidth. Furthermore, note that in the base and online phases of the GC-PSI protocol, only one party actually performs the computationally expensive task of garbling or evaluating the circuit. Therefore, if both parties are ready, the base and online phases of the GC-PSI protocol can be interleaved in a pipelined fashion, where the server sends the garbled circuits and the client evaluates them as soon as parts of them are available. This

Parameters		Time [s]		Comm. [MiB]	
N_c^{pre}	Protocol	WiFi	LTE	$S \rightarrow C$	$S \leftarrow C$
2^{10}	AES-GC-PSI	7.14	38.98	162.52	2.02
	LowMC-GC-PSI	1.85	6.57	22.01	2.02
	ECC-NR-PSI	**0.61**	**4.21**	**0.01**	**1.99**

Table 3: Base phase of our PSI protocols. Precomputation for checking N_c^{pre} client contacts. Best results marked in bold.

Parameters		Server Setup [s]	Transmission [s]		Comm. [MiB]
N_s	Protocol		WiFi	LTE	$S \rightarrow C$
2^{28}	AES-GC-PSI	**23.94**	32.66	211.30	1072
	LowMC-GC-PSI	1,869.13			
	ECC-NR-PSI	52,332.38			
2^{26}	AES-GC-PSI	**4.87**	8.13	52.55	268
	LowMC-GC-PSI	467.29			
	ECC-NR-PSI	12,787.79			
2^{24}	AES-GC-PSI	**1.12**	2.13	13.05	67
	LowMC-GC-PSI	116.66			
	ECC-NR-PSI	3,297.96			
2^{20}	AES-GC-PSI	**0.06**	0.25	0.63	4.19
	LowMC-GC-PSI	7.27			
	ECC-NR-PSI	241.54			

Table 4: Setup phase of our PSI protocols. Server setup run once for *all* clients. The Cuckoo filter parameters are set as described in §4.1 ($\varepsilon_{max} = 2^{-29.4}, v = 32, b = 3$). Best results marked in bold. Note that the size of the client set does not influence the runtime of the setup phase and the client does not send any data during the setup phase in any protocol.

method can reduce the runtime of the combined base and online phase to the runtime of the slower phase.

We observe that using LowMC instead of AES in the GC-PSI protocol leads to 7.4x less communication and thus to a much smaller runtime in the base phase, while the online phase of both protocol versions is very comparable. Only during the one-time setup phase, the AES version is more efficient due to AES-NI instructions. Using a hardware-accelerated implementation of LowMC could reduce this runtime close to the one of AES, but we again stress that the setup phase is a one-time cost. This confirms our choice of LowMC over AES as the PRF in GC-PSI.

ECC-NR-PSI is the most efficient protocol during the base phase since it does not send garbled circuits to the client: compared to the LowMC version of GC-PSI, it requires 12x less communication. The ECC-NR-PSI online phase is slightly slower than both GC-PSI protocols, while being the fastest for a single item. The one-time setup phase of the ECC-NR-PSI protocol is much slower than both GC-PSI protocol versions due to elliptic curve operations.

6.2 Comparison with Related Work

We now highlight differences to other works in the literature and compare our optimized GC- and NR-PSI protocols and implementations to other unbalanced PSI implementations available for Android in Tab. 6. Comparisons with implementations for the x86 architecture are given in App. D.

Parameters		Time [s]		Comm. [KiB]	
N_c	Protocol	WiFi	LTE	$S \rightarrow C$	$S \leftarrow C$
2^{10}	AES-GC-PSI	**1.43**	**1.86**	**2,048**	**16.00**
	LowMC-GC-PSI	1.71	2.02	**2,048**	**16.00**
	ECC-NR-PSI	2.31	2.32	4,147	**16.00**
2^8	AES-GC-PSI	**0.34**	**0.47**	**512**	**4.00**
	LowMC-GC-PSI	0.37	0.48	**512**	**4.00**
	ECC-NR-PSI	0.61	0.61	1,037	**4.00**
1	AES-GC-PSI	0.03	0.03	**2.00**	**0.02**
	LowMC-GC-PSI	0.04	0.05	**2.00**	**0.02**
	ECC-NR-PSI	**0.01**	**0.02**	4.06	0.04

Table 5: Online phase of our PSI protocols. Best results marked in bold. The influence of the server set size on runtime and communication is negligible and therefore not listed.

Chen et al. [18, 19]. The protocols of [18, 19] for unbalanced PSI are based on leveled fully homomorphic encryption (FHE). They both work as follows: the client encrypts all its items and sends them to the server, which then computes the intersection under encryption with all of its own items and returns the result in encrypted form. The client can then decrypt the received ciphertexts to find the intersection.

The protocol in [19] is only defined for 32 bit strings, a limitation that stems from the parameter choice of the FHE scheme. Since the universe of possible items is larger than 2^{32} in the use case of contact discovery, we exclude this protocol from further comparisons. However, this limitation was lifted in the subsequent work [18] where arbitrary length items are supported. The benefits of [18] compared to our protocols are that the client is not required to store any data and that the total communication is sublinear in the size of the server database. For example, for $N_s = 2^{28}$, the total communication in the protocol of [18] is only 18.4 MB.

However, there is a huge computational overhead during the online phase of the protocol: even on a high-end server it takes more than 12 s on 32 threads to compute the intersection with $N_c = 1024$ client elements. Unfortunately, the online phase needs to be repeated whenever there are updates on client or server side. Also, due to the employed FHE batching optimizations, the runtime for a single item is almost equal to the runtime for thousands of items. Assuming that each of the $N_s = 2^{28}$ registered clients runs one update per day, this would require the service provider to pay for $2^{28} \cdot 12.1 \cdot 32 \approx 28.9$ million core hours every day. In contrast, the online phases of our protocols run in ≈ 2 s for $N_c = 1024$ in the WiFi setting on a single-threaded smartphone and require no cryptographic operations on server side. The evaluation of [18] was performed on two servers with Intel Xeon CPUs in a 10 Gbit/s local network. Therefore, it is also unclear how the FHE encryption and decryption routines perform in a mobile setting on real smartphones.

Resende and de Freitas Aranha [59]. In [59], the authors present implementation improvements for the PSI protocol of [7]. For each element in the client's set, they perform 3

Parameters N_s	Parameters N_c	PSI Protocol	Base + Online Time [s] WiFi	Base + Online Time [s] LTE	Communication [MiB] $S \rightarrow C$	Communication [MiB] $S \leftarrow C$	Setup Communication / Client Storage [MiB]	Setup Transfer [s] WiFi	Setup Transfer [s] LTE	Server Setup [s]
2^{28}	1,024	AES-GC-PSI [40]	1,507.73	2,742.66	177.23	4.00	1,380.25	42.05	272.06	**26.70**
		NR-PSI [40]	171.23	221.20	64.25	2.02	1,380.25	42.05	272.06	194,130.21
		LowMC-GC-PSI (Ours)	3.54	8.59	22.01	2.02	**1,072.00**	**32.66**	**211.30**	1,869.13
		ECC-NR-PSI (Ours)	**2.92**	**6.53**	**4.07**	**2.00**	**1,072.00**	**32.66**	**211.30**	52,332.38
	1	AES-GC-PSI [40]	1.53	2.95	0.18	0.02	1,380.25	42.05	272.06	**26.70**
		NR-PSI [40]	0.17	0.21	0.06	**0.01**	1,380.25	42.05	272.06	194,130.21
		LowMC-GC-PSI (Ours)	0.17	0.18	0.04	0.02	**1,072.00**	**32.66**	**211.30**	1,869.13
		ECC-NR-PSI (Ours)	**0.13**	**0.13**	**0.01**	**0.01**	**1,072.00**	**32.66**	**211.30**	52,332.38
2^{24}	1,024	AES-GC-PSI [40]	1,507.73	2,742.66	177.23	4.00	86.26	2.74	16.80	**1.18**
		NR-PSI [40]	171.23	221.20	64.25	2.02	86.26	2.74	16.80	12,174.40
		LowMC-GC-PSI (Ours)	3.54	8.59	22.01	2.02	**67.00**	**2.13**	**13.05**	116.66
		ECC-NR-PSI (Ours)	**2.92**	**6.53**	**4.07**	**2.00**	**67.00**	**2.13**	**13.05**	3,297.96
	1	AES-GC-PSI [40]	1.53	2.95	0.18	0.02	86.26	2.74	16.80	**1.18**
		NR-PSI [40]	0.17	0.21	0.06	**0.01**	86.26	2.74	16.80	12,174.40
		LowMC-GC-PSI (Ours)	0.17	0.18	0.04	0.02	**67.00**	**2.13**	**13.05**	116.66
		ECC-NR-PSI (Ours)	**0.13**	**0.13**	**0.01**	**0.01**	**67.00**	**2.13**	**13.05**	3,297.96
2^{20}	1,024	AES-GC-PSI [40]	1,507.73	2,742.66	177.23	4.00	5.39	0.32	0.81	**0.05**
		NR-PSI [40]	171.23	221.20	64.25	2.02	5.39	0.32	0.81	758.40
		LowMC-GC-PSI (Ours)	3.54	8.59	22.01	2.02	**4.19**	**0.25**	**0.63**	7.27
		ECC-NR-PSI (Ours)	**2.92**	**6.53**	**4.07**	**2.00**	**4.19**	**0.25**	**0.63**	241.54
	1	AES-GC-PSI [40]	1.53	2.95	0.18	0.02	5.39	0.32	0.81	**0.05**
		NR-PSI [40]	0.17	0.21	0.06	**0.01**	5.39	0.32	0.81	758.40
		LowMC-GC-PSI (Ours)	0.17	0.18	0.04	0.02	**4.19**	**0.25**	**0.63**	7.27
		ECC-NR-PSI (Ours)	**0.13**	**0.13**	**0.01**	**0.01**	**4.19**	**0.25**	**0.63**	241.54

Table 6: Comparison of PSI protocols with smartphone implementations. Numbers for protocols of [40] are obtained by running their implementations in our benchmarking environment. In all tests $N_c^{\text{pre}} = N_c$. Best in class marked in bold.

point multiplications and transmit 2 group elements. This results in a lower communication than our approaches (64 B for 2 group elements vs. 22 KiB per garbled circuit vs. 6 KiB per item in NR-PSI). However, one major contribution of [59] is a significant optimization of the GLS-254 curve for x86 CPUs. It is therefore unclear how their protocol performs on smartphones with ARMv8-A hardware. Furthermore, their Cuckoo filters parameters allow for a false positive probability that is too high for real-world deployment (cf. §4.1). Finally, their protocol assumes semi-honest adversaries, and while a maliciously secure variant [38] of their basic protocol exists, its performance has not yet been evaluated.

Kiss et al. [40]. In [40], the authors consider various semi-honest PSI protocols, from which their GC-PSI and NR-PSI protocols are the foundation of our work. Their Android implementation (in pure Java) takes about 1.5 s for a single oblivious AES evaluation in their GC-PSI protocol. The authors therefore conclude that instead their ECC-DH-PSI protocol is most suited for the mobile use case since the evaluation time for a single item is 23 ms. However, both of our optimized protocols with security against malicious clients are more than competitive with an evaluation time of less than 2 ms for a single item. For $N_c = 1024$ client elements, the combined base and online time of our optimized GC- and NR-PSI protocols improves by more than a factor of 300x and 30x, respectively, compared to the unoptimized semi-honest implementations of [40] in both the WiFi and the LTE network setting. Also, the total communication during the base and online phase improves by factors 7.5x and 10.9x compared to the respective GC- and NR-PSI protocols of [40].

7 Conclusion

Our native implementations of our optimized NR- and GC-PSI protocols are two almost equivalently outstanding solutions for large-scale mobile private contact discovery with security against malicious clients. The Signal developers stated that to actually deploy PSI-based contact discovery, it would need to be able to handle a server database with 1 billion users while address books are assumed to contain up to 10,000 contacts. In terms of latency, lookups are required to take less than 2 s, while in terms of throughput a single core should be able to handle 1,600 contacts per second. Clearly, we cannot meet these demanding requirements yet. Therefore, as part of future work, we suggest to implement and evaluate our proposed extensions (especially the combination with PIR) to take the next important steps towards real-world deployment.

Acknowledgments

This work was co-funded by the DFG as part of project E4 within the CRC 1119 CROSSING and project A.1 within the RTG 2050 “Privacy and Trust for Mobile Users”, by the BMBF and the HMWK within CRISP, and by the European Union’s Horizon 2020 research and innovation programme under grant agreement No 644052 (HECTOR). Daniel Kales has been supported by iov42 Ltd.

References

[1] WhatsApp Legal Info. https://www.whatsapp.com/legal, 2019.

[2] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In *EUROCRYPT*, volume 9056 of *LNCS*, pages 430–454. Springer, 2015.

[3] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More Efficient Oblivious Transfer and Extensions for Faster Secure Computation. In *CCS*, pages 535–548. ACM, 2013.

[4] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More Efficient Oblivious Transfer Extensions with Security for Malicious Adversaries. In *EUROCRYPT*, volume 9056 of *LNCS*, pages 673–701. Springer, 2015.

[5] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More Efficient Oblivious Transfer Extensions. *Journal of Cryptology*, 30(3):805–858, 2017.

[6] N. Asokan, Alexandra Dmitrienko, Marcin Nagy, Elena Reshetova, Ahmad-Reza Sadeghi, Thomas Schneider, and Stanislaus Stelle. CrowdShare: Secure Mobile Resource Sharing. In *ACNS*, volume 7954 of *LNCS*, pages 432–440. Springer, 2013.

[7] Pierre Baldi, Roberta Baronio, Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. Countering GATTACA: Efficient and Secure Testing of Fully-Sequenced Human Genomes. In *CCS*, pages 691–702. ACM, 2011.

[8] Donald Beaver. Precomputing Oblivious Transfer. In *CRYPTO*, volume 963 of *LNCS*, pages 97–109. Springer, 1995.

[9] Donald Beaver. Correlated Pseudorandomness and the Complexity of Private Computations. In *STOC*, pages 479–488. ACM, 1996.

[10] Donald Beaver, Silvio Micali, and Phillip Rogaway. The Round Complexity of Secure Protocols (Extended Abstract). In *STOC*, pages 503–513. ACM, 1990.

[11] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient Garbling from a Fixed-Key Blockcipher. In *IEEE Symposium on Security and Privacy*, pages 478–492. IEEE Computer Society, 2013.

[12] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[13] Joan Boyar and René Peralta. A New Combinational Logic Minimization Technique with Applications to Cryptology. In *Symposium on Experimental Algorithms*, volume 6049 of *LNCS*, pages 178–189. Springer, 2010.

[14] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function Secret Sharing: Improvements and Extensions. In *CCS*, pages 1292–1303. ACM, 2016.

[15] Alexander Breslow and Nuwan Jayasena. Morton Filters: Faster, Space-Efficient Cuckoo Filters via Biasing, Compression, and Decoupled Logical Sparsity. *Proceedings of the VLDB Endowment (PVLDB)*, 11(9):1041–1055, 2018.

[16] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*, pages 991–1008. USENIX Association, 2018.

[17] Henry Carter, Benjamin Mood, Patrick Traynor, and Kevin R. B. Butler. Secure Outsourced Garbled Circuit Evaluation for Mobile Devices. In *USENIX Security*, pages 289–304. USENIX Association, 2013.

[18] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. Labeled PSI from Fully Homomorphic Encryption with Malicious Security. In *CCS*, pages 1223–1237. ACM, 2018.

[19] Hao Chen, Kim Laine, and Peter Rindal. Fast Private Set Intersection from Homomorphic Encryption. In *CCS*, pages 1243–1255. ACM, 2017.

[20] Tung Chou and Claudio Orlandi. The Simplest Protocol for Oblivious Transfer. In *LATINCRYPT*, volume 9230 of *LNCS*, pages 40–58. Springer, 2015.

[21] Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. PIR-PSI: Scaling Private Contact Discovery. *PoPETs*, 2018(4):159–178, 2018.

[22] Itai Dinur, Daniel Kales, Angela Promitzer, Sebastian Ramacher, and Christian Rechberger. Linear Equivalence of Block Ciphers with Partial Non-Linear Layers: Application to LowMC. In *EUROCRYPT*, volume 11476 of *LNCS*, pages 343–372. Springer, 2019.

[23] Itai Dinur, Yunwen Liu, Willi Meier, and Qingju Wang. Optimized Interpolation Attacks on LowMC. In *ASIACRYPT*, volume 9453 of *LNCS*, pages 535–560. Springer, 2015.

[24] Christoph Dobraunig, Maria Eichlseder, and Florian Mendel. Higher-Order Cryptanalysis of LowMC. In *ICISC*, volume 9558 of *LNCS*, pages 87–101. Springer, 2015.

[25] Yevgeniy Dodis and Aleksandr Yampolskiy. A Verifiable Random Function with Short Proofs and Keys. In *PKC*, volume 3386 of *LNCS*, pages 416–431. Springer, 2005.

[26] Jack Doerner, Yashvanth Kondi, Eysa Lee, and abhi shelat. Secure Two-party Threshold ECDSA from ECDSA Assumptions. In *IEEE Symposium on Security and Privacy*, pages 980–997. IEEE Computer Society, 2018.

[27] Bin Fan, David G. Andersen, Michael Kaminsky, and Michael Mitzenmacher. Cuckoo Filter: Practically Better Than Bloom. In *Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, pages 75–88. ACM, 2014.

[28] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword Search and Oblivious Pseudorandom Functions. In *TCC*, volume 3378 of *LNCS*, pages 303–324. Springer, 2005.

[29] Chun Guo, Jonathan Katz, Xiao Wang, and Yu Yu. Efficient and Secure Multiparty Computation from Fixed-Key Block Ciphers. *IACR Cryptology ePrint Archive*, 2019:074, 2019. https://ia.cr/2019/074.

[30] Carmit Hazay and Yehuda Lindell. Efficient Protocols for Set Intersection and Pattern Matching with Security Against Malicious and Covert Adversaries. In *TCC*, volume 4948 of *LNCS*, pages 155–175. Springer, 2008.

[31] Carmit Hazay and Yehuda Lindell. Efficient Protocols for Set Intersection and Pattern Matching with Security Against Malicious and Covert Adversaries. *Journal of Cryptology*, 23(3):422–456, 2010.

[32] Wilko Henecka and Thomas Schneider. Faster secure two-party computation with less memory. In *ASIACCS*, pages 437–446. ACM, 2013.

[33] Kashmir Hill. Facebook recommended that this psychiatrist's patients friend each other. https://splinternews.com/facebook-recommended-that-this-psychiatrists-patients-f-1793861472, 2016.

[34] Yan Huang, Peter Chapman, and David Evans. Privacy-Preserving Applications on Smartphones. In *HotSec*, pages 4–4. USENIX Association, 2011.

[35] Russell Impagliazzo and Steven Rudich. Limits on the Provable Consequences of One-way Permutations. In *STOC*, pages 44–61. ACM, 1989.

[36] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending Oblivious Transfers Efficiently. In *CRYPTO*, volume 2729 of *LNCS*, pages 145–161. Springer, 2003.

[37] Stanislaw Jarecki and Xiaomin Liu. Efficient Oblivious Pseudorandom Function with Applications to Adaptive OT and Secure Computation of Set Intersection. In *TCC*, volume 5444 of *LNCS*, pages 577–594. Springer, 2009.

[38] Stanislaw Jarecki and Xiaomin Liu. Fast Secure Computation of Set Intersection. In *SCN*, volume 6280 of *LNCS*, pages 418–435. Springer, 2010.

[39] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively Secure OT Extension with Optimal Overhead. In *CRYPTO*, volume 9215 of *LNCS*, pages 724–741. Springer, 2015.

[40] Ágnes Kiss, Jian Liu, Thomas Schneider, N. Asokan, and Benny Pinkas. Private Set Intersection for Unequal Set Sizes with Mobile Applications. *PoPETs*, 2017(4):177–197, 2017.

[41] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient Batched Oblivious PRF with Applications to Private Set Intersection. In *CCS*, pages 818–829. ACM, 2016.

[42] Vladimir Kolesnikov and Thomas Schneider. Improved Garbled Circuit: Free XOR Gates and Applications. In *ICALP*, volume 5126 of *LNCS*, pages 486–498. Springer, 2008.

[43] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A Programming Framework for Secure Computation. In *IEEE Symposium on Security and Privacy*, pages 359–376. IEEE Computer Society, 2015.

[44] Moxie Marlinspike. The Difficulty Of Private Contact Discovery. https://signal.org/blog/contact-discovery, 2014.

[45] Moxie Marlinspike. Technology Preview: Private Contact Discovery for Signal. https://signal.org/blog/private-contact-discovery, 2017.

[46] Benjamin Mood, Debayan Gupta, Kevin R. B. Butler, and Joan Feigenbaum. Reuse It Or Lose It: More Efficient Secure Computation Through Reuse of Encrypted Values. In *CCS*, pages 582–596. ACM, 2014.

[47] Moni Naor and Omer Reingold. Number-Theoretic Constructions of Efficient Pseudo-Random Functions. *Journal of the ACM*, 51(2):231–262, 2004.

[48] Rasmus Pagh and Flemming Friche Rodler. Cuckoo Hashing. In *Annual European Symposium on Algorithms*, volume 2161 of *LNCS*, pages 121–133. Springer, 2001.

[49] Panagiotis Papadopoulos, Antonios A. Chariton, Elias Athanasopoulos, and Evangelos P. Markatos. Where's Wally?: How to Privately Discover your Friends on the Internet. In *ASIACCS*, pages 425–430. ACM, 2018.

[50] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A Framework for Efficient and Composable Oblivious Transfer. In *CRYPTO*, volume 5157 of *LNCS*, pages 554–571. Springer, 2008.

[51] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private Set Intersection Using Permutation-based Hashing. In *USENIX Security*, pages 515–530. USENIX Association, 2015.

[52] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure Two-Party Computation Is Practical. In *ASIACRYPT*, volume 5912 of *LNCS*, pages 250–267. Springer, 2009.

[53] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. Efficient Circuit-based PSI with Linear Communication. In *EUROCRYPT*, volume 11476 of *LNCS*, pages 122–153. Springer, 2019.

[54] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. Efficient Circuit-Based PSI via Cuckoo Hashing. In *EUROCRYPT*, volume 10822 of *LNCS*, pages 125–157. Springer, 2018.

[55] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster Private Set Intersection Based on OT Extension. In *USENIX Security*, pages 797–812. USENIX Association, 2014.

[56] Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable Private Set Intersection Based on OT Extension. *ACM Transactions on Privacy and Security*, 21(2):7:1–7:35, 2018.

[57] Michael Rabin. How to Exchange Secrets with Oblivious Transfer. In *Technical Report TR-81*. Aiken Computation Laboratory: Harvard University, 1981.

[58] Christian Rechberger, Hadi Soleimany, and Tyge Tiessen. Cryptanalysis of Low-Data Instances of Full LowMCv2. *IACR Transactions on Symmetric Cryptology*, 2018(3):163–181, 2018.

[59] Amanda Cristina Davi Resende and Diego de Freitas Aranha. Faster Unbalanced Private Set Intersection. In *FC*, LNCS. Springer, 2018.

[60] Peter Rindal. libOTe: A fast, portable, and easy to use Oblivious Transfer Library. https://github.com/osu-crypto/libOTe.

[61] Statista. Most Popular Global Mobile Messenger Apps. https://www.statista.com/statistics/258749/most-popular-global-mobile-messenger-apps, 2019.

[62] Andrew Chi-Chih Yao. How to Generate and Exchange Secrets (Extended Abstract). In *FOCS*, pages 162–167. IEEE, 1986.

[63] Samee Zahur, Mike Rosulek, and David Evans. Two Halves Make a Whole - Reducing Data Transfer in Garbled Circuits Using Half Gates. In *EUROCRYPT*, volume 9057 of *LNCS*, pages 220–250. Springer, 2015.

A Protocol Extensions

We propose further extensions for improving practicality.

Combination with FHE Protocols. Protocols for unbalanced PSI based on fully homomorphic encryption (FHE), e.g., [18], are computationally expensive and thus much slower during the online phase than our protocols (cf. §6.2). However, their advantage is that the total amount of communication is sublinear in the size of the server database. When clients install a new messaging application and are not connected to a high-speed WiFi network, such FHE-based protocols likely produce faster contact discovery results, which leads to higher user satisfaction. Thus, we recommend the following hybrid use of contact discovery protocols: Directly after installation of a mobile messaging application, a FHE-based protocol (e.g., [18]) is used to perform the initial contact discovery. Then, while the phone is charging overnight and is connected to a WiFi network, the base and setup phase of one of our protocols is performed. This leads to very efficient online phases for future protocol runs, which are performed regularly when updates on client or server side happen (potentially over mobile data plans where communication matters). See also §6.2 for a more detailed comparison between FHE-based unbalanced PSI protocols and our work.

Dedicated Server for Cuckoo Filter Membership Tests. In many scenarios, a large number of clients is part of a single organization. For example, consider the mobile malware detection scenario discussed in [40], where all applications installed on a client's smartphone are checked against a database of malicious applications. When employing such a malware detection service in an enterprise context, a company usually buys a volume license for all of its employees.

To reduce the overall data communication, the company could host a dedicated server which would receive the large encrypted database of server items represented as a compressed Cuckoo filter once. If a client then wants to compute the intersection between installed and malicious applications, it only communicates with the malware detection service provider to

perform OPRF evaluations and then hands off the encrypted items to the trusted company server, which performs the set intersection on behalf of the clients and reports back the result. Since this trusted server does not have knowledge of the PRF key, it cannot directly deduce which items the client holds.

However, since the OPRF result is deterministic when using the same secret key, the trusted server can learn when multiple clients request the same item. Furthermore, it could interact with the malware detection service provider itself to obtain encryptions of known items, which it can compare to the encrypted items of the clients. However, this kind of leakage can be argued to be acceptable in many settings, such as the company-internal setting mentioned above.

Partitioning the Database. A simple solution to reduce the required communication during the setup phase is to partition the server database s.t. clients only download Cuckoo filters relevant for the contacts in their address book (for example w.r.t. number prefixes, states, countries, or regions).

Assuming that the majority of users has contacts in only very few such partitions, this approach leads to practical data transmission sizes even for services with billions of users. In the worst case (i.e., a user has contacts in all partitions or prefers to leak no information at all), multiple runs of our protocols can cover the worldwide user base.

However, this solution presents a significant performance / privacy trade-off since clients leak information about their social graph. For example, intelligence agencies might find it suspicious if US citizens evidently have contacts in middle eastern countries. How severe the privacy of users is threatened also depends on how fine-grained the chosen partitions are: if they are too small, it might even be possible to identify an individual just by observing Cuckoo filter downloads.

B ARM Cryptography Extensions (CE)

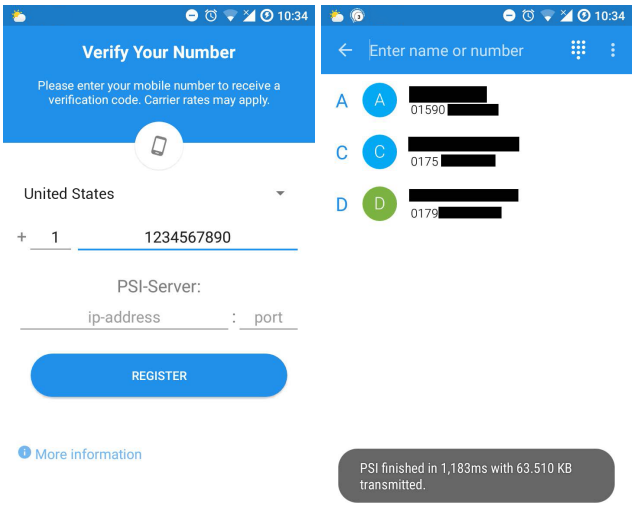
The wide availability of the ARM Cryptography Extensions (CE) in modern smartphone processors is highlighted in Tab. 7.

System-on-a-Chip (SoC)	Example Smartphones and Tablets	CE
Apple A4, A5, A6	iPhone 4, iPad, iPad 2, iPhone 5	✗
Apple A7, A8, A9	iPhone (5s,6), iPad Air, iPad mini 2	✓
Apple A10, A11, A12	iPhone (7,8,X,Xs), iPad (2018), iPad Pro	✓
Snapdragon 801	HTC One (E8), OnePlus One	✗
Snapdragon 805	Galaxy S5+, Nexus 6	✗
Snapdragon 808	Nexus 5X, LG G4, Moto X Style	✓
Snapdragon 810	OnePlus 2, Nexus 6P, Sony Xperia Z5	✓
Snapdragon 820	OnePlus 3, Galaxy S7, LG G5	✓
Snapdragon 821	Google Pixel (XL), LG G6	✓
Snapdragon 835	Google Pixel 2 (XL), Galaxy S8	✓
Snapdragon 845	OnePlus 6, Galaxy S9, Sony Xperia Z2	✓

Table 7: Availability of ARM Cryptography Extensions (CE) in modern smartphone and tablet systems-on-a-chip (SoCs).

C Signal Integration Demonstrator

As a proof-of-concept, we modified the client application of the open-source messenger Signal to perform contact discovery using our PSI protocols. To be able to run the modified client with the official servers, the integration works as follows: Whenever Signal triggers the contact discovery routine, we run one of the PSI protocols with our own PSI server[10]. The resulting matches are then used as input for the unmodified Signal contact discovery routine. This way, the official Signal server only learns the hashes of phone numbers which are already registered to the service. Our changes to the user interface of the Android version of the Signal application are depicted in Fig. 3.



(a) Signal registration. (b) Contact discovery result.

Figure 3: Screenshots of our prototype integration into the open-source messenger Signal.

D Comparison of Unbalanced PSI Protocols on the x86 Architecture

The goal of our paper is to provide efficient private contact discovery for mobile messaging applications via improved unbalanced PSI protocols with implementations optimized for smartphones. Therefore, we focus our implementation and evaluation efforts on the mobile use case and perform our experiments on real smartphones with ARMv8 architecture. However, to present the complete picture, we give a comparison to protocols for unbalanced PSI running on x86 hardware and communicating in a local network in Tab. 8.

[10] In practice, this PSI server would be run by Signal and use the actual database of Signal users.

Parameters N_s	Parameters N_c	Protocol	Online Time [s]	Online Communication [MiB]	Setup Communication / Client Storage [MiB]	Server Setup [s]
2^{28}	1,024	[59]	∗0.16	0.07	806	∗182
		[18]	∗12.10	18.57	0	∗4,628
		LowMC-GC-PSI (Ours)	0.93	24.01	1,072	1,869
		ECC-NR-PSI (Ours)	1.34	6.06	1,072	52,332
2^{24}	11,041	[59]	0.71	0.67	48	342
		[19]	44.70	23.20	0	71
		[18]	20.10	41.48	0	656
		LowMC-GC-PSI (Ours)	12.51	258.79	67	117
		ECC-NR-PSI (Ours)	11.94	65.24	67	3,298
	5,535	[59]	0.35	0.34	48	342
		[19]	40.10	20.10	0	64
		[18]	22.01	16.39	0	806
		LowMC-GC-PSI (Ours)	5.63	129.73	67	117
		ECC-NR-PSI (Ours)	5.93	32.71	67	3,298
2^{20}	11,041	[59]	0.71	0.67	3	22
		[19]	6.40	11.50	0	6.4
		[18]	4.49	14.34	0	43
		LowMC-GC-PSI (Ours)	12.51	258.79	4.2	7.3
		ECC-NR-PSI (Ours)	11.94	65.24	4.2	242
	5,535	[59]	0.35	0.34	3	22
		[19]	4.30	5.60	0	4.3
		[18]	4.23	11.50	0	43
		LowMC-GC-PSI (Ours)	5.63	129.73	4.2	7.3
		ECC-NR-PSI (Ours)	5.93	32.71	4.2	242

Table 8: Comparison of unbalanced PSI protocols in the LAN setting (10 Gbit/s, 0.02 ms RTT) on PC hardware (x86 architecture). Numbers for other protocols are taken from [18]. All numbers are from single-core executions, except those marked with ∗, which was an execution with 32 cores on the server side and 4 cores on the client side. The bit length α of all items is 128, except for [19], where $\alpha = 32$ due to limitations of the protocol.

EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats

Tahina Ramananandro* Antoine Delignat-Lavaud* Cédric Fournet* Nikhil Swamy*
Tej Chajed† Nadim Kobeissi‡ Jonathan Protzenko*
*Microsoft Research †Massachusetts Institute of Technology ‡Inria Paris

Abstract

We present EverParse, a framework for generating parsers and serializers from tag-length-value binary message format descriptions. The resulting code is verified to be safe (no overflow, no use after free), correct (parsing is the inverse of serialization) and non-malleable (each message has a unique binary representation). These guarantees underpin the security of cryptographic message authentication, and they enable testing to focus on interoperability and performance issues.

EverParse consists of two parts: LowParse, a library of parser combinators and their formal properties written in F*; and QuackyDucky, a compiler from a domain-specific language of RFC message formats down to low-level F* code that calls LowParse. While LowParse is fully verified, we do not formalize the semantics of the input language and keep QuackyDucky outside our trusted computing base. Instead, it also outputs a formal message specification, and F* automatically verifies our implementation against this specification.

EverParse yields efficient zero-copy implementations, usable both in F* and in C. We evaluate it in practice by fully implementing the message formats of the Transport Layer Security standard and its extensions (TLS 1.0–1.3, 293 datatypes) and by integrating them into MITLS, an F* implementation of TLS. We illustrate its generality by implementing the Bitcoin block and transaction formats, and the ASN.1 DER payload of PKCS #1 RSA signatures. We integrate them into C applications and measure their runtime performance, showing significant improvements over prior handwritten libraries.

1 Introduction

Because they are directly exposed to adversarial inputs, parsers are often among the most vulnerable components of security applications, and techniques to simplify their construction while ensuring their safety and correctness are valuable. Hence, developers prefer self-describing formats like JSON or XML (with universal implementations) or use automated tools and libraries to generate parsers from structured

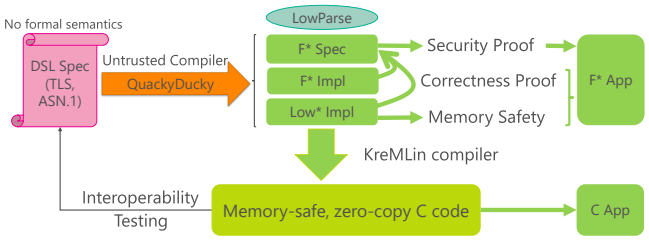


Figure 1: EverParse architecture

format specifications, or even from type declarations in Java or C++. However, when parsing is on the critical path of an application's performance, or because of requirements of the message format (such as compliance with a standard), developers may be forced to write and maintain their own parsers and serializers in low-level unsafe languages like C, increasing the risk of attacks triggered by malicious inputs.

More specifically, when the application authenticates messages in some way—using for instance cryptographic hashes, MACs, encryptions, or signatures—it is critical for security to ensure that the messages are verified, parsed, and interpreted by the receiver exactly as intended by the sender before serialization. This is often at odds with general-purpose formats and tools that may not provide such non-malleability guarantees.

This paper presents EverParse, a new framework to automatically generate efficient, low-level parsers and serializers in C from declarative descriptions of tag-length-value (TLV) binary message formats. The generated parsers and serializers are formally verified to be *safe* (no use-after-free, no buffer overruns, no integer overflows, ...), *functionally correct* (parsers and serializers are inverse of one another), and *non-malleable* (valid messages have unique representations). With EverParse, developers of low-level protocols can enjoy the ease of programming and maintenance expected from parser generators, and stop worrying about details of the message format and trade-offs between security and performance.

Architecture Overview. Figure 1 depicts the overall architecture of EverParse and its two main components: a simple, untrusted frontend (named QuackyDucky) for compiling message format descriptions; and a library of verified parsers and serializers (named LowParse).

Verification is based on F⋆ [48], a programming language and proof assistant. Whereas F⋆ is a high-level functional language, whose code extracts by default to OCaml or F#, it also embeds a language named Low⋆ for writing verified low-level code that extracts to C using a tool named KReMLin [38]. EverParse uses Low⋆ to program efficient, low-level parsers and serializers, proving them safe, correct and non-malleable in F⋆, and then extracting them to C. The resulting C code can be compiled using several off-the-shelf C compilers, including CompCert [28] for highest assurance, or more mainstream compilers like Clang or GCC.

The input of EverParse is a message format description for a collection of types, in the C-like language commonly used in RFCs and similar specifications. QuackyDucky translates this description into a collection of F⋆ modules, one for each input type, and F⋆ typechecks each of these modules to verify their safety and security. The modules produced by QuackyDucky include a formal parser specification (using high-level F⋆ datatypes) and two correct implementations of this specification: one high-level and the other in Low⋆, suitable for extraction to safe C code. This lower-level implementation enables efficient message processing; it automatically performs the same input validation as the high-level parser, but it operates in-place using interior pointers to binary representations within messages. Its performance is similar to handwritten C code—but its safety, correctness, and security are automatically verified. Hence, rather than verifying existing, ad hoc C code by hand, which would require much effort and expertise even for small protocols, our toolchain automatically yields C code verified by construction.

The code generated by QuackyDucky consists of applications of *combinators* in LowParse, which are higher-order functions on parsers. For instance, given a integer parser, one can build a parser for pairs of integers by applying the concatenation combinator to two copies of the integer parser. While parser combinators are widely used in functional languages [22, 27], they are usually more difficult to apply in languages that do not easily support higher-order programming with closures, like C. However, by employing partial evaluation within F⋆, we specialize higher-order code to efficient, ad hoc, first-order C code.

We carry out all proofs on combinators once and for all within LowParse. Only the conditions for composing them must be checked (by typing) in the code produced by QuackyDucky. LowParse is split into three layers: one for *specifications*, where we prove non-malleability, one for *high-level functional implementations*, which are proved functionally correct with respect to specifications, and one for *low-level implementations*, which operate on positions within buffers and are proved functionally correct and memory safe.

EverParse code can be used in two ways. A verified F⋆ application can use the formal specification for its security proof, and either the high-level or low-level implementation—this is the approach adopted for verifying protocols as part of the Everest project [6], and notably the MITLS [7] implementation of the TLS secure channel protocol. Alternatively, a native C/C++ application can use the interface extracted from the Low⋆ implementation by the KReMLin compiler—this is the approach taken in this paper for performance evaluation.

Our contributions We present the following contributions:

- A definition of message-format security, motivated by a discussion of several vulnerabilities whose root cause is malleability (§2).
- QuackyDucky, a compiler from tag-length-value message format descriptions to their high-level datatype specifications and low-level implementations. It provides the first implemented zero-copy and secure message format language that captures several existing protocols and standards, including TLS, PKCS #1 signature payloads, and Bitcoin (§4).
- LowParse, a library of verified parser and formatter combinators, with support for non-malleability proofs (§5).
- A qualitative evaluation of EverParse: we present a complete case study of applying EverParse to the message formats of all TLS versions, featuring many improvements and corrections over the standard's descriptions. We integrate the generated high-level implementation into MITLS, an implementation of the TLS protocol in F⋆. For a few select types, we also replace the high-level implementation with the Low⋆ one (§3).
- A quantitative evaluation of EverParse: we compare the performance of our extracted low-level parsers for Bitcoin blocks and the ASN.1 payload of PKCS #1 signatures with their counterparts in Bitcoin Core and mbedTLS. We find that our automatically generated code meets, and in some cases significantly exceeds, the performance of hand-written C/C++ reference code (§6).

All the components of EverParse and its dependencies are open-source and publicly available at `https://github.com/project-everest/everparse`

2 Parsing Security: Definitions & Attacks

In this paper, we focus on applications that authenticate the contents of serialized messages in some way. Cryptographic mechanisms provide (serialized) bytestring authentication, whereas applications rely on (parsed) message authentication. Hence, correctness and runtime safety are not sufficient to preserve authentication: a correct parser may accept inputs outside the range of the serializer, or multiple serializations of the same message, which may lead to subtle, and sometimes

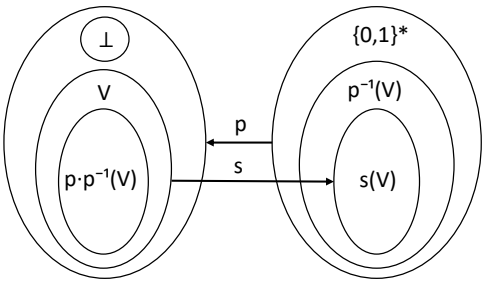


Figure 2: Parsing and serialization functions

devastating, vulnerabilities. We propose a security definition for authenticated message formats to prevent such vulnerabilities, and illustrate it using known high-impact attacks against popular applications.

2.1 What is a Secure Message Format?

We first set up notations for parsers and serializers, illustrated in Figure 2, and define their properties of interest. Given a set $\mathcal{V}$ of valid messages,

- a *parser* is a function $p : \{0,1\}^* \to \mathcal{V} \uplus \{\bot\}$ that returns either a message $m \in \mathcal{V}$ or a parsing error $\bot$;
- a *serializer*, or *formatter*, is a function $s : \mathcal{V} \to \{0,1\}^*$.

Informally, parsers and formatters are inverse of one another. A parser is *correct* with respect to a serializer when it yields back any formatted message: $\forall m \in \mathcal{V}, p(s(m)) = m$, and *exact* when it accepts only serialized messages: $p^{-1}(\mathcal{V}) = s(\mathcal{V})$.

Parsers may also be considered on their own. A parser is *non-malleable* (or *injective*) when it accepts at most one binary representation of each message: $\forall x, y \in \{0,1\}^*, p(x) = p(y) \Rightarrow (x = y \vee p(x) = \bot)$, and *complete* (or *surjective*) when it accepts at least one binary representation of each message: $p(\{0,1\}^*) \setminus \{\bot\} = \mathcal{V}$. If p is a non-malleable parser for $\mathcal{V}$, then p^{-1} is a serializer over $p(\{0,1\}^*) \setminus \{\bot\}$.

We say that p is a *secure parser* for $\mathcal{V}$ if p is non-malleable and complete. If p is secure, then it is also correct and exact with respect to the (unique) serializer p^{-1}. We say a serializer s is secure if there exists a secure parser p such that $s = p^{-1}$. (§5 provides more general definitions that account for parsers that do not consume their whole input.)

In the rest of the paper, we only consider parsers that operate on strings of bytes $\mathcal{B} = \{0,1\}^8$.

2.2 Attacks on Parsers

Heartbleed Unsurprisingly, the most common type of parser vulnerability is simply memory safety bugs. Indeed, one of the most impactful attacks in the past decade, Heartbleed (which is estimated to have affected up to 55% of the top internet websites [17]) is a simple buffer overrun caused by improper validation of the length field in the TLS messages defined in OpenSSL's implementation of the heartbeat protocol extension (shown in Figure 3). Interestingly, the specification of `HeartbeatMessage` is very unusual among TLS types (explained in detail in §3), because it contains a variable length field (`padding`) that is not prefixed by an explicit length (`padding_length` is not defined in the struct, but its value is defined semantically). Indeed, as specified, this type is not expressible in QuackyDucky because the padding length depends on a field of the parent `TLSPlaintext` type, and we only capture dependencies between fields that are concatenated. This forces applications to verify the `padding_length` semantically, increasing the risk of error. The Heartbleed disaster would likely have been averted if the format was specified using standard TLS constructors for variable length fields:

```
struct {
  HeartbeatMessageType type;
  uint16 payload_length;
  opaque payload[payload_length];
  opaque padding[padding_length];
} HeartbeatMessage;
The total length of a HeartbeatMessage MUST NOT exceed
2^14 or max_fragment_length when negotiated [RFC6066].
The padding is random content that MUST be ignored by
the receiver. The padding_length MUST be at least 16,
and equal to TLSPlaintext.length-payload_length-3 for
TLS and DTLSPlaintext.length-payload_length-3 for DTLS
The sender of a HeartbeatMessage MUST use a random
padding of at least 16 bytes. The padding of a
received HeartbeatMessage message MUST be ignored.
```

Figure 3: Specification of the Heartbeat message (fragment)

```
struct {
  HeartbeatMessageType type;
  opaque payload<0..2^14-21>;
  opaque padding<16..2^14-3>;
} HeartbeatMessage;
```

This example illustrates the benefits of writing message format descriptions in a constrained language: it encourages uniform patterns and enables automated analysis.

PKCS #1 signature forgery The PKCS #1 v1.5 signature format illustrates the risks of applying message parsing after a cryptographic operation (in this case, modular exponentiation). Given a public key $(N = pq, e)$ where p and q are large secret primes, the raw RSA signature σ over a message m is computed as $\sigma = m^d \bmod N$ where the secret exponent $d = e^{-1} \bmod (p-1)(q-1)$ is hard to compute without knowing p and q. As written, this scheme is not usable because m must be smaller than N, and it has the undesirable homomorphic property that the signature of the product of 2 messages is equal (modulo N) to the product of the signature of each message (thus, it is easy to forge new valid signatures from existing ones). To fix these shortcomings, PKCS #1 v1.5 defines a standard for hashing and padding the message to sign: given an arbitrary message m, it is first

hashed into a digest h, then stored together with the identifier a of the hash algorithm into an ASN.1 DER [36] structure. The distinguished encoding rules are supposed to ensure that the serializer ρ for this structure is secure. The final signature is obtained by applying raw RSA to $\rho(a,h)$ left-padded to the size of N with padding of the form `\x00\x01(\xFF)*\x00`.

The security of the scheme relies heavily on (the integer interpretation of) the padding: it is computationally hard to forge a valid signature σ because $\sigma^e \bmod N$ must be of the form $2^{\lceil \log_2(N) \rceil - 15} - 2^{\lceil \log_2(\rho(a,h)) \rceil + 1} + \rho(a,h)$ for some digest h and hash identifier a. It is hard to find such a value by brute force because all but the $\lceil \log_2(\rho(a,h)) \rceil$ last bits are fixed, and inverting the modular exponentiation by e is hard without knowing d. However, if the ASN.1 parser π used after exponentiation is malleable (or non exact), there may exist a large class of inputs x such that $\pi(x) = (a,h)$. If this class contains inputs that fill most of the $\lceil \log_2(N) \rceil$ bits of the message, the padding may be reduced to `\x00\x01\x00`. When e is small ($e = 3$ is commonly used by legacy public keys), it may be easy to find a value σ such that $\sigma^e \bmod N = 2^{\lceil \log_2(N) \rceil - 15} + x$ with $\pi(x) = (a,h)$ for any h. For instance, in Bleichenbacher's original description of the attack (retold by Finney [18]), the parser ignores the bytes that appear after the encoded ASN.1 structure, i.e. if $\pi(x) \neq \bot$, $\pi(x||z) = \pi(x)$ for all z. To forge a valid signature for h, one can simply compute the cubic root of $2^{\lceil \log_2(N) \rceil - \lceil \log_2(\rho(a,h)) \rceil - 16}(\texttt{\x0100}||\rho(a,h))$.

Ever since its publication, this attack has reappeared in dozens of implementations, including several recent examples (e.g. [9, 13, 37, 49]). Interestingly, the parser malleability bugs that cause the attack are diverse: unparsed extra bytes are tolerated at the end of the message [18, 49]; the parser accepts arbitrary parameters in the algorithm identifier [9]; and a length overflow causes only its last 4 bytes to be counted [13]. This diversity illustrates how difficult it is to write secure parsers and to detect malleability vulnerabilities—some of the attacks above have existed for years in popular libraries. All variants lead to universal signature forgery: an attacker can freely impersonate any client or server, sign malicious code updates, etc.

Bitcoin transaction malleability Another well-documented case of parser security attacks occurred against Bitcoin [34] transactions, which are signed by the sender, then hashed (after serialization) and stored in Merkle trees. Transactions are identified by their hash, which covers more data than what the sender signs (in particular, the hash includes the signature itself). The format of transactions is malleable in several ways: one example is the encoding of this signature, which originally did not mandate the ASN.1 DER rules for non-malleability. Another source comes from the ECDSA signing algorithm, which is a randomized scheme (hence, there are many valid signatures for the same message) that always has two valid representations: if (r,s) is a valid signature, then $(r,-s)$ also is, and can be trivially computed without knowledge of the private key. Other sources of malleability are related to the `scriptSig` construct of the Bitcoin Script language,[1] inasmuch as the signature is passed to a stack-based script to authorize spending. In total, BIP62 [50] lists 9 different sources of malleability. Each of them allows an attacker to alter a valid transaction t into a semantically-equivalent valid transaction t' such that $h(t) \neq h(t')$. One way to exploit this is to try to fool someone into believing that a transaction they submitted was rejected by the network, although in reality, it was accepted under a different transaction hash. The Mt. Gox bitcoin exchange blamed this attack for the loss of over 850,000 bitcoins (worth $473M at the time of bankruptcy) and although this claim is heavily disputed, later forensic examination of the blockchain by Decker et al. [12] revealed that in total, 300,000 bitcoins were spent over 30,000 transactions confirmed under a different identifier than originally submitted between Feb 1, 2014 and Feb 28, 2014.

Ambiguous TLS message Sometimes, the message specifications themselves are ambiguous, and cannot be implemented by a secure parser. This is the case of the `ServerKeyExchange` message in TLS:

```
enum {dh_anon, dhe, ecdhe, rsa,(255)} KeyExchange;
struct {
  select (KeyExchange) {
    case dh_anon: DHAnonServerKeyExchange;
    case dhe: SignedDHKeyExchange;
    case ecdhe: SignedECDHKeyExchange;
    case rsa: Fail; /* Force error: no SKE in RSA */
  } key_exchange;
} ServerKeyExchange;
```

This message represents an untagged union: the struct is missing a field of type `KeyExchange` that clarifies which case to use in the union. A parser for an untagged union can only be secure if the format of all cases share no common prefix. The specification of TLS assumes that the key exchange algorithm is available from the context (in this case, it is part of the negotiation process). However, it turns out that the security of the TLS negotiation depends itself on the `ServerKeyExchange` message. This leads to a real practical attack reported by Mavrogiannopoulos et al. [32], where a `SignedDHKeyExchange` is interpreted as a bogus `SignedECDHKeyExchange`. Worryingly, two other TLS types use untagged unions: `ClientKeyExchange` (in TLS 1.2) and `CertificateEntry` (in TLS 1.3).

3 Case Study: the TLS Message Format

We choose the TLS message format as our main case study for several reasons: the message format description of TLS is reasonably specified; it is designed to be secure and extensible; it defines hundreds of types that exercise the full range

[1] https://en.bitcoin.it/wiki/Script

```
uint32 word; /* Type declaration */
word digest[16]; /* Fixed-length array of 4 words */
word phrase<0..2^8-1>; /* List of 0 to 16 words */
struct {
  opaque id[32]; /* Array of 32 bytes */
  uint16 payload<2..8>; /* List of 1 to 4 uint16 */
  digest payload_digest;
} body; /* Struct with 3 fields */
enum {
  request (0x2300), /* Constant tag */
  response (0x2301),
  (65535) /* Indicates 16 bit representation */
} header; /* Enum with 2 defined cases */
struct {
  header tag;
  select(tag) { /* Tagged union */
    case request: body;
    case response: phrase;
  } x; /* Enum-dependent field type */
} message;
struct {
  uint24 len; /* Explicit length */
  message data[len]; /* Ensures length(data)=len */
} batch; /* Length encapsulation */
```

Figure 4: Sample type descriptions in TLS message format.

of available combinators in LowParse; and there exists a verified F* implementation of TLS that we can use to test the integration of the generated parsers (including the integration of parser security lemmas into the protocol security proof).

Language Description IETF's RFC 2246 [14] specification of TLS 1.0, published in 1999, includes a section that describes the presentation language of its message format, inspired by C and XDR [46], and illustrated in Figure 4. A description consists of a sequence of type declarations. The base types are unsigned fixed-length integers `uint8`, `uint16`, `uint24`, and `uint32`, with `opaque` being used instead of `uint8` to indicate raw bytes. The type constructors are fixed-length arrays, variable-length lists, structs, enums, and tagged unions. The length boundaries of arrays and lists are all counted in bytes rather than in elements: for instance, type `digest` in Figure 4 is an array of elements of type `word` whose binary representation takes 16 bytes in total; since each `word` takes 4 bytes, this array holds exactly 4 elements. Arrays can be constructed only from fixed-length types, whereas lists can be defined for any types: as illustrated by `answer` and `payload`, their format declares the range of their length; and their binary representation embeds their actual length within that range.

Following the convention of RFCs, we interpret types in terms of the byte sequences that represent their elements. The representation of a struct is the concatenation of the representations of each of its fields in sequence, without any padding. Arrays are the concatenation of elements whose total length in bytes is the array's annotated size. Lists are represented by a length field encoded in a fixed number of bytes (determined by the maximum length of the list, encoded in big endian), followed by a concatenation of the elements. A special case of structs are length-dependent fields, e.g., the `batch` type in Figure 4. In these types the first field describes the length of a single (variable-length) element of the specified type of the second field represented adjacently. The interpretation of enumerations contains the big endian encodings of its elements in a constant number of bytes determined by the size descriptor of the enum type. Tagged unions (like `message` in the figure) are encoded as the concatenation of the tag's enum representation followed by the encoding of the corresponding case's type. TLS messages are more compact than many TLV formats: explicit tags only appear for tagged unions, and lengths only for lists (or when ascribed). All structural information is erased, in contrast with BSON [33] (which encodes field names) or Protocol Buffers [20] (which encodes field numbers).

We automatically extracted the data format descriptions from the RFCs for TLS 1.2 [15] (including descriptions also for TLS 1.0 and TLS 1.1), for TLS 1.3 [40], for TLS extensions (RFC 6066), and from the TLS IANA parameter assignments, which defines additional constants for enumerations. We then merged them together by hand, and edited some of them to fix minor mistakes, avoid name clashes in the original descriptions, and gain precision (e.g. by adding length dependencies documented in the RFC text).

Extensibility A difficult issue for any format description languages is extensibility: as new versions of the protocol are defined, it is often necessary to extend messages with new fields and cases while maintaining compatibility with older implementations. To address this problem, TLS was designed with extensibility through open enumerations. As a simple example, TLS has an enum type that defines the possible cipher suites to negotiate. Receiving a value that doesn't match any of the defined cases of the enum is not a parsing error—instead, the value should be treated as unknown but valid, and the receiver should ignore it in the rest of the negotiation. This also applies to enums that act as a tag for unions. For instance, the hello messages contain a list of extensions tagged with an extension type. Although this is implicit in the standard, it is possible to define a default type for unknown values in a `select`. The protocol is extended by defining new values for enums (such as new cipher suites or new group names), and new defined cases for tagged unions (for instance, new extensions). Interestingly, many TLS implementations fail to understand this concept, and incorrectly reject unknown values. To fight this problem, Google recently introduced GREASE [4], which causes Chrome to randomly include undefined values in all extensible fields of the protocol, thereby enforcing that implementations that interoperate with Chrome be extensible.

Unfortunately, the TLS standard does not clearly say which

```
/* All TLS versions*/
struct {
  ExtensionType extension_type;
  opaque extension_data<0..2^16-1>;
 } Extension;

/* From RFC 5246, section 7.4.2 */
opaque ASN.1Cert<1..2^24-1>;
struct {
  ASN.1Cert certificate_list<0..2^24-1>;
} Certificate;

/* From RFC 8446, section 4.4.2 */
enum { X509(0), RawPublicKey(2), (255)} CertType;
opaque ASN1_subjectPublicKeyInfo<1..2^24-1>;
struct {
  select (certificate_type) {
    case RawPublicKey: ASN1_subjectPublicKeyInfo;
    case X509: ASN.1Cert; } cert;
  Extension extensions<0..2^16-1>;
} CertificateEntry;
struct {
  opaque certificate_request_context<0..2^8-1>;
  CertificateEntry certificate_list<0..2^24-1>;
} Certificate;
```

Figure 5: The `Certificate` type for TLS 1.2 vs TLS 1.3.

enums and which tagged unions are extensible in the message format, and instead explain their intended semantics in text. In QuackyDucky, we add an explicit annotation to mark which enums are extensible, and we extend the syntax of `select` to support `default` cases. For instance, we mark the tag of extensions `ExtensionType` with the `/*@open*/` attribute, but not the tag of messages `HandshakeType`, as the RFC states that receiving any unknown message is an error.

Protocol versions Another complication stems from version differences not captured by extensibility. Consider the Certificate message in TLS 1.2 [15] and TLS 1.3 [40]. Its format, listed in Figure 5, illustrates several problems with the TLS specification. First, the two definitions of the `Certificate` type are mutually incompatible, even though the `Certificate` message is defined in both versions using the same `handshakeType` tag. Second, the `CertificateEntry` type of TLS 1.3 uses an untagged union, where the value of the tag depends on the context rather than on a value sent over the network (as in `ServerKeyExchange` and `ClientKeyExchange` where the key exchange algorithm is omitted). Third, the `Extension` type is under-specified: there are complex rules and tables about which extension may appear in which message (see [40, §4.2]), and the type of each extension contents depends on which message it appears in. None of these constraints are currently captured in type definitions.

To address the issue of conflicting definitions across versions, we split the definitions of incompatible types such as

TLS 1.2	TLS 1.3

`struct {`
`  HandshakeType type;`
`  uint24 length;`
`  select (type) {`
`  case hello_request:`
`  Empty;`
`  case certificate:`
`  Certificate12;`
`  /* ... */`
`  case finished:`
`  Finished;`
`  } m[length];`
`} Handshake12;`
	`struct {`
	`  HandshakeType type;`
	`  uint24 length;`
	`  select (msg_type) {`
	`  case eoed:`
	`  Empty;`
	`  case certificate:`
	`  Certificate13;`
	`  /* ... */`
	`  case key_update:`
	`  KeyUpdate;`
	`  } m[length];`
	`} Handshake13;`

Figure 6: Specialized `Handshake` types for TLS 1.2 and 1.3

`Certificate` and we define version-specific variants of the handshake message type, shown in Figure 6. Before version negotiation, hello messages are parsed using a third, version-agnostic `Handshake` type. We then switch to parsers for the negotiated version, thus ensuring that the following messages are parsed using precise, version-specific types.

To address the problem of untagged unions, we introduce an `/*@implicit*/` attribute for tags, which instructs QuackyDucky to generate an interface where the value of the tag is passed as an additional argument to the parser and formatter. This approach is not compositional, and comes with a restriction: if a type that contains an implicit tag appears in another type, it must appear at a location that includes an explicit length. The parsing will be *staged*: when it reaches the surrounding length, it skips its contents, leaving it uninterpreted. The application needs to manually call the parser for the untagged union by providing the tag value. The presence of untagged unions is a clear mistake in message formats, as the application is responsible for providing the correct tag when it calls the parser, and thus, we only provide a conditional security guarantee in this case.

Similarly to what we did with message types, we split and specialize the definition of extension types for each message that may include them (hellos, hello retry, encrypted extensions, certificate, certificate request, new session ticket). This also reveals some interesting mistakes in RFCs. For instance, in [41, Appendix A], the authors fail to understand the purpose of the explicit length around extensions, and incorrectly believe it is redundant with the length of the list in the extension contents. They claim merging the two lengths makes the extension less ambiguous; in reality, their change makes the format *more ambiguous*: it is no longer possible to distinguish between receiving no ticket and receiving an empty ticket. To handle such corner cases, we add support in QuackyDucky to coerce `opaque` arrays with explicit lengths to `opaque` lists after parsing.

4 Compiling Message Format Descriptions

We now present our compiler from the message formats of §3 to parsers and serializers for processing these messages in Low⋆ [38]. We briefly review F⋆ and Low⋆ (§4.1), then explain the code generated by QuackyDucky in three parts: datatypes and parser specifications for verification purposes (§4.2); high-level functional parsers and serializers (§4.3); and lower-lever code for reading (§4.4) and writing (§4.5) messages.

QuackyDucky recursively descends through the structure of the format description, generating parsers and serializers for compound types from those previously generated, while keeping track of their properties (notably their length boundaries). QuackyDucky mostly composes the combinators provided by LowParse, described in §5. In contrast with this library, which involves complex proofs for a few generic combinators, the generated code is verbose but shallow, enabling us to automatically verify its safety, correctness, and non-malleability using the properties verified in LowParse.

4.1 Verified Programming in Low⋆ (Review)

The F⋆ language and proof assistant We carry out our specification, implementation and proofs using F⋆ [48], a functional language and proof assistant based on dependent types. A simple form of dependent types supported in F⋆ is *refinement types*, to represent types of values satisfying additional properties: whereas int is the F⋆ type of mathematical integers, the type nat of non-negative mathematical integers is defined as the refinement type (x: int { x ≥ 0 }). F⋆ supports types that depend on other types and values. It also supports functions where argument types can depend on the values of the previous arguments and the type of the return value of a function can depend on the values of its arguments. For instance, the integer division function, which would have a simple type int → int → int in a functional language such as OCaml, can be given a more precise type, such as (a: int) → (b: int {b > 0}) → (q: int { 0 ≤ a − b ∗ q < b }) meaning that F⋆ will reject, at typechecking time, any call to such a function if it cannot prove the second argument is strictly positive (b > 0). Conversely, the caller can use the postcondition on its return value 0 ≤ a − b ∗ q < b to prove further goals.

F⋆ is not just a proof assistant: it is also a programming language enjoying automatic translation (*extraction*) into executable languages such as OCaml or C; in this extraction process, the user can mark some of their F⋆ functions as *ghost*, to have them erased at extraction time, such as lemmas and proofs, but also auxiliary functions that need not, or cannot, be executed at run time. To this end, F⋆ equips its function types with an *effect* system. By default, given two types t and u, all functions from type t to u, in the type t → u, will be extracted, one says that they are in the Tot effect, so their type can be written as t → Tot u; to mark a function ghost, the user needs to use the GTot effect, hence using the function type t → GTot u instead. Stateful functions that operate on mutable objects (such as a mutable array of bytes, i.e. `uint8_t*` in C) live in the ST effect and are extracted to stateful OCaml or C code.

All F⋆ proofs rely on automatic encoding of proof obligations to first-order logic, which are then solved by automatic theorem provers such as Z3, leading to reduced overall proof effort. F⋆ also has a tactic system [31] that can be used in cooperation with, or in replacement of, Z3-based proofs.

The Low⋆ subset of F⋆ and its extraction to C code Users who wish to use our parsers and serializers not only demand performance, but also the ability to integrate our code into their codebases. Therefore, OCaml is oftentimes not a viable option; instead, we compile our F⋆ code to C.

As shown in Figure 1, specifications are first implemented in F⋆. This implementation can already be compiled to C using a dedicated compiler, KReMLin. However, lacking further restrictions, this F⋆ implementation relies on lists and bytes as values, meaning that compiling the F⋆ implementation through KReMLin generates code that (i) is inefficient, because it uses functional byte copies and linked lists; (ii) is not idiomatic: no C programmer would write such code; and (iii) requires a garbage collector, since lists and bytes are persistent values with no lifetime (as in, say, OCaml).

To avoid these shortcomings, the Low⋆ [38] subset of F⋆ defines a C-like memory model that talks about the stack and the heap, along with corresponding abstractions and libraries, e.g. for mutable arrays, machine integers and in-place loops. Using Low⋆ requires the programmer to rewrite their code and reason about spatial and temporal safety; in exchange for these added constraints, the F⋆ code is compiled by KReMLin to idiomatic, readable, efficient C code that does not require a garbage collector. The Low⋆ restrictions only apply to executable code; computationally-irrelevant portions of the program, such as proofs and ghost code, retain the full power of F⋆, since they are eliminated when compiled to C.

4.2 Datatypes and Parser Specifications

Continuing from §2.1, we use F⋆ types to represent sets of parsed values. Hence, QuackyDucky generates a parsed type *t* for each named format description in its input.

Our F⋆ types for parser and serializer specifications are listed below.

```
type parser (t:Type) (k:meta) =
  p: (input: seq uint8 → GTot (option (t * l: nat{l ≤ length input})))
  { parser_prop k p }

type serializer (#t:Type) (#k:meta) (p: parser t k) =
  s: (t → GTot (seq uint8))
  { ∀ (x:t) . p (s v) == Some (v, length (s v)) }
```

The parser type definition is parameterized by a parsed type *t* and some metadata *k* (explained below) that records verified properties of the parser. It states that a parser is a pure function

that takes as input a sequence of bytes (of arbitrary length) and returns an optional result that consists of some parsed value of type t and the number of input bytes consumed. GTot states that this parser is ghost, that is, used only for verification; option indicates that it may return nothing in case parsing fails; the refinement $l \leq$ length input ensures that it consumes at most a prefix of its input; the refinement parser_prop k p states that p is *non-malleable* (see formal definition in §5) and that it satisfies the properties recorded in k.

The metadata k includes a verified range of lengths of input that the parser may consume. This range provides useful bounds for programming with this parser, for instance to wait for a minimal number of input bytes before parsing, or to allocate I/O buffers of adequate sizes. Internally, QuackyDucky also relies on this information to select more efficient implementations (for example when the input length is fixed) and to require that some parsers consume at least one byte (for example to compute the length of a list from its binary format). The metadata may include additional properties, indicating for instance that the parser always fails; or that it always succeeds given enough input bytes; or that it is no-lookahead (see §5).

The serializer type definition is indexed by a *parser* for t, not just by t (the # makes this parameter an implicit argument, as it can be inferred from p). It states that a serializer specification is a pure, total, ghost function from values of type t to sequences of bytes, such that parsing its output for any value v of type t succeeds and yields back v and its binary length.

Running Example Consider the following excerpt of the TLS 1.3 wire format description [40] for the body of its first ClientHello message.

```
struct {
  ProtocolVersion version;
  opaque random[32];
  opaque session_id<0..32>;
  CipherSuite cipher_suites<2..2^16-2>;
  Compression compressions<1..2^8-1>;
  ClientHelloExtension extensions<0..2^16-1>;
} ClientHello;
```

This message includes a 2-byte protocol version, a 32-byte random nonce, a variable-length session identifier for resumption, and variable-length list of proposed cipher suites, compression methods, and extensions. QuackyDucky translates it to a corresponding F* record type:

```
let in_range x min max = min ≤ x ∧ x ≤ max
type clientHello = {
  version : protocolVersion;
  random : (b:bytes {Bytes.length b == 32});
  session_id : (b:bytes{in_range (Bytes.length b) 0 32});
  cipher_suites : (l:list cipherSuite{in_range (List.length x) 1 32767});
  compressions : (l: list compression{in_range (List.length l) 1 255});
  extensions : (l: list extension{in_range (extensions_bytesize l) 0 65535});}
```

This high-level type includes precise length information, coded as refinements. Since the elements of the first three lists have constant binary lengths, QuackyDucky computes precise bounds on their numbers of elements. Conversely, the extensions in the last list are themselves of variable lengths, hence QuackyDucky captures the bounds on its total binary size using an auxiliary function extension_list_bytesize previously defined from the extension serializer. The main benefit of capturing these constraints is to ensure that *all* messages of type clientHello can be serialized to a standard-compliant bytestring.

QuackyDucky also generates the corresponding metadata, parser and serializer specifications.

```
let clientHello_meta = {low=43;high=131396; ...}
val clientHello_parser: parser clientHello_meta clientHello
val clientHello_serializer: serializer clientHello_parser
```

The parser metadata is exposed in the generated interface (indicating, e.g., that the shortest TLS clientHello body message takes 43 bytes) whereas the parser and serializer specs are kept abstract—the interface gives their types, but hides the details of their wire format. Thus, the three lines above state the abstract, joint properties of our generated parser and serializer specs (including non-malleability and round-trip properties) and typechecking these specs ensures these properties hold.

Anticipating on the combinators defined in §5, we give below an outline of the generated definition of clientHello_parser, which parses our sample message by successively calling the parsers corresponding to each of its fields:

```
let clientHello_parser =
  ((protocolVersion_parser × clientHello_random_parser) × ...)
    synth (fun ((pv, r), ...) → { protocolVersion = pv; random = r; ... })
```

These definitions are used only as reference implementations and are not extracted to C. In a second stage, QuackyDucky generates actual parsers and serializers, and typechecking their code ensures they safely implement these specs.

4.3 Functional Parsers and Serializers

QuackyDucky generates high-level functional parsers and serializers, with the following interface. (The "32" suffix indicates that their code uses 32-bit machine integers instead of the unbounded integers in their specs.)

```
val clientHello_parser32: parser32 clientHello_parser
val clientHello_serializer32: serializer32 clientHello_serializer
```

We systematically index our implementations by their specifications. Here, for instance, the type definition parser32 p used above simply states that a high-level parser for the parser-specification p is a pure function that takes a (bounded, immutable) F* bytestring and returns the same result as p on the corresponding sequence of bytes except that the consumed length is an unsigned 32-bit integer.

We give below an F* code sample illustrating their use: a 'reader' function that counts the number of cipher suites in a given ClientHello message in wire format, and a 'writer' function that builds a message given a configuration.

```
let count_ciphersuites (input: bytes): UInt32.t =
  match clientHello_parser32 input with
  | None → 0ul
  | Some ch → List.length ch.ciphersuites

let compute_extensions config: (l: list extension {...} ) = ...

let hello (cfg: config) : bytes =
  clientHello_serializer32 {
    version = TLS_1p3;
    random = config.random;
    ...;
    extensions = compute_extensions config; }
```

This code and its supporting parsers and formatters operate on immutable bytestrings. Although it can be safely extracted to C, it is inefficient, and the implicit allocations and copies mandate the use of a garbage-collector. For example, clientHello_parser32 allocates 4 lists and briefly uses only one.

4.4 Low-Level Accessors and Readers

To provide more efficient implementations, QuackyDucky also generates code for a lower-level API that enables in-place processing of messages in their binary formats.

We begin with a low-level alternative to parsing. For each parser specification p, QuackyDucky provides functions that operate on an input buffer. A *validator* reads the input buffer and returns either the number of bytes that p would consume by successfully parsing its contents, or an error code. Thus, successful validation ensures the existence of a high-level message in binary format, but does not construct it. Assuming the input buffer is valid, then, for each field of the message, an *accessor* computes the position of the first byte in its binary representation. This guarantees in particular that this representation of this element of the message is also valid. Accessor computations are similar to pointer arithmetic in C, or "get element pointer" computations in LLVM, but they sometimes require reading the lengths of intermediate parts in order to skip them. For each base type (e.g. 16-byte unsigned integers), a *reader* takes an input buffer and position and actually parses and returns a value of that type.

Continuing with our example, QuackyDucky produces a validator for clientHello and an accessor for every field (shown below only for its cipherSuites).

```
val clientHello_validator : validator clientHello_parser
val accessor_clientHello_cipherSuites :
  accessor
    clientHello_parser
    clientHello_cipherSuites
    clientHello_cipherSuites_parser
```

The type definitions validator and accessor are still parameterized by parser specifications, but they are more complex, since they describe functions that operate on pointers to mutable buffers. We represent their input as a *slice*, that is, a Low* buffer (§4.1) and its length, and a *position* within this slice. (Experimentally, computations on integer positions based on a single pointer are simpler to verify, and better optimized by C compilers.) Accordingly, our validators return either the final position in the slice after successful validation, or an error coded as a large integer. We illustrate their use by re-implementing the count_cipherSuites example of §4.3 in Low*.

```
let count_ciphersuites_inplace (input:slice) (pos:UInt32.t) =
  let pos_final = clientHello_validator input pos in
  if max_length < pos_final then 0ul (* invalid input *)
  else
    let pos_ciphersuites = accessor_clientHello_ciphersuites input pos in
    clientHello_cipherSuites_count input pos_ciphersuites
```

The last line calls another QuackyDucky-generated function that returns the length of a list of cipher suites in wire format. In this case, since each cipher suite takes exactly two bytes, this length is computed without actually reading the list content, by dividing its binary length by two.

Unsurprisingly, this function yields C code of the form:

```
// A slice is the pair of a byte array and its size
typedef struct {uint8_t * base; uint32_t len; } slice;
uint32 count_ciphersuites_inplace(slice input, uint32 pos) {
  uint32 pos_final = clientHello_validator(input,pos);
  if (max_length < pos_final) return 0;
  else {
    uint32 pos_ciphersuites = accessor_clientHello_ciphersuites(input,pos);
    return clientHello_cipherSuites_count(input,pos_ciphersuites);
  }
```

Once compiled by Clang, we can check on the resulting machine code that the 'else' branch eventually boils down to (1) adding 34 to pos to skip the first two fields; (2) reading and adding the one-byte length of the third field; (3) reading the two-byte length of the fourth field and shifting it by one.

Validators, accessors, jumpers and readers are specified using a *validity* predicate, $\mathsf{valid}(p,m,b,i)$ stating that the parser p succeeds when provided the bytes in buffer b starting from offset i in memory state m. If this predicate holds, then, thanks to the injectivity property of p, there is a unique value $\mathsf{contents}(p,m,b,i)$ returned by the parser, and an offset $\mathsf{getpos}(p,m,b,i)$ within b one past the end of the representation of that value. This validity predicate is the post-condition of validators when they succeed, and is the precondition of jumpers and readers; accessors for struct fields have the validity predicate for the struct (resp. field) parser as a precondition (resp. postcondition). We give below the type definitions for validators and readers:

```
type validator (#k: meta) (#t: Type) (p: parser k t) =
  (input: slice) → (pos: U32.t) → ST U32.t
  (requires (fun m → live_slice input pos ∧ input.len ≤ max_length))
  (ensures (fun m pos' m' → m ≡ m' ∧
    (valid(p, m, input, pos) ⇔ pos' ≤ max_length) ∧
    (pos' ≤ max_length ⇒ pos' == getpos(p, m, input, pos))))

type reader (#k: meta) (#t: Type) (p: parser k t) =
  (input: slice) → (pos: U32.t) → ST t
  (requires (fun m → valid(p, m, input, pos)))
  (ensures (fun m res m' → m ≡ m' ∧ res == contents(p, m, input, pos)))
```

4.5 Low-Level Writers

We finally describe our low-level API for serializing a message in an output buffer. Our goal is to avoid intermediate allocations and copies; for example, our high-level hello function constructs the whole message before serializing. To this end, QuackyDucky generates families of low-level writers and auxiliary functions that take as parameter an output slice (that is, a buffer and a length) and a write position, modify the buffer between this position and the end of the buffer, and return a new write position. These functions either require that the output buffer is large enough (based on parser metadata) or may also return an error in case the buffer is too small.

In contrast to accessors, which enable random access to validated input in binary format, it is not generally possible to know in advance where to write data before writing any preceding variable-length data. Thus, our API assumes that data will be written sequentially, with the flexibility for the programmer to use an intermediate buffer whenever they choose to write data out of order. A notable exception is for encoding the lengths of variable-length data, which is usually known only after writing the raw data itself. To this end, QuackyDucky provides a *finalizer* that takes two positions in the output buffer, requires as a precondition that the buffer contain a placeholder for the length followed by a valid binary representation of the raw data, computes and writes its length, and ensures as a post-condition that the buffer now contains a valid variable-length representation of this data.

We illustrate these different cases on a low-level variant of the hello function, whose Low⋆ and extracted C code are shown in Figure 7. The first field, version, is an enumeration formatted in a fixed two-byte format: it is directly written using the QuackyDucky writer for protocolVersions. The second field, random, is a fixed-length, previously-allocated bytestring that can be copied from the configuration. Omitting intermediate fields, which may be handled similarly, the last field is a complex list of extensions. The list itself is written by repeatedly calling the extension writer on each element, after skipping the 2 bytes required for their total length. The computed length of the list is finally written by calling a QuackyDucky finalizer.

As a cumulative post-condition of all these steps, we know that the output buffer now contains the concatenation of a valid binary format for each of its fields, and we can conclude that it thus also contains a valid representation of a clientHello message by calling the *validity lemma* clientHello_valid also generated by QuackyDucky and verified by F⋆. The call to this lemma is erased before extraction to C code.

As an important simplification, our sample code requires as static precondition that the output buffer be large enough to hold any valid clientHello message. In more realistic code, one would need to dynamically check this length. (Each of these writer functions are fail-safe, but their errors still need to be propagated.)

```
let write_extension_list cfg output pos = ...
  (* write a list of extensions computed from the configuration *)

let write_hello cfg output pos =
  (* write 2 bytes of protocol version *)
  let pos_after_protocol_version =
    write_protocolVersion output pos TLS_1p3 in
  (* copy 32 bytes from the configuration *)
  memcpy cfg.random 0ul out.base pos_after_protocol_version 32ul;
  let pos_after_random = pos_after_protocol_version + 32ul in
  (* similarly write or copy the other fields *)
  let pos_after_session_id = ... in
  let pos_after_ciphersuites = ... in
  let pos_after_compressions = ... in
  (* leave two bytes for the total length of the extension list *)
  let pos_list = pos_after_compressions in
  (* calls an auxiliary function to write the extension list in-place *)
  let pos_after_extensions =
    write_extension_list cfg output pos_list in
  (* computes and writes the extensions length at pos_after_compressions *)
  finalize_clientHelloExtensions
    output pos_after_compressions pos_after_extensions;
  (* call the validity lemma for the clientHello message *)
  let m = get () in clientHello_valid m output pos;
  (* return the final position *)
  pos_after_extensions
```

```
uint32_t write_extension_list(config cfg, slice output, uint32_t pos);
uint32_t write_hello(config cfg, slice output, uint32_t pos)
{
  uint32_t pos_after_protocol_version =
    write_protocolVersion(output, pos0, TLS_1p3);
  memcpy(cfg.random + 0, output.base + pos_after_protocol_version, 32);
  uint32_t pos_after_random = pos_after_protocol_version + 32;
  ...
  uint32_t pos_list = pos_after_compressions + 2;
  uint32_t pos_after_extensions =
    write_extension_list(cfg, output, pos_after_compressions);
  finalize_clientHelloExtensions
    (output, pos_after_compressions, pos_after_extensions);
  return(pos_after_extensions);
}
```

Figure 7: Sample Low⋆ code for writing a TLS client hello (above) and its translation to C (below).

5 LowParse: Secure Parser Combinators

As we have seen in §4.2, QuackyDucky produces parser implementations by composing basic parsers using *combinators*, which are higher-order functions on parsers. For example, a combinator for pairs may take parsers for types t and u and yield a parser for type $t \times u$. Its implementation may first parse a message of type t, then parse a message of type u.

LowParse is our library of parser combinators, based on the long tradition of *monadic* parser combinators [22] in the functional programming community. However, LowParse is unique in that it is tailored to support the verification of non-malleable, correct parsers. We focus on combinators at the

QuackyDucky Syntax	Data Type	Parser Combinator
uint$N$, $N \in \{8,16,32,64\}$	Unsigned integer within $0..2^N-1$	parse_uN
t[N], $N \in \mathbb{N}$	Fixed-size array of ts of length N	plist[p] truncN
t<M..N>	List of ts, of variable length $M..N$	vldata(plist[p], M, N)
t{M..N}	List of ts of variable element count $M..N$	$(\text{parse_u}k \text{ filter } (n \mapsto M \le n \le N)) \triangleright (n \mapsto p^n)$ where $k = 8 \times \log_{256} N$
struct{t_1 x_1; …; t_n x_n;}	Record with n fields named (x_i) of type (t_i)	$(p_1 \times \cdots \times p_n)$ synth $((v_1,\ldots,v_n) \mapsto \{x_1 = v_1; \ldots; x_n = v_n\})$
struct {…; uint$N$ x; t y[x]; …}	Variable-length field y prefixed by its length x	vldata$(p, 0, 256^{N/8}-1)$
enum {$E_1(N_1)$, …, $E_n(N_n)$, (M)}	Constant integer enumeration (with maximal value $M = 2^N - 1$)	penum(parse_uN, $[(E_1,N_1);\ldots;(E_n,N_n)]$)
struct {t x; select(x) { case E_1: t_1; …; case E_n: t_n } y}	Tagged union (t must be an enum type)	$p \triangleright_f q$ where $f(E_i,x) = E_i$ and $q(E_i) = p_i$ synth $(y \mapsto (E_i,y))$

Figure 8: The QuackyDucky input language and the corresponding LowParse combinators: everywhere in this table, p_i is the parser for type t_i. All lengths are counted in bytes except otherwise mentioned.

specification level and their security properties, then discuss more briefly their implementations. For each specification combinator, we prove non-malleability and inverse properties; for each implementation combinator, we prove both safety and correctness. All properties are verified by typing the library.

Figure 8 summarizes the QuackyDucky input language and the corresponding LowParse combinators. We designed QuackyDucky and LowParse in a modular way, making it easy to extend the surface syntax of QuackyDucky by providing additional combinators. For instance, the t x{M..N} syntax for variable-size lists prefixed with their number of elements is a late addition to support the Bitcoin application in §6.2 but is not required for TLS.

We first define the properties attached to the specifications of §4.2. We prove a stronger version of non-malleability than the one given in §2.1, extending the definition there to handle parsers that may not consume all their input.

Definition 1 *A parser p for type t is* non-malleable *if, whenever it succeeds and returns the same parsed value on two inputs, it also returns the same number of consumed bytes, and the two inputs coincide of these bytes.*

We also rely on the following no-lookahead property:

Definition 2 *A parser p has the* strong prefix property *when, if it succeeds on an input and consumes ℓ bytes, then it returns the same result on any inputs with the same first ℓ bytes.*

For a serializer to exist for a format that requires concatenating two value representations valid with respect to two parsers p_1, p_2 (such as pairs, lists, tagged unions, or variable-length data), p_1 is required to have the strong prefix property. Consider for instance serializing a pair of two pieces of data x_1, x_2 using serializers s_1, s_2 correct with respect to parsers p_1, p_2. We would like to prove that the serialization $s(x_1,x_2) = s_1(x_1) \cdot s_2(x_2)$ obtained by concatenating the two serializations, is correct with respect to the parser for pairs. By correctness of s_1, $p_1(s_1(x_1))$ succeeds and returns $(x_1, |s_1(x_1)|)$, but this is not enough to know that $p_1(s_1(x_1) \cdot s_2(x_2))$ succeeds and also returns $(x_1, |s_1(x_1)|)$ (so that we can cut the input after $|s_1(x_1)|$ and apply p_2 on the remainder, $s_2(x_2)$), unless p_1 has the strong prefix property.

These properties are included in the definition of parser_prop on the metadata generated by QuackyDucky, hence enforced by typing for all its parser specifications.

5.1 Specification Combinators

LowParse is an extensible library of combinators. For each parser specification combinator, we attach a corresponding metadata combinator; then, we define, when possible, a serializer combinator.

Parser combinators We define primitive parser combinators below. For each of them, we prove injectivity and any relevant additional properties indicated in their metadata, such as the strong prefix property. We also define derived combinators; in contrast, all their properties are established automatically as the result of their definitions (by type unification and matching on their metadata). The code produced by QuackyDucky only inserts annotations to prove their composability conditions, for instance, by computing the length boundaries of the derived metadata, which are then verified by F*.

We start by defining primitive parser combinators: fail, which consumes no input and fails; ret[x], which consumes no input and succeeds returning x; read_byte, which consumes and returns a single byte of input; and and_then, which sequentially composes two parsers where the second parser depends on the value parsed by the first parser (i.e., monadic composition). For each of these basic combinators, we prove non-malleability and/or the strong prefix property under suitable conditions. For instance, p and_then q has the strong prefix property provided that p has it, $q[x]$ has it for all x, and, moreover, if $q[x_1]$ and $q[x_2]$ succeed on inputs b_1 and b_2, respectively, and return the same value, then $x_1 = x_2$. (Otherwise, consider for example $p =$ read_byte and $q =$ ret[0].)

Using those primitive combinators, we define derived combinators, for which verification of non-malleability and metadata correctness automatically follows by typing. Given parsers p_0 and p_1 for t_0 and t_1, respectively, we can derive a parser for pairs of type $t_0 \times t_1$ using $p \times q$; mapping functions over parsed results using $p\,\mathsf{synth}\,f$; filtering parsed results by some predicate using $p\,\mathsf{filter}\,f$; etc. proving non-malleability and the strong prefix property for them under suitable conditions.

More specifically, we derive parsers for fixed-length machine integers, and we prove their non-malleability for both endiannesses. For instance, we define little-endian 16-bit parsing as $(\mathsf{read_byte} \times \mathsf{read_byte})\,\mathsf{synth}\,((x,y) \mapsto x + 256 \times y)$.

Our next combinators support variable-length data and lists:

- Given a parser p for type t, the parser $\mathsf{plist}[p]$ is defined by repeatedly applying p to its input. It fails as soon as p fails or consumes zero bytes. If succeeds when p eventually consumes its whole input and then returns the resulting list of values.
- Given a parser p for type t and $n > 0$, the parser $p\,\mathsf{trunc}\,n$ succeeds when p succeeds on its input truncated to its first n bytes and consumes exactly n bytes.

The parser $\mathsf{plist}[p]$ does not have the strong prefix property, but it consumes all its input. The parser $p\,\mathsf{trunc}\,n$ always has the strong prefix property, even if p does not. If s is a correct serializer for p at type t, then $p\,\mathsf{trunc}\,n$ is a parser for type $x : t\{|s(x)| = n\}$ and s is its correct serializer at that type.

We finally present further derived combinators, whose properties are automatically verified by construction:

Tagged unions: if p is a parser for type t and $f : u \to t$ and $q[x]$ is a parser for type $(y : u\{f(y) = x\})$ for every $x : t$, then:

$$p \triangleright_f q = p\,\mathsf{and_then}\,(x \mapsto q[x]\,\mathsf{synth}\,(y \mapsto y))$$

is a parser for type u. This combinator is a strengthening of and_then that enforces non-malleability of q by making its codomain dependent: u is the union type, and t is the tag type, and f gives the tag of an element of the union type. From there, we define a combinator for *sum types*, which can be used for tagged unions.

Enum types: if l is a list of key-value pairs where each key and each value only appear once, then it defines both a closed enum type (whose elements are the keys that appear in l) and an open enum type (whose elements are the known keys that appear in l and the unknown values that do not appear in l). We define parsers for both variants $\mathsf{penum}(p,l)$ (where p is the value parser), using filter, synth and the dictionary function on key-value pair lists.

Variable-sized data: formats such as TLS often specify variable-length data as a payload prefixed by its size in bytes. If p is a parser for the payload, and if s is a serializer correct with respect to p, then we define

$$\begin{aligned}\mathsf{vldata}(p,l,h) = \quad & \mathsf{parse_u}_\ell \circ \mathsf{filter}(n \mapsto l \le n \le h) \\ & \triangleright_f (n \mapsto p\,\mathsf{trunc}\,n)\end{aligned}$$

as a parser for the refined type $(x : t\{l \le |s(x)| \le h\})$, where $\ell = 8 \times |\log_{256}(h)|$, is the bit size of the size integer prefix, and $f(x) = |s(x)|$. Such parsers inherit the strong prefix property from the parser for the prefix size, regardless of whether it holds for p.

Correct Serializers Not all parser specifications have correct serializers. For instance, $\mathsf{ret}[x]$ and and_then do not have a generic serializer. So, in LowParse, we provide serializer combinators for read_byte, fail, plist, synth, and $\triangleright$, for each of which we prove correctness with respect to its corresponding parser combinator (i.e., that they are inverse of one another). We also easily prove that a correct serializer for p is also correct for $p\,\mathsf{trunc}\,n$ and $p\,\mathsf{filter}\,f$ (once its domain is restricted accordingly). From there, we derive correct serializers for $\times$, nlist, vldata, etc. for which the correctness proof automatically follows by typing.

5.2 Implementation Combinators

For each parser-specification combinator, LowParse provides combinators for its high-level parser and for its low-level validators and jumpers (and similarly for serializers). For primitive combinators, we implement their corresponding validators jumpers and serializers; for each of them we prove memory safety and functional correctness with respect to their specification. We implement most derived combinators by following the same construction as for their specs, by assembling the corresponding implementation combinators. Thus, their memory safety and functional correctness automatically follow by typing. We also define accessor combinators for synth and tagged unions, and accessors for pair elements, from which QuackyDucky derives accessors for struct fields and sum types.

By design, our combinators are inherently higher-order and so they cannot directly be extracted to C. Instead, we rely on meta-programming features of F* and KReMLin, based on source code annotations, to ensure that all combinator code is inlined and specialized before extraction. In most cases, this is achieved by annotating our source code. In other cases, we extend LowParse with F* tactics [31], pieces of F* metaprograms written once and for all and evaluated at typechecking time to automatically generate Low* validators from some type definitions. For example, our validators for enum values and tagged unions are specified using constant key-value lists. Instead of programming a loop on these lists, we meta-program their unrolling at compile-time, which yields a cascade of `if`s automatically turned into a `switch` by many C compilers. In rare cases, such as unions tagged with an enum value, we write additional validator combinators to more precisely control their inlining by F* and KReMLin.

In addition, metadata allow us to provide some generic validator combinators that apply regardless of the actual parser combinator. For example, if we know that a parser consumes a constant n bytes and always succeeds, then we can use a

	QD	F* LoC	Verify	Extract	C LoC	Obj.
TLS	1601	69,534	46m	25m	192,229	717KB
Bitcoin	31	1,925	1m56s	1m14s	1,344	8KB
PKCS #1	117	4,452	2m14s	2m39s	3,368	26KB
LowParse	N/A	32,210	3m5s	1m5s	185	739 B

Table 1: Overview of EverParse Applications

validator that just jumps *n* bytes. QuackyDucky selects these combinators based on the metadata it computes.

6 Integration and Evaluation

We evaluate the integration of EverParse-generated parsers for three applications: the TLS message format, integrated into MITLS; the Bitcoin block and transaction format, integrated into the Bitcoin Core benchmark; and the ASN.1 payload of PKCS #1 signatures, integrated into mbedTLS.

Table 1 shows for each application the lines of QuackyDucky input specification, the amount of F* code generated, the time required for verification and KReMLin extraction, and the size of the C code and compiled objects. The Bitcoin evaluation was performed on a 28-core Xeon E5-2680 v4 CPU with 128GB of RAM, running with turbo boost and all but one core disabled. The rest of the figures were collected on a 10-core Xeon W-2155 CPU with 128GB of RAM, running F* commit 7b6d77 with Z3 4.5.1 and GCC 7.4.

6.1 TLS Message Format

As described in §3, we have specified the TLS message format for all versions of TLS from 1.0 to 1.3. However, integrating the generated parsers presents some major challenges: implementations tend to define their own representations of messages, with field and tag names that differ from the RFC, and some of them like mbedTLS interleave the parsing and processing of messages. MITLS [7] uses functional, high-level parser implementations and types, operating on values. Most of the basic data types (such as cipher suite names) are defined in a module called `TLSConstants`, while some specialized ones scattered in other modules (e.g. group names in `CommonDH`). Extension types and parsers are in the `Extensions` module, while message types and parsers are in `HandshakeMessage`. We noticed that these files contain many assumptions and incomplete proofs, many of which have been completed for earlier drafts of TLS 1.3, but not updated as the formats changed (with EverParse, such updates and extensions only require a few changes to the format description).

In total, in order to switch to the high-level implementation produced by QuackyDucky, we update or rename over 200 types (and propagate these changes), which requires 2,865 additions and 3,266 deletions over 38 files (according to our Github pull request). Unlike LowParse, MITLS individually proves the non-malleability of each parser as a lemma separate from parser definitions instead of a refinement; the MITLS proofs for such lemmas are lengthy and intricate. So, we define a `LowParseWrappers` module to replace such proofs with a uniform call to LowParse parser property lemmas. Our changes do not break other existing proofs, but several generated types are more precise than the handwritten ones (notably, all lists are refined to ensure they can be serialized), which leads to additional conditions to prove in many functions. The generated parsers are also a lot stricter: for instance, we now check at parsing which extensions can appear in a message, and which messages can appear for the negotiated version.

To test the impact of EverParse parsers, we run the simple HTTP client and server tool distributed with MITLS to compare how many requests can be served, using the default algorithm choices. This tool is not optimized for production and processes requests sequentially. We compare the time to process 500 requests between the original MITLS parsers and EverParse high-level parser implementations.

	MITLS	MITLS-EverParse
HTTP requests	49.8 req/s	53.3 req/s

Integrating the low-level Low* implementations into MITLS requires a large effort, as many functions that are currently pure (operating on values such as lists) become stateful (the buffer that contains the valid positions matching each value must be live). To anticipate the benefits of this effort, we run a synthetic benchmark that validates all messages from a public dataset of TLS handshakes published by Lumen [39]. This dataset contains handshake produced by a wide range of clients and servers, and contains over 13GB of data (including the BSON overhead). As a baseline, we compare in-place validation time with the cost of checking the message length, allocating a buffer of the message size, and copying the contents of the message in the buffer.

Memcpy		EverParse	
1,864 MB/s	1.761 cy/B	2,684 MB/s	1.177 cy/B

6.2 Bitcoin Blocks and Transactions

To show that EverParse is extensible and evaluate the performance of its low-level parsers, we implement the Bitcoin block and transaction format, listed in Figure 9. We do not implement Segregated Witness ("segwit"), an extension that overloads the semantics of a length in the block format to conditionally add a new field to the block structure, because it requires a very ad-hoc combinator. Bitcoin requires two LowParse extensions: one for the encoding of "compact integers" (`bitcoin_varint`), and one for lists prefixed by their size in elements rather than in bytes.

For compact integers, the representation may either use 1, 3, 5, or 9 bytes depending on whether the value of the first byte is respectively less than 252, 253, 254, or 255. It is not clear from the Bitcoin documentation and wiki that the format of compact integer is not malleable (e.g. 4636

```
opaque sha256[32];
struct {
  sha256 prev_hash; uint32_le prev_idx;
  opaque scriptSig<0..10000 : bitcoin_varint>;
  uint32_le seq_no;
} txin;
struct {
  uint64_le value;
  opaque scriptPubKey<0..10000 : bitcoin_varint>;
} txout;
struct {
  uint32_le version;
  txin inputs{0..1000 : bitcoin_varint};
  txout outputs{0..11110 : bitcoin_varint};
  uint32_le lock_time;
} transaction;
struct {
  uint32_le version;
  sha256 prev_block; sha256 merkle_root;
  uint32_le timestamp;
  uint32_le bits; uint32_le nonce;
  transaction tx{0..2^16 : bitcoin_varint};
} block;
```

Figure 9: QuackyDucky specification of Bitcoin blocks

could be represented as `fd121c`, or `fe0000121c`). However, we checked that the Core implementation enforces the shortest representation in the `ReadCompactSize` function. Additionally, we allow list types to specify in their range the type of integer used to encode the prefix length or size (e.g. `txin inputs{0..2^14 : bitcoin_varint}`). A drawback of prefixing lists by their number of elements is that the theoretical maximum length of the formatted list can get extremely large. For instance, the maximal size of a well-formed Bitcoin block is over 2^{320} bytes (in practice, it is well-known that non-segwit blocks are at most 1MB). To avoid overflowing OCaml's 63-bit arithmetic in the parser metadata length computations in QuackyDucky, we must write more conservative boundaries. Scripts are known to be at most 10,000 bytes. Historically, all non-segwit blocks in the main chain contain less than 2^{16} transactions (although the maximum is higher). It is more difficult to bound the number of inputs and outputs of a transaction. If we assume a transaction is standard (at most 100,000 bytes) and all inputs are signed (their script is at least 64 bytes), there are less than 1000 inputs. Since outputs can be as short as 9 bytes, a transaction can have over 11000. Our test data is blocks 100,000 to 110,000 of the Bitcoin blockchain, totaling 21MB. To experimentally check those assumptions, we parsed all of these blocks and confirmed they are accepted by our validator.

For benchmarking, we measure: first, the performance of our zero-copy block validator compared with the built-in deserialization function of the Bitcoin Core client (commit cbe7efe); second, the performance variations of our zero-copy block validator across compilers and optimization levels; third, the performance impact of fine-grained code-generation options passed to KReMLin (Figure 10). For each one of those benchmarks, we report numbers in kiB/s, i.e. the throughput; we only occasionally report cycles per byte since most of our validators run at less than 1 cy/B.

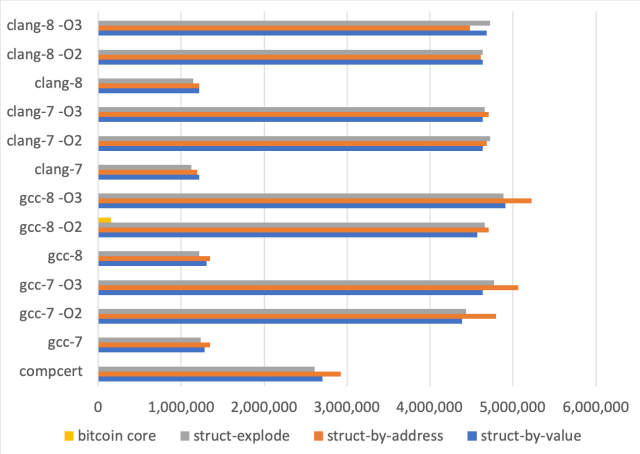


Figure 10: Synthetic performance comparison for validating 10,001 Bitcoin blocks. Throughput in kiB/s, higher is better.

The first measurement compares the performance of our code against a reference implementation, namely, Bitcoin Core. Bitcoin uses a custom template for serializing C++ objects. This template is well-optimized and tries to rely on casts and the in-memory representation of base types as much as possible. However, it is not zero-copy: parsing relies on the memory allocated for the C++ object, and serialization requires a copy to the output buffer. The benefit is that the data can be accessed using standard data structure libraries such as `std::vector` for lists. Bitcoin provides a built-in benchmarking tool for many of its features, including block deserialization and validation in `src/bench/checkblock.cpp`.

We modify this benchmark to use our test data of 10,001 blocks and deserialize all of them in each run. The benchmark deserializes 130 times, and reports the median over 5 runs. We keep the default compiler options (gcc-8, O2 optimization level). The measured throughput is 152,786 kiB/s, which translates to 15 cy/B on the test machine used for the Bitcoin measurements. We then validate the same 21MB of data using our validators, with the same compiler and optimization levels. We obtain a throughput of 4,568,632 kiB/s, which is less than a cycle per byte, for the default KReMLin configuration.

While the validation performance of our code is excellent, we do not claim that this benchmark is representative of real application usage, as it doesn't account for the overhead of accessor functions to read the block and transaction contents. Nevertheless, this shows that our verified low-level implementation is competitive with hand-optimized formatters.

Second, we measure performance variations across compiler versions. The performance is comparable between the two most recent versions of Clang and GCC, for optimization levels O2 and O3. Unsurprisingly, the default setting without optimization yields much slower code, but even then, we remain considerably faster than the original Bitcoin code. We also measure the performance of our code compiled with CompCert [28]. We find that CompCert is consistently 42% slower than GCC and Clang with optimizations, but still more than twice as fast as GCC or Clang without optimizations. We conclude that our code lends itself well to optimizations by modern compilers, and that users do not need to enable the (risky) O3 performance level to get maximum performance out of Clang or GCC.

Third, we experiment with various compilation schemes of the KReMLin compiler for the core type `LowParse.slice.slice`, a two-field C struct (representing a byte buffer through its base pointer and length) which in the default configuration is passed by value (§4.4). Two alternate compilation schemes are considered. First, passing both the base pointer and length as separate arguments to functions; this is the "struct-explode" category, and yields no performance improvement. Second, we pass those structures by address, relying on an unverified transformation in the KReMLin compiler, similar to CompCert's `-fstruct-passing` feature. This yields modest performance improvements for GCC 7 and GCC 8 at the higher optimization levels (3% to 9%). We conclude that our generated C code is satisfactory and that we don't need to either rewrite our code to pass slices by address (a substantial proof burden) or instruct KReMLin to perform this transformation (which would increase the trusted computing base).

Finally, we perform fuzz testing on the X64 machine code of our generated bitcoin-block validator as compiled with gcc-8 -O2. (Although our verification results ensure memory safety for all inputs, fuzzing may still, in principle, detect bugs in our toolchain and the C compilers we use.) We use SAGE [19], a fuzzer specialized to parsers, which generates random input, valid or not, and feeds them to the validator which SAGE automatically instruments to check for buffer overflows. As expected, SAGE reported no bugs after 21,664,448 inputs tested at an average rate of 599 inputs per minute.

6.3 ASN.1 Payload of PKCS #1 Signatures

Our last example is the payload of PKCS #1 signatures introduced in § 2.2. We extend LowParse with a combinator for the encoding of ASN.1 DER lengths. This encoding is particularly convoluted: if a length is less than 127, it is represented over a single byte. Otherwise, the 7 least significant bits of the first byte encode the length in byte of the shortest big endian representation of the length. This means the length can be at most $2^{2^{1016}} - 1$. To avoid overflows, we only support values of the first byte less than 132 (i.e. 32-bit lengths). An issue with the specification is the lack of dependency between the object identifier of the hash algorithm and the octet string of the actual digest: the application is required to check the digest is of the correct length if it tries to parse the signature contents. We capture this dependency by only making the outermost sequence variable length, and by parsing the object identifier as a constant tag of an union of fixed-length arrays. (Note that this is for illustration only, the recommended approach is to serialize the computed hash, and use a constant time comparison with the un-padded signature contents instead).

We integrate our code into `pkcs1_v15_verify` function of mbedTLS, and modify the built-in benchmarking tool to measure the PKCS #1 signature verification time instead of the raw public key and private key operation time measured by default. In addition, we also export the internal function to format the ASN.1 payload of the signature (`pkcs1_v15_encode`), and compare it with our extracted formatter functions. The following table compares the amount of operations per second and cycles per operation for complete signature verification, and for the encoding of the ASN.1 payload:

Operation	mbedTLS		EverParse	
Verify	79K op/s	5,700 cy/op	79K op/s	5,649 cy/op
Encode	31M op/s	14 cy/op	134M op/s	3 cy/op

As expected, the verification time is dominated by the cost of the RSA exponentiation: even though our validator is over 4 times faster and avoids the allocation of a modulus-sized intermediate buffer to compare the expected and computed digests, the impact on overall validation performance is negligible. For signing and encoding, the constant constant parts of the signature payload must be written manually, and separate finalizers must be called for to write the bytes we depend on for the algorithm choice and the outermost ASN.1 length.

We tested our implementation against all variants of the Bleichenbacher's attack listed in §2.2 and confirmed they are properly rejected.

7 Related work

Parsing combinators are widely used in functional programming languages, and there exist several libraries for network protocols [29], including TLS and X.509 [30].

For well-behaved language classes (e.g. regular, context-free), there is a long history on verification of parser correctness with respect to simple specifications (regular expressions, grammars). Jourdan et al. [25] propose a certifying compiler for LR(1) grammars, which translates the grammar into a pushdown automaton and a certificate of language equivalence between the grammar and the automaton. The certificate is checked by a validator verified in Coq [1], while the automaton is interpreted by a verified interpreter. Barthwal et al. [3] propose a verified grammar compiler and automaton interpreter for the simpler class of SLR languages, verified in HOL [42]. For regular languages, Koprowski et al.

introduced TRX [26], an interpreter for regular expressions verified in Coq. All of these works require runtime interpretation, which greatly degrades the performance compared to compilation. Furthermore, they target garbage-collected functional language runtimes like OCaml, which cannot easily be integrated into high-performance, native C applications.

For TLV languages, there have been some attempts [2] to create context-free or even regular specifications for X.509. However, due to the context-sensitive nature of ASN.1, these efforts rely on discretizations of some fields (such as variable-length integers) and drastic simplifications of the format (such as limiting the choice of extensions to a known subset). The combinatorial explosion required to achieve interoperability makes these approaches impractical for real implementations, although some authors claim otherwise [21].

For runtime safety, fuzzing techniques [19, 43] are widely deployed and often included into test suites for cryptographic libraries. Although best practice, fuzzing is by nature incomplete, and may be difficult to apply to authenticated messages (as fuzzing invalidates hashes, signatures and MACs). Dynamic analysis tools like Valgrind [35] or AddressSanitizer [44] are widely used but also incomplete, while static analysis tools like Frama-C [11] require higher expertise, a significant time investment, and tend to scale poorly with large codebases. Because of past attacks, specific tools have been created for TLS and cryptographic libraries, including TLS-Attacker [45], FlexTLS [5], and Wycheproof [8], but their focus is to uncover known vulnerability patterns in protocol implementations rather than prove formal guarantees on their message formats.

Another related line of work [10, 16] applies abstract interpretation and symbolic execution to study the properties of parsers, such as whether two implementations of a format accept the same message. These techniques can be applied to existing implementations, but cannot generate new ones.

Narcissus [47] also constructs correct binary parsers from a verified library of combinators written in Coq. There are two major differences with EverParse: first, Narcissus only proves the correctness of its parsers, while we also prove parser security; second, Narcissus only generates higher-order, functional implementations while our compiled approach means that our parsers are entirely specialized at F⋆ extraction, and can be compiled in zero-copy mode. Building on Narcissus, Ye and Delaware [51] build a verified compiler in Coq for parsers and formatters described using Protocol Buffers [20]. Like EverParse, their parsers and formatters are proven to be correct. Their library produces high-level functional code, which is memory-safe by construction—in contrast, EverParse produces low-level C code, together with memory safety proofs. Further, due to the inherent structure of the Protocol Buffers format, their work does not consider non-malleability.

Jim and Mandelbaum [23, 24] have formalized and developed parser generators for a wide class of context-free grammars extended with data dependency, including tag-length-value encodings, tagged unions, and other forms of dependence supported by QuackyDucky. They also provide tooling, like QuackyDucky, to automatically extract message format descriptions from RFCs and have applied their work to network message formats like IMAP, the popular mail protocol. While the input language of their framework is significantly more expressive than ours, EverParse, in contrast, produces provably safe, secure and functionally correct parsers. Jim and Mandelbaum also do not address message formatting.

8 Limitations and Future Work

Trusted computing base: we statically guarantee at the F⋆ source level memory safety, functional correctness, and non-malleability for all code generated by QuackyDucky. Preserving non-malleability down to machine code requires only preserving functional correctness, since non-malleability is a specification-level guarantee. All our verification results, including preservation of memory safety and functional correctness down to machine code, relies on a trusted computing base (TCB) that includes:

- the F⋆ proof assistant and the Z3 theorem prover, although work by Swamy et al. [48] provides a model of a subset of F⋆ and proves its soundness;
- the KReMLin compiler from Low⋆ to C, although work by Protzenko et al. [38] provides a model of a subset of Low⋆, its compilation to CompCert Clight, and proofs (on paper) that compilation to C preserves memory safety and functional correctness;
- the C compiler, although one can use the CompCert [28] verified C compiler, which ensures the preservation of memory safety and functional correctness, at the expense of some performance.

This trusted base is comparable to Coq-based verified implementations, which trust Coq, the Coq extraction to OCaml, and the OCaml compiler and runtime. Ongoing research aims to reduce this TCB by verifying Coq extraction; similar efforts could, in principle, be applied to F⋆ and KReMLin.

Conversely, neither LowParse nor QuackyDucky are in the TCB. LowParse is fully verified. The input format specification of QuackyDucky is trusted for liveness, but not for security: if there is a mistake in the format specification, the worse that can happen is that the generated messages are incompatible with implementations of the correct format. We rely on interoperability testing to detect such mistakes. Conversely, EverParse can be used during the standardization of a new message format, as it can prove that the specification is secure regardless of the generated implementation.

Expressiveness QuackyDucky currently focuses on supporting tag-length-value encodings of non-malleable data formats. We show that the message formats of several important protocols and standards, including TLS, PKCS #1 signature payloads and Bitcoin, fall into this class. LowParse,

being the target language of QuackyDucky's translation, is also currently restricted to supporting non-malleable data formats. However, it would be straightforward to make non-malleability conditional on a flag set in the parser metadata in order to define combinators for zero-copy malleable formats, including MessagePack, CBOR, Apache Arrow, Cap'n proto, and Protocol Buffers which are malleable at least by default (some have canonical representation rules). Generalized to support malleable formats, LowParse, being a library of verified monadic parser combinators, would support parsing with arbitrary data dependence and lookahead, beyond the class of context-free languages—however, coming up with *efficient* verified implementations of parsers for such language classes is an open question. In the future, we will also consider generalizing QuackyDucky to target the class of languages supported by LowParse.

Side-channel attacks: the implementation produced by EverParse branches on values read from the input buffer, which may leak (through timing side-channels) information when used on confidential data. We may in principle verify properties such as constant-time execution for the processing of simple message formats, reusing F* and KReMLin techniques and libraries for side-channel protection of cryptographic algorithms. For example, we may provide constant-time combinators for fixed-length secret bytestrings. We leave such extensions for future work.

Fuzzing: since we expect our extracted C code to be compiled by unverified toolchains (such as GCC and LLVM, with optimizations), fuzz testing can provide additional assurance that the compilation from F* to binary does not break our verified safety properties. We started using fuzzers optimized for parsers, such as SAGE [19], to fuzz the generated bitcoin block validator; we plan to extend their use to fuzz application code that uses generated validators and accessors.

Integration: we have integrated the high-level implementation of EverParse TLS parsers into miTLS, but our goal is to transition to the low-level implementation, thus avoiding many unnecessary heap allocations and copies. This is a major step towards making miTLS practical in performance-sensitive deployments.

9 Conclusion

Developers should prefer the convenience and robustness of writing high-level format specifications compiled by parser generation tools to programming tedious and error-prone custom parsers, although the latter is sometimes required for performance reasons. EverParse offers a unique combination of high performance, zero-copy implementations and high-assurance formal verification of the generated parsers.

Acknowledgments We thank the anonymous reviewers and Prateek Saxena for their helpful comments, which improved the writing of this paper. We thank Barry Bond, Christoph Wintersteiger and the Everest team for their help in testing EverParse. We thank Clément Pit-Claudel and Benjamin Delaware for insightful discussions on the goals of verified parsing. Tej Chajed and Nadim Kobeissi completed their work during internships at Microsoft Research.

References

[1] The Coq proof assistant. http://coq.inria.fr, 1984–2019.

[2] A. Barenghi, N. Mainardi, and G. Pelosi. Systematic parsing of X.509: eradicating security issues with a parse tree. *CoRR*, abs/1812.04959, 2018.

[3] A. Barthwal and M. Norrish. Verified, executable parsing. In *European Symposium on Programming*, pages 160–174. Springer, 2009.

[4] D. Benjamin. Applying GREASE to TLS extensibility. IETF Draft, 2016.

[5] B. Beurdouche, A. Delignat-Lavaud, N. Kobeissi, A. Pironti, and K. Bhargavan. FLEXTLS: A tool for testing TLS implementations. In *Usenix Workshop on Offensive Technologies (WOOT15)*, 2015.

[6] K. Bhargavan, B. Bond, A. Delignat-Lavaud, C. Fournet, C. Hawblitzel, C. Hritcu, S. Ishtiaq, M. Kohlweiss, R. Leino, J. R. Lorch, K. Maillard, J. Pan, B. Parno, J. Protzenko, T. Ramananandro, A. Rane, A. Rastogi, N. Swamy, L. Thompson, P. Wang, S. Z. Béguelin, and J. K. Zinzindohoue. Everest: Towards a verified, drop-in replacement of HTTPS. In *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*, pages 1:1–1:12, 2017. https://project-everest.github.io.

[7] K. Bhargavan, C. Fournet, and M. Kohlweiss. miTLS: Verifying protocol implementations against real-world attacks. *IEEE Security & Privacy*, 14(6):18–25, Nov 2016. https://github.com/project-everest/mitls-fstar.

[8] D. Bleichenbacher, T. Duong, E. Kasper, and Q. Nguyen. Project Wycheproof: Scaling crypto testing. In *Real World Crypto Symposium, New York, USA*, 2017.

[9] S. Y. Chau. The OID parser in the ASN.1 code in GMP allows any number of random bytes after a valid OID. Available from MITRE CVE-2018-16151, 2018.

[10] P. Cousot and R. Cousot. *Grammar Analysis and Parsing by Abstract Interpretation*, pages 175–200. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[11] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C. In *International Conference on Software Engineering and Formal Methods*, pages 233–247. Springer, 2012.

[12] C. Decker and R. Wattenhofer. Bitcoin transaction malleability and MtGox. In *European Symposium on Research in Computer Security*, pages 313–326. Springer, 2014.

[13] A. Delignat-Lavaud. RSA signature forgery attack in NSS due to incorrect parsing of ASN.1 encoded DigestInfo. MITRE CVE-2014-1569, 2014.

[14] T. Dierks and C. Allen. The TLS 1.0 protocol. IETF RFC 2246, 1999.

[15] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.2. IETF RFC 5246, 2008.

[16] K.-G. Doh, H. Kim, and D. A. Schmidt. *Abstract LR-Parsing*, pages 90–109. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[17] Z. Durumeric, F. Li, J. Kasten, J. Amann, et al. The Matter of Heartbleed. In *Proceedings of the 2014 Internet Measurement Conference*, pages 475–488. ACM, 2014.

[18] H. Finney. Bleichenbacher's RSA signature forgery based on implementation error, 2006.

[19] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012.

[20] Google. Protocol buffers. github.com/protocolbuffers.

[21] R. D. Graham and P. C. Johnson. Finite state machine parsing for internet protocols: Faster than you think. In *Security and Privacy Workshops (SPW), 2014 IEEE*, pages 185–190. IEEE, 2014.

[22] G. Hutton. Higher-order functions for parsing. *Journal of functional programming*, 2(3):323–343, 1992.

[23] T. Jim and Y. Mandelbaum. Efficient earley parsing with regular right-hand sides. *Electr. Notes Theor. Comput. Sci.*, 253(7):135–148, 2010.

[24] T. Jim and Y. Mandelbaum. A new method for dependent parsing. In *Programming Languages and Systems - 20th European Symposium on Programming (ESOP)*, pages 378–397, 2011.

[25] J.-H. Jourdan, F. Pottier, and X. Leroy. Validating LR(1) parsers. In *Proceedings of the 21st European Conference on Programming Languages and Systems*, ESOP'12, pages 397–416, Berlin, Heidelberg, 2012. Springer-Verlag.

[26] A. Koprowski and H. Binsztok. TRX: A formally verified parser interpreter. In *European Symposium on Programming*, pages 345–365. Springer, 2010.

[27] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. 2001.

[28] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM symposium on Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

[29] O. Levillain. Parsifal: A pragmatic solution to the binary parsing problems. In *2014 IEEE Security and Privacy Workshops*, pages 191–197, May 2014.

[30] A. Madhavapeddy and D. J. Scott. Unikernels: the rise of the virtual library operating system. *Communications of the ACM*, 57(1):61–69, 2014.

[31] G. Martínez, D. Ahman, V. Dumitrescu, N. Giannarakis, C. Hawblitzel, C. Hritcu, M. Narasimhamurthy, Z. Paraskevopoulou, C. Pit-Claudel, J. Protzenko, T. Ramananandro, A. Rastogi, and N. Swamy. Meta-F*: Proof automation with SMT, tactics, and metaprograms. In *28th European Symposium on Programming*, 2019.

[32] N. Mavrogiannopoulos, F. Vercauteren, V. Velichkov, and B. Preneel. A cross-protocol attack on the TLS protocol. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 62–72, 10 2012.

[33] MongoDB. BSON. http://bsonspec.org/.

[34] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[35] N. Nethercote and J. Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[36] G. Neufeld and S. Vuong. An overview of ASN.1. *Computer Networks and ISDN Systems*, 23(5):393–415, 1992.

[37] Y. Oiwa, K. Kobara, and H. Watanabe. A new variant for an attack against RSA signature verification using parameter field. In J. Lopez, P. Samarati, and J. L. Ferrer, editors, *Public Key Infrastructure*, pages 143–153, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[38] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hriţcu, K. Bhargavan, C. Fournet, and N. Swamy. Verified low-level programming embedded in F*. *PACMPL*, 1(ICFP):17:1–17:29, Sept. 2017.

[39] A. Razaghpanah, A. Akhavan Niaki, N. Vallina-Rodriguez, S. Sundaresan, J. Amann, and P. Gill. Tls handshake data collected by Lumen, Sept. 2017. https://haystack.mobi/datasets.

[40] E. Rescorla. The transport layer security (TLS) protocol version 1.3. IETF RFC 8446, 2018.

[41] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig. Transport layer security (TLS) session resumption without server-side state. IETF RFC 5077, 2008.

[42] N. Schirmer. *Verification of sequential imperative programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.

[43] K. Serebryany. OSS-Fuzz: Google's continuous fuzzing service for open source software. 2017.

[44] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *Usenix Annual Technical Conference (ATC12)*, pages 309–318, 2012.

[45] J. Somorovsky. Systematic fuzzing and testing of TLS libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1492–1504. ACM, 2016.

[46] R. Srinivasan. XDR: External data representation. IETF RFC 1832, 1995.

[47] S. Suriyakarn, B. Delaware, A. Chlipala, et al. Narcissus: Deriving correct-by-construction decoders and encoders from binary formats. *arXiv preprint arXiv:1803.04870*, 2018.

[48] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *ACM Symposium on Principles of Programming Languages*, pages 256–270, 2016. https://www.fstar-lang.org.

[49] F. Valsorda. Bleichenbacher'06 signature forgery in Python-RSA, 2016.

[50] P. Wuille et al. BIP62: Dealing with malleability, 2014.

[51] Q. Ye and B. Delaware. A verified protocol buffer compiler. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2019, pages 222–233, 2019.

Blind Bernoulli Trials: A Noninteractive Protocol For Hidden-Weight Coin Flips

Emma Connor and Max Schuchard

University of Tennessee

Abstract

We introduce the concept of a "Blind Bernoulli Trial," a noninteractive protocol that allows a set of remote, disconnected users to individually compute one random bit each with probability p defined by the sender, such that no receiver learns any more information about p than strictly necessary. We motivate the problem by discussing several possible applications in secure distributed systems. We then formally define the problem in terms of correctness and security definitions and explore possible solutions using existing cryptographic primitives. We prove the security of an efficient solution in the standard model. Finally, we implement the solution and give performance results that show it is practical with current hardware.

1 Introduction

Distributed systems sometimes require users to make random choices to drive network behavior. For example, peer-to-peer anonymous systems such as Freenet [12], AP3 [26], and DiscountANODR [36] employs a random "coin flip" in routing decisions to help obscure information about the path of a request from an observer. Opportunistic routing protocols may use a random decision on whether to forward or cache certain content or not. Systems that do rely on users making coin flips usually model the coin flip as a random trial that produces a single bit with a fixed probability. They distribute the probability of the trial's two possible outcomes either as a static, pre-defined parameter known to the whole network, or as a dynamic parameter distributed to users as cleartext.

However, in some instances the trial probability itself may be sensitive information. For example, if we use dynamic trial probabilities to prioritize certain content in a network (i.e., have some content forwarded with higher probability than others), then observers can distinguish and target higher-priority content. If an anonymous communication system varied the probability of forwarding to fine-tune the trade-off between performance and anonymity for individual messages, then a malicious node could selectively attempt to deanonymize easier traffic.

For such applications we can envision a cryptographic solution that allows each user to carry out only a single trial and obtain a random bit with some weighted probability, while learning as little as possible about the overall probability of each outcome. The user should not be able to repeat the trial for a different result, since users can easily approximate the probability using multiple results. Nor should they be able to use the trial parameters to learn anything more about the actual probability of the outcomes. Users should be able to perform a trial noninteractively, as they would if the probability were distributed in cleartext. In other words, we want a way to distribute a weighted coin that each user can flip once, while revealing *as little information as possible* about the weight of the coin. We call this construction a "Blind Bernoulli Trial, or *BBT*."

Specifically, we propose a definition where an authority generates and distributes unique keys to individual users. For each trial, the authority generates an encrypted *tag* that corresponds to the desired probability for that trial. Given a user key and a tag, one can noninteractively compute the outcome of exactly one trial without learning the overall probability.

We formalize the security of this system with a simulation-based definition inspired by the usual definition of semantic security for a cipher. Informally, the definition states that any function that can efficiently be computed by some number of identities and trial parameters could also be computed only knowing the trial results. The definition also includes a "leakage function" to allow schemes that leak some information but still have near-ideal security. The leakage function formally quantifies and places an upper bound on the amount of information an attacker can gain.

This paper evaluates three BBT schemes. First we develop a very simple protocol that meets our definitions and is based on a semantically secure cipher. This scheme essentially encrypts one trial result per user. We discuss why this trivial solution is unsatisfactory and then show an alterna-

tive construction from a general functional encryption primitive. Finally, since no practical functional encryption scheme for general functions is known, we examine more specific functional encryption schemes that support only a limited class of functions. We show how to construct a near-ideal BBT scheme from functional encryption supporting only inner product predicate functions, for which practical schemes currently exist.

In section 6, we compare the security of the ideal BBT schemes with the inner product construction using a quantitative attack analysis, discussing what an attacker can learn about the trial probability given a certain number of trial results. We design and evaluate a simulation of an attacker's perspective on both possible schemes, using the different information available to the adversary in each case. The attack simulations show that the information gained by an attacker for the inner product scheme, on average, is very similar to the information gained in the ideal case.

Since efficiency is a major concern, we discuss both the running time and storage requirements for the inner product scheme. To evaluate the feasibility of current inner product functional encryption schemes, we implement a recently proposed scheme in software (to our knowledge, the first implementation of this scheme) and provide benchmarks for each algorithm involved in a Blind Bernoulli trial scheme. The benchmarks show that a Blind Bernoulli Trial scheme based on inner product encryption can run in a reasonable amount of time on current hardware.

Finally, we explore in more detail some potential applications of this new cryptographic concept. We discuss two possible distributed-systems scenarios where random behavior is used and the probability of that random behavior is sensitive information. In these cases, Blind Bernoulli trials can enhance privacy in the distributed system by hiding the weighted probabilities from users.

1.1 Related Work

Protocols for remote parties to agree on a random bit in a way that is fair and verifiable go back decades in cryptography [6]. These protocols differ from BBT in that their goal is to prevent either party from biasing the result. BBT is almost the opposite: here we explicitly want one party to be able to bias the result and for the other party to be unable to determine the bias.

More generally, secure multi-party computation (MPC) encompasses a wide body of related work that deals with allowing remote parties to interactively perform arbitrary computations together. MPC focuses on protocols that enable distrusting parties to jointly compute a function on private inputs, without revealing the inputs to each other. It allows for private inputs from both parties, and protocols are interactive, proceeding in multiple rounds. Existing MPC schemes can be practical [31]. Our formulation of BBT does not allow interactive protocols, and the only private input is the probability of the trial. Therefore BBT is not compatible with MPC solutions.

While BBT does not fall under the area of MPC, it does fit squarely within the functional encryption model. Section 4 gives additional background on functional encryption and shows how BBT can be instantiated using general functional encryption.

In contrast with MPC, no practical general functional encryption is known. Recent works have proposed general functional encryption schemes, although these are not yet practical [15]. Other works have focused on implementing efficient functional encryption for specific classes of functions such as inner products and polynomials [18]. This paper primarily focuses on building an efficient construction specifically for BBTs.

2 Blind Bernoulli Trials

A Bernoulli trial models a random process with two possible results, where each result occurs with a fixed probability. This has applications in some distributed systems. For example, it provides a very simple means for one user to direct the behavior of a certain percentage of others without knowing exactly how many there are and without needing direct communication. An authority can distribute the parameters for a trial, and users can run a Bernoulli trial with the given parameters to self-organize into groups of approximately the desired proportions.

However, in some cases it may be important to the security goals of the system that individual users do not learn the overall probability of success. In these cases it is not acceptable for an authority to distribute the parameters for a trial, since this directly reveals the overall probability of success to all users. In response to this need, we formulate the concept of a *Blind Bernoulli Trial*, or *BBT*, which allows each user to obtain a single pseudo-random trial result without revealing additional information about the overall probability of success.

At first glance, it might appear that trivial solution would be for the authority to run the trials on a trusted computer and individually send a different trial result to each user. Since a user sees only their result, this scheme is secure. However, this scheme does not meet the requirement that a BBT be noninteractive. This leaves it with an important drawback compared to an unencrypted Bernoulli trial (publishing the probability parameter in plaintext). For an unencrypted Bernoulli trial, the authority can publish the probability parameter once, and any number of users can run a trial or forward the trial parameters to other users. Instead, the trivial interactive solution forces the authority to open an individual communication channel for each user. As a result, this interactive solution presents scalability concerns for systems

where communication between the authority and users may be intermittent or costly.

Ideally, a BBT scheme should more closely mirror the properties we get with a noninteractive unencrypted Bernoulli trial. The authority should be able to publish an encrypted object that represents the trial parameters and any number of users should be able to use this object to obtain a trial result without further interaction with the authority. Therefore, a Blind Bernoulli Trial scheme will try to construct "tags" that represent encrypted trial parameters of varying probabilities. Users will be able to use these tags to conduct trials without interaction with the authority.

In order to hide the overall probability of success for a trial, users must be able to obtain only one trial result per tag. If users could run multiple pseudo-random trials with the same tag they could quickly approximate the probability of success. To avoid this, we require that trial results are deterministic on a per user basis. In other words, the same user will always compute the same result for a given tag. Since Blind Bernoulli Trials must be deterministic, they are not true Bernoulli trials and do not have a "probability" of success in the same sense. Instead, when using a BBT, we are more interested in the overall probability of a trial's success across a distribution of users. Accordingly, when we speak of the "probability" of an outcome of a Blind Bernoulli Trial, we are referring to the probability of that outcome when a trial is performed with a user key that is selected at random from the set of users. For schemes that require the authority to store key material for each user, the "set of users" refers to the set of user keys kept by the authority. Otherwise, it refers to the set of all possible user keys.

Just as a single user must always get the same result for the same tag, it is also necessary to prevent one entity from controlling many user identities and utilizing them to perform multiple trials on a tag. For this reason, it must be impractical for an adversary to create multiple working user identities. We avoid this by introducing a master key that is required to generate new user identities. These identities take the form of a user key, which is combined with the tag to conduct a single trial. In general, an adversary should not be able to create a user key without the master key. The impact of collusion is discussed at length in Section 6.

Taking into account these properties, we can arrive at a clearer picture for what our scheme must look like: in a cryptographic Blind Bernoulli Trial scheme, an *authority* uses a private *master key* to generate and distribute *user keys* representing individual user identities and *tags*, each representing a Bernoulli trial with a fixed probability of success prescribed by the authority. Given a user key and a tag, there exists a public, deterministic procedure to compute the result of a single Bernoulli trial (either "success" or "failure") without revealing to the user the probability of success associated with the trial.

Formally, a Blind Bernoulli Trial encryption scheme consists of the following algorithms:

- $\mathsf{Setup}(1^\lambda)$: Accepts a security parameter λ and returns a master key sk and public parameters pk.
- $\mathsf{KeyGen}(\mathsf{sk})$: returns a user key uk.
- $\mathsf{TagGen}(x)$: takes a probability parameter x and returns a tag t; the exact form of the probability parameter can vary depending on the construction. In order to be useful, there must be at least two possible probability parameters that create tags with different probabilities of success.
- $\mathsf{Trial}(\mathsf{uk},\mathsf{t})$: returns a single bit b indicating success or failure.

This definition does not allow the trivial interactive solution mentioned earlier, where the authority carries out trials and directly communicates results to each user individually. This reflects a key design goal: that users must be able to obtain trial results without online communication with a centralized infrastructure. Users must be able to transfer tags to each other and each tag must be usable by all users. This allows individuals in a disconnected distributed system to obtain trial results without needing a direct intermediary.

2.1 Security Definition

Inherently, a BBT must reveal some information about the underlying priority. For example, an adversary that seeks to distinguish high-probability trials from low-probability ones could, after generating a trial result for a tag with their user key, guess "high-probability" for successful trials and "low-probability" for unsuccessful ones. Such a trivial adversary could already achieve non-negligible advantage in distinguishing between two types of tags.

Since each trial result unavoidably reveals *some* information about the underlying probability of success (a single successful trial means that the trial is more likely to have a higher probability of success), our security definition must take into account this inherent information leakage. Also, our definition must take into account collusion, so that the scheme remains as secure as possible even when a single adversary controls multiple user identities. Therefore, we compare the information an adversary learns from some number of user keys to that learned by an adversary that learns only the trial results corresponding to those keys.

Informally, a Blind Bernoulli Trial scheme is secure if an adversary with access to x user keys and y tags learns no more about the probabilities of success of any tags than he would by being given only the results of x trials for each tag. Since a BBT scheme intends to reveal the outcome of 1 trial per key, clearly this is the best any scheme could hope to do. In Section 6 we discuss some possible attacks when an

adversary controls multiple keys and quantify the amount of information gained by such an adversary.

Formally, we use a simulation-based definition to capture the idea that any function which is efficiently computable from a trial tag and a set of user keys must also be efficiently computable using only the trial results. We also include some allowance for additional information leaked, as this will be useful later in constructing a scheme that achieves near-ideal security with much greater efficiency compared to other schemes.

Definition 2.1 (Security with leakage)**.** *A Blind Bernoulli trial scheme is secure with respect to a leakage function $\mathscr{L}$ if for all probabilistic polynomial time (PPT) algorithms $\mathscr{A}$, there exists a PPT algorithm $\mathscr{B}$ such that for all polynomially-bounded functions f,h, the advantage of $\mathscr{A}$, defined as:*

$$\Pr[\mathscr{A}(1^\lambda, \mathsf{uk}_1, \mathsf{uk}_2, \ldots, \mathsf{uk}_n, \mathsf{t}, h(1^\lambda, x))) = f(1^\lambda, x)] - \Pr[\mathscr{B}(1^\lambda, \mathsf{uk}_1, \mathsf{uk}_2, \ldots, \mathsf{uk}_n, \mathsf{Trial}(\mathsf{uk}_1, \mathsf{t}), \mathsf{Trial}(\mathsf{uk}_2, \mathsf{t}), \ldots, \mathsf{Trial}(\mathsf{uk}_n, \mathsf{t}), \mathscr{L}(\mathsf{t}), h(1^\lambda, x)) = f(1^\lambda, x)] \quad (1)$$

is negligible in the security parameter, where x is the probability parameter and $\mathsf{t} = \mathsf{TagGen}(x)$.

This definition is closely-related to the usual definition of semantic security for private-key encryption and formalizes the idea that an adversary should learn as little as possible about a tag beyond the results of the trials of all keys known to the adversary. The leakage function places an upper bound on the amount of information that an adversary can learn from a tag because the definition states that any function that can be efficiently computed with the tag t can also be efficiently computed with only the trial results (which are intentionally revealed) and $\mathscr{L}(\mathsf{t})$.

Implicit in this security definition is the design requirement that an adversary can not forge additional user keys. If a BBT system allowed an adversary to forge a non-zero number of additional keys, that adversary would gain access to an extra set of trial results beyond those generated from their originally controled keys. Such a system fails to meet our security definition.

2.2 Other Design Goals

Security is a necessary property, but it is not the only design goal. To be usable, a BBT scheme must be efficient, both in terms of the running times of the algorithms and the space complexity of keys and tags. As stated previously, we also require that the protocol is noninteractive; that is, that tags can be freely transmitted from user to user and that users can obtain trial results from a tag without direct communication with the authority.

Each algorithm must be efficient enough to run in a reasonable amount of time. The running time of the Trial algorithm is particularly important, as we expect this algorithm to be run most frequently. Each user must run a trial for each tag received. Also, several applications of BBT feature users with lower computing resources compared to the authority. The other algorithms that comprise a BBT scheme are likely to be run less often: Setup is run only once, or only when the system needs to be re-keyed. And if n tags are created and m users then we expect the number of trials run to be on the order of nm if most users receive most tags.

The size of objects in the scheme must also be efficient. "Efficient" tags and user keys should require space logarithmic or at least sublinear in the number of users. In order to minimize storage requirements for the authority, we would also prefer that the tag generation algorithm does not depend on the current state of users. This eliminates the need for the authority to keep a database of users, and also allows user keys to be used even with tags that were generated before the key. This is particularly important in distributed systems applications that are disconnected or high churn, where new users may regularly encounter tags that were generated before the user key.

Another potentially desirable property would be the ability for users to generate tags. We consider this property desirable because if it is not wanted it can easily be removed by composing tags with any cryptographic signature scheme. Users can then simply reject tags that do not have a valid signature from the authority. On the other hand, it is not clear how to add this property to a scheme that does not support it, so we consider a scheme that does allow user tag generation to be more flexible.

A less-obvious but important property of a BBT scheme is the possible probability values for a tag. There is no requirement that a scheme support an arbitrary probability, but only that TagGen accepts some parameter that increases or decreases the probability of success for trials resulting from the generated tag. A scheme that allows more fine-grained control of the probability level is preferable over one that supports more limited probability levels.

3 Construction from Semantically-Secure Encryption

A simple BBT scheme can be trivially constructed from any symmetric or asymmetric encryption scheme that is semantically secure. In short, the authority can simply generate and store a random key for each user and send a tag consisting of a different ciphertext for each user, which that user can decrypt to obtain a trial result with the corresponding user key. To run a trial, users simply decrypt the ciphertext corresponding to their key. The security of this scheme follows immediately from the semantic security of the underlying en-

cryption system.

Either a public-key or symmetric system can be used here, as it long it meets the definition for semantic security. A symmetric-key system will be especially efficient, but a public-key system has the advantage that users can generate tags themselves. On the other hand, if a symmetric-key system is used, then the same keys that create tags can also decrypt them, which means that only the authority can hold the keys needed to create tags.

The individual algorithms are described as follows:

Setup

The authority initializes sk as an empty list of encryption keys.

Generating User Keys

The authority generates a decryption key uk for the underlying cryptosystem, gives it to the user, and appends its corresponding encryption key to sk (in the case of a symmetric system, the encryption key may be the same as the decryption key).

Generating Tags

A single tag consists of a set of ciphertexts, with one ciphertext per user. The authority generates it as follows:

1. The authority randomly selects a subset S containing x of $|\mathsf{sk}|$ users.
2. For each uk_i in sk, the authority computes $\mathsf{ct}_i \leftarrow \mathsf{Encrypt}_{\mathsf{uk}_i}(m_{\mathsf{success}})$ if $\mathsf{uk}_i \in S$, otherwise $\mathsf{ct}_i \leftarrow \mathsf{Encrypt}_{\mathsf{uk}_i}(m_{\mathsf{fail}})$
3. The tag is a tuple of all ct_i: $\mathsf{t} \leftarrow (\mathsf{ct}_1, \mathsf{ct}_2, \ldots, \mathsf{ct}_{|\mathsf{sk}|})$
4. The probability of success for the tag is $x/|\mathsf{sk}|$.

Trials

To perform a trial, a user selects the ciphertext corresponding to that user's key from the set of ciphertexts that forms the tag. The user then decrypts that ciphertext to obtain the trial result:

1. $m \leftarrow \mathsf{Decrypt}_{\mathsf{uk}_i}(\mathsf{ct}_i)$.
2. Return 1 if $m = m_{\mathsf{success}}$.
3. Otherwise, return 0.

3.1 Discussion

Since a trial consists only of a single decryption, trials are very efficient. User key generation is likewise extremely efficient as it requires only choosing a random key. Trials and user key generation are both $O(1)$. However, generating a tag is linear in the number of users, requiring l encryptions for l users. The space complexity of tags is also $O(l)$. When the number of users is known to be small, this may be acceptable. However, especially because BBT schemes are designed for applications where network resources are extremely limited, the linear space complexity of tags may quickly become a concern as the number of users increases.

This type of BBT scheme also allows for the most fine-grained possible control of probability. The tag generator can select any subset of users of any size for a successful trial. This is contrast to the schemes proposed in Sections 4 and 5, which are both limited in the possible subsets of users that observe a successful trial.

This scheme does not meet the design goal that tag generation does not depend on user state. The authority must maintain a central database of all user keys. If an asymmetric key system is used to allow tag generation by users, this key storage burden is also placed on each user. Each tag will be valid only for the user keys that existed in the authority's database at the time the tag was generated, so it will not be possible for newly-created users to run older tags.

4 Construction From Functional Encryption

A BBT scheme with ideal security can be constructed from any functional encryption scheme that supports arbitrary functions. In functional encryption, given a key k and ciphertext ct, one can learn the output of a function of the plaintext $f_k(m)$ without learning anything else about the plaintext [8, 30, 2]. To construct a BBT scheme from a functional encryption primitive, we define the user key functions using a pseudorandom function family (PRF) and a comparison. The tag plaintext consists of a random seed and a threshold value which determines the tag's likelihood of success. The authority encrypts tags under the functional encryption scheme and distributes the ciphertexts to users. Each user key corresponds to a function f that is defined as:

$$\begin{aligned} f(s,t) &= 1 \text{ if } h(s) < t \\ &= 0 \text{ otherwise} \end{aligned} \tag{2}$$

where h is a function selected at random from a PRF for each user key. Although the domain and range of functions in PRFs are typically viewed as bit strings, for our purposes it is more convenient to view them as integers in binary representation.

Setup

The authority initializes the functional encryption scheme and retains the master key sk which allows the creation of function keys.

Generating User Keys

The authority selects h at random from a PRF and generates the user key uk as the function key for f, as described above.

Generating Tags

The authority selects a seed s at random from the domain of each function in the PRF. The threshold t controls the probability p of the tag and is computed as $p * \mathsf{max}(\mathsf{range}(h))$. The authority then encrypts the tuple (s,t) under the functional encryption scheme to obtain the ct.

Trials

To perform a trial, a user computes $f(s,t)$ using the tag ct and the user key uk. The trial result is the function output.

4.1 Discussion

Although this construction achieves best-case security and allows fine-grained choice of success probabilities, its description relies on functional encryption for arbitrary functions. While such schemes do exist, they in turn rely on other heavy-handed approaches such as fully homomorphic encryption for which practical implementations are not yet available [15]. Therefore, a more practical solution is needed.

5 Construction from Inner Product Encryption

In this section we show how to use any fully attribute-hiding inner product encryption (IPE) scheme to construct a BBT scheme that is secure with respect to the leakage function $\mathscr{L}(\mathsf{t}) = \mathsf{pk}$, where pk is the public key of the IPE scheme used.

5.1 Background

The term "inner product encryption" has been applied to multiple related but distinct cryptographic concepts [21, 5, 28, 29, 27, 19, 3], In this context, we use it to refer to a specific form of predicate encryption where ciphertexts and keys are each associated with vectors, and the associated predicate is the inner product function. Predicate encryption is a generalized form of public key encryption where each key k is associated with a predicate function f_k, and each ciphertext is associated with an attribute y [18]. A ciphertext with attribute y can be decrypted with key k if and only if $f_k(y)$ is true.

In general, an IPE scheme operates on n-dimensional vectors of integers modulo a prime p. In an IPE scheme, each key $\mathsf{sk}_{\vec{k}}$ is associated with a vector $\vec{k} \in \mathbb{Z}_p^n$, and each ciphertext $\mathsf{ct}_{\vec{y}}$ is associated with an attribute vector $\vec{y} \in \mathbb{Z}_p^n$. The associated predicate is $f_{\vec{k}}(\mathsf{ct}_{\vec{y}}) = \vec{k} \cdot \vec{y} \stackrel{?}{=} 0$. In other words, given $\mathsf{sk}_{\vec{k}}$ and a ciphertext $\mathsf{ct}_{\vec{y}}$, one can compute the plaintext m if and only if $\vec{k} \cdot \vec{y} = 0$. An IPE scheme consists of the following functions:

- $\mathsf{Setup}(1^\lambda)$ outputs public key pk and secret key sk.
- $\mathsf{KeyGen}(\vec{k}, \mathsf{sk})$ accepts the secret key sk and a vector $\vec{k} \in \mathbb{Z}_p^n$ and outputs $\mathsf{sk}_{\vec{k}}$.
- $\mathsf{Encrypt}(m, \vec{y}, \mathsf{pk})$ outputs $\mathsf{ct}_{\vec{y}}$.
- $\mathsf{Decrypt}(\mathsf{ct}_{\vec{y}}, \mathsf{sk}_{\vec{k}}, \mathsf{pk})$ outputs m if $\vec{k} \cdot \vec{y} = 0$, otherwise outputs $\perp$.

Attribute-hiding IPE additionally requires that the vector $\vec{y}$ associated with each ciphertext is hidden. Partially attribute hiding schemes hide $\vec{y}$ from users who are not authorized to decrypt the associated ciphertext, while fully attribute-hiding schemes hide $\vec{y}$ even in the case where $\vec{k} \cdot \vec{y} = 0$.

IPE security is defined by a game between a challenger and an adversary [28].

Definition 5.1 (Attribute-hiding IPE Security)**.** *The security of a fully attribute-hiding IPE scheme is defined by the following game between the challenger and an admissible adversary $\mathscr{A}$*

1. *The challenger runs* $\mathsf{Setup}_{\mathsf{IPE}}$ *and gives* pk *to $\mathscr{A}$, retaining* sk.
2. *$\mathscr{A}$ adaptively makes any polynomial number of key queries for key vectors $\vec{k}_i$. The challenger gives $\mathscr{A}$ $\mathsf{sk}_{\vec{k}_i} \leftarrow \mathsf{KeyGen}(\vec{k}_i, \mathsf{sk})$*
3. *$\mathscr{A}$ chooses challenge attribute vectors $(\vec{y}_0, \vec{y}_1)$ and challenge plaintexts (m_0, m_1),.*
4. *The challenger randomly selects a bit $b = 0$ or $b = 1$.*
5. *The challenger gives $\mathscr{A}$ $\mathsf{Encrypt}(m_b, \vec{y}_b, \mathsf{pk})$*
6. *$\mathscr{A}$ can again adaptively make a polynomial of key queries for additional key vectors $\vec{k}_i$.*
7. *$\mathscr{A}$ outputs a guess b' and wins the game if $b' = b$.*

Here, an admissible adversary is defined as one whose queries adhere to at least one of the following conditions:

1. $\vec{k}_i \cdot \vec{y}_0 \neq 0$ and $\vec{k}_i \cdot \vec{y}_1 \neq 0$ for all $\vec{k}_i$

2. $m_0 = m_1$ and either ($\vec{k}_i \cdot \vec{y}_1 \neq 0$ and $\vec{k}_i \cdot \vec{y}_0 \neq 0$) or ($\vec{k}_i \cdot \vec{y}_0 = 0$ and $\vec{k}_i \cdot \vec{y}_1 = 0$) for all $\vec{k}_i$.

Without these restrictions an adversary can trivially infer b by submitting a challenge attribute pair that will be possible to decrypt for one value of b and not possible to decrypt for another, or a challenge message pair that can be decrypted to a different value depending on b.

5.2 Construction

With attribute-hiding IPE and its security now defined, we can show how to construct a BBT scheme using it. Intuitively, we will construct user keys and tags from randomly sampled vectors. Tags will correspond to IPE ciphertexts, user keys correspond to IPE user keys, and trials correspond to IPE decryptions. A successful decryption means a successful trial, while a failed decryption indicates a failed trial. We will vary the number of nonzero components in a tag's associated vector to control the probability that a randomly-selected user key will be able to decrypt it. Because the IPE scheme is fully attribute-hiding, the vector associated with tags is hidden from users, regardless of trial result.

Setup

The Setup function for IPE-based BBT additionally accepts a paramter a that determines the number of nonzero components in each user key. The authority runs the following procedure:

1. Run $\mathsf{Setup}_{\mathsf{IPE}}(1^\lambda, n)$ to obtain the private key sk and public key pk.
2. Store a as a public parameter.
3. Select m_{success} randomly from the message space of the underlying IPE scheme, and store it as a public parameter.

Generating User Keys

Every user key has the same number of nonzero components, which is parameterized as a.

1. $\vec{k}$ is randomly selected from the set of all vectors with a nonzero entries.
2. uk is computed as $\mathsf{KeyGen}_{\mathsf{IPE}}(\vec{k}, \mathsf{sk})$

Generating Tags

In this scheme, TagGen accepts the integer probability parameter $0 < x < n$, which represents the number of nonzero components in the tag vector:

1. $\vec{t}$ is randomly selected from the set of all vectors with x nonzero entries.
2. t is computed as $\mathsf{Enc}_{\mathsf{IPE}}(m_{\mathsf{success}}, \vec{t}, \mathsf{pk})$.

Trials

1. $m \leftarrow \mathsf{Dec}_{\mathsf{IPE}}(\mathsf{t}, \mathsf{uk}, \mathsf{pk})$.
2. Return 1 if $m = m_{\mathsf{success}}$.
3. Otherwise, return 0.

5.3 Security

Theorem 1. *The IPE-based BBT scheme is secure with respect to the leakage function* $\mathscr{L}(\mathsf{t}) = \mathsf{pk}$.

Proof. The proof is simulation-based. For all PPT adversaries $\mathscr{A}(1^\lambda, \mathsf{uk}_1, \mathsf{uk}_2, \ldots, \mathsf{uk}_n, \mathsf{t}, h(1^\lambda, \vec{t}))) = f(1^\lambda, \vec{t})$ there exists a PPT simulator that achieves the same advantage using only the trial results and the public key:

$$\mathscr{B}(1^\lambda, \mathsf{uk}_1, \mathsf{uk}_2, \ldots, \mathsf{uk}_n, \mathsf{Trial}(\mathsf{uk}_1, \mathsf{t}), \mathsf{Trial}(\mathsf{uk}_2, \mathsf{t}), \ldots, \mathsf{Trial}(\mathsf{uk}_n, \mathsf{t}), \mathsf{pk}, h(1^\lambda, \vec{t})) = f(1^\lambda, \vec{t})$$

The simulator $\mathscr{B}$ produces an output that is computationally indistinguishable from that of $\mathscr{A}$. The algorithm for $\mathscr{B}$ proceeds as follows:

$\vec{s} \leftarrow \langle 1, 1, \ldots, 1, 1 \rangle$
for $1 \leq j \leq n$ **do**
  $\vec{v}_j \leftarrow \langle 0, 0, \ldots, 1, \ldots, 0, 0 \rangle$ where only the jth element is 1.
  $\mathsf{t}_j \leftarrow \mathsf{Encrypt}(\vec{v}_j, \mathsf{pk})$
end for
for all uk_i **do**
  if $\mathsf{Trial}(\mathsf{uk}_i, \mathsf{t})$ is success **then**
    for $1 \leq j \leq n$ **do**
      if $\mathsf{Trial}(\mathsf{uk}_i, \mathsf{t}_j)$ is not success **then**
        $\vec{s}_j \leftarrow 0$
      end if
    end for
  end if
end for
$\mathsf{s} \leftarrow \mathsf{Encrypt}(\vec{s}, \mathsf{pk})$
Run $\mathscr{A}(1^\lambda, \mathsf{uk}_1, \mathsf{uk}_2, \ldots, \mathsf{uk}_n, \mathsf{s}, h(1^\lambda, \vec{t})))$ and output the result.

The output of algorithm $\mathscr{B}$ described above must be computationally indistinguishable from the output of $\mathscr{A}$; otherwise, an adversary could leverage the difference in the two to break the security of the underlying IPE scheme as follows:

1. Choose $\vec{t}$ as an arbitrary vector.

2. Submit arbitrary key vectors $\vec{k}_1, \vec{k}_2, \ldots, \vec{k}_n$.

3. Choose $\vec{s}$ as it would be computed by $\mathscr{B}$ (i.e., the vector with the maximal number of non-zero entries that is still orthogonal to all $\vec{k}_i$ orthogonal to $\vec{t}$).

4. Choose plaintext $m = m_{\mathsf{success}}$.

5. Submit challenge attribute vector $(\vec{t}, \vec{s})$ and challenge plaintext (m, m) and receive x, which is t if $b = 0$ or s if $b = 1$. Note that these submissions are admissible under the security definition of IPE because $\vec{s}$ and $\vec{t}$ are specifically constructed such that $\vec{s} \cdot \vec{k}_i = \vec{t} \cdot \vec{k}_i$ for all $\vec{k}_i$, as is required when $m_0 = m_1$.

6. Compute $\mathsf{out}_{\mathscr{A}} \leftarrow \mathscr{A}(1^\lambda, \mathsf{uk}_1, \mathsf{uk}_2, \ldots, \mathsf{uk}_n, \mathsf{x}, h(1^\lambda, \vec{t}))$.

7. If $\mathsf{out}_{\mathscr{A}}$ is as $\mathscr{A}(1^\lambda, \mathsf{uk}_1, \mathsf{uk}_2, \ldots, \mathsf{uk}_n, \mathsf{t}, h(1^\lambda, \vec{t}))$, output 0.

8. Otherwise, if the output $\mathsf{out}_{\mathscr{A}}$ is as $\mathscr{A}(1^\lambda, \mathsf{uk}_1, \mathsf{uk}_2, \ldots, \mathsf{uk}_n, \mathsf{s}, h(1^\lambda, \vec{t}))$, output 1.

Clearly, if the adversary has non-negligible advantage in distinguishing the outputs of $\mathscr{A}(1^\lambda, \mathsf{uk}_1, \mathsf{uk}_2, \ldots, \mathsf{uk}_n, \mathsf{t}, h(1^\lambda, \vec{t}))$ (which is exactly the output of the adversary $\mathscr{A}$) and $\mathscr{A}(1^\lambda, \mathsf{uk}_1, \mathsf{uk}_2, \ldots, \mathsf{uk}_n, \mathsf{s}, h(1^\lambda, \vec{t}))$ (which is exactly the output of the simulator $\mathscr{B}$), then the adversary also wins the IPE security game with non-negligible advantage. But, for a secure IPE scheme no such adversary can exist. Thus no PPT algorithm exists that can distinguish the output of the simulator $\mathscr{B}$ from the output of the $\mathscr{A}$.

□

5.4 Choice of Parameters

Besides the choice of underlying IPE scheme, the IPE-based BBT scheme also allows the choice of parameters for the dimension of the vector space n and the number of nonzero components a in each user key. Choices of these parameters will affect the number users that the system can support as well as the available choices for tag probabilities.

The IPE construction uses the number of nonzero components in a tag vector to control the probability of a successful trial. Therefore, for a system of dimension n there are n discrete probability "tiers" where tags in the ith tier have i nonzero components. Given an IPE scheme of dimension n, a nonzero components in each user key, and x nonzero components in a tag, the odds of a successful trial are:

$$\Pr[\text{success}] = \binom{n-x}{a} / \binom{n}{a}$$

Here, the numerator counts the number of ways to choose a user key that is orthogonal to the tag, and the denominator represents the total number of user keys possible.

This means that the probability tiers are not distributed uniformly. There are more tiers with lower probabilities of success than there are tiers with higher probabilities of success. For applications that require higher probabilities, we can simply invert the result of all trials to get a more favorable distribution. The remainder of this section follows this convention of inverting trial results. As a concrete example, figure 1 visualizes the case where $n = 64$ components are used.

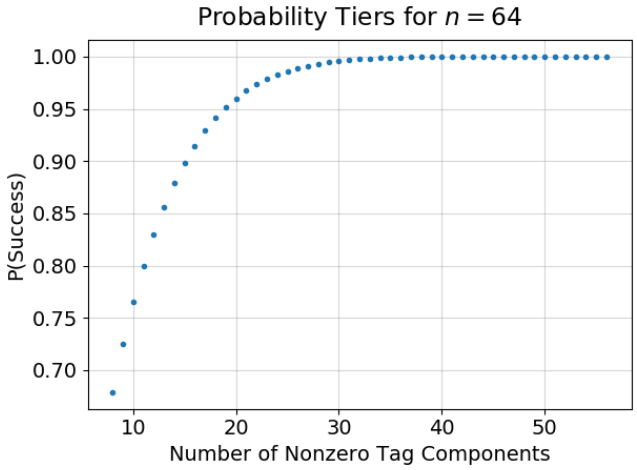


Figure 1: The distribution of probability tiers is biased toward the upper end of [0, 1] (using inverted trial results with $n = 64$, $a = 8$).

Two user keys uk_1 and uk_2 are called functionally unique if there exists a tag t such that $\mathsf{Trial}(\mathsf{uk}_1, \mathsf{t}) \neq \mathsf{Trial}(\mathsf{uk}_2, \mathsf{t})$. In other words, at least one of the associated key vectors has at least one non-zero component that is zero in the other vector, so that it is possible to construct a vector that is orthogonal to one but not the other. The number of functionally unique user keys depends on the number of components n and the choice of number of non-zero components in each user key a and is given simply as:

$$\binom{n}{a} = \frac{n!}{a!(n-a)!}$$

Table 5.4 compares the tag size in bits for IPE-BBT and the alternative scheme described in section 3. For the underlying IPE scheme, we used the state-of-the-art attribute-hiding IPE scheme due to Chen et al. [10] (our implementation using this scheme is discussed further in section 7). We assume a 1024-bit prime is used, for security equivalent to a symmetric key of 112 bits [16]. Chen's IPE scheme requires $4n + 4$ group elements for a ciphertext in an n-dimensional IPE scheme. We assume that group elements can be represented compactly by specifying only the x coordinate plus one bit indicating the y coordinate [25]. Thus the total size

Dimension	Users	IPE Size	ElGamal Size
9	9	5.1 KB	0.5 KB
10	45	5.6 KB	2.5 KB
11	165	6.2 KB	9.3 KB
12	495	6.7 KB	27.8 KB
16	12870	8.7 KB	723.9 KB
32	1.1×10^7	16.9 KB	591.7 MB
64	4.4×10^9	33.3 KB	249.0 GB

Table 1: Sizes of tags in a system supporting a given number of users in IPE-based BBT using Chen's IPE scheme, compared with semantically secure cipher construction using ECC ElGamal variant.

of a tag using Chen's IPE scheme is $(1024+1)(4n+4)$ for IPE dimension n.

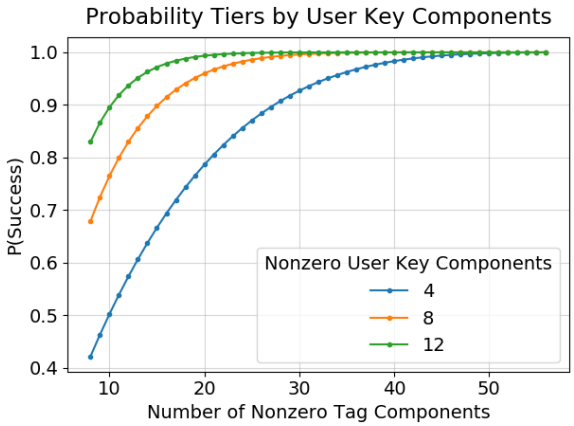
For comparison, we selected a public-key cryptosystem that represents minimal realistic storage requirements for a public key scheme at a comparable security level. We instantiated the semantically-secure encryption with an ECC variant of the ElGamal cryptosystem, which has been proven secure under elliptic curve discrete log assumptions [20]. This cryptosystem requires 2 group elements per ciphertext. We assume a 224-bit curve for a comparable level of security with the IPE scheme, again equivalent to a symmetric key strength of 112 [4]. This requires a total of 450 bits per ciphertext. The IPE scheme that supports 165 users (11 components with 8 non-zero user key components) uses less space than the corresponding public key scheme.

The number of possible tags is defined in the same way as it is for user keys. Two tags t_1 and t_2 are functionally unique if there exists a user key uk such that $\mathsf{Trial}(\mathsf{uk},\mathsf{t}_1) \neq \mathsf{Trial}(\mathsf{uk},\mathsf{t}_2)$. The number of functionally unique tags is different at each probability tier and depends on the total number of components in the vector space n and the number of nonzero components x used for that probability tier:

$$\binom{n}{x}$$

Therefore, it may be desirable to restrict the minimum and maximum probability tiers used so that the number of functionally unique tags at any probability tier does not fall below a chosen minimum. This means that the number of practically usable probability tiers may be less than n. For example, if one uses $n = 64$ components then one may only use tags with at least 8 components and no more than 56, which ensures that the number of functionally unique tags in any probability tier is at least $\binom{64}{8} \approx 2^{32}$.

The number of nonzero components a in each user key also affects the range and number of probability tiers: lower values of a allow a wider range of probability tiers, but fewer functionally unique user keys. Individual applications will need to determine a suitable trade-off. Figure 5.4 visualizes



Nonzero Components	Users (to nearest power of 2)
4	2^{19}
8	2^{32}
12	2^{41}

Figure 2: Comparison of available probability tiers with IPE dimension $n = 64$ with various values of a (nonzero user key components). Lower values of a give greater flexibility in probability choice but support fewer users.

the available probability tiers and number of user keys by the number of nonzero user key components.

6 Practical Security

In practice, the security of any BBT scheme will require that an adversary does not have access to too many keys. With enough keys, it is possible for an adversary to approximate the trial probability. As with any distributed system, the security of BBT in a system will break down if an adversary compromises enough nodes. In this section we first consider ways an adversary might attempt to compromise a system and then develop a model that quantifies the amount of information an attacker learns about trials based on the number of compromised nodes.

One of the most obvious way an adversary may attempt to compromise a system is a Sybil attack. In a Sybil attack, a single adversary creates multiple fake identities and appears to the network as many users instead of one [14]. Fortunately, there are wide range of known defenses against Sybil attacks for different domains. The authority can attempt to manually attempt to verify node identities before issuing user keys. In cases where this is impractical, automated defenses exist. Social network-based defenses such as SybilGuard [38], SybilLimit [37], and SybilInfer [13] are capable of detecting Sybil nodes using the social relationships between nodes in the network, under the assumption that attackers are unable to create many trust relationships with legitimate users. Behavior-based schemes seek to distinguish

between real and Sybil nodes via behaviors such as network activity and movement. For example, both work by Abbas et al. [1] and Jan et al. [17] utilize the heterogeneity of radio signals to detect Sybil attackers in MANETs and Wireless Sensor Networks respectively. Puzzle-based defenses such as SybilControl [22] utilize a proof of work based approach to mitigating Sybil attacks. Lastly, new approaches which leverage smart contracts, such as that proposed by Bochem et al. [7] put an economic price on Sybil identities.

Another simple attack is possible if a protocol misuses BBTs. For example, if a protocol requires multiple trials at the same probability protocol, then even a single user may be able to gain significant information about the probability of the associated tags. If a user observes several tags and knows (either from protocol specification or otherwise) that the tags all use the same probability parameter, then the user may use the differing results from the tags to approximate their shared probability.

In the event that an attacker does obtain multiple trial results, it is critical that we understand how much can be learned. The following subsections analyze the amount of knowledge that an attacker gains from multiple trial results, in both the ideal and the practical IPE schemes. Whether or not this amount of information leakage is considered acceptable is ultimately application-dependent.

Attacks on the Ideal Scheme

In an ideal BBT construction, the adversary learns only the trial results corresponding to the keys that it holds. If the adversary with n keys has no auxiliary information about the underlying distribution of success probabilities, then its best estimate for the success probability of a tag is $\frac{x}{n}$ where x is the number of observed successes. A confidence interval can also be computed to measure the uncertainty in this estimate. As an example, Figure 3 shows the upper and lower bounds of a 95% confidence interval on a tag with $p = 0.5$ as the number of the adversary's keys increases. The confidence interval is computed assuming that the adversary's trial results approximate the true tag probability as closely as possible, which is the best possible case for an attacker. We use a normal approximation to compute the confidence interval. Figure 3 shows that the attacker's knowledge increases with more trials. The attacker gains information rapidly at first, but trials beyond the first 10-15 bring diminishing returns. After 100 trials, the adversary is 95% confident that $0.4 < p < 0.6$.

However, if the adversary has a priori information on the underlying distribution of success probabilities, then the calculation is different. For example, if an adversary knows that all tags are drawn from discrete probability tiers (as in the case of the IPE-based scheme), then the adversary can use Bayesian inference to compute the likelihood that a tag comes from any tier given the a priori knowledge and the

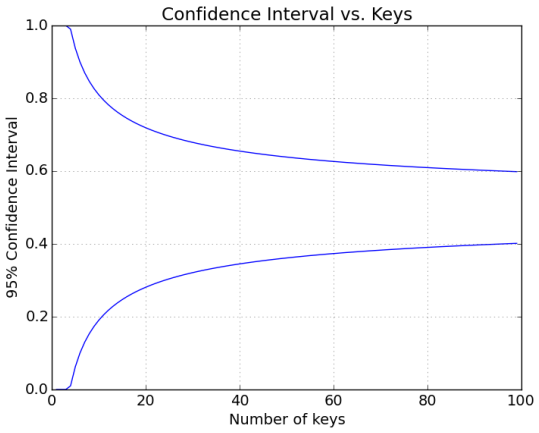


Figure 3: The upper and lower bounds of a 95% confidence interval for the probability of a tag with $p = 0.5$, as computed by an attacker whose trial results approximate a 50% success rate as closely as possible.

trial outcomes. Bayes' theorem gives the likelihood that a tag is from tier T given trial results R as:

$$P(T|R) = \frac{P(T)P(R|T)}{P(R)} \tag{3}$$

where $P(T)$ is the prior likelihood of a probability tier T, $P(R|T)$ can be modeled as a binomial distribution, and $P(R)$ can be computed as $\sum P(R|T_i)P(T_i)$ for each probability tier T_i. This represents, from the adversary's point of view, the likelihood that a tag comes from a given probability tier given the observed trial results. This serves as an effective measure of what the adversary knows about the probability of a trial associated with the tag.

Figure 4 shows the expected view of an attacker in the ideal discrete case for two different tags in a simplified model that includes only 4 probability tiers. The tiers used are selected at roughly equal intervals from the tiers available in the IPE scheme with $n = 64$, and are listed in table 2. The attacker's confidence in each probability tier was computed using the Bayesian model outlined above and taken as an average over all possible attacker trial results (weighted using the binomial distribution for the likelihood of each result). We assume that each probability tier is equally likely to an attacker as a prior likelihood. As the number of trial results available to the attacker increases, the confidence in the true probability tier increases while the confidence in other tiers decreases.

Attacks on the IPE Scheme

We know that with ideal security, only the trial results are learned. In the IPE scheme, additional information is leaked (constrained by leakage function in the security proof). We

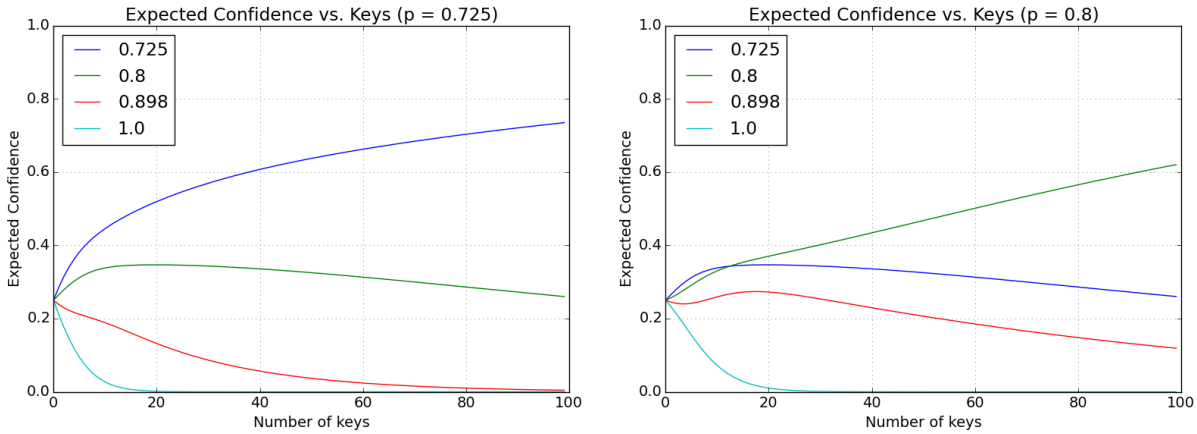


Figure 4: An attacker's view of two tags in the ideal discrete case, using a simplified model with 4 probability tiers. Each graph shows the attacker's expected confidence in each probability tier as a function of the number of trial results available to the attacker. The left graph shows the case where the true probability of the tag is $p = 0.725$, while the right graph shows a tag with $p = 0.8$.

Nonzero Components	p	Prior Likelihood
9	0.725	0.25
11	0.800	0.25
16	0.898	0.25
56	1.000	0.25

Table 2: Distribution of tags used in analysis.

wish to quantify how much an attacker can learn from the trial results, and how much more can be learned from the leaked information in the IPE scheme.

Essentially, the attacker can use information about the location of nonzero components in user keys to narrow down the set of possible tags. Knowing the components in each user key, together with their trial results for a tag, allows the attacker to quickly rule out any tag configurations that are inconsistent with the observed trial results.

The leakage function of the security proof takes this into account by allowing the public key of the underlying IPE scheme to leak. Since the public key is intentionally public in an IPE scheme, clearly this does not break the security guarantees of the underlying IPE; however, it does allow an adversary to potentially learn more about tags than the ideal case. Recall that the IPE public key allows one to encrypt a message under an arbitrary vector and obtain the ciphertext. In a BBT scheme, the IPE ciphertexts corresponds to BBT tags. Therefore, an adversary with the IPE public key can generate arbitrary tags, which allows the adversary to test user keys (but not tags) for the presence of any non-zero components. By repeated testing an adversary can determine exactly which components are zero and nonzero in each user key. In the rest of this analysis we assume the worst-case; that is, that the adversary already has access to the upper bound of information allowed by our security proof.

If an adversary knows which components in each user key are nonzero, then it can narrow the set of possible tags to those that give the same trial results for the same keys. Now, the adversary can estimate $P(R|T)$ as the proportion of possible tags from a tier that produce the same results when combined with same set of keys. For example, if trial result of testing a tag with one user key indicates that the two vectors are orthogonal, then any tags that share a nonzero component with the key are eliminated as possibilities. The attacker can count the number of consistent tags at each probability tier, and divide by the total number of possible tags in that tier to obtain a better estimate of $P(R|T)$. Again, $P(R)$ can be computed as $\sum P(R|T_i)P(T_i)$. The adversary can then again compute the overall likelihood that a tag comes from a given probability tier using the Bayesian inference described by equation 3.

Comparison

In order to determine the true impact of this attack, we compare the security of the IPE scheme to the ideal case by modelling two adversaries that each calculate the likelihood of tags differently. The component-aware adversary uses the full knowledge of the user keys components to compute the exact number of tags in each probability tier that could have produced the observed trial results, and then combines this with the prior likelihood of each probability tier to produce a confidence that a given tag comes from a given tier. Remember that no PPT adversary could hope to further distinguish between possible tags that would have produced the same trial results, since this directly contradicts the IPE security

definition.

On the other hand, the naive adversary uses only the number of success and number of failures to compute the likelihood of probability tiers. The likelihood of an observed result given a probability tier is modeled only as a binomial distribution. This is the best that an adversary could hope to do under the ideal security definition, where only trial results are revealed.

Figures 5 and 6 show a comparison of the two attacks in one case. For simulating the two attacks we chose parameters that provide a reasonable balance of performance, security, and number of users supported: $n = 64$ as the dimension of the IPE scheme and $a = 8$ for the number of nonzero components per user key. For simplicity, we limited tags to only a set of a few that provided roughly evenly-spaced probability tiers from about 0.72 to 1.0, in 0.10 intervals. Table 2 lists the exact tags used. For the prior distribution of tags, each probability tier was assumed to be equally likely.

For each attack, we sampled a given number of random keys, ran the attack with the keys on a randomly sampled tag at each probability tier, and then reported the resulting computed distribution of tag likelihood for each tag. We repeated this process many times to obtain an average over uniformly random sampled n keys and tags sampled randomly from our distribution of probability tiers. Shannon entropy, defined as $-\sum p_i \log(p_i)$ can be used as a measure of the uncertainty over a distribution [33]. After each sampled attack, we computed the entropy of the computed probability tier distribution. We then computed the average expected entropy over the sampled sets of user keys and tags and graphed it as a function of number of user keys held by an attacker. Figure 5 shows that the difference in the attacker uncertainty is minimal between the ideal and IPE schemes. The expected entropy in a tag distribution from an attacker's point of view diminishes with each additional key known, and it diminishes slightly faster for the IPE-based scheme than it does in an ideal scenario.

We also computed the expected Kullback-Leibler divergence between the component-aware model and the naive model, using the same tag distribution and random sampling method. The Kullback-Leibler divergence between two distributions P and Q is defined as $-\sum p_i \log(\frac{p_i}{q_i})$. This divergence measures the amount of information that an attacker gains about a tag probability by exploiting the information leakage in an IPE-based BBT scheme. Figure 6 shows that the additional information leaked to an attacker is minimal and quickly levels off around 0.02 bits.

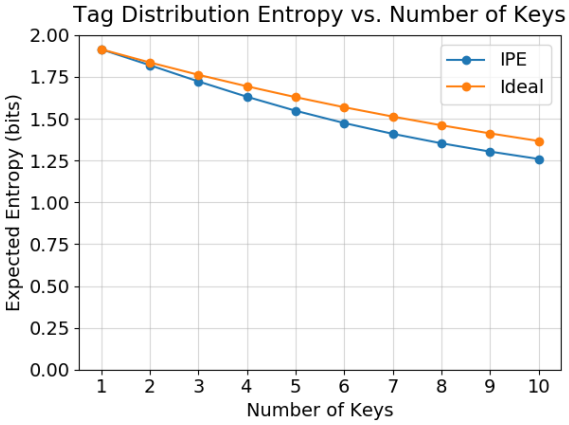


Figure 5: Expected entropy of computed tag likelihood as a function of attacker's number of keys.

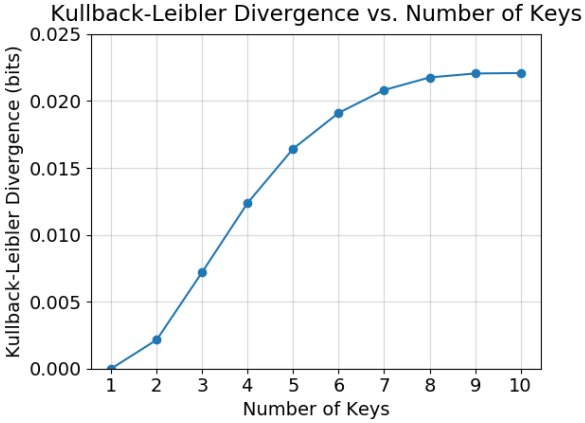


Figure 6: Expected Kullback-Leibler divergence between the component-aware model and the naive model as a function of attacker's number of keys.

7 Evaluation

We implemented IPE-based BBT using the adaptively attribute-hiding IPE scheme by Chen, Gong, and Wee, which is the current state of the art [10]. Chen et al. propose multiple variations with different performance characteris-

Object	Size (in elements)
Public key	$8n+16$
User key	7
Tag	$4n+4$

Table 3: Size of objects in IPE-BBT implemented with IPE scheme from Chen et al., in number of group elements.

tics using different standard assumptions. We implemented the variant described in Section 4.4 of their paper, which is proved secure under the external decisional linear assumption.

Under this scheme, tags and the public key both require space that is $O(n)$ in the number of dimensions, or $O(\log(l))$ in the number of functionally unique user keys. User keys have a constant space requirement of 7 group elements. Table 3 details the exact space requirements for each object.

Setup, trials, user key generation, and tag generation all run in $O(n)$ time. For a typical number of dimensions, the trial run time is dominated by the pairing operations. Crucially, trials in this scheme require only 7 pairing operations, regardless of the number of dimensions.

We tested the speed of this implementation on a single core of an Intel Xeon E5-2680 v4 CPU clocked at 2.40GHz. For pairing operations, we used the Stanford Pairing-Based Cryptography (PBC) library [23, 24]. The curve used was of PBC's "Type A," which are curves of the form $y^2 = x^3 + x$ over the field $\mathbb{F}_q$, where q is a prime such that $q = 3 \bmod 4$. q was chosen as a random 1024-bit prime, and the parameter r was chosen as a 224-bit number. Since the curve has embedding degree $k = 2$, these parameters are equivalent to the strength of an 112-bit symmetric key, according to the IEEE Standards for pairing-based cryptography [16].

Although the performance of all BBT steps must be reasonable, the Trial step is of the most concern. Trials are expected to be carried out by clients who may have limited resources, such as mobile devices. In contrast, Setup, TagGen, and KeyGen are all expected to be carried out by the single authority which would likely have access to significantly more resources.

Figure 7 shows the runtime of each BBT algorithm in our implementation using Chen, Gong, and Wee's IPE scheme. As expected, each algorithm shows a clear linear trend as the dimension is increased. As previously mentioned, the performance of the trial algorithm is the most critical, since disconnected clients with limited computing resources will be running it. Our results show that the trial algorithm is quite practical with parameters that can support a large number of users. For example, with $n = 64$ dimensions and $a = 8$ nonzero components per user key, there are approximately 2^{32} functionally unique user keys and a trial takes about 29 ms.

8 Applications

Any system that requires participants to take actions probabilistically can use BBTs to enhance privacy. Specifically, we envision several possible applications for BBTS in secure distributed systems. We provide two example scenarios that could benefit from deployment of blinded Bernoulli trials.

Probabilistic Forwarding of Content in a Network

Some networks (especially peer-to-peer networks) employ random walks based on probabilistic forwarding of content for privacy reasons. For example, in the anonymous communication systems such as AP3 [26] and DiscountANODR [36], when presented with a message, nodes randomly decide to either forward messages to another node in the mix or to send the message directly to its destination. The random process of forwarding obscures the origin of a message: when a node receives a message, it does not know if the message originated from the immediate preceding node or if it comes from 1, 2, or more hops away from that node. The use of the random coin rather than full circuit specification relieves the sending node from having to maintain state about the network topology outside of its immediate neighbors. The network uses a parameter p to specify the probability that nodes forward messages on to another node.

This approach introduces a trade-off between anonymity and network overhead: for lower values of p, messages take shorter paths (on average) through the network, but an observer can narrow down the set of likely originators to a smaller set based on the overlay distance to mix nodes and the distribution of random walk lengths. Higher values of p increase the number of possible nodes that might be the originator, but reduce network performance due to longer random walks. The authors of AP3 propose p between 0.5 and 0.9.

Using BBTs we can construct a network that allows for differential service, providing some users faster traffic, while still retaining the anonymity of longer paths. For example, consider a system with two classes of traffic. The Priority Class wants higher performance, and therefore shorter random walks, achievable with a low value of p. On the other hand, the Slow Class has no performance demands, so it can tolerate a higher value of p, increasing the size of its anonymity set. Without BBTs the traffic classes are trivial to distinguish, and as a result Priority traffic can be analyzed in a vacuum, leading to small anonymity sets. Marking a message's p with a BBT means that the two classes can not be distinguished based on this value, and in turn the faster class can benefit from the adversary's uncertainty about which peers should be included in the set of possible originators. This approach works especially well when the majority of the traffic falls into the Slow Class.

To evaluate the utility of BBT in this system, we simulated

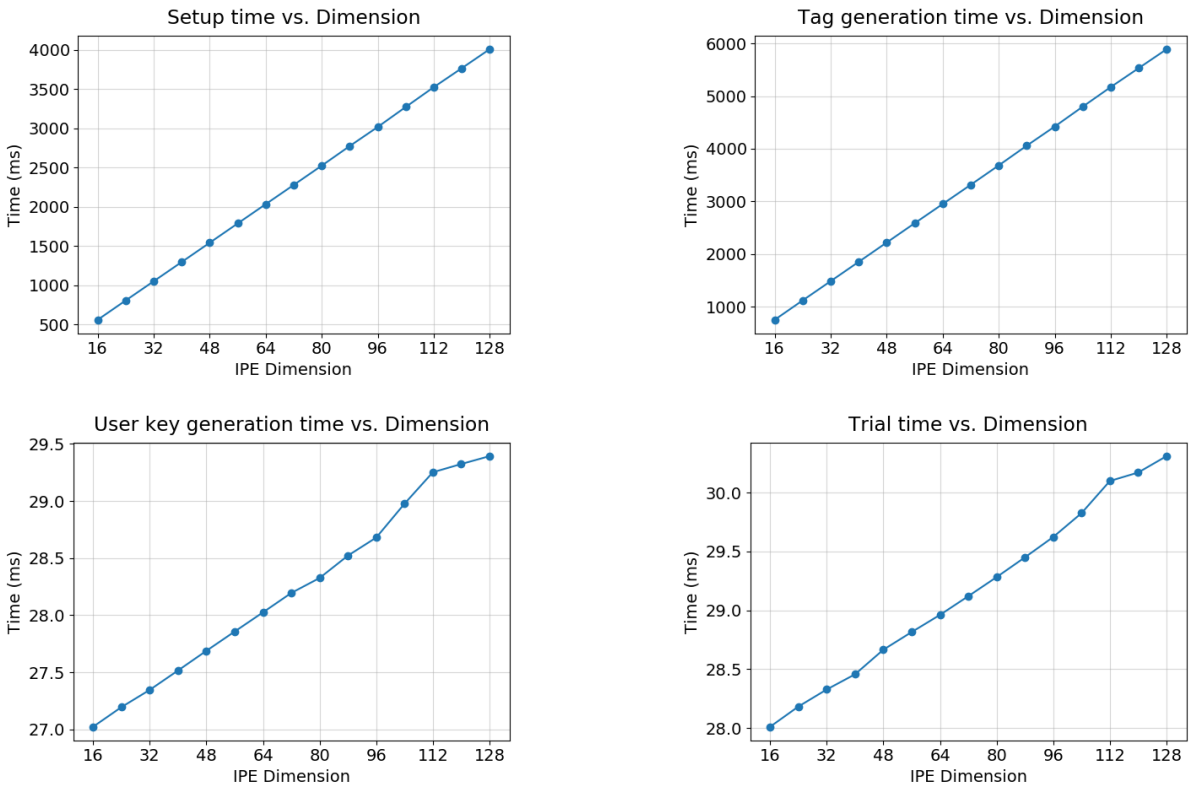


Figure 7: Runtimes for individual steps of IPE-based BBT implementation, by the dimension of the IPE.

a network that uses probabilistic forwarding with two traffic classes. Priority traffic uses $p = 0.5$, while Slow traffic in our test network uses $p = 0.9$. We simulated the operation of a network containing 1,000 nodes, with each node maintaining at least 4 connections to other nodes. The resulting graph had an average mixing time of slightly more than 7. In this network, Priority traffic represents 10% of total messages, and it is assumed that the adversary knows both the overall proportion of Priority traffic in the network and the full network topology. As expected, Figure 8 shows that Slow traffic takes drastically longer paths through the network, while Priority traffic reaches its destination much quicker. If an observer can distinguish Priority traffic, then this creates a privacy concern: because Priority traffic originates from nearby nodes with high probability, a node that receives it and recognizes it as Priority traffic has a high degree of confidence that the sender is in the small set of nodes that are nearby in the topology. However, if a BBT scheme is used to blind the priority class of traffic in the network, then Priority traffic can benefit from the reduced latency without resorting to unacceptably small anonymity sets. As Figure 9 illustrates, using BBT in the network increases the anonymity set sizes to levels that are near those of the Slow class. On average, anonymity sets for the merged class of traffic resulting from BBT blinding of priority is slightly lower than if only slow traffic is considered. This is because the adversary knows the relative frequency of fast and slow traffic, and can adjust computation of likely nodes based on this information.

Beyond the example systems of AP3 and DiscountANODR, many anonymous communication protocols feature a system parameter taking the form of a probability. Examples include Freenet [12], Crowds [35], and Imprecise Routing [11]. In each of these protocols, BBT can be used in a similar manner to adjust network behavior, allowing for different security properties while blending in with standard traffic.

Intrusion Detection in Wireless Sensor Networks

Another application for BBTs is masking how many nodes are conducting a specific behavior. As an example, consider a wireless sensor networks comprised of a large number of low-cost embedded devices conducting measurements in an environment [32]. They rely on short-range wireless communication between low-power devices (often battery-powered), which makes power consumption a top concern. Because of the communication range constraints and large number of devices over a wide area, direct connectivity is limited; instead sensors distribute messages from device to device over multiple hops in the wireless network.

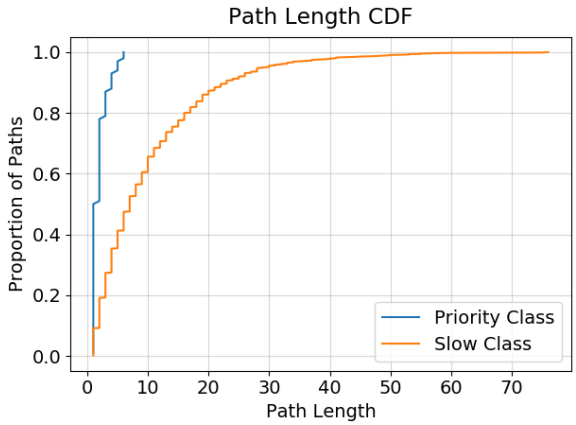


Figure 8: The distribution of path lengths for different traffic classes in the simulated network. Priority traffic takes significantly fewer hops to reach its destination, which translate to improved reliability and decreased latency.

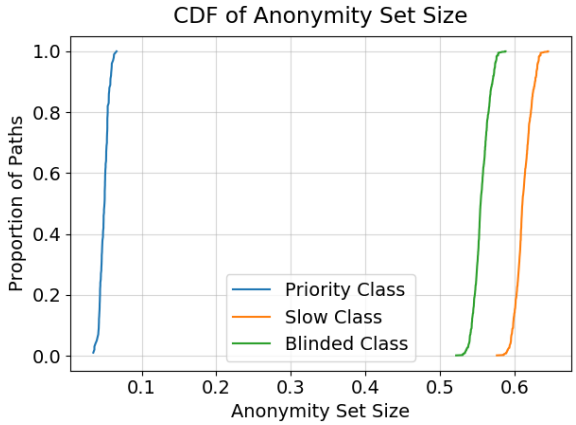


Figure 9: The distribution of anonymity set sizes for traffic classes. Without BBT, the anonymity set for nodes sending Priority traffic is less than 10% of the network. If the classes are indistinguishable, the anonymity set size is more than 50% of the network.

Sensor networks deployed in a hostile environment face the additional complication that certain compromised sensors may not be trustworthy. Adversaries may attempt to falsify sensor readings from compromised nodes. Recent work has examined methods for detecting compromised nodes in sensor networks [9, 34, 39]. In general, prior work has dealt with detecting falsified results by comparing them to a consensus of trustworthy results, under the assumption that an adversary is unable to compromise enough nodes to form a false consensus.

Of course, building a consensus requires a large number of measurements. Again, this leads to a trade-off between security and efficiency: employing many nodes in redundant measurements to build a consensus raises the bar for an attacker, but each sensor measurement uses energy (which is an extremely limited resource in wireless sensor networks). An obvious compromise would be to perform some fraction of measurements using only a small number of sensors, and sometimes use large-scale "audit" measurements to provide the necessary data for intrusion detection. If the metadata associated with a measurement reveals the number of nodes involved in the measurement, then this solution is vulnerable to an obvious attack: an adversary can simply refrain from lying during the audit measurements.

BBTs mitigate the aforementioned attack by limiting the adversary's ability to distinguish typical measurements from audit measurements. The authority can control the scope of the measurement using the probability of a tag, and nodes can decide their involvement in the measurement with a trial. This prevents the attacker from selectively influencing measurements without detection. In addition, the authority can easily verify that nodes are not returning results for spurious measurements by duplicating the deterministic trial results and verifying that the node originating the measurement was in fact one of the nodes directed to take the measurement.

We evaluated the effect of applying this technique on a small network of sensor nodes. We modeled the network as 100 independent sensor nodes, each capable of a fixed finite number of measurements before it is considered expired. The adversary controls one node. We assume that the malicious node is detected if it lies on a measurement that is performed by at least half of the nodes. For normal measurements, the network uses a tag with $p = 0.05$. For audit measurements, the network sets $p = 0.6$ (this ensures with high probability that at least half of the network does actually perform the measurement). We also assume that the authority verifies that each returned result is tied to an actual successful trial. This can be accomplished in general by requiring that nodes attach their user key to each result, encrypted so that only the authority can read it. Note that this is consistent with the threat model which already assumes that the authority has unlimited access to user keys.

By varying the "audit rate" (the fraction of measurements that are audit measurements), the authority can select an arbi-

Audit Rate	Probability of Detection	Relative Lifetime
0.0	0.00	12.0
0.2	0.71	3.7
0.4	0.86	2.2
0.6	0.93	1.6
0.8	0.97	1.2
1.0	0.99	1.0

Table 4: The trade-off between sensor lifetime and detection probability. Here, the audit rate is the proportion of measurements that are audit measurements; the detection probability is the probability that a single lie is detected; and the relative lifetime is the lifetime of the entire network relative to the base case where all measurements are audit measurements. Allowing a small probability of an undetected attack can significantly increase the lifetime of the network.

trary trade-off between efficiency (conserving resources for more useful measurements) and security (performing redundant measurements to detect attacks). As the audit rate increases, so does the probability of detecting a malicious measurement; on the other hand, as the audit rate decreases, the lifetime of the network increases. Table 4 shows this trade-off. We sampled audit rates and computed both the resulting sensor lifetime and the probability that the adversary is detected each time it lies. The expected sensor lifetime is reported relative to the lifetime in the "safe" network (that is, one with an audit rate of 1). Even at a relatively low audit rate of 0.2, the probability of detection is high (71%) from the perspective of the sensor. This is because the number of sensors employed for normal measurements is much smaller than the number of sensors involved in an audit measurement. As a result, a given sensor is more likely to be selected via a large audit measurement than it is to be selected for a normal measurement.

Because the adversary cannot distinguish normal and audit measurements, it cannot selectively lie. This effectively limits the number of times an adversary can lie without detection (with overwhelming probability). Without using BBT, the adversary is only prevented from lying during audit measurements; otherwise, it can forge measurements without detection indefinitely. With BBT, if the adversary has a 50% chance of detection per measurement (for example) then it can expect to lie only about twice before detection, on average.

Discussion

In practice, the bandwidth and computation overhead of a BBT scheme will determine its usefulness for any particular application. In the first application, we assume that the overhead of computing a trial is low relative to the work required for a message forward. This assumption is reasonable, for example, in AP3, where a single forward requires a distributed hash table lookup and therefore multiple round-trip messages with peers. In this scenario the combined latency of one forward can be significantly slower than a trial in the IPE-based BBT construction.

For the sensor network application, the energy savings of skipping measurements will have to be weighed against the cost of performing the trials at each sensor. Depending on resource availability, different BBT constructions might be more appropriate. For example, if bandwidth is cheap but computational resources are constrained, then the IND-CPA construction presented 3 might actually be more suitable.

9 Conclusion

Although many distributed systems make use of probabilistic actions, systems so far either specify the probability as a fixed parameter or reveal the varying probabilities to each user. Blind Bernoulli trials are a privacy-enhancing measure that preserves the semantics of Bernoulli trials across a set of nodes, while hiding the exact parameters from individuals.

Fundamentally, Blind Bernoulli trials reveal the trial outcome without revealing the trial parameters. We create a definition that formalizes the idea that users should learn "no more" about the trial parameters than they would by receiving only the trial results corresponding to the keys held, and explore some possible solutions that meet our proposed definition.

Since BBTs are a special case of functional encryption (FE), they can easily be implemented with any FE primitive that allows arbitrary functions. However, since practical general functional encryption is not currently available, there is a need for a specific scheme that achieves the same results with a more efficient algorithm. Existing forms of functional encryption for specific classes of functions can be used to instantiate much more practical Blind Bernoulli trials, albeit with some security loss. Specifically, we can construct a near-ideal BBT scheme from inner product encryption by varying the number of nonzero components in tags to control the probability of their trials.

We prove the near-ideal security of the IPE-based scheme under our definition by showing a reduction to the security of the underlying IPE scheme. This definition takes into account the security loss and places an upper bound on exactly how much information is revealed to an adversary, even one who controls multiple keys. By simulating the attacker's point of view on a large number of trials and keys, we can measure the average uncertainty towards the distribution of possible tags for an adversary with multiple keys: the expected entropy of a tag distribution decreases steadily with the number of trial results known. We compare the entropy loss in the ideal case to the IPE-based BBT scheme and show that the additional entropy loss in the IPE case is small.

Finally, we implement the IPE-based scheme in software and analyze its efficiency. We show that, in a system with realistic parameters, trials can be executed in a reasonable amount of time. Also, tags and keys in the IPE scheme are small (logarithmic in the number of users). Even with a relatively small number of users (on the order of 100), this is less storage than our simple semantically-secure cipher solution with linear keys.

We conclude that Blind Bernoulli trials can be efficiently implemented using IPE, and that they are an effective way obscure probability parameter metadata. This paper proposes two potential applications where this can prevent attacks that would otherwise exploit knowledge of the probability parameter. We hope that others in the security and distributed systems communities will explore additional uses for the primitive.

References

[1] ABBAS, S., MERABTI, M., LLEWELLYN-JONES, D., AND KIFAYAT, K. Lightweight sybil attack detection in manets. *IEEE systems journal 7*, 2 (2013), 236–248.

[2] AGRAWAL, S., AGRAWAL, S., BADRINARAYANAN, S., KUMARASUBRAMANIAN, A., PRABHAKARAN, M., AND SAHAI, A. Function Private Functional Encryption and Property Preserving Encryption - New Definitions and Positive Results. *IACR Cryptology ePrint Archive* (2013).

[3] AGRAWAL, S., FREEMAN, D. M., AND VAIKUNTANATHAN, V. Functional Encryption for Inner Product Predicates from Learning with Errors. In *Advances in Cryptology – EUROCRYPT 2010*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 21–40.

[4] BARKER, E. SP 800-57 Part 1 Rev. 4: Recommendation for Key Management Part 1: General, Jan. 2016.

[5] BISHOP, A., JAIN, A., AND KOWALCZYK, L. Function-Hiding Inner Product Encryption. In *Advances in Cryptology – ASIACRYPT 2015*. Springer, Berlin, Heidelberg, Berlin, Heidelberg, Nov. 2015, pp. 470–491.

[6] BLUM, M. Coin flipping by telephone a protocol for solving impossible problems. *ACM SIGACT News 15*, 1 (1983), 23–27.

[7] BOCHEM, A., LEIDING, B., AND HOGREFE, D. Unchained identities: Putting a price on sybil nodes in mobile ad hoc networks. *Security and Privacy in Communication Networks (SecureComm 2018). Singapore (August 2018)* (2018).

[8] BONEH, D., SAHAI, A., AND WATERS, B. Functional Encryption - Definitions and Challenges. *TCC 6597*, Chapter 16 (2011), 253–273.

[9] CHATZIGIANNAKIS, V., PAPAVASSILIOU, S., GRAMMATIKOU, M., AND MAGLARIS, B. Hierarchical anomaly detection in distributed large-scale sensor networks. In *Computers and Communications, 2006. ISCC'06. Proceedings. 11th IEEE Symposium on* (2006), IEEE, pp. 761–767.

[10] CHEN, J., GONG, J., AND WEE, H. Improved inner-product encryption with adaptive security and full attribute-hiding. In *International Conference on the Theory and Application of Cryptology and Information Security* (2018), Springer, pp. 673–702.

[11] CIACCIO, G. Improving sender anonymity in a structured overlay with imprecise routing. In *International Workshop on Privacy Enhancing Technologies* (2006), Springer, pp. 190–207.

[12] CLARKE, I., SANDBERG, O., WILEY, B., AND HONG, T. W. Freenet - A Distributed Anonymous Information Storage and Retrieval System. *Workshop on Design Issues in Anonymity and Unobservability* (2000).

[13] DANEZIS, G., AND MITTAL, P. Sybilinfer: Detecting sybil nodes using social networks. In *NDSS* (2009), San Diego, CA, pp. 1–15.

[14] DOUCEUR, J. R. The Sybil Attack. *IPTPS* (2002).

[15] GOLDWASSER, S., KALAI, Y., POPA, R. A., VAIKUNTANATHAN, V., AND ZELDOVICH, N. Reusable garbled circuits and succinct functional encryption. In *the 45th annual ACM symposium* (New York, New York, USA, 2013), ACM Press, pp. 555–564.

[16] 1363.3-2013 - IEEE Standard for Identity-Based Cryptographic Techniques using Pairings. IEEE, 2013.

[17] JAN, M. A., NANDA, P., HE, X., AND LIU, R. P. A sybil attack detection scheme for a centralized clustering-based hierarchical network. In *Trustcom/BigDataSE/ISPA, 2015 IEEE* (2015), vol. 1, IEEE, pp. 318–325.

[18] KATZ, J., SAHAI, A., AND WATERS, B. Predicate Encryption Supporting Disjunctions, Polynomial Equations, and Inner Products. *EUROCRYPT 4965*, Chapter 9 (2008), 146–162.

[19] KAWAI, Y., AND TAKASHIMA, K. Predicate- and Attribute-Hiding Inner Product Encryption in a Public Key Setting. *Pairing 8365*, Chapter 7 (2013), 113–130.

[20] KOBLITZ, N. Elliptic Curve Cryptosystems. *Mathematics of Computation 48*, 177 (Jan. 1987), 203–209.

[21] LEWKO, A., OKAMOTO, T., SAHAI, A., TAKASHIMA, K., AND WATERS, B. Fully Secure Functional Encryption: Attribute-Based Encryption and (Hierarchical) Inner Product Encryption. In *Advances in Cryptology – EUROCRYPT 2010*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 62–91.

[22] LI, F., MITTAL, P., CAESAR, M., AND BORISOV, N. Sybilcontrol: Practical sybil defense with computational puzzles. In *Proceedings of the seventh ACM workshop on Scalable trusted computing* (2012), ACM, pp. 67–78.

[23] LYNN, B. Stanford Pairing-Based Cryptography Library. `https://crypto.stanford.edu/pbc/`, 2006–2013.

[24] LYNN, B. On the implementation of pairing-based cryptosystems. Dissertation, 2007. `https://crypto.stanford.edu/pbc/thesis.pdf`.

[25] MENEZES, A., AND VANSTONE, S. A. Elliptic Curve Cryptosystems and Their Implementations. *Journal of Cryptology* (1993).

[26] MISLOVE, A., OBEROI, G., POST, A., REIS, C., DRUSCHEL, P., AND WALLACH, D. S. Ap3: Cooperative, decentralized anonymous communication. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop* (2004), ACM, p. 30.

[27] OKAMOTO, T., AND TAKASHIMA, K. Achieving Short Ciphertexts or Short Secret-Keys for Adaptively Secure General Inner-Product Encryption. In *Advances in Cryptology – EUROCRYPT 2010*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 138–159.

[28] OKAMOTO, T., AND TAKASHIMA, K. Adaptively attribute-hiding (hierarchical) inner product encryption. EUROCRYPT 2012, 2012. `https://eprint.iacr.org/2011/543`.

[29] OKAMOTO, T., AND TAKASHIMA, K. Fully Secure Unbounded Inner-Product and Attribute-Based Encryption. *ASIACRYPT 7658*, Chapter 22 (2012), 349–366.

[30] O'NEILL, A. Definitional Issues in Functional Encryption. *IACR Cryptology ePrint Archive* (2010).

[31] PINKAS, B., SCHNEIDER, T., SMART, N. P., AND WILLIAMS, S. C. Secure two-party computation is practical. In *International Conference on the Theory and Application of Cryptology and Information Security* (2009), Springer, pp. 250–267.

[32] POTTIE, G. J., AND KAISER, W. J. Wireless integrated network sensors. *Communications of the ACM 43*, 5 (2000), 51–58.

[33] RÈNYI, A. On measures of entropy and information. In *Proceedings of the Fourth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Contributions to the Theory of Statistics* (Berkeley, Calif., 1961), University of California Press, pp. 547–561.

[34] SHENG, B., LI, Q., MAO, W., AND JIN, W. Outlier detection in sensor networks. In *Proceedings of the 8th ACM international symposium on Mobile ad hoc networking and computing* (2007), ACM, pp. 219–228.

[35] SHIELDS, C., AND LEVINE, B. N. A protocol for anonymous communication over the internet. In *Proceedings of the 7th ACM conference on Computer and communications security* (2000), ACM, pp. 33–42.

[36] YANG, L., JAKOBSSON, M., AND WETZEL, S. Discount anonymous on demand routing for mobile ad hoc networks. In *Securecomm and Workshops, 2006* (2006), IEEE, pp. 1–10.

[37] YU, H., GIBBONS, P. B., KAMINSKY, M., AND XIAO, F. Sybillimit: A near-optimal social network defense against sybil attacks. In *2008 IEEE Symposium on Security and Privacy (sp 2008)* (2008), IEEE, pp. 3–17.

[38] YU, H., KAMINSKY, M., GIBBONS, P. B., AND FLAXMAN, A. D. SybilGuard - defending against sybil attacks via social networks. *IEEE/ACM Trans. Netw.* (2008).

[39] ZHANG, K., SHI, S., GAO, H., AND LI, J. Unsupervised outlier detection in sensor networks using aggregation tree. In *International Conference on Advanced Data Mining and Applications* (2007), Springer, pp. 158–169.

XONN: XNOR-based Oblivious Deep Neural Network Inference

M. Sadegh Riazi
UC San Diego

Mohammad Samragh
UC San Diego

Hao Chen
Microsoft Research

Kim Laine
Microsoft Research

Kristin Lauter
Microsoft Research

Farinaz Koushanfar
UC San Diego

Abstract

Advancements in deep learning enable cloud servers to provide inference-as-a-service for clients. In this scenario, clients send their raw data to the server to run the deep learning model and send back the results. One standing challenge in this setting is to ensure the privacy of the clients' sensitive data. Oblivious inference is the task of running the neural network on the client's input without disclosing the input or the result to the server. This paper introduces XONN (pronounced /zʌn/), a novel end-to-end framework based on Yao's Garbled Circuits (GC) protocol, that provides a paradigm shift in the conceptual and practical realization of oblivious inference. In XONN, the costly matrix-multiplication operations of the deep learning model are replaced with XNOR operations that are essentially free in GC. We further provide a novel algorithm that customizes the neural network such that the runtime of the GC protocol is minimized without sacrificing the inference accuracy.

We design a user-friendly high-level API for XONN, allowing expression of the deep learning model architecture in an unprecedented level of abstraction. We further provide a compiler to translate the model description from high-level Python (i.e., Keras) to that of XONN. Extensive proof-of-concept evaluation on various neural network architectures demonstrates that XONN outperforms prior art such as Gazelle (USENIX Security'18) by up to 7×, MiniONN (ACM CCS'17) by 93×, and SecureML (IEEE S&P'17) by 37×. State-of-the-art frameworks require one round of interaction between the client and the server for each layer of the neural network, whereas, XONN requires a *constant* round of interactions for *any* number of layers in the model. XONN is first to perform oblivious inference on Fitnet architectures with up to 21 layers, suggesting a new level of scalability compared with state-of-the-art. Moreover, we evaluate XONN on four datasets to perform privacy-preserving medical diagnosis. The datasets include breast cancer, diabetes, liver disease, and Malaria.

1 Introduction

The advent of big data and striking recent progress in artificial intelligence are fueling the impending industrial automation revolution. In particular, Deep Learning (DL) —a method based on learning Deep Neural Networks (DNNs) —is demonstrating a breakthrough in accuracy. DL models outperform human cognition in a number of critical tasks such as speech and visual recognition, natural language processing, and medical data analysis. Given DL's superior performance, several technology companies are now developing or already providing DL as a service. They train their DL models on a large amount of (often) proprietary data on their own servers; then, an inference API is provided to the users who can send their data to the server and receive the analysis results on their queries. The notable shortcoming of this remote inference service is that the inputs are revealed to the cloud server, breaching the privacy of sensitive user data.

Consider a DL model used in a medical task in which a health service provider withholds the prediction model. Patients submit their plaintext medical information to the server, which then uses the sensitive data to provide a medical diagnosis based on inference obtained from its proprietary model. A naive solution to ensure patient privacy is to allow the patients to receive the DL model and run it on their own trusted platform. However, this solution is not practical in real-world scenarios because: (i) The DL model is considered an essential component of the service provider's intellectual property (IP). Companies invest a significant amount of resources and funding to gather the massive datasets and train the DL models; hence, it is important to service providers not to reveal the DL model to ensure their profitability and competitive advantage. (ii) The DL model is known to reveal information about the underlying data used for training [1]. In the case of medical data, this reveals sensitive information about other patients, violating HIPAA and similar patient health privacy regulations.

Oblivious inference is the task of running the DL model on the client's input without disclosing the input or the re-

sult to the server itself. Several solutions for oblivious inference have been proposed that utilize one or more cryptographic tools such as Homomorphic Encryption (HE) [2, 3], Garbled Circuits (GC) [4], Goldreich-Micali-Wigderson (GMW) protocol [5], and Secret Sharing (SS). Each of these cryptographic tools offer their own characteristics and tradeoffs. For example, one major drawback of HE is its *computational complexity*. HE has two main variants: Fully Homomorphic Encryption (FHE) [2] and Partially Homomorphic Encryption (PHE) [3, 6]. FHE allows computation on encrypted data but is computationally very expensive. PHE has less overhead but only supports a subset of functions or depth-bounded arithmetic circuits. The computational complexity drastically increases with the circuit's depth. Moreover, non-linear functionalities such as the ReLU activation function in DL cannot be supported.

GC, on the other hand, can support an arbitrary functionality while requiring only a *constant* round of interactions regardless of the depth of the computation. However, it has a high communication cost and a significant overhead for multiplication. More precisely, performing multiplication in GC has quadratic computation and communication complexity with respect to the bit-length of the input operands. It is well-known that the complexity of the contemporary DL methodologies is dominated by matrix-vector multiplications. GMW needs less communication than GC but requires many rounds of *interactions* between the two parties.

A standalone SS-based scheme provides a computationally inexpensive multiplication yet requires three or more independent (non-colluding) computing servers, which is a strong assumption. Mixed-protocol solutions have been proposed with the aim of utilizing the best characteristics of each of these protocols [7, 8, 9, 10]. They require secure conversion of secrets from one protocol to another in the middle of execution. Nevertheless, it has been shown that the cost of secret conversion is paid off in these hybrid solutions. Roughly speaking, the number of interactions between server and client (i.e., round complexity) in existing hybrid solutions is *linear* with respect to the depth of the DL model. Since depth is a major contributor to the deep learning accuracy [11], scalability of the mixed-protocol solutions with respect to the number of layers remains an unsolved issue for more complex, many-layer networks.

This paper introduces XONN, a novel end-to-end framework which provides a paradigm shift in the conceptual and practical realization of privacy-preserving interference on deep neural networks. The existing work has largely focused on the development of customized security protocols while using conventional fixed-point deep learning algorithms. XONN, for the first time, suggests leveraging the concept of the Binary Neural Networks (BNNs) in conjunction with the GC protocol. In BNNs, the weights and activations are restricted to binary (i.e, ± 1) values, substituting the costly multiplications with simple XNOR operations during the inference phase. The XNOR operation is known to be *free* in the GC protocol [12]; therefore, performing oblivious inference on BNNs using GC results in the removal of costly multiplications. Using our approach, we show that oblivious inference on the standard DL benchmarks can be performed with minimal, if any, decrease in the prediction accuracy.

We emphasize that an effective solution for oblivious inference should take into account the deep learning algorithms and optimization methods that can tailor the DL model for the security protocol. Current DL models are designed to run on CPU/GPU platforms where many multiplications can be performed with high throughput, whereas, bit-level operations are very inefficient. In the GC protocol, however, bit-level operations are inexpensive, but multiplications are rather costly. As such, we propose to train deep neural networks that involve many bit-level operations but *no* multiplications in the inference phase; using the idea of learning binary networks, we achieve an average of $21\times$ reduction in the number of gates for the GC protocol.

We perform extensive evaluations on different datasets. Compared to the Gazelle [10] (the prior best solution) and MiniONN [9] frameworks, we achieve $7\times$ and $93\times$ lower inference latency, respectively. XONN outperforms DeepSecure [13] (prior best GC-based framework) by $60\times$ and CryptoNets [14], an HE-based framework, by $1859\times$. Moreover, our solution renders a *constant* round of interactions between the client and the server, which has a significant effect on the performance on oblivious inference in Internet settings. We highlight our contributions as follows:

- Introduction of XONN, the *first* framework for privacy preserving DNN inference with a *constant* round complexity that does not need expensive matrix multiplications. Our solution is the first that can be scalably adapted to ensure security against malicious adversaries.

- Proposing a novel conditional addition protocol based on Oblivious Transfer (OT) [15], which optimizes the costly computations for the network's input layer. Our protocol is $6\times$ faster than GC and can be of independent interest. We also devise a novel network trimming algorithm to remove neurons from DNNs that minimally contribute to the inference accuracy, further reducing the GC complexity.

- Designing a high-level API to readily automate fast adaptation of XONN, such that users only input a high-level description of the neural network. We further facilitate the usage of our framework by designing a compiler that translates the network description from Keras to XONN.

- Proof-of-concept implementation of XONN and evaluation on various standard deep learning benchmarks. To demonstrate the scalability of XONN, we perform oblivious inference on neural networks with as many as 21 layers for the first time in the oblivious inference literature.

2 Preliminaries

Throughout this paper, scalars are represented as lowercase letters ($x \in \mathbb{R}$), vectors are represented as bold lowercase letters ($\mathbf{x} \in \mathbb{R}^n$), matrices are denoted as capital letters ($X \in \mathbb{R}^{m \times n}$), and tensors of more than 2 ways are shown using bold capital letters ($\mathbf{X} \in \mathbb{R}^{m \times n \times k}$). Brackets denote element selection and the colon symbol stands for all elements —$W[i,:]$ represents all values in the i-th row of W.

2.1 Deep Neural Networks

The computational flow of a deep neural network is composed of multiple computational layers. The input to each layer is either a vector (i.e., $\mathbf{x} \in \mathbb{R}^n$) or a tensor (i.e., $\mathbf{X} \in \mathbb{R}^{m \times n \times k}$). The output of each layer serves as the input of the next layer. The input of the first layer is the raw data and the output of the last layer represents the network's prediction on the given data (i.e., inference result). In an image classification task, for instance, the raw image serves as the input to the first layer and the output of the last layer is a vector whose elements represent the probability that the image belongs to each category. Below we describe the functionality of neural network layers.

Linear Layers: Linear operations in neural networks are performed in Fully-Connected (FC) and Convolution (CONV) layers. The vector dot product (VDP) between two vectors $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{w} \in \mathbb{R}^n$ is defined as follows:

$$\mathsf{VDP}\ (\mathbf{x}, \mathbf{w}) = \sum_{i=1}^{n} \mathbf{w}[i] \cdot \mathbf{x}[i]. \tag{1}$$

Both CONV and FC layers repeat VDP computation to generate outputs as we describe next. A fully connected layer takes a vector $\mathbf{x} \in \mathbb{R}^n$ and generates the output $\mathbf{y} \in \mathbb{R}^m$ using a linear transformation:

$$\mathbf{y} = W \cdot \mathbf{x} + \mathbf{b}, \tag{2}$$

where $W \in \mathbb{R}^{m \times n}$ is the weight matrix and $\mathbf{b} \in \mathbb{R}^m$ is a bias vector. More precisely, the i-th output element is computed as $\mathbf{y}[\mathbf{i}] = \mathsf{VDP}\ (W[i,:], \mathbf{x}) + \mathbf{b}[i]$.

A convolution layer is another form of linear transformation that operates on images. The input of a CONV layer is represented as multiple rectangular channels (2D images) of the same size: $\mathbf{X} \in \mathbb{R}^{h1 \times h2 \times c}$, where $h1$ and $h2$ are the dimensions of the image and c is the number of channels. The CONV layer maps the input image into an output image $\mathbf{Y} \in \mathbb{R}^{h1' \times h2' \times f}$. A CONV layer consists of a weight tensor $\mathbf{W} \in \mathbb{R}^{k \times k \times c \times f}$ and a bias vector $\mathbf{b} \in \mathbb{R}^f$. The i-th output channel in a CONV layer is computed by sliding the kernel $\mathbf{W}[:,:,:,i] \in \mathbb{R}^{k \times k \times c}$ over the input, computing the dot product between the kernel and the windowed input, and adding the bias term $\mathbf{b}[i]$ to the result.

Non-linear Activations: The output of linear transformations (i.e., CONV and FC) is usually fed to an activation layer, which applies an element-wise non-linear transformation to the vector/tensor and generates an output with the same dimensionality. In this paper, we particularly utilize the Binary Activation (BA) function for hidden layers. BA maps the input operand to its sign value (i.e., $+1$ or -1).

Batch Normalization: A batch normalization (BN) layer is typically applied to the output of linear layers to normalize the results. If a BN layer is applied to the output of a CONV layer, it multiplies all of the i-th channel's elements by a scalar $\boldsymbol{\gamma}[i]$ and adds a bias term $\boldsymbol{\beta}[i]$ to the resulting channel. If BN is applied to the output of an FC layer, it multiplies the i-th element of the vector by a scalar $\boldsymbol{\gamma}[i]$ and adds a bias term $\boldsymbol{\beta}[i]$ to the result.

Pooling: Pooling layers operate on image channels outputted by the CONV layers. A pooling layer slides a window on the image channels and aggregates the elements within the window into a single output element. Max-pooling and Average-pooling are two of the most common pooling operations in neural networks. Typically, pooling layers reduce the image size but do not affect the number of channels.

2.2 Secret Sharing

A secret can be securely shared among two or multiple parties using Secret Sharing (SS) schemes. An SS scheme guarantees that each share does not reveal any information about the secret. The secret can be reconstructed using all (or subset) of shares. In XONN, we use additive secret sharing in which a secret S is shared among two parties by sampling a random number $\widehat{S}_1 \in_R \mathbb{Z}_{2^b}$ (integers modulo 2^b) as the first share and creating the second share as $\widehat{S}_2 = S - \widehat{S}_1 \ mod\ 2^b$ where b is the number of bits to describe the secret. While none of the shares reveal any information about the secret S, they can be used to reconstruct the secret as $S = \widehat{S}_1 + \widehat{S}_2 \ mod\ 2^b$. Suppose that two secrets $S^{(1)}$ and $S^{(2)}$ are shared among two parties where party-1 has $\widehat{S}_1^{(1)}$ and $\widehat{S}_1^{(2)}$ and party-2 has $\widehat{S}_2^{(1)}$ and $\widehat{S}_2^{(2)}$. Party-i can create a share of the sum of two secrets as $\widehat{S}_i^{(1)} + \widehat{S}_i^{(2)} \ mod\ 2^b$ without communicating to the other party. This can be generalized for arbitrary (more than two) number of secrets as well. We utilize additive secret sharing in our Oblivious Conditional Addition (OCA) protocol (Section 3.3).

2.3 Oblivious Transfer

One of the most crucial building blocks of secure computation protocols, e.g., GC, is the Oblivious Transfer (OT) protocol [15]. In OT, two parties are involved: a sender and a receiver. The sender holds n different messages $m_j,\ j = 1...n$, with a specific bit-length and the receiver holds an index (ind) of a message that she wants to receive. At the end of the protocol, the receiver gets m_{ind} with no additional knowledge about the other messages and the sender learns nothing about the selection index. In GC, 1-out-of-2 OT is used where $n = 2$ in which case the selection index is only one bit. The initial realizations of OT required costly public key

encryptions for each run of the protocol. However, the OT Extension [16, 17, 18] technique enables performing OT using more efficient symmetric-key encryption in conjunction with a *fixed* number of base OTs that need public-key encryption. OT is used both in the OCA protocol as well as the Garbled Circuits protocol which we discuss next.

2.4 Garbled Circuits

Yao's Garbled Circuits [4], or GC in short, is one of the generic two-party secure computation protocols. In GC, the result of an arbitrary function $f(.)$ on inputs from two parties can be computed without revealing each party's input to the other. Before executing the protocol, function $f(.)$ has to be described as a Boolean circuit with two-input gates.

GC has three main phases: garbling, transferring data, and evaluation. In the first phase, only one party, the Garbler, is involved. The Garbler starts by assigning two randomly generated l-bit binary strings to each wire in the circuit. These binary strings are called *labels* and they represent semantic values 0 and 1. Let us denote the label of wire w corresponding to the semantic value x as L_x^w. For each gate in the circuit, the Garbler creates a four-row garbled table as follows. Each label of the output wire is encrypted using the input labels according to the truth table of the gate. For example, consider an AND gate with input wires a and b and output wire c. The last row of the garbled table is the encryption of L_1^c using labels L_1^a and L_1^b.

Once the garbling process is finished, the Garbler sends all of the garbled tables to the Evaluator. Moreover, he sends the correct labels that correspond to input wires that represent his inputs to the circuit. For example, if wire w^* is the first input bit of the Garbler and his input is 0, he sends L_0^*. The Evaluator acquires the labels corresponding to her input through 1-out-of-2 OT where Garbler is the sender with two labels as his messages and the Evaluator's selection bit is her input for that wire. Having all of the garbled tables and labels of input wires, the Evaluator can start decrypting the garbled tables one by one until reaching the final output bits. She then learns the plaintext result at the end of the GC protocol based on the output labels and their relationships to the semantic values that are received from the Garbler.

3 The XONN Framework

In this section, we explain how neural networks can be trained such that they incur a minimal cost during the oblivious inference. The most computationally intensive operation in a neural network is matrix multiplication. In GC, each multiplication has a *quadratic* computation and communication cost with respect to the input bit-length. This is the major source of inefficiency in prior work [13]. We overcome this limitation by changing the learning process such that the trained neural network's weights become binary. As a result, costly multiplication operations are replaced with XNOR gates which are essentially free in GC. We describe the training process in Section 3.1. In Section 3.2, we explain the operations and their corresponding Boolean circuit designs that enable a very fast oblivious inference. In Section 4, we elaborate on XONN implementation.

3.1 Customized Network Binarization

Numerical optimization algorithms minimize a specific cost function associated with neural networks. It is well-known that neural network training is a non-convex optimization, meaning that there exist many locally-optimum parameter configurations that result in similar inference accuracies. Among these parameter settings, there exist solutions where both neural network parameters and activation units are restricted to take binary values (i.e., either $+1$ or -1); these solutions are known as Binary Neural Netowrks (BNNs) [19].

One major shortcoming of BNNs is their (often) low inference accuracy. In the machine learning community, several methods have been proposed to modify BNN functionality for accuracy enhancement [20, 21, 22]. These methods are devised for *plaintext* execution of BNNs and are not efficient for oblivious inference with GC. We emphasize that, when modifying BNNs for accuracy enhancement, one should also take into account the implications in the corresponding GC circuit. With this in mind, we propose to modify the number of channels and neurons in CONV and FC layers, respectively. Increasing the number of channels/neurons leads to a higher accuracy but it also increases the complexity of the corresponding GC circuit. As a result, XONN provides a trade-off between the accuracy and the communication/runtime of the oblivious inference. This tradeoff enables cloud servers to customize the complexity of the GC protocol to optimally match the computation and communication requirements of the clients. To customize the BNN, XONN configures the per-layer number of neurons in two steps:

- Linear Scaling: Prior to training, we scale the number of channels/neurons in all BNN layers with the same factor (s), e.g., $s = 2$. Then, we train the scaled BNN architecture.
- Network Trimming: Once the (uniformly) scaled network is trained, a post-processing algorithm removes redundant channels/neurons from each hidden layer to reduce the GC cost while maintaining the inference accuracy.

Figure 1 illustrates the BNN customization method for an example baseline network with four hidden layers. Network trimming (pruning) consists of two steps, namely, Feature Ranking and Iterative Pruning which we describe next.

Feature Ranking: In order to perform network trimming, one needs to sort the channels/neurons of each layer based on their contribution to the inference accuracy. In conventional neural networks, simple ranking methods sort features based

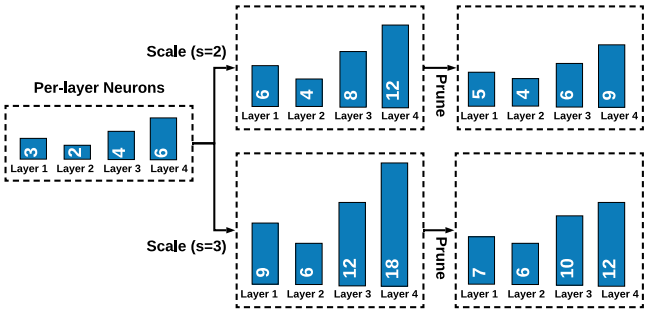


Figure 1: Illustration of BNN customization. The bars represent the number of neurons in each hidden layer.

on absolute value of the neurons/channels [23]. In BNNs, however, the weights/features are either $+1$ or -1 and the absolute value is not informative. To overcome this issue, we utilize first order Taylor approximation of neural networks and sort the features based on the magnitude of the gradient values [24]. Intuitively, the gradient with respect to a certain feature determines its importance; a high (absolute) gradient indicates that removing the neuron has a destructive effect on the inference accuracy. Inspired by this notion, we develop a feature ranking method described in Algorithm 1.

Iterative Pruning: We devise a step-by-step algorithm for model pruning which is summarized in Algorithm 2. At each step, the algorithm selects one of the BNN layers l^* and removes the first p^* features with the lowest importance (line 17). The selected layer l^* and the number of pruned neurons p^* maximize the following reward (line 15):

$$reward(l,p) = \frac{c_{curr} - c_{next}}{e^{a_{curr} - a_{next}}}, \tag{3}$$

where c_{curr} and c_{next} are the GC complexity of the BNN before and after pruning, whereas, a_{curr} and a_{next} denote the corresponding validation accuracies. The numerator of this reward encourages higher reduction in the GC cost while the denominator penalizes accuracy loss. Once the layer is pruned, the BNN is fine-tuned to recover the accuracy (line 18). The pruning process stops once the accuracy drops below a pre-defined threshold.

3.2 Oblivious Inference

BNNs are trained such that the weights and activations are binarized, i.e., they can only have two possible values: $+1$ or -1. This property allows BNN layers to be rendered using a simplified arithmetic. In this section, we describe the functionality of different layer types in BNNs and their Boolean circuit translations. Below, we explain each layer type.

Binary Linear Layer: Most of the computational complexity of neural networks is due to the linear operations in CONV and FC layers. As we discuss in Section 2.1, linear operations are realized using vector dot product (VDP). In BNNs, VDP operations can be implemented using simplified circuits. We categorize the VDP operations of this work into

Algorithm 1 XONN Channel Sorting for CONV Layers

Inputs: Trained BNN with loss function $\mathscr{L}$, CONV layer l with output shape of $h1 \times h2 \times f$, subsampled validation data and labels $\{(\mathbf{X_1}, z_1), \ldots, (\mathbf{X_k}, z_k)\}$
Output: Indices of the sorted channels: $\{i_0, \ldots, i_f\}$

1: $\mathbf{G} \leftarrow zeros(k \times h1 \times h2 \times f)$ ▷ define gradient tensor
2: **for** $i = 1, \ldots, k$ **do**
3: $\quad \mathscr{L} = \mathscr{L}(\mathbf{X_i}, z_i)$ ▷ evaluate loss function
4: $\quad \nabla_Y = \frac{\partial \mathscr{L}}{\partial Y^l}$ ▷ compute gradient w.r.t. layer output
5: $\quad \mathbf{G}[i,:,:,:] \leftarrow \nabla_Y$ ▷ store gradient
6: **end for**
7: $\mathbf{G_{abs}} \leftarrow |\mathbf{G}|$ ▷ take elementwise absolute values
8: $\mathbf{g_s} \leftarrow zeros(f)$ ▷ define sum of absolute values
9: **for** $i = 1, \ldots, f$ **do**
10: $\quad \mathbf{g_s}[i] \leftarrow sum(\mathbf{G_{abs}}[:,:,:,i])$
11: **end for**
12: $\{i_0, \ldots, i_f\} \leftarrow sort(\mathbf{g_s})$
13: **return** $\{i_0, \ldots, i_f\}$

two classes: (i) Integer-VDP where only one of the vectors is binarized and the other has integer elements and (ii) Binary-VDP where both vectors have binary (± 1) values.

Integer-VDP: For the first layer of the neural network, the server has no control over the input data which is not necessarily binarized. The server can only train binary weights and use them for oblivious inference. Consider an input vector $\mathbf{x} \in \mathbb{R}^n$ with integer (possibly fixed-point) elements and a weight vector $\mathbf{w} \in \{-1,1\}^n$ with binary values. Since the elements of the binary vector can only take $+1$ or -1, the Integer-VDP can be rendered using additions and subtractions. In particular, the binary weights can be used in a selection circuit that decides whether the pertinent integer input should be added to or subtracted from the VDP result.

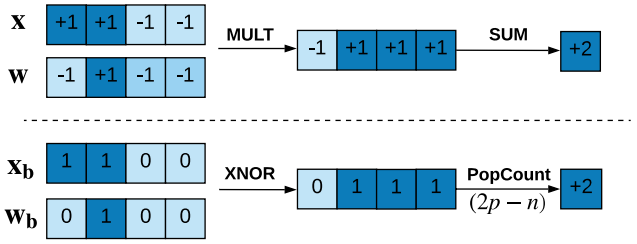


Figure 2: Equivalence of Binary-VDP and **XnorPopcount**.

Binary-VDP: Consider a dot product between two binary vectors $\mathbf{x} \in \{-1,+1\}^n$ and $\mathbf{w} \in \{-1,+1\}^n$. If we encode each element with one bit (i.e., $-1 \rightarrow 0$ and $+1 \rightarrow 1$), we obtain binary vectors $\mathbf{x}_b \in \{0,1\}^n$ and $\mathbf{w}_b \in \{0,1\}^n$. It has been shown that the dot product of $\mathbf{x}$ and $\mathbf{w}$ can be efficiently computed using an *XnorPopcount* operation [19]. Figure 2 depicts the equivalence of VDP$(\mathbf{x},\mathbf{w})$ and

Algorithm 2 XONN Iterative BNN Pruning

Inputs: Trained BNN with n overall CONV and FC layers, minimum accuracy threshold θ, number of pruning trials per layer t, subsampled validation data and labels $data_V$, training data and labels $data_T$
Output: BNN with pruned layers

$\mathbf{p} \leftarrow zeros(n-1)$ ▷ current number of pruned neurons/channels per layer
$a_{curr} \leftarrow Accuracy(BNN, data_V|\mathbf{p})$ ▷ current BNN validation accuracy
$c_{curr} \leftarrow Cost(BNN|\mathbf{p})$ ▷ current GC cost
while $a_{curr} > \theta$ **do** ▷ repeat until accuracy drops below θ
  for $l = 1, \ldots, n-1$ **do** ▷ search over all layers
    $inds \leftarrow Rank(BNN, l, data_V)$ ▷ rank features via Algorithm 1
    $f \leftarrow$ Number of neurons/channels ▷ number of output neurons/channels
    for $p = \mathbf{p}[l], \mathbf{p}[l] + \frac{f}{t}, \ldots, f$ **do** ▷ search over possible pruning rates
      $BNN_{next} \leftarrow Prune(BNN, l, p, inds)$ ▷ prune p features with lowest ranks from the l-th layer
      $a_{next} \leftarrow Accuracy(BNN_{next}, data_V|\mathbf{p}[\mathbf{1}], \ldots, \mathbf{p}[\mathbf{l}] = p, \ldots, \mathbf{p}[\mathbf{n-1}])$ ▷ validation accuracy if pruned
      $c_{next} \leftarrow Cost(BNN_{next}|\mathbf{p}[\mathbf{1}], \ldots, \mathbf{p}[\mathbf{l}] = p, \ldots, \mathbf{p}[\mathbf{n-1}])$ ▷ GC cost if pruned
      $reward(l,p) = \frac{c_{curr} - c_{next}}{e^{(a_{curr} - a_{next})}}$ ▷ compute reward given that p features are pruned from layer l
    end for
  end for
  $\{l^*, p^*\} \leftarrow \arg\max_{l,p} reward(l,p)$ ▷ select layer l^* and pruning rate p^* that maximize the reward
  $\mathbf{p}[\mathbf{l}^*] \leftarrow p^*$ ▷ update the number of pruned features in vector $\mathbf{p}$
  $BNN \leftarrow Prune(BNN, l^*, p^*, inds)$ ▷ prune p^* features with lowest ranks from the l^*-th layer
  $BNN \leftarrow Fine\text{-}tune(BNN, data_T)$ ▷ fine-tune the pruned model using training data to recover accuracy
  $a_{curr} \leftarrow Accuracy(BNN, data_V|\mathbf{p})$ ▷ update current BNN validation accuracy
  $c_{curr} \leftarrow Cost(BNN|\mathbf{p})$ ▷ update current GC cost
end while
return BNN

$XnorPopcount(\mathbf{x}_b, \mathbf{w}_b)$ for a VDP between 4-dimensional vectors. First, element-wise XNOR operations are performed between the two binary encodings. Next, the number of set bits p is counted, and the output is computed as $2p-n$.

Binary Activation Function: A Binary Activation (BA) function takes input x and maps it to $y = Sign(x)$ where $Sign(\cdot)$ outputs either $+1$ or -1 based on the sign of its input. This functionality can simply be implemented by extracting the most significant bit of x.

Binary Batch Normalization: in BNNs, it is often useful to normalize feature x using a Batch Normalization (BN) layer before applying the binary activation function. More specifically, a BN layer followed by a BA is equivalent to:

$$y = Sign(\gamma \cdot x + \beta) = Sign(x + \frac{\beta}{\gamma}),$$

since γ is a positive value. The combination of the two layers (BN+BA) is realized by a comparison between x and $-\frac{\beta}{\gamma}$.

Binary Max-Pooling: Assuming the inputs to the max-pooling layers are binarized, taking the maximum in a window is equivalent to performing logical OR over the binary encodings as depicted in Figure 3. Note that average-pooling layers are usually not used in BNNs since the average of multiple binary elements is no longer a binary value.

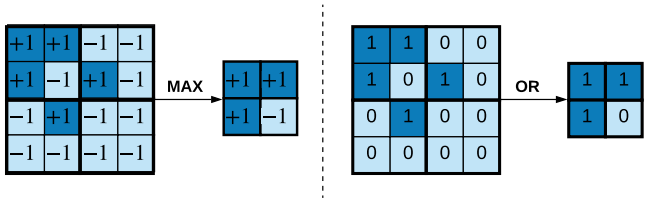


Figure 3: The equivalence between Max-Pooling and Boolean-OR operations in BNNs.

Figure 4 demonstrates the Boolean circuit for Binary-VDP followed by BN and BA. The number of non-XOR gates for binary-VDP is equal to the number of gates required to render the tree-adder structure in Figure 4. Similarly, Figure 5 shows the Integer-VDP counterpart. In the first level of the tree-adder of Integer-VDP (Figure 5), the binary weights determine whether the integer input should be added to or subtracted from the final result within the "Select" circuit. The next levels of the tree-adder compute the result of the integer-VDP using "Adder" blocks. The combination of BN and BA is implemented using a single *comparator*. Compared to Binary-VDP, Integer-VDP has a high garbling cost which is linear with respect to the number of bits. To mitigate this problem, we propose an alternative solution based on Oblivious Transfer (OT) in Section 3.3.

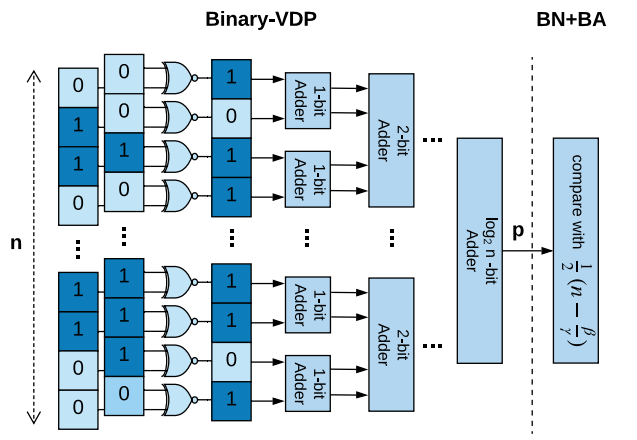


Figure 4: Circuit for binary-VDP followed by comparison for batch normalization (BN) and binary activation (BA).

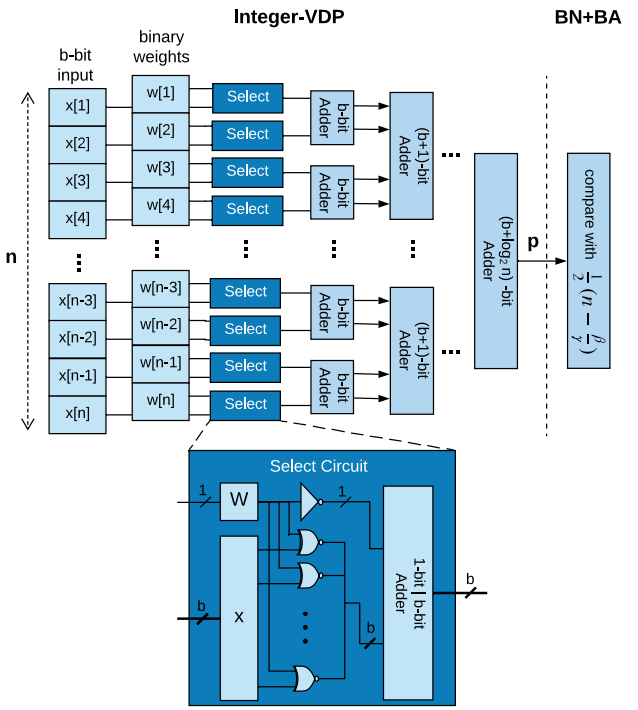


Figure 5: Circuit for Integer-VDP followed by comparison for batch normalization (BN) and binary activation (BN).

3.3 Oblivious Conditional Addition Protocol

In XONN, all of the activation values as well as neural network weights are binary. However, the input to the neural network is provided by the user and is not necessarily binary. The first layer of a typical neural network comprises either an FC or a CONV layer, both of which are evaluated using oblivious Integer-VDP. On the one side, the user provides her input as non-binary (integer) values. On the other side, the network parameters are binary values representing -1 and 1. We now demonstrate how Integer-VDP can be described as an OT problem. Let us denote the user's input as a vector $\mathbf{v_1}$ of n (b-bit) integers. The server holds a vector of n binary values denoted by $\mathbf{v_2}$. The result of Integer-VDP is a number "y" that can be described with

$$b' = \left\lceil \log_2(n \cdot (2^b - 1)) \right\rceil$$

bits. Figure 6 summarizes the steps in the OCA protocol. The first step is to *bit-extend* $\mathbf{v_1}$ from b-bit to b'-bit. In other words, if $\mathbf{v_1}$ is a vector of *signed* integer/fixed-point numbers, the most significant bit should be repeated $(b'-b)$-many times, otherwise, it has to be zero-padded for most significant bits. We denote the bit-extended vector by $\mathbf{v_1^*}$. The second step is to create the two's complement vector of $\mathbf{v_1^*}$, called $\overline{\mathbf{v_1^*}}$. The client also creates a vector of n (b'-bit) randomly generated numbers, denoted as $\mathbf{r}$. She computes element-wise vector subtractions $\mathbf{v_1^*} - \mathbf{r} \ mod \ 2^{b'}$ and $\overline{\mathbf{v_1^*}} - \mathbf{r} \ mod \ 2^{b'}$. These two vectors are n-many pair of messages that will be used as input to n-many 1-out-of-two OTs. More precisely, $\overline{\mathbf{v_1^*}} - \mathbf{r} \ mod \ 2^{b'}$ is a list of first messages and $\mathbf{v_1^*} - \mathbf{r} \ mod \ 2^{b'}$ is a list of second messages. The server's list of selection bits is $\mathbf{v_2}$. After n-many OTs are finished, the server has a list of n transferred numbers called $\mathbf{v_t}$ where

$$\mathbf{v_t}[i] = \begin{cases} \overline{\mathbf{v_1^*}}[i] - \mathbf{r}[i] \ mod \ 2^{b'} & if \ \ \mathbf{v_2}[i] = 0 \\ \mathbf{v_1^*}[i] - \mathbf{r}[i] \ mod \ 2^{b'} & if \ \ \mathbf{v_2}[i] = 1 \end{cases} \quad i = 1, \ \ldots \ , n.$$

Finally, the client computes $y_1 = \sum_{i=1}^{n} \mathbf{r}[i] \ mod \ 2^{b'}$ and the server computes $y_2 = \sum_{i=1}^{n} \mathbf{v_t}[i] \ mod \ 2^{b'}$. By OT's definition, the receiver (server) gets only one of the two messages from the sender. That is, based on each selection bit (a binary weight), the receiver gets an additive share of either the sender's number or its two's complement. Upon adding all of the received numbers, the receiver computes an additive share of the Integer-VDP result. Now, even though the sender does not know which messages were selected by the receiver, she can add all of the randomly generated numbers $\mathbf{r}[i]$s which is equal to the other additive share of the Integer-VDP result. Since all numbers are described in the two's complement format, subtractions are equivalent to the addition of the two's complement values, which are created by the sender at the beginning of OCA. Moreover, it is possible that as we accumulate the values, the bit-length of the final Integer-VDP result grows accordingly. This is supported due to the bit-extension process at the beginning of the protocol. In other words, all additions are performed in a larger ring such that the result does not overflow.

Note that all numbers belong to the ring $\mathbb{Z}_{2^{b'}}$ and by definition, a ring is closed under addition, therefore, y_1 and y_2 are true additive shares of $y = y_1 + y_2 \ mod \ 2^{b'}$. We described the OCA protocol for one Integer-VDP computation. As we outlined in Section 3.2, all linear operations in the first layer of the DL model (either FC or CONV) can be formulated as a series of Integer-VDPs.

Sender:
(1) Bit-extends all elements of $\mathbf{v_1}$ and creates $\mathbf{v_1^*}$
(2) Creates two's complement of $\mathbf{v_1^*}$: $\overline{\mathbf{v_1^*}}$
(3) Creates random vector $\mathbf{r}$: same size as $\overline{\mathbf{v_1^*}}$
(4) Creates list of first messages as $\mathbf{m^2} = \overline{\mathbf{v_1^*}} - \mathbf{r}$ *mod* $2^{b'}$
(5) Creates list of second messages as $\mathbf{m^1} = \mathbf{v_1^*} - \mathbf{r}$ *mod* $2^{b'}$
Sender & Receiver:
(6) Parties engage in Oblivious Transfer (OT)
Sender puts $\mathbf{m^1}$ and $\mathbf{m^2}$ as message vectors
Receiver puts $\mathbf{v_2}$ vector as selection bits
Receiver:
(7) Gets vector $\mathbf{v_t}$ where:

$$\mathbf{v_t}[i] = \begin{cases} \overline{\mathbf{v_1^*}}[i] - \mathbf{r}[i] \ mod \ 2^{b'} \ (\text{if } \mathbf{v_2}[i] = 0) \\ \mathbf{v_1^*}[i] - \mathbf{r}[i] \ mod \ 2^{b'} \ (\text{if } \mathbf{v_2}[i] = 1) \end{cases}$$

Sender:
(8) Computes her additive share of VDP result as:

$$y_1 = \sum_{i=1}^{n} \mathbf{r}[i] \ mod \ 2^{b'}$$

Receiver:
(9) Computes his additive share of VDP result as:

$$y_2 = \sum_{i=1}^{n} \mathbf{v_t}[i] \ mod \ 2^{b'}$$

Figure 6: Oblivious Conditional Addition (OCA) protocol.

In traditional OT, public-key encryption is needed for each OT invocation which can be computationally expensive. Thanks to the Oblivious Transfer Extension technique [16, 17, 18], one can perform many OTs using symmetric-key encryption and only a fixed number of public-key operations.

Required Modification to the Next Layer. So far, we have shown how to perform Integer-VDP using OT. However, we need to add an "addition" layer to reconstruct the true value of y from its additive shares before further processing it. The overhead of this layer, as well as OT computations, are discussed next. Note that OCA is used only for the first layer and it does not change the overall constant round complexity of XONN since it is performed only once regardless of the number of layers in the DL model.

Comparison to Integer-VDP in GC. Table 1 shows the computation and communication costs for two approaches: (i) computing the first layer in GC and (ii) utilizing OCA. OCA removes the GC cost of the first layer in XONN. However, it adds the overhead of a set of OTs and the GC costs associated with the new ADD layer.

Table 1: Computation and communication cost of OCA.

Costs {Sender, Receiver}	**GC**	**OCA**	
		OT	ADD Layer
Comp. (AES ops)	$(n+1)\cdot b \cdot \{2,4\}$	$n \cdot \{1,2\}$	$b' \cdot \{2,4\}$
Comm. (bit)	$(n+1)\cdot b \cdot 2 \cdot 128$	$n \cdot b$	$b' \cdot 2 \cdot 128$

3.4 Security of XONN

We consider the Honest-but-Curious (HbC) adversary model consistent with all of the state-of-the-art solutions for oblivious inference [7, 8, 9, 10, 13, 25]. In this model, neither of the involved parties is trusted but they are assumed to follow the protocol. Both server and client cannot infer any information about the other party's input from the entire protocol transcript. XONN relies solely on the GC and OT protocols, both of which are proven to be secure in the HbC adversary model in [26] and [15], respectively. Utilizing binary neural networks does not affect GC and OT protocols in any way. More precisely, we have changed the function $f(.)$ that is evaluated in GC such that it is more efficient for the GC protocol: drastically reducing the number of AND gates and using XOR gates instead. Our novel Oblivious Conditional Addition (OCA) protocol (Section 3.3) is also based on the OT protocol. The sender creates a list of message pairs and puts them as input to the OT protocol. Each message is an additive share of the sender's private data from which the secret data cannot be reconstructed. The receiver puts a list of selection bits as input to the OT. By OT's definition, the receiver learns nothing about the unselected messages and the sender does not learn the selection bits.

During the past few years, several attacks have been proposed that extract some information about the DL model by querying the server many times [1, 27, 28]. It has been shown that some of these attacks can be effective in the black-box setting where the client only receives the prediction results and does not have access to the model. Therefore, considering the definition of an oblivious inference, these type of attacks are out of the scope of oblivious inference frameworks. However, in Appendix B, we show how these attacks can be thwarted by adding a simple layer at the end of the neural network which adds a negligible overhead.

Security Against Malicious Adversaries. The HbC adversary model is the standard security model in the literature. However, there are more powerful security models such as security against covert and malicious adversaries. In the malicious security model, the adversary (either the client or server) can deviate from the protocol at any time with the goal of learning more about the input from the other party. One of the main distinctions between XONN and the state-of-the-art solutions is that XONN can be automatically adapted to the malicious security using cut-and-choose techniques [29, 30, 31]. These methods take a GC protocol in HbC and readily extend it to the malicious security model. This modification increases the overhead but enables a higher security level. To the best of our knowledge, there is no practical solution to extend the customized mixed-protocol frameworks [7, 9, 10, 25] to the malicious security model. Our GC-based solution is more efficient compared to the mixed-protocol solutions and can be upgraded to the malicious security at the same time.

4 The XONN Implementation

In this section, we elaborate on the garbling/evaluation implementation of XONN. All of the optimizations and techniques proposed in this section do not change the security or correctness in anyway and only enable the framework's scalability for large network architectures.

We design a new GC framework with the following design principles in mind: (i) *Efficiency:* XONN is designed to have a minimal data movement and low cache-miss rate. (ii) *Scalability:* oblivious inference inevitably requires significantly higher memory usage compared to plaintext evaluation of neural networks. High memory usage is one critical shortcoming of state-of-the-art secure computation frameworks. As we show in our experimental results, XONN is designed to scale for very deep neural networks that have higher accuracy compared to networks considered in prior art. (iii) *Modularity:* our framework enables users to create Boolean description of different layers separately. This allows the hardware synthesis tool to generate more optimized circuits as we discuss in Section 4.1. (iv) *Ease-to-use:* XONN provides a very simple API that requires few lines of neural network description. Moreover, we have created a compiler that takes a Keras description and automatically creates the network description for XONN API.

XONN is written in C++ and supports all major GC optimizations proposed previously. Since the introduction of GC, many optimizations have been proposed to reduce the computation and communication complexity of this protocol. Bellare et al. [32] have provided a way to perform garbling using efficient fixed-key AES encryption. Our implementation benefits from this optimization by using Intel AES-NI instructions. Row-reduction technique [33] reduces the number of garbled tables from four to three. Half-Gates technique [34] further reduces the number of rows in the garbled tables from three to two. One of the most influential optimizations for the GC protocol is the *free-XOR* technique [12] which makes XOR, XNOR, and NOT almost free of cost. Our implementation for Oblivious Transfer (OT) is based on libOTe [35].

4.1 Modular Circuit Synthesis and Garbling

In XONN, each layer is described as multiple invocations of a *base* circuit. For instance, linear layers (CONV and FC) are described by a VDP circuit. MaxPool is described by an OR circuit where the number of inputs is the window size of the MaxPool layer. BA/BN layers are described using a comparison (CMP) circuit. The memory footprint is significantly reduced in this approach: we only create and store the base circuits. As a result, the connection between two invocations of two different base circuits is handled at the software level.

We create the Boolean circuits using TinyGarble [36] hardware synthesis approach. TinyGarble's technology libraries are optimized for GC and produce circuits that have low number of non-XOR gates. Note that the Boolean circuit description of the contemporary neural networks comprises between millions to billions of Boolean gates, whereas, synthesis tools cannot support circuits of this size. However, due to XONN modular design, one can synthesize each base circuit separately. Thus, the bottleneck transfers from the synthesis tool's maximum number of gates to the system's memory. As such, XONN effectively scales for any neural network complexity regardless of the limitations of the synthesis tool as long as enough memory (i.e., RAM) is available. Later in this section, we discuss how to increase the scalability by dynamically managing the allocated memory.

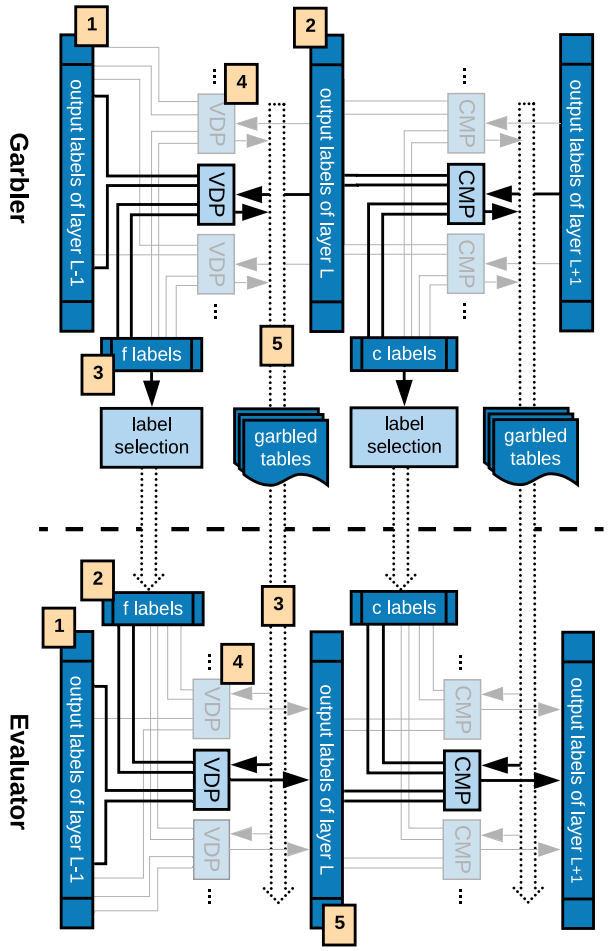


Figure 7: XONN modular and pipelined garbling engine.

Pipelined GC Engine. In XONN, computation and communication are pipelined. For instance, consider a CONV layer followed by an activation layer. We garble/evaluate these layers by multiple invocations of the VDP and CMP circuits (one invocation per output neuron) as illustrated in Figure 7. Upon finishing the garbling process of layer $L-1$, the Garbler starts garbling the Lth layer and creates the random labels for output wires of layer L. He also needs to create the random labels associated with his input (i.e., the weight

parameters) to layer L. Given a set of input and output labels, Garbler generates the garbled tables, and sends them to the Evaluator as soon as one is ready. He also sends one of the two input labels for his input bits. At the same time, the Evaluator has computed the output labels of the $(L-1)^{th}$ layer. She receives the garbled tables as well as the Garbler's selected input labels and decrypts the tables and stores the output labels of layer L.

Dynamic Memory Management. We design the framework such that the allocated memory for the labels is released as soon as it is no longer needed, reducing the memory usage significantly. For example, without our dynamic memory management, the Garbler had to allocate 10.41GB for the labels and garbled tables for the entire garbling of BC1 network (see Section 7 for network description). In contrast, in our framework, the size of memory allocation never exceeds 2GB and is less than 0.5GB for most of the layers.

4.2 Application Programming Interface (API)

XONN provides a simplified and easy-to-use API for oblivious inference. The framework accepts a high-level description of the network, parameters of each layer, and input structure. It automatically computes the number of invocations and the interconnection between all of the base circuits. Figure 8 shows the complete network description that a user needs to write for a sample network architecture (the BM3 architecture, see Section 7). All of the required circuits are automatically generated using TinyGarble [36] synthesis libraries. It is worth mentioning that for the task of oblivious inference, our API is much simpler compared to the recent *high-level* EzPC framework [25]. For example, the required lines of code to describe BM1, BM2, and BM3 network architectures (see Section 7) in EzPC are 78, 88, and 154, respectively. In contrast, they can be described with only 6, 6, and 10 lines of code in our framework.

```
1  INPUT 28 1 8
2  CONV 5 16 1 0 OCA
3  ACT
4  MAXPOOL 2
5  CONV 5 16 1 0
6  ACT
7  MAXPOOL 2
8  FC 100
9  ACT
10 FC 10
```

```
Description:

INPUT #input_feature #channels #bit-length

CONV #filter_size #filters #stride
       #Pad #OCA (optional)

MAXPOOl #window_size

FC #output_neurons
```

Figure 8: Sample snippet code in XONN.

Keras to XONN Translation. To further facilitate the adaptation of XONN, a compiler is created to translate the description of the neural network in Keras [37] to the XONN format. The compiler creates the `.xonn` file and puts the network parameters into the required format (`HEX` string) to be read by the framework during the execution of the GC protocol. All of the parameter adjustments are also automatically performed by the compiler.

5 Related Work

CryptoNets [14] is one of the early solutions that suggested the adaptation of Leveled Homomorphic Encryption (LHE) to perform oblivious inference. LHE is a variant of Partially HE that enables evaluation of depth-bounded arithmetic circuits. DeepSecure [13] is a privacy-preserving DL framework that relies on the GC protocol. CryptoDL [38] improves upon CryptoNets [14] and proposes more efficient approximation of the non-linear functions using low-degree polynomials. Their solution is based on LHE and uses mean-pooling in replacement of the max-pooling layer. Chou et al. propose to utilize the sparsity within the DL model to accelerate the inference [39].

SecureML [8] is a privacy-preserving machine learning framework based on homomorphic encryption, GC, and secret sharing. SecureML also uses customized activation functions and supports privacy-preserving training in addition to inference. Two non-colluding servers are used to train the DL model where each client XOR-shares her input and sends the shares to both servers. MiniONN [9] is a mixed-protocol framework for oblivious inference. The underlying cryptographic protocols are HE, GC, and secret sharing.

Chameleon [7] is a more recent mixed-protocol framework for machine learning, i.e., Support Vector Machines (SVMs) as well as DNNs. Authors propose to perform low-depth non-linear functions using the Goldreich-Micali-Wigderson (GMW) protocol [5], high-depth functions by the GC protocol, and linear operations using additive secret sharing. Moreover, they propose to use correlated randomness to more efficiently compute linear operations. EzPC [25] is a secure computation framework that enables users to write high-level programs and translates it to a protocol-based description of both Boolean and Arithmetic circuits. The backend cryptographic engine is based on the ABY framework.

Shokri and Shmatikov [40] proposed a solution for privacy-preserving collaborative deep learning where the training data is distributed among many parties. Their approach, which is based on *differential privacy*, enables clients to train their local model on their own training data and update the central model's parameters held by a central server. However, it has been shown that a malicious client can learn significant information about the other client's private data [41]. Google [42] has recently introduced a new approach for securely aggregating the parameter updates from multiple users. However, none of these approaches [40, 42] study the oblivious inference problem. An overview of related frameworks is provided in [43, 44].

Frameworks such as ABY^3 [45] and SecureNN [46] have different computation models and they rely on three (or four) parties during the oblivious inference. In contrast, XONN does not require an additional server for the computation. In E2DM framework [47], the model owner can encrypt and outsource the model to an untrusted server to perform obliv-

ious inference. Concurrently and independently of ours, in TAPAS [48], Sanyal et al. study the binarization of neural networks in the context of oblivious inference. They report inference latency of 147 seconds on MNIST dataset with 98.6% prediction accuracy using custom CNN architecture. However, as we show in Section 7 (BM3 benchmark), XONN outperforms TAPAS by close to *three orders of magnitude*.

Gazelle [10] is the previously most efficient oblivious inference framework. It is a mixed-protocol approach based on additive HE and GC. In Gazelle, convolution operations are performed using the packing property of HE. In this approach, many numbers are packed inside a single ciphertext for faster convolutions. In Section 6, we briefly discuss one of the essential requirements that the Gazelle protocol has to satisfy in order to be secure, namely, *circuit privacy*.

High-Level Comparison. In contrast to prior work, we propose a DL-secure computation co-design approach. To the best of our knowledge, DeepSecure [13] is the only solution that preprocesses the data and network before the secure computation protocol. However, this preprocessing step is unrelated to the underlying cryptographic protocol and compacts the network and data. Moreover, in this mode, some information about the network parameters and structure of data is revealed. Compared to mixed-protocol solutions, not only XONN provides a more efficient solution but also maintains the *constant* round complexity regardless of the number of layers in the neural network model. It has been shown that round complexity is one of the important criteria in designing secure computation protocols [49] since the performance can significantly be reduced in Internet settings where the network latency is high. Another important advantage of our solution is the ability to upgrade to the security against malicious adversaries using cut-and-choose techniques [29, 30, 31]. As we show in Section 7, XONN outperforms all previous solutions in inference latency. Table 2 summarizes a high-level comparison between state-of-the-art oblivious inference frameworks.

Table 2: High-Level Comparison of oblivious inference frameworks. "**C**"onstant round complexity. "**D**"eep learning/secure computation co-design. "**I**"ndependence of secondary server. "**U**"pgradeable to malicious security using standard solutions. "**S**"upporting any non-linear layer.

Framework	Crypto. Protocol	C	D	I	U	S
CryptoNets [14]	HE	✓	✗	✓	✗	✗
DeepSecure [13]	GC	✓	✓	✓	✓	✓
SecureML [8]	HE, GC, SS	✗	✗	✗	✗	✗
MiniONN [9]	HE, GC, SS	✗	✗	✓	✗	✓
Chameleon [7]	GC, GMW, SS	✗	✗	✗	✗	✓
EzPC [25]	GC, SS	✗	✗	✓	✗	✓
Gazelle [10]	HE, GC, SS	✗	✗	✓	✗	✓
XONN (This work)	GC, SS	✓	✓	✓	✓	✓

6 Circuit Privacy

In Gazelle [10], for each linear layer, the protocol starts with a vector $\mathbf{m}$ that is secret-shared between client $\mathbf{m_1}$ and server $\mathbf{m_2}$ ($\mathbf{m} = \mathbf{m_1} + \mathbf{m_2}$). The protocol outputs the secret shares of the vector $\mathbf{m}' = A \cdot \mathbf{m}$ where A is a matrix known to the server but not to the client. The protocol has the following procedure: (i) Client generates a pair (pk, sk) of public and secret keys of an additive homomorphic encryption scheme HE. (ii) Client sends $\mathsf{HE}.Enc_{pk}(\mathbf{m_1})$ to the server. Server adds its share ($\mathbf{m_2}$) to the ciphertext and recovers encryption of $\mathbf{m}$: $\mathsf{HE}.Enc_{pk}(\mathbf{m})$. (iii) Server homomorphically evaluates the multiplication with A and obtains the encryption of $\mathbf{m}'$. (iv) Server secret shares $\mathbf{m}'$ by sampling a random vector $\mathbf{r}$ and returns ciphertext $\mathbf{c} = \mathsf{HE}.Enc_{pk}(\mathbf{m}' - \mathbf{r})$ to the client. The client can decrypt $\mathbf{c}$ using private key sk and obtain $\mathbf{m}' - \mathbf{r}$.

Gazelle uses the Brakerski-Fan-Vercauteren (BFV) scheme [50, 51]. However, the vanilla BFV scheme does not provide circuit privacy. At high-level, the circuit privacy requirement states that the ciphertext $\mathbf{c}$ should not reveal any information about the private inputs to the client (i.e., A and $\mathbf{r}$) other than the underlying plaintext $A \cdot \mathbf{m} - \mathbf{r}$. Otherwise, some information is leaked. Gazelle proposes two methods to provide circuit privacy that are not incorporated in their implementation. Hence, we need to scale up their performance numbers for a fair comparison.

The first method is to let the client and server engage in a two-party secure decryption protocol, where the input of client is sk and input of server is $\mathbf{c}$. However, this method adds communication and needs extra rounds of interaction. A more widely used approach is *noise flooding*. Roughly speaking, the server adds a large noise term to $\mathbf{c}$ before returning it to the client. The noise is big enough to drown any extra information contained in the ciphertext, and still small enough to so that it still decrypts to the same plaintext.

For the concrete instantiation of Gazelle, one needs to triple the size of ciphertext modulus q from 60 bits to 180 bits, and increase the ring dimension n from 2048 to 8192. The (amortized) complexity of homomorphic operations in the BFV scheme is approximately $O(\log n \log q)$, with the exception that some operations run in $O(\log q)$ amortized time. Therefore, adding noise flooding would result in a *3-3.6 times slow down* for the HE component of Gazelle. To give some concrete examples, we consider two networks used for benchmarking in Gazelle: MNIST-D and CIFAR-10 networks. For the MNIST-D network, homomorphic encryption takes 55% and 22% in online and total time, respectively. For CIFAR-10, the corresponding figures are 35%, and 10%[1]. Therefore, we estimate that the total time for MNIST-D will grow from 0.81s to 1.16-1.27s (network BM3 in this paper). In the case of CIFAR-10 network, the total time will grow from 12.9s to 15.48-16.25s.

[1] these percentage numbers are obtained through private communication with the authors.

7 Experimental Results

We evaluate XONN on MNIST and CIFAR10 datasets, which are two popular classification benchmarks used in prior work. In addition, we provide four healthcare datasets to illustrate the applicability of XONN in real-world scenarios. For training XONN, we use Keras [37] with Tensorflow backend [52]. The source code of XONN is compiled with GCC 5.5.0 using O3 optimization. All Boolean circuits are synthesized using Synopsys Design Compiler 2015. Evaluations are performed on (Ubuntu 16.04 LTS) machines with Intel-Core i7-7700k and 32GB of RAM. The experimental setup is comparable (but has less computational power) compared to the prior art [10]. Consistent with prior frameworks, we evaluate the benchmarks in the LAN setting.

7.1 Evaluation on MNIST

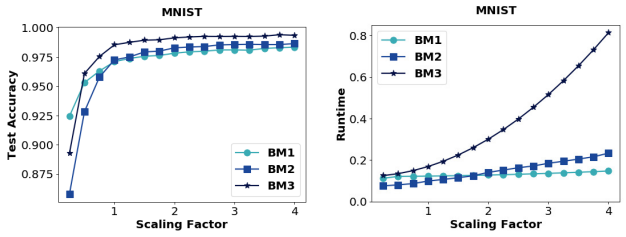
There are mainly three network architectures that prior works have implemented for the MNIST dataset. We convert these reference networks into their binary counterparts and train them using the standard BNN training algorithm [19]. Table 3 summarizes the architectures for the MNIST dataset.

Table 3: Summary of the trained binary network architectures evaluated on the MNIST dataset. Detailed descriptions are available in Appendix A.2, Table 13.

Arch.	Previous Papers	Description
BM1	SecureML [8], MiniONN [9]	3 FC
BM2	CryptoNets [14], MiniONN [9], DeepSecure [13], Chameleon [7]	1 CONV, 2 FC
BM3	MiniONN [9], EzPC [25]	2 CONV, 2MP, 2FC

Analysis of Network Scaling: Recall that the classification accuracy of XONN is controlled by scaling the number of neurons in all layers (Section 3.1). Figure 9a depicts the inference accuracy with different scaling factors (more details in Table 11 in Appendix A.2). As we increase the scaling factor, the accuracy of the network increases. This accuracy improvement comes at the cost of a higher computational complexity of the (scaled) network. As a result, increasing the scaling factor leads to a higher runtime. Figure 9b depicts the runtime of different BNN architectures as a function of the scaling factor s. Note that the runtime grows (almost) quadratically with the scaling factor due to the quadratic increase in the number of *Popcount* operations in the neural network (see $BM3$). However, for the $BM1$ and $BM2$ networks, the overall runtime is dominated by the constant initialization cost of the OT protocol ($\sim$ 70 millisecond).

GC Cost and the Effect of OCA: The communication cost of GC is the key contributor to the overall runtime of XONN. Here, we analyze the effect of the scaling factor on the total message size. Figure 10 shows the communication cost of



(a) (b)

Figure 9: Effect of scaling factor on (a) accuracy and (b) inference runtime of MNIST networks. No pruning was applied in this evaluation.

GC for the BM1 and BM2 network architectures. As can be seen, the message size increases with the scaling factor. We also observe that the OCA protocol drastically reduces the message size. This is due to the fact that the first layer of BM1 and BM2 models account for a large portion of the overall computation; hence, improving the first layer with OCA has a drastic effect on the overall communication.

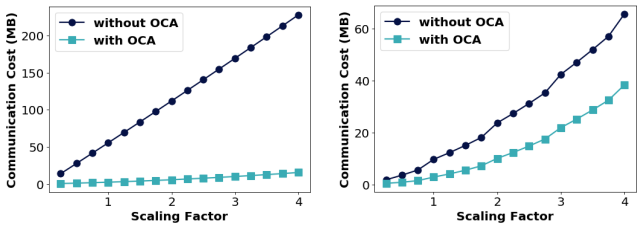


Figure 10: Effect of OCA on the communication of the BM1 (left) and BM2 (right) networks for different scaling factors. No pruning was applied in this evaluation.

Comparison to Prior Art: We emphasize that, unlike previous work, the accuracy of XONN can be customized by tuning the scaling factor (s). Furthermore, our channel/neuron pruning step (Algorithm 2) can reduce the GC cost in a post-processing phase. To provide a fair comparison between XONN and prior art, we choose a proper scaling factor and trim the pertinent scaled BNN such that the corresponding BNN achieves the same accuracy as the previous work. Table 4 compares XONN with the previous work in terms of accuracy, latency, and communication cost (a.k.a., message size). The last column shows the scaling factor (s) used to increase the width of the hidden layers of the BNN. Note that the scaled network is further trimmed using Algorithm 2.

In XONN, the runtime for oblivious transfer is at least $\sim$ 0.07 second for initiating the protocol and then it grows linearly with the size of the garbled tables; As a result, in very small architectures such as $BM1$, our solution is slightly slower than previous works since the constant runtime dominates the total runtime. However, for the $BM3$ network which has higher complexity than $BM1$ and $BM2$, XONN

achieves a more prominent advantage over prior art. In summary, our solution achieves up to 7.7× faster inference (average of 3.4×) compared to Gazelle [10]. Compared to MiniONN [9], XONN has up to 62× lower latency (average of 26×) Table 4. Compared to EzPC [25], our framework is 34× faster. XONN achieves 37.5×, 1859×, 60.4×, and 14× better latency compared to SecureML [8], CryptoNets [14], DeepSecure [13], and Chameleon [7], respectively.

Table 4: Comparison of XONN with the state-of-the-art for the MNIST network architectures.

Arch.	Framework	Runtime (s)	Comm. (MB)	Acc. (%)	s
BM1	SecureML	4.88	-	93.1	-
	MiniONN	1.04	15.8	97.6	-
	EzPC	0.7	76	97.6	-
	Gazelle	0.09	0.5	97.6	-
	XONN	0.13	4.29	97.6	1.75
BM2	CryptoNets	297.5	372.2	98.95	-
	DeepSecure	9.67	791	98.95	-
	MiniONN	1.28	47.6	98.95	-
	Chameleon	2.24	10.5	99.0	-
	EzPC	0.6	70	99.0	-
	Gazelle	0.29	8.0	99.0	-
	XONN	0.16	38.28	98.64	4.00
BM3	MiniONN	9.32	657.5	99.0	-
	EzPC	5.1	501	99.0	-
	Gazelle	1.16	70	99.0	-
	XONN	0.15	32.13	99.0	2.00

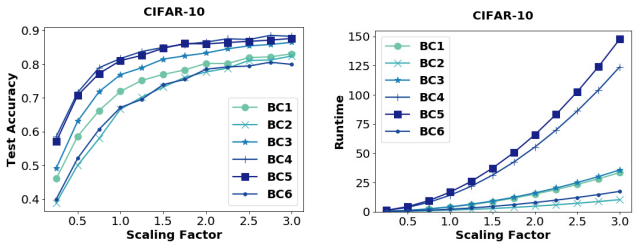
7.2 Evaluation on CIFAR-10

In Table 5, we summarize the network architectures that we use for the CIFAR-10 dataset. In this table, BC1 is the binarized version of the architecture proposed by MiniONN. To evaluate the scalability of our framework to larger networks, we also binarize the Fitnet [53] architectures, which are denoted as BC2-BC5. We also evaluate XONN on the popular VGG16 network architecture (BC6). Detailed architecture descriptions are available in Appendix A.2, Table 13.

Table 5: Summary of the trained binary network architectures evaluated on the CIFAR-10 dataset.

Arch.	Previous Papers	Description
BC1	MiniONN[9], Chameleon [7], EzPC [25], Gazelle [10]	7 CONV, 2 MP, 1 FC
BC2	Fitnet [53]	9 CONV, 3 MP, 1 FC
BC3	Fitnet [53]	9 CONV, 3 MP, 1 FC
BC4	Fitnet [53]	11 CONV, 3 MP, 1 FC
BC5	Fitnet [53]	17 CONV, 3 MP, 1 FC
BC6	VGG16 [54]	13 CONV, 5 MP, 3 FC

Analysis of Network Scaling: Similar to the analysis on the MNIST dataset, we show that the accuracy of our binary models for CIFAR-10 can be tuned based on the scaling factor that determines the number of neurons in each layer. Figure 11a depicts the accuracy of the BNNs with different scaling factors. As can be seen, increasing the scaling factor enhances the classification accuracy of the BNN. The runtime also increases with the scaling factor as shown in Figure 11b (more details in Table 12, Appendix A.2).



(a) (b)

Figure 11: (a) Effect of scaling factor on accuracy for CIFAR-10 networks. (b) Effect of scaling factor on runtime. No pruning was applied in this evaluation.

Comparison to Prior Art: We scale the BC2 network with a factor of $s = 3$, then prune it using Algorithm 2. Details of pruning steps are available in Table 10 in Appendix A.1. The resulting network is compared against prior art in Table 6. As can be seen, our solution achieves 2.7×, 45.8×, 9.1×, and 93.1× lower latency compared to Gazelle, EzPC, Chameleon, and MiniONN, respectively.

Table 6: Comparison of XONN with prior art on CIFAR-10.

Framework	Runtime (s)	Comm. (MB)	Acc. (%)	s
MiniONN	544	9272	81.61	-
Chameleon	52.67	2650	81.61	-
EzPC	265.6	40683	81.61	-
Gazelle	15.48	1236	81.61	-
XONN	5.79	2599	81.85	3.00

7.3 Evaluation on Medical Datasets

One of the most important applications of oblivious inference is medical data analysis. Recent advances in deep learning greatly benefit many complex diagnosis tasks that require exhaustive manual inspection by human experts [55, 56, 57, 58]. To showcase the applicability of oblivious inference in real-world medical applications, we provide several benchmarks for publicly available healthcare datasets summarized in Table 7. We split the datasets into validation and training portions as indicated in the last two columns of Table 7. All datasets except Malaria Infection are normalized to have 0 mean and standard deviation of 1 per feature. The images of Malaria Infection dataset are resized to 32×32 pictures. The normalized datasets are quantized up to 3 decimal digits. Detailed architectures are available in Appendix A.2, Table 13 We report the validation accuracy along with inference time and message size in Table 8.

Table 7: Summary of medical application benchmarks.

Task	Arch.	Description	# of Samples	
			Tr.	Val.
Breast Cancer [59]	BH1	3 FC	453	113
Diabetes [60]	BH2	3 FC	615	153
Liver Disease [61]	BH3	3 FC	467	116
Malaria Infection [62]	BH4	2 CONV, 2 MP, 2 FC	24804	2756

Table 8: Runtime, communication cost (Comm.), and accuracy (Acc.) for medical benchmarks.

Arch.	Runtime (ms)	Comm. (MB)	Acc. (%)
BH1	82	0.35	97.35
BH2	75	0.16	80.39
BH3	81	0.3	80.17
BH4	482	120.75	95.03

8 Conclusion

We introduce XONN, a novel framework to automatically train and use deep neural networks for the task of oblivious inference. XONN utilizes Yao's Garbled Circuits (GC) protocol and relies on binarizing the DL models in order to translate costly matrix multiplications to XNOR operations that are free in the GC protocol. Compared to Gazelle [10], prior best solution, XONN achieves 7× lower latency. Moreover, in contrast to Gazelle that requires one round of interaction for each layer, our solution needs a constant round of interactions regardless of the number of layers. Maintaining constant round complexity is an important requirement in Internet settings as a typical network latency can significantly degrade the performance of oblivious inference. Moreover, since our solution relies on the GC protocol, it can provide much stronger security guarantees such as security against malicious adversaries using standard cut-and-choose protocols. XONN high-level API enables clients to utilize the framework with a minimal number of lines of code. To further facilitate the adaptation of our framework, we design a compiler to translate the neural network description in Keras format to that of XONN.

Acknowledgements We would like to thank the anonymous reviewers for their insightful comments.

References

[1] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction APIs. In *USENIX Security*, 2016.

[2] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. *SIAM Journal on Computing*, 43(2):831–871, 2014.

[3] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):13, 2014.

[4] Andrew Yao. How to generate and exchange secrets. In *FOCS*, 1986.

[5] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 218–229. ACM, 1987.

[6] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 223–238. Springer, 1999.

[7] M Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *ASIACCS'18*, 2018.

[8] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *IEEE S&P*, 2017.

[9] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via MiniONN transformations. In *ACM CCS*, 2017.

[10] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. *USENIX Security*, 2018.

[11] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, et al. Going deeper with convolutions. CVPR, 2015.

[12] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP*, 2008.

[13] Bita Darvish Rouhani, M Sadegh Riazi, and Farinaz Koushanfar. DeepSecure: Scalable provably-secure deep learning. *DAC*, 2018.

[14] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy. In *ICML*, 2016.

[15] Michael O Rabin. How to exchange secrets with oblivious transfer. *IACR Cryptology ePrint Archive*, 2005:187, 2005.

[16] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In *Annual International Cryptology Conference*, pages 145–161. Springer, 2003.

[17] Donald Beaver. Correlated pseudorandomness and the complexity of private computations. In *STOC*, 1996.

[18] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *ACM CCS*, 2013.

[19] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[20] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[21] Mohammad Ghasemzadeh, Mohammad Samragh, and Farinaz Koushanfar. ReBNet: Residual binarized neural network. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 57–64. IEEE, 2018.

[22] Xiaofan Lin, Cong Zhao, and Wei Pan. Towards accurate binary convolutional neural network. In *Advances in Neural Information Processing Systems*, pages 345–353, 2017.

[23] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

[24] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*, 2016.

[25] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. EzPC: Programmable, efficient, and scalable secure two-party computation. *IACR Cryptology ePrint Archive*, 2017/1109, 2017.

[26] Yehuda Lindell and Benny Pinkas. A proof of security of Yao's protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, 2009.

[27] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *ACM CCS*. ACM, 2015.

[28] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *S&P*. IEEE, 2017.

[29] Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. *Journal of Cryptology*, 25(4):680–722, 2012.

[30] Yan Huang, Jonathan Katz, and David Evans. Efficient secure two-party computation using symmetric cut-and-choose. In *Advances in Cryptology–CRYPTO 2013*, pages 18–35. Springer, 2013.

[31] Yehuda Lindell. Fast cut-and-choose-based protocols for malicious and covert adversaries. *Journal of Cryptology*, 29(2):456–490, 2016.

[32] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *IEEE S&P*, 2013.

[33] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *ACM Conference on Electronic Commerce*, 1999.

[34] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole. In *EUROCRYPT*, 2015.

[35] Peter Rindal. libOTe: an efficient, portable, and easy to use Oblivious Transfer Library. `https://github.com/osu-crypto/libOTe`.

[36] Ebrahim M Songhori, Siam U Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. TinyGarble: Highly compressed and scalable sequential garbled circuits. In *IEEE S&P*, 2015.

[37] François Chollet et al. Keras. `https://keras.io`, 2015.

[38] Ehsan Hesamifard, Hassan Takabi, Mehdi Ghasemi, and Rebecca N Wright. Privacy-preserving machine learning as a service. *Proceedings on Privacy Enhancing Technologies*, 2018(3):123–142, 2018.

[39] Edward Chou, Josh Beal, Daniel Levy, Serena Yeung, Albert Haque, and Li Fei-Fei. Faster CryptoNets: Leveraging sparsity for real-world encrypted inference. *arXiv preprint arXiv:1811.09953*, 2018.

[40] Reza Shokri and Vitaly Shmatikov. Privacy-preserving deep learning. In *ACM CCS*, 2015.

[41] Briland Hitaj, Giuseppe Ateniese, and Fernando Pérez-Cruz. Deep models under the GAN: information leakage from collaborative deep learning. In *ACM CCS*, 2017.

[42] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In *ACM CCS*, 2017.

[43] M Sadegh Riazi, Bita Darvish Rouhani, and Farinaz Koushanfar. Deep learning on private data. *IEEE Security and Privacy (S&P) Magazine.*, 2019.

[44] M Sadegh Riazi and Farinaz Koushanfar. Privacy-preserving deep learning and inference. In *Proceedings of the International Conference on Computer-Aided Design*, page 18. ACM, 2018.

[45] Payman Mohassel and Peter Rindal. ABY3: a mixed protocol framework for machine learning. In *ACM CCS*, 2018.

[46] Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: Efficient and private neural network training, 2018.

[47] Xiaoqian Jiang, Miran Kim, Kristin Lauter, and Yongsoo Song. Secure outsourced matrix computation and application to neural networks. In *ACM CCS*, 2018.

[48] Amartya Sanyal, Matt Kusner, Adria Gascon, and Varun Kanade. TAPAS: Tricks to accelerate (encrypted) prediction as a service. In *International Conference on Machine Learning*, pages 4497–4506, 2018.

[49] Aner Ben-Efraim, Yehuda Lindell, and Eran Omri. Optimizing semi-honest secure multiparty computation for the internet. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 578–590. ACM, 2016.

[50] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *Advances in cryptology–crypto 2012*, pages 868–886. Springer, 2012.

[51] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.

[52] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Operating Systems Design and Implementation (OSDI)*, 2016.

[53] Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*, 2014.

[54] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[55] Andre Esteva, Alexandre Robicquet, Bharath Ramsundar, Volodymyr Kuleshov, Mark DePristo, Katherine Chou, Claire Cui, Greg Corrado, Sebastian Thrun, and Jeff Dean. A guide to deep learning in healthcare. *Nature medicine*, 25(1):24, 2019.

[56] Andre Esteva, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau, and Sebastian Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *Nature*, 542(7639):115, 2017.

[57] Babak Alipanahi, Andrew Delong, Matthew T Weirauch, and Brendan J Frey. Predicting the sequence specificities of dna-and rna-binding proteins by deep learning. *Nature biotechnology*, 33(8):831, 2015.

[58] Alvin Rajkomar, Eyal Oren, Kai Chen, Andrew M Dai, Nissan Hajaj, Michaela Hardt, Peter J Liu, Xiaobing Liu, Jake Marcus, Mimi Sun, et al. Scalable and accurate deep learning with electronic health records. *npj Digital Medicine*, 1(1):18, 2018.

[59] Breast Cancer Wisconsin, accessed on 01/20/2019. `https://www.kaggle.com/uciml/breast-cancer-wisconsin-data`.

[60] Pima Indians Diabetes, accessed on 01/20/2019. `https://www.kaggle.com/uciml/pima-indians-diabetes-database`.

[61] Indian Liver Patient Records, accessed on 01/20/2019. `https://www.kaggle.com/uciml/indian-liver-patient-records`.

[62] Malaria Cell Images, accessed on 01/20/2019. `https://www.kaggle.com/iarunava/cell-images-for-detecting-malaria`.

A Experimental Details

A.1 Network Trimming Examples

Table 9 and 10 summarize the trimming steps for the MNIST and CIFAR-10 benchmarks, respectively.

Table 9: Trimming MNIST architectures.

Network	Property	Trimming Step				Change
		initial	step 1	step 2	step 3	
BM1 (s=1.75)	Acc. (%)	97.63	97.59	97.28	97.02	-0.61%
	Comm. (MB)	4.95	4.29	3.81	3.32	1.49× less
	Lat. (ms)	158	131	114	102	1.54× faster
BM2 (s=4)	Acc. (%)	98.64	98.44	98.37	98.13	-0.51%
	Comm. (MB)	38.28	28.63	24.33	15.76	2.42× less
	Lat. (ms)	158	144	134	104	1.51× faster
BM3 (s=2)	Acc. (%)	99.22	99.11	98.96	99.00	-0.22%
	Comm. (MB)	56.08	42.51	37.34	32.13	1.75× less
	Lat. (ms)	190	165	157	146	1.3× faster

Table 10: Trimming the BC2 network for CIFAR-10.

Property	Trimming Step				Change
	initial	step 1	step 2	step 3	
Acc. (%)	82.40	82.39	82.41	81.85	-0.55%
Com. (GB)	3.38	3.05	2.76	2.60	1.30× less
Lat. (s)	7.59	6.87	6.23	5.79	1.31× faster

A.2 Accuracy, Runtime, and Communication

Runtime and communication reports are available in Table 11 and Table 12 for MNIST and CIFAR-10 benchmarks, respectively. The corresponding neural network architectures are provided in Table 13. Entries corresponding to a communication of more than 40GB are estimated using numerical runtime models.

Table 11: Accuracy (Acc.), communication (Comm.), and latency (Lat.) for MNIST dataset. Channel/neuron trimming is not applied.

Arch.	s	Acc. (%)	Comm. (MB)	Lat. (s)
BM1	1	97.10	2.57	0.12
	1.5	97.56	4.09	0.13
	2	97.82	5.87	0.13
	3	98.10	10.22	0.14
	4	98.34	15.62	0.15
BM2	1	97.25	2.90	0.10
	1.50	97.93	5.55	0.12
	2	98.28	10.09	0.14
	3	98.56	21.90	0.18
	4	98.64	38.30	0.23
BM3	1	98.54	17.59	0.17
	1.5	98.93	36.72	0.22
	2	99.13	62.77	0.3
	3	99.26	135.88	0.52
	4	99.35	236.78	0.81

Table 12: Accuracy (Acc.), communication (Comm.), and latency (Lat.) for CIFAR-10 dataset. Channel/neuron trimming is not applied.

Arch.	s	Acc. (%)	Comm. (MB)	Lat. (s)
BC1	1	0.72	1.26	3.96
	1.5	0.77	2.82	8.59
	2	0.80	4.98	15.07
	3	0.83	11.15	33.49
BC2	1	0.67	0.39	1.37
	1.5	0.73	0.86	2.78
	2	0.78	1.53	4.75
	3	0.82	3.40	10.35
BC3	1	0.77	1.35	4.23
	1.5	0.81	3.00	9.17
	2	0.83	5.32	16.09
	3	0.86	11.89	35.77
BC4	1	0.82	4.66	14.12
	1.5	0.85	10.41	31.33
	2	0.87	18.45	55.38
	3	0.88	41.37	123.94
BC5	1	0.81	5.54	16.78
	1.5	0.85	12.40	37.29
	2	0.86	21.98	65.94
	3	0.88	49.30	147.66
BC6	1	0.67	0.65	2.15
	1.5	0.74	1.46	4.55
	2	0.78	2.58	7.91
	3	0.80	5.77	17.44

Table 13: Evaluated network architectures.

BM1

1	**FC** [input: 784, output: $128s$] + **BN** + **BA**
2	**FC** [input: $128s$, output: $128s$] + **BN** + **BA**
3	**FC** [input: $128s$, output: 10] + **BN** + **Softmax**

BM2

1	**CONV** [input: $28 \times 28 \times 1$, window: 5×5, stride: 2, kernels: $5s$, output: $12 \times 12 \times 5s$] + **BN** + **BA**
2	**FC** [input: $720s$, output: $100s$] + **BN** + **BA**
3	**FC** [input: $100s$, output: 10] + **BN** + **Softmax**

BM3

1	**CONV** [input: $28 \times 28 \times 1$, window: 5×5, stride: 1, kernels: $16s$, output: $24 \times 24 \times 16s$] + **BN** + **BA**
2	**MP** [input: $24 \times 24 \times 16s$, window: 2×2, output: $12 \times 12 \times 16s$]
3	**CONV** [input: $12 \times 12 \times 16s$, window: 5×5, stride: 1, kernels: $16s$, output: $8 \times 8 \times 16s$] + **BN** + **BA**
4	**MP** [input: $8 \times 8 \times 16s$, window: 2×2, output: $4 \times 4 \times 16s$]
5	**FC** [input: $256s$, output: $100s$] + **BN** + **BA**
6	**FC** [input: $100s$, output: 10] + **BN** + **Softmax**

BC1

1	**CONV** [input: $32 \times 32 \times 3$, window: 3×3, stride: 1, kernels: $64s$, output: $30 \times 30 \times 64s$] + **BN** + **BA**
2	**CONV** [input: $30 \times 30 \times 64s$, window: 3×3, stride: 1, kernels: $64s$, output: $28 \times 28 \times 64s$] + **BN** + **BA**
3	**MP** [input: $28 \times 28 \times 64s$, window: 2×2, output: $14 \times 14 \times 64s$]
4	**CONV** [input: $14 \times 14 \times 64s$, window: 3×3, stride: 1, kernels: $64s$, output: $12 \times 12 \times 64s$] + **BN** + **BA**
5	**CONV** [input: $12 \times 12 \times 64s$, window: 3×3, stride: 1, kernels: $64s$, output: $10 \times 10 \times 64s$] + **BN** + **BA**
6	**MP** [input: $10 \times 10 \times 64s$, window: 2×2, output: $5 \times 5 \times 64s$]
7	**CONV** [input: $5 \times 5 \times 64s$, window: 3×3, stride: 1, kernels: $64s$, output: $3 \times 3 \times 64s$] + **BN** + **BA**
8	**CONV** [input: $3 \times 3 \times 64s$, window: 1×1, stride: 1, kernels: $64s$, output: $3 \times 3 \times 64s$] + **BN** + **BA**
9	**CONV** [input: $3 \times 3 \times 64s$, window: 1×1, stride: 1, kernels: $16s$, output: $3 \times 3 \times 16s$] + **BN** + **BA**
10	**FC** [input: $144s$, output: 10] + **BN** + **Softmax**

BC2

1	**CONV** [input: $32 \times 32 \times 3$, window: 3×3, stride: 1, kernels: $16s$, output: $32 \times 32 \times 16s$] + **BN** + **BA**
2	**CONV** [input: $32 \times 32 \times 16s$, window: 3×3, stride: 1, kernels: $16s$, output: $32 \times 32 \times 16s$] + **BN** + **BA**
3	**CONV** [input: $32 \times 32 \times 16s$, window: 3×3, stride: 1, kernels: $16s$, output: $32 \times 32 \times 16s$] + **BN** + **BA**
4	**MP** [input: $32 \times 32 \times 16s$, window: 2×2, output: $16 \times 16 \times 16s$]
5	**CONV** [input: $16 \times 16 \times 16s$, window: 3×3, stride: 1, kernels: $32s$, output: $16 \times 16 \times 32s$] + **BN** + **BA**
6	**CONV** [input: $16 \times 16 \times 32s$, window: 3×3, stride: 1, kernels: $32s$, output: $16 \times 16 \times 32s$] + **BN** + **BA**
7	**CONV** [input: $16 \times 16 \times 32s$, window: 3×3, stride: 1, kernels: $32s$, output: $16 \times 16 \times 32s$] + **BN** + **BA**
8	**MP** [input: $16 \times 16 \times 32s$, window: 2×2, output: $8 \times 8 \times 32s$]
9	**CONV** [input: $8 \times 8 \times 32s$, window: 3×3, stride: 1, kernels: $48s$, output: $6 \times 6 \times 48s$] + **BN** + **BA**
10	**CONV** [input: $6 \times 6 \times 48s$, window: 3×3, stride: 1, kernels: $48s$, output: $4 \times 4 \times 48s$] + **BN** + **BA**
11	**CONV** [input: $4 \times 4 \times 48s$, window: 3×3, stride: 1, kernels: $64s$, output: $2 \times 2 \times 64s$] + **BN** + **BA**
12	**MP** [input: $2 \times 2 \times 64s$, window: 2×2, output: $1 \times 1 \times 64s$]
13	**FC** [input: $64s$, output: 10] + **BN** + **Softmax**

BC3

1	**CONV** [input: $32 \times 32 \times 3$, window: 3×3, stride: 1, kernels: $16s$, output: $32 \times 32 \times 16s$] + **BN** + **BA**
2	**CONV** [input: $32 \times 32 \times 16s$, window: 3×3, stride: 1, kernels: $32s$, output: $32 \times 32 \times 32s$] + **BN** + **BA**
3	**CONV** [input: $32 \times 32 \times 32s$, window: 3×3, stride: 1, kernels: $32s$, output: $32 \times 32 \times 32s$] + **BN** + **BA**
4	**MP** [input: $32 \times 32 \times 32s$, window: 2×2, output: $16 \times 16 \times 32s$]
5	**CONV** [input: $16 \times 16 \times 32s$, window: 3×3, stride: 1, kernels: $48s$, output: $16 \times 16 \times 48s$] + **BN** + **BA**
6	**CONV** [input: $16 \times 16 \times 48s$, window: 3×3, stride: 1, kernels: $64s$, output: $16 \times 16 \times 64s$] + **BN** + **BA**
7	**CONV** [input: $16 \times 16 \times 64s$, window: 3×3, stride: 1, kernels: $80s$, output: $16 \times 16 \times 80s$] + **BN** + **BA**
8	**MP** [input: $16 \times 16 \times 80s$, window: 2×2, output: $8 \times 8 \times 80s$]
9	**CONV** [input: $8 \times 8 \times 80s$, window: 3×3, stride: 1, kernels: $96s$, output: $6 \times 6 \times 96s$] + **BN** + **BA**
10	**CONV** [input: $6 \times 6 \times 96s$, window: 3×3, stride: 1, kernels: $96s$, output: $4 \times 4 \times 96s$] + **BN** + **BA**
11	**CONV** [input: $4 \times 4 \times 96s$, window: 3×3, stride: 1, kernels: $128s$, output: $2 \times 2 \times 128s$] + **BN** + **BA**
12	**MP** [input: $2 \times 2 \times 128s$, window: 2×2, output: $1 \times 1 \times 128s$]
13	**FC** [input: $128s$, output: 10] + **BN** + **Softmax**

BC4

1	**CONV** [input: $32 \times 32 \times 3$, window: 3×3, stride: 1, kernels: $32s$, output: $32 \times 32 \times 32s$] + **BN** + **BA**
2	**CONV** [input: $32 \times 32 \times 32s$, window: 3×3, stride: 1, kernels: $48s$, output: $32 \times 32 \times 48s$] + **BN** + **BA**
3	**CONV** [input: $32 \times 32 \times 48s$, window: 3×3, stride: 1, kernels: $64s$, output: $32 \times 32 \times 64s$] + **BN** + **BA**
4	**CONV** [input: $32 \times 32 \times 64s$, window: 3×3, stride: 1, kernels: $64s$, output: $32 \times 32 \times 64s$] + **BN** + **BA**
5	**MP** [input: $32 \times 32 \times 64s$, window: 2×2, output: $16 \times 16 \times 64s$]
6	**CONV** [input: $16 \times 16 \times 64s$, window: 3×3, stride: 1, kernels: $80s$, output: $16 \times 16 \times 80s$] + **BN** + **BA**
7	**CONV** [input: $16 \times 16 \times 80s$, window: 3×3, stride: 1, kernels: $80s$, output: $16 \times 16 \times 80s$] + **BN** + **BA**
8	**CONV** [input: $16 \times 16 \times 80s$, window: 3×3, stride: 1, kernels: $80s$, output: $16 \times 16 \times 80s$] + **BN** + **BA**
9	**CONV** [input: $16 \times 16 \times 80s$, window: 3×3, stride: 1, kernels: $80s$, output: $16 \times 16 \times 80s$] + **BN** + **BA**
10	**MP** [input: $16 \times 16 \times 80s$, window: 2×2, output: $8 \times 8 \times 80s$]
11	**CONV** [input: $8 \times 8 \times 80s$, window: 3×3, stride: 1, kernels: $128s$, output: $6 \times 6 \times 128s$] + **BN** + **BA**
12	**CONV** [input: $6 \times 6 \times 128s$, window: 3×3, stride: 1, kernels: $128s$, output: $4 \times 4 \times 128s$] + **BN** + **BA**
13	**CONV** [input: $4 \times 4 \times 128s$, window: 3×3, stride: 1, kernels: $128s$, output: $2 \times 2 \times 128s$] + **BN** + **BA**
14	**MP** [input: $2 \times 2 \times 128s$, window: 2×2, output: $1 \times 1 \times 128s$]
15	**FC** [input: $128s$, output: 10] + **BN** + **Softmax**

BC5

1	**CONV** [input: $32 \times 32 \times 3$, window: 3×3, stride: 1, kernels: $32s$, output: $32 \times 32 \times 32s$] + **BN** + **BA**
2	**CONV** [input: $32 \times 32 \times 32s$, window: 3×3, stride: 1, kernels: $32s$, output: $32 \times 32 \times 32s$] + **BN** + **BA**
3	**CONV** [input: $32 \times 32 \times 32s$, window: 3×3, stride: 1, kernels: $32s$, output: $32 \times 32 \times 32s$] + **BN** + **BA**
4	**CONV** [input: $32 \times 32 \times 32s$, window: 3×3, stride: 1, kernels: $48s$, output: $32 \times 32 \times 48s$] + **BN** + **BA**
5	**CONV** [input: $32 \times 32 \times 48s$, window: 3×3, stride: 1, kernels: $48s$, output: $32 \times 32 \times 48s$] + **BN** + **BA**
6	**MP** [input: $32 \times 32 \times 48s$, window: 2×2, output: $16 \times 16 \times 48s$]
7	**CONV** [input: $16 \times 16 \times 48s$, window: 3×3, stride: 1, kernels: $80s$, output: $16 \times 16 \times 80s$] + **BN** + **BA**
8	**CONV** [input: $16 \times 16 \times 80s$, window: 3×3, stride: 1, kernels: $80s$, output: $16 \times 16 \times 80s$] + **BN** + **BA**
9	**CONV** [input: $16 \times 16 \times 80s$, window: 3×3, stride: 1, kernels: $80s$, output: $16 \times 16 \times 80s$] + **BN** + **BA**
10	**CONV** [input: $16 \times 16 \times 80s$, window: 3×3, stride: 1, kernels: $80s$, output: $16 \times 16 \times 80s$] + **BN** + **BA**
11	**CONV** [input: $16 \times 16 \times 80s$, window: 3×3, stride: 1, kernels: $80s$, output: $16 \times 16 \times 80s$] + **BN** + **BA**
12	**CONV** [input: $16 \times 16 \times 80s$, window: 3×3, stride: 1, kernels: $80s$, output: $16 \times 16 \times 80s$] + **BN** + **BA**
13	**MP** [input: $16 \times 16 \times 80s$, window: 2×2, output: $8 \times 8 \times 80s$]
14	**CONV** [input: $8 \times 8 \times 80s$, window: 3×3, stride: 1, kernels: $128s$, output: $8 \times 8 \times 128s$] + **BN** + **BA**
15	**CONV** [input: $8 \times 8 \times 128s$, window: 3×3, stride: 1, kernels: $128s$, output: $8 \times 8 \times 128s$] + **BN** + **BA**
16	**CONV** [input: $8 \times 8 \times 128s$, window: 3×3, stride: 1, kernels: $128s$, output: $8 \times 8 \times 128s$] + **BN** + **BA**
17	**CONV** [input: $8 \times 8 \times 128s$, window: 3×3, stride: 1, kernels: $128s$, output: $6 \times 6 \times 128s$] + **BN** + **BA**
18	**CONV** [input: $6 \times 6 \times 128s$, window: 3×3, stride: 1, kernels: $128s$, output: $4 \times 4 \times 128s$] + **BN** + **BA**
19	**CONV** [input: $4 \times 4 \times 128s$, window: 3×3, stride: 1, kernels: $128s$, output: $2 \times 2 \times 128s$] + **BN** + **BA**
20	**MP** [input: $2 \times 2 \times 128s$, window: 2×2, output: $1 \times 1 \times 128s$]
21	**FC** [input: $128s$, output: 10] + **BN** + **Softmax**

BC6

1	**CONV** [input: $32 \times 32 \times 3$, window: 3×3, stride: 1, kernels: $16s$, output: $32 \times 32 \times 16s$] + **BN** + **BA**
2	**CONV** [input: $32 \times 32 \times 16s$, window: 3×3, stride: 1, kernels: $16s$, output: $32 \times 32 \times 16s$] + **BN** + **BA**
3	**MP** [input: $32 \times 32 \times 16s$, window: 2×2, output: $16 \times 16 \times 16s$]
4	**CONV** [input: $16 \times 16 \times 16s$, window: 3×3, stride: 1, kernels: $32s$, output: $16 \times 16 \times 32s$] + **BN** + **BA**
5	**CONV** [input: $16 \times 16 \times 32s$, window: 3×3, stride: 1, kernels: $32s$, output: $16 \times 16 \times 32s$] + **BN** + **BA**
6	**MP** [input: $16 \times 16 \times 32s$, window: 2×2, output: $8 \times 8 \times 32s$]
7	**CONV** [input: $8 \times 8 \times 32s$, window: 3×3, stride: 1, kernels: $64s$, output: $8 \times 8 \times 64s$] + **BN** + **BA**
8	**CONV** [input: $8 \times 8 \times 64s$, window: 3×3, stride: 1, kernels: $64s$, output: $8 \times 8 \times 64s$] + **BN** + **BA**
9	**CONV** [input: $8 \times 8 \times 64s$, window: 3×3, stride: 1, kernels: $64s$, output: $8 \times 8 \times 64s$] + **BN** + **BA**
10	**MP** [input: $8 \times 8 \times 64s$, window: 2×2, output: $4 \times 4 \times 64s$]
11	**CONV** [input: $4 \times 4 \times 64s$, window: 3×3, stride: 1, kernels: $128s$, output: $4 \times 4 \times 128s$] + **BN** + **BA**
12	**CONV** [input: $4 \times 4 \times 128s$, window: 3×3, stride: 1, kernels: $128s$, output: $4 \times 4 \times 128s$] + **BN** + **BA**
13	**CONV** [input: $4 \times 4 \times 128s$, window: 3×3, stride: 1, kernels: $128s$, output: $4 \times 4 \times 128s$] + **BN** + **BA**
14	**MP** [input: $4 \times 4 \times 128s$, window: 2×2, output: $2 \times 2 \times 128s$]
15	**CONV** [input: $2 \times 2 \times 128s$, window: 3×3, stride: 1, kernels: $128s$, output: $2 \times 2 \times 128s$] + **BN** + **BA**
16	**CONV** [input: $2 \times 2 \times 128s$, window: 3×3, stride: 1, kernels: $128s$, output: $2 \times 2 \times 128s$] + **BN** + **BA**
17	**CONV** [input: $2 \times 2 \times 128s$, window: 3×3, stride: 1, kernels: $128s$, output: $2 \times 2 \times 128s$] + **BN** + **BA**
18	**MP** [input: $2 \times 2 \times 128s$, window: 2×2, output: $1 \times 1 \times 128s$]
19	**FC** [input: $128s$, output: $512s$] + **BN** + **BA**
20	**FC** [input: $512s$, output: $512s$] + **BN** + **BA**
21	**FC** [input: $512s$, output: 10] + **BN** + **Softmax**

BH1

1	**FC** [input: 30, output: 16] + **BN** + **BA**
2	**FC** [input: 16, output: 16] + **BN** + **BA**
3	**FC** [input: 16, output: 2] + **BN** + **Softmax**

BH2

1	**FC** [input: 8, output: 20] + **BN** + **BA**
2	**FC** [input: 20, output: 20] + **BN** + **BA**
3	**FC** [input: 20, output: 2] + **BN** + **Softmax**

BH3

1	**FC** [input: 10, output: 32] + **BN** + **BA**
2	**FC** [input: 32, output: 32] + **BN** + **BA**
3	**FC** [input: 32, output: 2] + **BN** + **Softmax**

BH4

1	**CONV** [input: $32 \times 32 \times 3$, window: 5×5, stride: 1, kernels: 36, output: $28 \times 28 \times 36$] + **BN** + **BA**
2	**MP** [input: $28 \times 28 \times 36$, window: 2×2, output: $14 \times 14 \times 36$]
3	**CONV** [input: $14 \times 14 \times 36$, window: 5×5, stride: 1, kernels: 36, output: $10 \times 10 \times 36$] + **BN** + **BA**
4	**MP** [input: $10 \times 10 \times 36$, window: 2×2, output: $5 \times 5 \times 36$]
5	**FC** [input: 900, output: 72] + **BN** + **BA**
6	**FC** [input: 72, output: 2] + **BN** + **Softmax**

B Attacks on Deep Neural Networks

In this section, we review three of the most important attacks against deep neural networks that are relevant to the context of oblivious inference [1, 27, 28]. In all three, a client-server model is considered where the client is the adversary and attempts to learn more about the model held by the server. The client sends many inputs and receives the inference results . He then analyzes the results to infer more information about either the network parameters or the training data that has been used in the training phase of the model. We briefly review each attack and illustrate a simple defense mechanism with negligible overhead based on the suggestions provided in these works.

Model Inversion Attack [27]. In the black-box access model of this attack (which fits the computational model of this work), an adversarial client attempts to learn about a prototypical sample of one of the classes. The client iteratively creates an input that maximizes the confidence score corresponding to the target class. Regardless of the specific training process, the attacker can learn significant information by querying the model many times.

Model Extraction Attack [1]. In this type of attack, an adversary's goal is to estimate the parameters of the machine learning model held by the server. For example, in a logistic regression model with n parameters, the model can be extracted by querying the server n times and upon receiving the confidence values, solving a system of n equations. Model extraction can diminish the pay-per-prediction business model of technology companies. Moreover, it can be used as a pre-step towards the model inversion attack.

Membership Inference Attack [28]. The objective of this attack is to identify whether a given input has been used in the training phase of the model or not. This attack raises certain privacy concerns. The idea behind this attack is that the neural networks usually perform better on the data that they were trained on. Therefore, two inputs that belong to the same class, one used in the training phase and one not, will have noticeable differences in the confidence values. This behavior is called *overfitting*. The attack can be mitigated using regularization techniques that reduce the dependency of the DL model on a single training sample. However, overfitting is not the only contributor to this information leakage.

Defense Mechanisms. In the prior state-of-the-art oblivious inference solution [9], it has been suggested to limit the number of queries from a specific client to limit the information leakage. However, in practice, an attacker can impersonate himself as many different clients and circumvent this defense mechanism. Note that all three attacks rely on the fact that along with the inference result, the server provides the confidence vector that specifies how likely the client's input belongs to each class. Therefore, as suggested by prior work [1, 27, 28], it is recommended to augment a *filter* layer that (i) rounds the confidence scores or (ii) selects the index of a class that has the highest confidence score.

1. *Rounding the confidence values:* Rounding the values simply means omitting one (or more) of the Least Significant Bit (LSB) of all of the numbers in the last layer. This operation is in fact *free* in GC since it means Garbler has to avoid providing the mapping for those LSBs.
2. *Reporting the class label:* This operation is equivalent to computing `argmax` on the last layer. For a vector of size c where each number is represented with b bits, `argmax` is translated to $c \cdot (2b+1)$ many non-`XOR` (`AND`) gates. For example, in a typical architecture for MNIST (e.g., BM3) or CIFAR-10 dataset (e.g., BC1), the overhead is 1.68E-2% and 1.36E-4%, respectively.

Note that the two aforementioned defense mechanisms can be augmented to any framework that supports non-linear functionalities [7, 9, 13]. However, we want to emphasize that compared to mixed-protocol solutions, this means that another round of communication is usually needed to support the filter layer. Whereas, in XONN the filter layer does not increase the number of rounds and has negligible overhead compared to the overall protocol.

JEDI: Many-to-Many End-to-End Encryption and Key Delegation for IoT

Sam Kumar, Yuncong Hu, Michael P Andersen, Raluca Ada Popa, and David E. Culler
University of California, Berkeley

Abstract

As the Internet of Things (IoT) emerges over the next decade, developing secure communication for IoT devices is of paramount importance. Achieving end-to-end encryption for large-scale IoT systems, like smart buildings or smart cities, is challenging because multiple principals typically interact *indirectly* via intermediaries, meaning that the recipient of a message is not known in advance. This paper proposes JEDI (**J**oining **E**ncryption and **D**elegation for **I**oT), a many-to-many end-to-end encryption protocol for IoT. JEDI encrypts and signs messages end-to-end, while conforming to the decoupled communication model typical of IoT systems. JEDI's keys support expiry and fine-grained access to data, common in IoT. Furthermore, JEDI allows principals to delegate their keys, restricted in expiry or scope, to other principals, thereby granting access to data and managing access control in a scalable, distributed way. Through careful protocol design and implementation, JEDI can run across the spectrum of IoT devices, including ultra low-power deeply embedded sensors severely constrained in CPU, memory, and energy consumption. We apply JEDI to an existing IoT messaging system and demonstrate that its overhead is modest.

1 Introduction

As the Internet of Things (IoT) has emerged over the past decade, smart devices have become increasingly common. This trend is only expected to continue, with tens of billions of new IoT devices deployed over the next few years [30]. The IoT vision requires these devices to communicate to discover and use the resources and data provided by one another. Yet, these devices collect privacy-sensitive information about users. A natural step to secure privacy-sensitive data is to use *end-to-end encryption* to protect it during transit.

Existing protocols for end-to-end encryption, such as SSL/TLS and TextSecure [44], focus on *one-to-one* communication between two principals: for example, Alice sends a message to Bob over an insecure channel. Such protocols, however, appear not to be a good fit for large-scale industrial IoT systems. Such IoT systems demand *many-to-many* communication among **decoupled** senders and receivers, and require **decentralized delegation of access** to enforce which devices can communicate with which others.

We investigate existing IoT systems, which currently do not encrypt data end-to-end, to understand the requirements on an end-to-end encryption protocol like JEDI. We use *smart cities* as an example application area, and data-collecting sensors in a large organization as a concrete use case. We identify three central requirements, which we treat in turn below:



Figure 1: IoT comprises a diverse set of devices, which span more than four orders of magnitude of computing power (estimated in Dhrystone MIPS).[1]

▷ **Decoupled senders and receivers.** IoT-scale systems could consist of thousands of principals, making it infeasible for consumers of data (e.g., applications) to maintain a separate session with each producer of data (e.g., sensors). Instead, senders are typically **decoupled** from receivers. Such decoupling is common in *publish-subscribe* systems for IoT, such as MQTT, AMQP, XMPP, and Solace [76]. In particular, many-to-many communication based on publish-subscribe is the *de-facto* standard in smart buildings, used in systems like BOSS [36], VOLTTRON [82], Brume [66] and bw2 [5], and adopted commercially in AllJoyn and IoTivity. Senders publish messages by addressing them to *resources* and sending them to a *router*. Recipients *subscribe* to a resource by asking the router to send them messages addressed to that resource.

Many systems for smart buildings/cities, like sMAP [35], SensorAct [7], bw2 [5], VOLTTRON [82], and BAS [56], organize resources as a **hierarchy**. A resource hierarchy matches the organization of IoT devices: for instance, smart cities contain buildings, which contain floors, which contain rooms, which contain sensors, which produce streams of readings. We represent each resource—a leaf in the hierarchy—as a Uniform Resource Indicator (**URI**), which is like a file path. For example, a sensor that measures temperature and humidity might send its readings to the two URIs `buildingA/floor2/roomLHall/sensor0/temp` and `buildingA/floor2/roomLHall/sensor0/hum`. A user can subscribe to a URI prefix, such as `buildingA/floor2/roomLHall/*`, which represents a subtree of the hierarchy. He would then receive all sensor readings in room "LHall."

▷ **Decentralized delegation.** Access control in IoT needs to be fine-grained. For example, if Bob has an app that needs

[1] Image credits: https://tweakers.net/pricewatch/1275475/asus-f5401a-dm1201t.html, https://www.lg.com/uk/mobile-phones/lg-H791, https://www.bestbuy.com/site/nest-learning-thermostat-3rd-generation-stainless-steel/4346501.p?skuId=4346501, https://www.macys.com/shop/product/fitbit-charge-2-heart-rate-fitness-wristband?ID=2999458

access to temperature readings from a single sensor, that app should receive the decryption key for only that one URI, even if Bob has keys for the entire room. In an IoT-scale system, it is not scalable for a central authority to individually give fine-grained decryption keys to each person's devices. Moreover, as we discuss in §2, such an approach would pose increased security and privacy risks. Instead, Bob, who himself has access to readings for the entire room, should be able to delegate temperature-readings access to the app. Generally, a principal with access to a set of resources can give another principal access to a subset of those resources.

Vanadium [77] and bw2 [5] introduced *decentralized delegation* (SPKI/SDSI [31] and Macaroons [13]) in the smart buildings space. Since then, decentralized delegation has become the state-of-the-art for access control in smart buildings, especially those geared toward large-scale commercial buildings or organizations [42,52]. In these systems, a principal can access a resource if there exists a *chain* of delegations, from the owner of the resource to that principal, granting access. At each link in the chain, the extent of access may be qualified by *caveats*, which add restrictions to which resources can be accessed and when. While these systems provide delegation of permissions, they do not provide protocols for encrypting and decrypting messages end-to-end.

▷ **Resource constraints.** IoT devices vary greatly in their capabilities, as shown in Fig. 1. This includes devices constrained in CPU, memory, and energy, such as wearable devices and low-cost environmental sensors.

In smart buildings/cities, one application of interest is *indoor environmental sensing*. Sensors that measure temperature, humidity, or occupancy may be deployed in a building; such sensors are *battery-powered* and transmit readings using a *low-power* wireless network. To see ubiquitous deployment, they must cost only *tens of dollars* per unit and have *several years* of battery life. To achieve this price/power point, sensor platforms are heavily resource-constrained, with mere *kilobytes* of memory (farthest right in Fig. 1) [3,4,26,41,49,59,69]. The *power consumption* of encryption is a serious challenge, even more so than its latency on a slower CPU; the CPU and radio must be used sparingly to avoid consuming energy too quickly [55, 89]. For example, on the sensor platform used in our evaluation, an average CPU utilization of merely 5% would result in less than a year of battery life, *even if the power cost of using the transducers and network were zero*.

1.1 Overview of JEDI

This paper presents JEDI, a *many-to-many* end-to-end encryption protocol compatible with the above three requirements of IoT systems. JEDI encrypts messages end-to-end for confidentiality, signs them for integrity while preserving anonymity, and supports delegation with caveats, all while allowing senders and receivers to be decoupled via a resource hierarchy. JEDI differs from existing encryption protocols like SSL/TLS, requiring us to overcome a number of *challenges*:

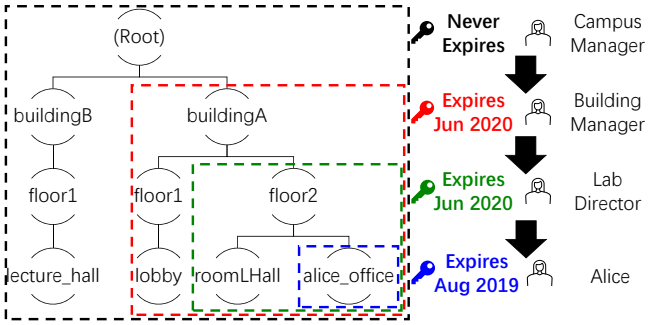


Figure 2: JEDI keys can be qualified and delegated, supporting decentralized, cryptographically-enforced access control via key delegation. Each person has a decryption key for the indicated resource subtree that is valid until the indicated expiry time. Black arrows denote delegation.

1. Formulating a new system model for end-to-end encryption to support **decoupled senders and receivers** and **decentralized delegation** typical of IoT systems (§1.1.1)
2. Realizing this expressive model while working within the **resource constraints** of IoT devices (§1.1.2)
3. Allowing receivers to verify the integrity of messages, while preserving the anonymity of senders (§1.1.3)
4. Extending JEDI's model to support revocation (§1.1.4)

Below, we explain how we address each of these challenges.

1.1.1 JEDI's System Model (§2)

Participants in JEDI are called *principals*. Any principal can create a **resource hierarchy** to represent some resources that it owns. Because that principal owns all of the resources in the hierarchy, it is called the *authority* of that hierarchy.

Due to the setting of **decoupled senders and receivers**, the sender can no longer encrypt messages with the receiver's public key, as in traditional end-to-end encryption. Instead, JEDI models principals as interacting with resources, rather than with other principals. Herein lies the key difference between JEDI's model and other end-to-end encryption protocols: the publisher of a message encrypts it according to the URI to which it is published, not the recipients subscribed to that URI. Only principals permitted to subscribe to a URI are given keys that can decrypt messages published to that URI.

IoT systems that support **decentralized delegation** (Vanadium, bw2), as well as related non-IoT authorization systems (e.g., SPKI/SDSI [31] and Macaroons [13]) provide principals with tokens (e.g., certificate chains) that they can present to prove they have access to a certain resource. Providing tokens, however, is not enough for end-to-end encryption; unlike these systems, JEDI allows *decryption keys* to be distributed via chains of delegations. Furthermore, the URI prefix and expiry time associated with each JEDI key can be restricted at each delegation. For example, as shown in Fig. 2, suppose Alice, who works in a research lab, needs access to sensor readings in her office. In the past, the campus facilities manager, who is the authority for the hierarchy, granted a key for `buildingA/*` to the building manager, who granted a key

for `buildingA/floor2/*` to the lab director. Now, Alice can obtain the key for `buildingA/floor2/alice_office/*` directly from her local authority (the lab director).

1.1.2 Encryption with URIs and Expiry (§3)

JEDI supports *decoupled* communication. The resource to which a message is published acts as a *rendezvous point* between the senders and receivers, used by the underlying system to route messages. Central to JEDI is the challenge of finding an analogous *cryptographic rendezvous point* that senders can use to encrypt messages without knowledge of receivers. A number of IoT systems [70, 74] use only simple cryptography like AES, SHA2, and ECDSA, but these primitives are not expressive enough to encode JEDI's rendezvous point, which must support hierarchically-structured resources, non-interactive expiry, and decentralized delegation.

Existing systems [83–85] with similar expressivity to JEDI use Attribute-Based Encryption (ABE) [12, 48]. Unfortunately, ABE is not suitable for JEDI because it is too expensive, especially in the context of **resource constraints** of IoT devices. Some IoT systems rule it out due to its latency alone [74]. In the context of low-power devices, encryption with ABE would also consume too much power. JEDI circumvents the problem of using ABE or basic cryptography with two insights: (1) Even though ABE is too heavy for low-power devices, this does not mean that we must resort to only symmetric-key techniques. We show that certain IBE schemes [1] can be made practical for such devices. (2) **Time is another resource hierarchy**: a timestamp can be expressed as `year/month/day/hour`, and in this hierarchical representation, any time range can be represented efficiently as a logarithmic number of subtrees. With this insight, we can simultaneously support URIs and expiry via a nonstandard use of a certain type of IBE scheme: WKD-IBE [1]. Like ABE, WKD-IBE is based on bilinear groups (pairings), but it is an order-of-magnitude less expensive than ABE as used in JEDI. To make JEDI practical on low-power devices, we design it to invoke WKD-IBE *rarely*, while relying on AES most of the time, much like session keys. Thus, JEDI achieves expressivity commensurate to IoT systems that do not encrypt data—significantly more expressive than AES-only solutions—while allowing several years of battery life for low-power low-cost IoT devices.

1.1.3 Integrity and Anonymity (§4)

In addition to being encrypted, messages should be signed so that the recipient of a message can be sure it was not sent by an attacker. This can be achieved via a certificate chain, as in SPKI/SDSI or bw2. Certificates can be distributed in a decentralized manner, just like encryption keys in Fig. 2.

Certificate chains, however, are insufficient if anonymity is required. For example, consider an office space with an occupancy sensor in each office, each publishing to the same URI `buildingA/occupancy`. In aggregate, the occupancy sensors could be useful to inform, e.g., heating/cooling in the building, but individually, the readings for each room could be considered privacy-sensitive. The occupancy sensors in different rooms could use different certificate chains, if they were authorized/installed by different people. This could be used to deanonymize occupancy readings. To address this challenge, we adapt the WKD-IBE scheme that we use for end-to-end encryption to achieve an *anonymous* signature scheme that can encode the URI and expiry and support decentralized delegation. Using this technique, anonymous signatures are practical even on low-power embedded IoT devices.

1.1.4 Revocation (§5)

As stated above, JEDI keys support expiry. Therefore, it is possible to achieve a lightweight revocation scheme by delegating each key with short expiry and periodically renewing it to extend the expiry. To revoke a key, one simply does not renew it. We expect this expiry-based revocation to be sufficient for most use cases, especially for low-power devices, which typically just "sense and send."

Enforcing revocation cryptographically, without relying on expiration, is challenging. As we discuss in §5, any cryptographically-enforced scheme that provides immediate revocation (i.e., keys can be revoked without waiting for them to expire) is subject to the fundamental limitation that the sender of a message must know which recipients are revoked when it encrypts the message. JEDI provides a form of immediate revocation, subject to this constraint. We use techniques from tree-based broadcast encryption [37, 67] to encrypt in such a way that all decryption keys for that URI, *except for ones on a revocation list*, can be used to decrypt. Achieving this is nontrivial because we have to combine broadcast encryption with JEDI's semantics of hierarchical resources, expiry, and delegation. First, we modify broadcast encryption to support delegation, in such a way that if a key is revoked, all delegations made with that key are also implicitly revoked. Then, we integrate broadcast revocation, in a *non-black-box* way, with JEDI's encryption and delegation, as a third resource hierarchy alongside URIs and expiry.

1.2 Summary of Evaluation

For our evaluation, we use JEDI to encrypt messages transmitted over bw2 [5, 27], a deployed open-source messaging system for smart buildings, and demonstrate that JEDI's overhead is small in the critical path. We also evaluate JEDI for a commercially available sensor platform called "Hamilton" [49], and show that a Hamilton-based sensor sending one sensor reading every 30 seconds would see several years of battery lifetime when sending sensor readings encrypted with JEDI. As Hamilton is among the least powerful platforms that will participate in IoT (farthest to the right in Fig. 1), this validates that JEDI is practical across the IoT spectrum.

2 JEDI's Model and Threat Model

A principal can post a message to a resource in a hierarchy by encrypting it according to the resource's URI, hierarchy's public parameters, and current time, and passing it to the un-

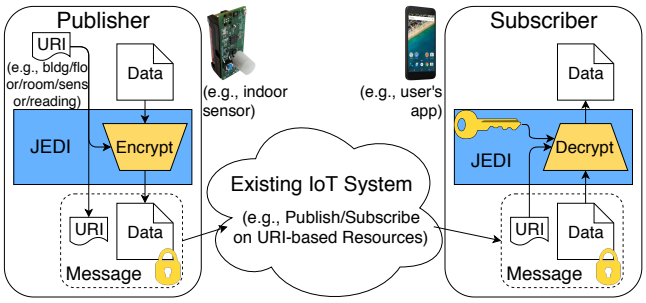


Figure 3: Applying JEDI to a smart buildings IoT system. Components introduced by JEDI are shaded. The subscriber's key is obtained via JEDI's decentralized delegation (Fig. 2).

derlying system that delivers it to the relevant subscribers. Given the secret key for a resource subtree and time range, a principal can generate a secret key for a subset of those resources and subrange of that time range, and give it to another principal, as in Fig. 2. The receiving principal can use the delegated key to decrypt messages that are posted to a resource in that subset at a time during that subrange.

JEDI does not require the structure of the resource hierarchy to be fixed in advance. In Fig. 2, the campus facilities manager, when granting access to `buildingA/*` to the building manager, need not be concerned with the structure of the subtree rooted at `buildingA`. This allows the building manager to organize `buildingA/*` independently.

2.1 Trust Assumptions

A principal is trusted for the resources it owns or was given access to (for the time ranges for which it was given access). In other words, an adversary who compromises a principal can read all resources that principal can read and forge new messages as if it were that principal. In particular, an adversary who compromises the authority for a resource hierarchy gains control over that resource hierarchy.

JEDI allows each principal to act as an authority for its own resource hierarchy in its own trust domain, without a single authority spanning all hierarchies. In particular, *principals* are not organized hierarchically; a principal may be delegated multiple keys, each belonging to a different resource hierarchy. In the example in Fig. 2, Alice might also receive JEDI keys from her landlord granting access to resources in her apartment building, in a separate hierarchy where her landlord is the authority. If Alice owns resources she would like to delegate to others, she can set up her own hierarchy to represent those resources. Existing IoT systems with decentralized delegation, like bw2 and Vanadium, use a similar model.

2.2 Applying JEDI to an Existing System

As shown in Fig. 3, JEDI can be applied as a wrapper around existing many-to-many communication systems, including publish-subscribe systems for smart cities. The transfer of messages from producers to consumers is handled by the existing system. A common design used by such systems is to have a central broker (or router) forward messages; however, an adversary who compromises the broker can read all messages. In this context, JEDI's end-to-end encryption protects data from such an adversary. Publishers encrypt their messages with JEDI before passing them to the underlying communication system (without knowledge of who the subscribers are), and subscribers decrypt them with JEDI after receiving them from the underlying communication system (without knowledge of who the publishers are).

2.3 Comparison to a Naïve Key Server Model

To better understand the benefits of JEDI's model, consider the natural strawman of a trusted key server. This key server generates a key for every URI and time. A publisher encrypts each message for that URI with the same key. A subscriber requests this key from the trusted key server, which must first check if the subscriber is authorized to receive it. The subscriber can decrypt messages for a URI using this key, and contact the key server for a new key when the key expires. JEDI's model is better than this key server model as follows:

- *Improved security.* Unlike the trusted key server, which must always be online, the authority in JEDI can delegate qualified keys to some principals *and then go offline*, leaving these principals to qualify and delegate keys further. While the authority is offline, it is more difficult for an attacker to compromise it and easier for the authority to protect its secrets because it need only access them rarely. This reasoning is the basis of root Certificate Authorities (CAs), which access their master keys infrequently. In contrast, the trusted key server model requires a central trusted party (key server) to be online to grant/revoke access to any resource.
- *Improved privacy.* No single participant sees all delegations in JEDI. An adversary in JEDI who steals an authority's secret key can decrypt all messages for that hierarchy, but still does not learn who has access to which resource, and cannot access separate hierarchies to which the first authority has no access. In contrast, an adversary who compromises the key server learns who has access to which resource and can decrypt messages for all hierarchies.
- *Improved scalability.* In the campus IoT example above, if a building admin receives access to all sensors and all their different readings for a building, the admin must obtain a potentially very large number of keys, instead of one key for the entire building. Moreover, the campus-wide key server needs to grant decryption keys to each application owned by each employee or student at the university. Finally, the campus-wide key server must understand which delegations are allowed at lower levels in the hierarchy, requiring the entire hierarchy to be centrally administered.

2.4 IoT Gateways

Low-power wireless embedded sensors, due to power constraints, often do not use network protocols like Wi-Fi, and instead use specialized low-power protocols such as Bluetooth or IEEE 802.15.4. It is common for these devices to rely on an *application-layer gateway* to send data to computers

outside of the low-power network [91]. This gateway could be in the form of a phone app (e.g., Fitbit), or in the form of a specialized border router [25, 92]. In some traditional setups, the gateway is responsible for performing encryption/authentication [70]. JEDI accepts that gateways may be necessary for Internet connectivity, but does not rely on them for security—JEDI's cryptography is lightweight enough to run directly on the low-power sensor nodes. This approach prevents the gateway from becoming a single point of attack; an attacker who compromises the gateway cannot see or forge data for any device using that gateway.

2.5 Generalizability of JEDI's Model

Since JEDI decouples senders from receivers, it has no requirements on what happens at any intermediaries (e.g., does not require messages to be forwarded from publishers to subscribers in any particular way). Thus, JEDI works even when messages are exchanged in a broadcast medium, e.g., multicast. This also means that JEDI is more broadly applicable to systems with hierarchically organized resources. For example, URIs could correspond to filepaths in a file system, or URLs in a RESTful web service.

2.6 Security Goals

JEDI's goal is to ensure that principals can only read messages from or send messages to resources they have been granted access to receive from or send to. In the context of publish-subscribe, JEDI also hides the content of messages from an adversary who controls the router.

JEDI does not attempt to hide metadata relating to the actual transfer of messages (e.g., the URIs on which messages are published, which principals are publishing or subscribing to which resources, and timing). Hiding this metadata is a complementary task to achieving delegation and end-to-end encryption in JEDI, and techniques from the secure messaging literature [29, 32, 81] will likely be applicable.

3 End-to-End Encryption in JEDI

A central question answered in this section is: How should publishers encrypt messages before passing them to the underlying system for delivery (§3.4)? As explained in §1.1.2, although ABE, the obvious choice, is too heavy for low-power devices, we identify WKD-IBE, a more lightweight identity-based encryption scheme, as sufficient to achieve JEDI's properties. The primary challenge is to encode a sufficiently expressive rendezvous point in the WKD-IBE ID (called a *pattern*) that publishers use to encrypt messages (§3.4).

3.1 Building Block: WKD-IBE

We first explain WKD-IBE [1], the encryption scheme that JEDI uses as a building block. Throughout this paper, we denote the security parameter as κ.

In WKD-IBE, messages are encrypted with *patterns*, and keys also correspond to patterns. A pattern is a list of values: $P = (\mathbb{Z}_p^* \cup \{\perp\})^\ell$. The notation $P(i)$ denotes the ith component of P, 1-indexed. A pattern P_1 *matches* a pattern P_2 if, for all $i \in [1,\ell]$, either $P_1(i) = \perp$ or $P_1(i) = P_2(i)$. In other words, if P_1 specifies a value for an index i, P_2 must match it at i. Note that the "matches" operation is not commutative; "P_1 matches P_2" does not imply "P_2 matches P_1".

We refer to a component of a pattern containing an element of $\mathbb{Z}_p^*$ as *fixed*, and to a component that contains $\perp$ as *free*. To aid our presentation, we define the following sets:

Definition 1. *For a pattern S, we define:*

$$\mathit{fixed}(S) = \{(i, S(i)) \mid S(i) \neq \perp\}$$
$$\mathit{free}(S) = \{i \mid S(i) = \perp\}$$

A key for pattern P_1 can decrypt a message encrypted with pattern P_2 if $P_1 = P_2$. Furthermore, a key for pattern P_1 can be used to derive a key for pattern P_2, as long as P_1 matches P_2. In summary, the following is the syntax for WKD-IBE.

- **Setup**$(1^\kappa, 1^\ell) \to \mathsf{Params}, \mathsf{MasterKey}$;
- **KeyDer**$(\mathsf{Params}, \mathsf{Key}_{\mathsf{Pattern_A}}, \mathsf{Pattern_B}) \to \mathsf{Key}_{\mathsf{Pattern_B}}$, derives a key for $\mathsf{Pattern_B}$, where either $\mathsf{Key}_{\mathsf{Pattern_A}}$ is the $\mathsf{MasterKey}$, or $\mathsf{Pattern_A}$ matches $\mathsf{Pattern_B}$;
- **Encrypt**$(\mathsf{Params}, \mathsf{Pattern}, m) \to \mathsf{Ciphertext}_{\mathsf{Pattern},m}$;
- **Decrypt**$(\mathsf{Key}_{\mathsf{Pattern}}, \mathsf{Ciphertext}_{\mathsf{Pattern},m}) \to m$.

We use the WKD-IBE construction in §3.2 of [1], based on BBG HIBE [17]. Like the BBG construction, it has constant-size ciphertexts, but requires the maximum pattern length ℓ to be known at Setup time. In this WKD-IBE construction, patterns containing $\perp$ can only be used in **KeyDer**, not in **Encrypt**; we extend it to support encryption with patterns containing $\perp$. We include the WKD-IBE construction with our optimizations in the appendix of our extended paper [57].

3.2 Concurrent Hierarchies in JEDI

WKD-IBE was originally designed to allow delegation in a *single* hierarchy. For example, the original suggested use case of WKD-IBE was to generate secret keys for a user's email addresses in all valid subdomains, such as `sysadmin@*.univ.edu` [1].

JEDI, however, uses WKD-IBE in a nonstandard way to simultaneously support *multiple* hierarchies, one for URIs and one for expiry (and later in §5, one for revocation), each in the vein of HIBE. We think of the ℓ components of a WKD-IBE pattern as "slots" that are initially empty, and are progressively filled in with calls to **KeyDer**. To combine a hierarchy of maximum depth ℓ_1 (e.g., the URI hierarchy) and a hierarchy of maximum depth ℓ_2 (e.g., the expiry hierarchy), one can **Setup** WKD-IBE with the number of slots equal to $\ell = \ell_1 + \ell_2$. The first ℓ_1 slots are filled in left-to-right for the first hierarchy and the remaining ℓ_2 slots are filled in left-to-right for the second hierarchy (Fig. 4).

3.3 Overview of Encryption in JEDI

Each principal maintains a **key store** containing WKD-IBE decryption keys. To create a resource hierarchy, any principal

can call the WKD-IBE **Setup** function to create a resource hierarchy. It releases the *public parameters* and stores the *master secret key* in its key store, making it the authority of that hierarchy. To delegate access to a URI prefix for a time range, a principal (possibly the authority) searches its key store for a set of keys for a superset of those permissions. It then qualifies those keys using **KeyDer** to restrict them to the specific URI prefix and time range (§3.5), and sends the resulting keys to the recipient of the delegation.[2] The recipient accepts the delegation by adding the keys to its key store.

Before sending a message to a URI, a principal encrypts the message using WKD-IBE. The pattern used to encrypt it is derived from the URI and the current time (§3.4), which are included along with the ciphertext. When a principal receives a message, it searches its key store, using the URI and time included with the ciphertext, for a key to decrypt it.

In summary, JEDI provides the following API:

Encrypt(Message, URI, Time) → Ciphertext
Decrypt(Ciphertext, URI, Time, KeyStore) → Message
Delegate(KeyStore, URIPrefix, TimeRange) → KeySet
AcceptDelegation(KeyStore, KeySet) → KeyStore′

Note that the WKD-IBE public parameters are an implicit argument to each of these functions. Finally, although the above API lists the arguments to **Delegate** as URIPrefix and TimeRange, JEDI actually supports succinct delegation over more complex sets of URIs and timestamps (see §3.7).

3.4 Expressing URI/Time as a Pattern

A message is encrypted using a pattern derived from (1) the URI to which the message is addressed, and (2) the current time. Let $H : \{0,1\}^* \to \mathbb{Z}_p^*$ be a collision-resistant hash function. Let $\ell = \ell_1 + \ell_2$ be the pattern length in the hierarchy's WKD-IBE system. We use the first ℓ_1 slots to encode the URI, and the last ℓ_2 slots to encode the time.

Given a URI of length d, such as `a/b/c` ($d = 3$ in this example), we split it up into individual components, and append a special terminator symbol `$`: `("a", "b", "c", $)`. Using H, we map each component to $\mathbb{Z}_p^*$, and then put these values into the first $d+1$ slots. If S is our pattern, we would have $S(1) = H(\texttt{"a"})$, $S(2) = H(\texttt{"b"})$, $S(3) = H(\texttt{"c"})$, and $S(4) = H(\$)$ for this example. Now, we encode the time range into the remaining ℓ_2 slots. Any timestamp, with the granularity of an hour, can be represented hierarchically as `(year, month, day, hour)`. We encode this into the pattern like the URI: we hash each component, and assign them to consecutive slots. The final ℓ_2 slots encode the time, so the depth of the time hierarchy is ℓ_2. The terminator symbol `$` is not needed to encode the time, because timestamps always have exactly ℓ_2 components. For example, suppose that a principal sends a message to `a/b` on June 8, 2017 at 6 AM. The message is encrypted with the pattern in Fig. 4.

$S(i)$	H("a")	H("b")	H($)	⊥	H("17")	H("Jun")	H("08")	H("06")
i	1	2	3	4	5	6	7	8
	$\ell_1 = 4$ slots for URI Hierarchy				$\ell_2 = 4$ slots for Time Hierarchy			

Figure 4: Pattern S used to encrypt message sent to `a/b` on June 08, 2017 at 6 AM. The figure uses 8 slots for space reasons; JEDI is meant to be used with more slots (e.g., 20).

3.5 Producing a Key Set for Delegation

Now, we explain how to produce a key set corresponding to a URI prefix and time range. To express a URI prefix as a pattern, we do the same thing as we did for URIs, without the terminator symbol `$`. For example, `a/b/*` is encoded in a pattern S as $S(1) = H(\texttt{"a"})$, $S(2) = H(\texttt{"b"})$, and all other slots free. Given the private key for S, one can use WKD-IBE's **KeyDer** to fill in slots $3 \ldots \ell_1$. This allows one to generate the private key for `a/b`, `a/b/c`, etc.—any URI for which `a/b` is a prefix. To grant access to only a specific resource (a full URI, not a prefix), the `$` is included as before.

In encoding a time range into a pattern, single timestamps (e.g., granting access for an hour) are done as before. The hierarchical structure for time makes it possible to succinctly grant permission for an entire day, month, or year. For example, one may grant access for all of 2017 by filling in slot ℓ_2 with $H(\texttt{"2017"})$ and leaving the final $\ell_2 - 1$ slots, which correspond to month, day, and year, free. Therefore, to grant permission over a time range, *the number of keys granted is logarithmic in the length of the time range*. For example, to delegate access to a URI from October 29, 2014 at 10 PM until December 2, 2014 at 1 AM, the following keys need to be generated: `2014/Oct/29/23`, `2014/Oct/29/24`, `2014/Oct/30/*`, `2014/Oct/31/*`, `2014/Nov/*`, `2014/Dec/01/*`, and `2014/Dec/02/01`. The tree can be chosen differently to support longer time ranges (e.g., additional level representing decades), change the granularity of expiry (e.g., minutes instead of hours), trade off encryption time for key size (e.g., deeper/shallower tree), or use a more regular structure (e.g., binary encoding with logarithmic split). For example, our implementation uses a depth-6 tree (instead of depth-4), to be able to delegate time ranges with fewer keys.

In summary, to produce a key set for delegation, first determine which subtrees in the time hierarchy represent the time range. For each one, produce a separate pattern, and encode the time into the last ℓ_2 slots. Encode the URI prefix in the first ℓ_1 slots of each pattern. Finally, generate the keys corresponding to those patterns, using keys in the key store.

3.6 Optimizations for Low-Power Devices

On low-power embedded devices, performing a single WKD-IBE encryption consumes a significant amount of energy. Therefore, we design JEDI with optimizations to WKD-IBE.

[2] JEDI does not govern *how* the key set is transferred to the recipient, as there are existing solutions for this. One can use an existing protocol for one-to-one communication (e.g., TLS) to securely transfer the key set. Or, one can encrypt the key set with the recipient's (normal, non-WKD-IBE) public key, and place it in a common storage area.

3.6.1 Hybrid Encryption and Key Reuse

JEDI uses WKD-IBE in a hybrid encryption scheme. To encrypt a message m in JEDI, one samples a symmetric key k, and encrypts k with JEDI to produce ciphertext c_1. The pattern used for WKD-IBE encryption is chosen as in §3.4 to encode the *rendezvous point*. Then, one encrypts m using k to produce ciphertext c_2. The JEDI ciphertext is (c_1, c_2).

For subsequent messages, one reuses k and c_1; the new message is encrypted with k to produce a new c_2. One can keep reusing k and c_1 until the WKD-IBE pattern for encryption changes, which happens at the end of each hour (or other interval used for expiry). At this time, JEDI performs *key rotation* by choosing a new k, encrypting it with WKD-IBE using the new pattern, and then proceeding as before. Therefore, *most messages only incur cheap symmetric-key encryption*.

This also reduces the load on subscribers. The JEDI ciphertexts sent by a publisher during a single hour will all share the same c_1. Therefore, the subscriber can decrypt c_1 once for the first message to obtain k, and *cache* the mapping from c_1 to k to avoid expensive WKD-IBE decryptions for future messages sent during that hour.

Thus, expensive WKD-IBE operations are only performed upon key rotation, which happens *rarely*—once an hour (or other granularity chosen for expiry) for each resource.

3.6.2 Precomputation with Adjustment

Even with hybrid encryption and key reuse to perform WKD-IBE encryption rarely, WKD-IBE contributes significantly to the overall power consumption on low-power devices. Therefore, this section explores how to perform individual WKD-IBE encryptions more efficiently.

Most of the work to encrypt under a pattern S is in computing the quantity $Q_S = g_3 \cdot \prod_{(i,a_i)\in\text{fixed}(S)} h_i^{a_i}$, where g_3 and the h_i are part of the WKD-IBE public parameters. One may consider computing Q_S once, and then reusing its value when computing future encryptions under the same pattern S. Unfortunately, this alone does not improve efficiency because the pattern S used in one WKD-IBE encryption is different from the pattern T used for the next encryption.

JEDI, however, observes that S and T are similar; they match in the ℓ_1 slots corresponding to the URI, and the remaining ℓ_2 slots will correspond to adjacent leaves in the time tree. JEDI takes advantage of this by efficiently *adjusting* the precomputed value Q_S to compute Q_T as follows:

$$Q_T = Q_S \cdot \prod_{\substack{(i,b_i)\in\text{fixed}(T)\\ i\in\text{free}(S)}} h_i^{b_i} \cdot \prod_{\substack{(i,a_i)\in\text{fixed}(S)\\ i\in\text{free}(T)}} h_i^{-a_i} \cdot \prod_{\substack{(i,a_i)\in\text{fixed}(S)\\ (i,b_i)\in\text{fixed}(T)\\ a_i\neq b_i}} h_i^{b_i-a_i}$$

This requires one $\mathbb{G}_1$ exponentiation per differing slot between S and T (i.e., the Hamming distance). Because S and T usually differ in only the final slot of the time hierarchy, this will usually require one $\mathbb{G}_1$ exponentiation total, substantially faster than computing Q_T from scratch. Additional exponentiations are needed at the end of each day, month, and year, but they can be eliminated by maintaining additional precomputed values corresponding to the start of the current day, current month, and current year.

The protocol remains secure because a ciphertext is distributed identically whether it was computed from a precomputed value Q_S or via regular encryption.

3.7 Extensions

Via simple extensions, JEDI can support (1) wildcards in the *middle* of a URI or time, and (2) forward secrecy. We describe these extensions in the appendix of our extended paper.

3.8 Security Guarantee

We formalize the security of JEDI's encryption below.

Theorem 1. *Suppose JEDI is instantiated with a Selective-ID CPA-secure [1, 16], history-independent (defined in our extended paper [57]) WKD-IBE scheme. Then, no probabilistic polynomial-time adversary $\mathcal{A}$ can win the following security game against a challenger $\mathcal{C}$ with non-negligible advantage:*
Initialization. *$\mathcal{A}$ selects a (URI, time) pair to attack.*
Setup. *$\mathcal{C}$ gives $\mathcal{A}$ the public parameters of the JEDI instance.*
Phase 1. *$\mathcal{A}$ can make three types of queries to $\mathcal{C}$:*
1. $\mathcal{A}$ asks $\mathcal{C}$ to create a principal; $\mathcal{C}$ returns a name in $\{0,1\}^*$, which $\mathcal{A}$ can use to refer to that principal in future queries. A special name exists for the authority.
*2. $\mathcal{A}$ asks $\mathcal{C}$ for the key set of any principal; $\mathcal{C}$ gives $\mathcal{A}$ the keys that the principal has. At the time this query is made, the requested key may **not** contain a key whose URI and time are both prefixes of the challenge (URI, time) pair.*
3. $\mathcal{A}$ asks $\mathcal{C}$ to make any principal delegate a key set of $\mathcal{A}$'s choice to another principal (specified by names in $\{0,1\}^*$).
Challenge. *When $\mathcal{A}$ chooses to end Phase 1, it sends $\mathcal{C}$ two messages, m_0 and m_1, of the same length. Then $\mathcal{C}$ chooses a random bit $b \in \{0,1\}$, encrypts m_b under the challenge (URI, time) pair, and gives $\mathcal{A}$ the ciphertext.*
Phase 2. *$\mathcal{A}$ can make additional queries as in Phase 1.*
Guess. *$\mathcal{A}$ outputs $b' \in \{0,1\}$, and wins the game if $b = b'$. The advantage of an adversary $\mathcal{A}$ is $\left|\Pr[\mathcal{A}\ wins] - \frac{1}{2}\right|$.*

We prove this theorem in our extended paper [57]. Although we only achieve selective security in the standard model (like much prior work [1, 17]), one can achieve adaptive security if the hash function H in §3.5 is modeled as a random oracle [1]. It is sufficient for JEDI to use a CPA-secure (rather than CCA-secure) encryption scheme because JEDI messages are signed, as detailed below in §4.

4 Integrity in JEDI

To prevent an attacker from flooding the system with messages, spoofing fake data, or actuating devices without permission, JEDI must ensure that a principal can only send a message on a URI if it has permission. For example, an application subscribed to `buildingA/floor2/roomLHall/sensor0/temp` should be able to verify that the readings it is receiving are produced by `sensor0`, not an attacker. In addition to subscribers, an intermediate party (e.g., the router in a

publish-subscribe system) may use this mechanism to filter out malicious traffic, without being trusted to read messages.

4.1 Starting Solution: Signature Chains

A standard solution in the existing literature, used by SPKI/SDSI [31], Vanadium [77], and bw2 [5], is to include a certificate chain with each message. Just as permission to subscribe to a resource is granted via a chain of delegations in §3, permission to publish to a resource is also granted via a chain of delegations. Whereas §3 includes WKD-IBE keys in each delegation, these integrity solutions delegate signed certificates. To send a message, a principal encrypts it (§3), signs the ciphertext, and includes a certificate chain that proves that the signing keypair is authorized for that URI and time.

4.2 Anonymous Signatures

The above solution reveals the sender's identity (via its public key) and the particular chain of delegations that gives the sender access. For some applications this is acceptable, and its auditability may even be seen as a benefit. For other applications, the sender must be able to send a message anonymously. See §1.1.3 for an example. How can we reconcile *access control* (ensuring the sender has permission) and *anonymity* (hiding who the sender is)?

4.2.1 Starting Point: WKD-IBE Signatures

Our solution is to use a signature scheme based on WKD-IBE. Abdalla et al. [1] observe that WKD-IBE can be extended to a signature scheme in the same vein as has been done for IBE [18] and HIBE [46]. To sign a message $m \in \mathbb{Z}_p^*$ with a key for pattern S, one uses **KeyDer** to fill in a slot with m, and presents the decryption key as a signature.

This is our starting point for designing anonymous signatures in JEDI. A message can be signed by first hashing it to $\mathbb{Z}_p^*$ and signing the hash as above. Just as consumers receive decryption keys via a chain of delegations (§3), publishers of data receive these signing keys via chains of delegations.

4.2.2 Anonymous Signatures in JEDI

The construction in §4.2.1 has two shortcomings. First, signatures are *large*, linear in the number of fixed slots of the pattern. Second, it is unclear if they are truly *anonymous*.

Signature size. As explained in §3, we use a construction of WKD-IBE based on BBG HIBE [17]. BBG HIBE supports a property called *limited delegation* in which a secret key can be reduced in size, in exchange for limiting the depth in the hierarchy at which subkeys can be generated from it. We observe that the WKD-IBE construction also supports this feature. Because we need not support **KeyDer** for the decryption key acting as a signature, we use limited delegation to compress the signature to just two group elements.

Anonymity. The technique in §4.2.1 transforms an encryption scheme into a signature scheme, but the resulting signature scheme is not necessarily anonymous. For the particular construction of WKD-IBE that we use, however, we prove that the resulting signature scheme is indeed anonymous. Our insight is that, for this construction of WKD-IBE, keys are *history-independent* in the following sense: **KeyDer**, for a fixed Params and $\mathsf{Pattern}_B$, returns a private key $\mathsf{Key}_{\mathsf{Pattern}_B}$ with the *exact same distribution* regardless of $\mathsf{Key}_{\mathsf{Pattern}_A}$ (see §3.1 for notation). Because signatures, as described in §4.2.1, are private keys generated with **KeyDer**, they are also history-independent; a signature for a pattern has the same distribution regardless of the key used to generate it. This is precisely the anonymity property we desire.

4.3 Optimizations for Low-Power Devices

As in §3.6.1, we must avoid computing a WKD-IBE signature for every message. A simple way to do this is to sample a digital signature keypair each hour, sign the verifying key with WKD-IBE at the beginning of the hour, and sign messages during the hour with the corresponding signing key.

Unfortunately, this may still be too expensive for low-power embedded devices because it requires a digital signature, which requires asymmetric-key cryptography, for *every* message. We can circumvent this by instead (1) choosing a *symmetric* key k every hour, (2) signing k at the start of each hour (using WKD-IBE for anonymity), and (3) using k in an *authenticated broadcast protocol* to authenticate messages sent during the hour. An authenticated broadcast protocol, like μTESLA [70], generates a MAC for each message using a key whose hash is the previous key; thus, the single signed key k allows the recipient to verify later messages, whose MACs are generated with hash preimages of k. In general, this design requires stricter time synchronization than the one based on digital signatures, as the key used to generate the MAC depends on the time at which it is sent. However, for the sense-and-send use case typical of smart buildings, sensors anyway publish messages on a fixed schedule (e.g., one sample every x seconds), allowing the key to depend only on the message index. Thus, timely message delivery is the only requirement. Our scheme differs from μTESLA because the first key (end of the hash chain) is signed using WKD-IBE.

Additionally, we use a technique similar to precomputation with adjustment (§3.6.2) for anonymous signatures. Conceptually, **KeyDer**, which is used to produce signatures, can be understood as a two-step procedure: (1) produce a key of the correct form and structure (called **NonDelegableKeyDer**), and (2) re-randomize the key so that it can be safely delegated (called **ResampleKey**). Re-randomization can be accelerated using the same precomputed value Q_S that JEDI uses for encryption (§3.6.2), which can be efficiently adjusted from one pattern to the next. The result of **NonDelegableKeyDer** can also be adjusted to obtain the corresponding result for a similar pattern more efficiently. We fully explain our adjustment technique for signatures in our extended paper [57].

Finally, WKD-IBE signatures as originally proposed (§4.2.1) are verified by encrypting a random message under the pattern corresponding to the signature, and then attempting to decrypt it using the key acting as a signature. We

provide a more efficient signature verification algorithm for this construction of WKD-IBE in our extended paper [57].

4.4 Security Guarantee

The integrity guarantees of the method in this section can be formalized using a game very similar to the one in Theorem 1, so we do not present it here for brevity. We do, however, formalize the anonymous aspect of WKD-IBE signatures:

Theorem 2. *For any well-formed keys k_1, k_2 corresponding to the same (URI, time) pair in the same resource hierarchy, and any message $m \in \mathbb{Z}_p^*$, the distribution of signatures over m produced using k_1 is information-theoretically indistinguishable from (i.e., equal to) the distribution of signatures over m produced using k_2.*

This implies that even a powerful adversary who observes the private keys held by all principals cannot distinguish signatures produced by different principals, for a fixed message and pattern. No computational assumptions are required. We prove Theorem 2 in the appendix of our extended paper [57].

5 Revocation in JEDI

This section explains how JEDI keys may be revoked.

5.1 Simple Solution: Revocation via Expiry

A simple solution for revocation is to rely on expiration. In this solution, all keys are time-limited, and delegations are periodically refreshed, according to a higher layer protocol, by granting a new key with a later expiry time. In this setup, the principal who granted a key can easily revoke it by not refreshing that delegation when the key expires. We expect this solution to be sufficient for many applications of JEDI.

5.2 Immediate Revocation

Some disadvantages of the solution in §5.1 are that (1) principals must periodically come online to refresh delegations, and (2) revocation only takes effect when the delegated key expires. We would like a solution without these disadvantages.

However, any revocation scheme that does not wait for keys to expire is subject to set of *inherent* limitations. The recipient of the revoked delegation still has the revoked decryption key, so it can still decrypt messages encrypted in the same way. This means that we must either (1) rely on intermediate parties to modify ciphertexts so that revoked keys cannot decrypt them, or (2) require senders to be aware of the revocation, and encrypt messages in a different way so that revoked keys cannot decrypt them. Neither solution is ideal: (1) makes assumptions about how messages are delivered, which we have avoided thus far (§2), and requires trust in an intermediary to modify ciphertexts, and (2) weakens the decoupling of senders and receivers (§1.1). We adopt the second compromise: while senders will not need to know who are the receivers, they will need to know who has been revoked.

5.3 Immediate Revocation in JEDI

We extend tree-based broadcast encryption [37,67] to support decentralized delegation of decryption keys, and incorporate it into JEDI. We use tree-based broadcast encryption because it only requires senders to know about *revoked* users when encrypting messages, as opposed to *all* users in the system (as is required by other broadcast encryption schemes).

5.3.1 Tree-based Broadcast Encryption

Existing work [37,67] proposes two methods of tree-based broadcast encryption: Complete Subtree (CS) and Subset Difference (SD). We focus on the CS method here.

The CS method is based on a binary tree (Fig. 5) where each node corresponds to a separate keypair. Each user corresponds to a leaf of the tree and has the secret keys for all nodes on the root-to-leaf path. To encrypt a message that is decryptable by a subset of users, one finds a collection of subtrees that include all leaves except those corresponding to revoked users and encrypts the message multiple times using the public keys corresponding to the root of each subtree. By associating each node with an ID and encrypting with IBE, one can avoid generating a separate keypair for each node.

5.3.2 Modifying Broadcast Encryption for Delegation

Users in broadcast encryption do not map one-to-one to users in JEDI. To avoid confusion, we refer to "users" in broadcast encryption as "leaves" (abbreviated lf).

We modify the CS method to support delegation, as follows. Each key corresponds to a range of consecutive leaves. When a user qualifies a key to delegate to another principal, she produces a new key corresponding to a subrange of the leaves of the original key. When a key is revoked, publishers are informed of the range of leaves corresponding to the revoked key. Then, they encrypt new messages using the CS method, choosing subtrees that cover all leaves except those corresponding to revoked leaves. If a key is revoked, that key and all keys derived from it can no longer decrypt messages, which is a property that we want. Thus, if Alice has k leaves, she must store secret keys for $O(k+\log n)$ nodes, where n is the total number of leaves (so the depth of the tree is $\log n$).

In JEDI, we reduce this to $O(\log n)$ secret keys by using HIBE. We give each node v_i an identifier $\mathsf{id}(v_i) \in \{0,1\}^*$ that describes the path from the root of the tree to that node. In particular, if v_j is an ancestor of v_i, then $\mathsf{id}(v_j)$ is a prefix of $\mathsf{id}(v_i)$. Note that if we use HIBE with these IDs directly, a user with the secret key for the root can generate keys for all nodes in the tree. To fix this, we use a property called *limited delegation*, introduced by prior work [17], to generate a HIBE key that is unqualifiable (i.e., cannot be extended). For example, if Alice has leaves lf_3 to lf_4 in Fig. 5, she stores an unqualifiable key for node v_1 and a qualifiable key for node v_3. In general, each user must store $O(\log k)$ qualifiable keys and $O(\log n)$ unqualifiable keys, thus $O(\log k + \log n)$ total.

5.3.3 Using Delegable Broadcast Encryption in JEDI

Secret keys in our modified broadcast encryption scheme consist of HIBE keys, so incorporating it into JEDI is simple. As discussed in §3.2, JEDI uses WKD-IBE in a way that provides multiple concurrent hierarchies, each in the vein of

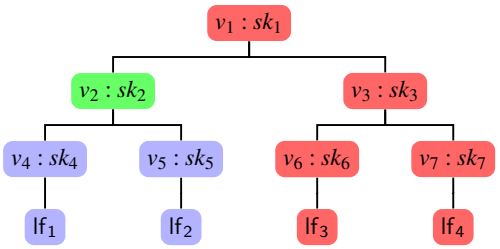


Figure 5: Key management of the CS method. Red nodes indicate nodes associated with revoked leaves. The green node is the root of the subtree covering unrevoked leaves.

HIBE. Therefore, we can instantiate a third hierarchy of depth $\ell_3 = \log n$ and use it for revocation.

Let r be the number of revoked keys. The CS method has $O(r \log \frac{n}{r})$-size ciphertexts, so JEDI ciphertexts grow to this size when revocation is used. When encrypting a message, senders use the same encryption protocol from §3 for the first $\ell_1 + \ell_2$ slots, and repeat the process, filling in the remaining ℓ_3 slots with the ID of each node used for broadcast encryption. The size of secret keys is $O(\log k + \log n)$ after our modifications to the CS method, so JEDI keys grow by this factor, to a total of $O((\log k + \log n) \cdot \log T)$ WKD-IBE keys, where T is the length of the time range for expiry.

The construction in this section works to revoke decryption keys, but cannot be used with anonymous signatures (§4.2). Extensions of tree-based broadcast encryption to signatures exist [60, 61], and we expect them to be useful to develop a construction for anonymous signatures.

How can JEDI inform publishers which leaves are revoked? One simple option is to have a global revocation list, which principals can append to. However, storing this information in a single list becomes a central point of attack, which we have avoided in our system thus far (§2). To avoid this, one can store the revocation list in a global-scale blockchain, such as Bitcoin or Ethereum, which would require an adversary to be exceptionally powerful to mount a successful attack. When revoking a set of leaves, a principal uses those keys to sign a predetermined object (as in §4.2), proving it owns an ancestor of that key in the hierarchy. To keep the revocation list private, one can use JEDI's encryption to ensure that only principals with permission to publish to a particular resource can see which keys are revoked for that resource (since publishers too have signing keys, as described in §4).

5.4 Security Guarantee

The security guarantee for immediate revocation can be stated as a modification to the game in Theorem 1. In the Initialization Phase, when $\mathcal{A}$ gives $\mathcal{C}$ the challenge (URI, time), $\mathcal{A}$ additionally submits a list of revoked leaves. Furthermore, $\mathcal{A}$ may compromise principals in possession of private keys that can decrypt the challenge (URI, time) pair during Phases 1 and 2, as long as all leaves corresponding to those keys are in the revocation list submitted in the Initialization Phase. We provide a proof in the appendix of the extended paper [57].

5.5 Optimizing JEDI's Immediate Revocation

A single JEDI ciphertext, with revocation enabled, consists of $O(r \log \frac{n}{r})$ WKD-IBE ciphertexts. To compute them efficiently, we observe that there is a large overlap in the patterns used in individual WKD-IBE encryptions, allowing us to use the "precomputation with adjustment" strategy from §3.6.2.

Even with the above optimization, immediate revocation substantially increases the cost of JEDI's cryptography. To reduce this cost, we make three observations. First, to extend JEDI's hybrid encryption to work with revocation, it is sufficient to additionally rotate keys whenever the revocation list changes, in addition to the end of each hour (as in §3.6.1). This means that, in the common case where the revocation list does not change in between two messages, efficient symmetric-key encryption can be used. Second, the revocation list used to encrypt a message need only contain revoked leaves for the *particular URI* to which the message is sent. This not only makes the broadcast encryption more efficient (smaller r), but also causes the effective revocation list for a stream of data to change even more rarely, allowing JEDI to benefit more from hybrid encryption. Third, we can do the same thing as above using the expiry time rather than the URI, allowing us to *cull* the revocation list by removing keys from it once they expire.

The efficiency of hybrid encryption depends on the revocation list changing *rarely*. We believe this is a reasonable assumption; most revocation will be handled by expiry, so immediate revocation is only needed if a principal must lose access *unexpectedly*. In the smart buildings use case (§1), for example, a key would need to be revoked if a principal unexpectedly transfers to another job.

The SD method for tree-based broadcast encryption can also be extended to support delegation and incorporated into JEDI (described in the appendix of our extended paper [57]), The SD method has smaller ciphertexts but larger keys.

6 Implementation

We implemented JEDI as a library in the Go programming language. We expect JEDI's key delegation to be computed on relatively powerful devices, like laptops, smartphones, or Raspberry Pis; less powerful devices (e.g., right half of Fig. 1) will primarily send and receive messages, rather than generate keys for delegation. Therefore, our focus for low-power platforms was on the "sense-and-send" use case [26, 38, 41] typical of indoor environmental sensing, where a device periodically publishes sensor readings to a URI. Whereas our Go library provides higher-level abstractions, we expect low-power devices to use JEDI's crypto library directly.

6.1 C/C++ Library for JEDI's Cryptography

As part of JEDI, we implemented a cryptography library optimized in assembly for three different architectures typical of IoT platforms (Fig. 1). It implements WKD-IBE and JEDI's optimizations and modifications (in §3.6, §4.3, and our full paper). The construction of WKD-IBE is based on a bilinear

group in which the Bilinear Diffie-Hellman Exponent assumption holds. We use the recent BLS12-381 elliptic curve [24].

State-of-the-art cryptography libraries implement BLS12-381, but none of them, to our knowledge, optimize for microarchitectures typical of low-power embedded platforms. To improve energy consumption, we implemented BLS12-381 in C/C++, profiled our implementation, and re-wrote performance-critical routines in assembly. We focus on ARM Cortex-M, an IoT-focused family of 32-bit microprocessors typical of contemporary low-power embedded sensor platforms [28, 49, 53]. Cortex-M processors have been used in billions of devices, including commercial IoT offerings such as Fitbit and Nest Protect. Our assembly targets Cortex-M0+, which is among the least powerful of processors in the Cortex-M series, and of those used in IoT devices (farthest to the right in Fig. 1). By demonstrating the practicality of JEDI on Cortex-M0+, we establish that JEDI is viable across the spectrum of IoT devices (Fig. 1).

The main challenge in targeting Cortex-M0+ is that the 32-bit multiply instruction provides only the lower 32 bits of the product. Even on more powerful microarchitectures without this limitation (e.g., Intel Core i7), most CPU time ($\geq 80\%$) is spent on multiply-intensive operations (e.g., BigInt multiplication and Montgomery reduction), so the lack of such an instruction was a performance bottleneck. As a workaround, our assembly code emulates multiply-accumulate with carry in 23 instructions. Cortex-M3 and Cortex-M4, which are more commonly used than Cortex-M0+, have instructions for 32-bit multiply-accumulate which produce the entire 64-bit result; we expect JEDI to be more efficient on those processors.

We also wrote assembly to optimize BLS12-381 for x86-64 and ARM64, representative of server/laptop and smartphone/Raspberry Pi, respectively (first two tiers in Fig. 1). Thus, our Go library, which runs on these non-low-power platforms, also benefits from low-level assembly optimizations.

6.2 Application of JEDI to bw2

We used our JEDI library to implement end-to-end encryption in bw2, a syndication and authorization system for IoT. bw2's syndication model is based on publish-subscribe, explained in §1. Here we discuss bw2's authorization model. Access to resources is granted via certificate chains from the authority of a resource hierarchy to a principal. Individual certificates are called Declarations of Trust (DOTs). bw2 maintains a publicly accessible registry of DOTs, implemented using blockchain smart contracts, so that principals can find the DOTs they need to form DOT chains. A *trusted* router enforces permissions granted by DOTs. Principals must present DOT chains when publishing/subscribing to resources, and the router verifies them. Note that a compromised router can read messages.

We use JEDI to enforce bw2's authorization semantics with end-to-end encryption. DOTs granting permission to subscribe now contain WKD-IBE keys to decrypt messages. By default, DOTs granting permission to publish to a URI remain unchanged, and are used as in §4.1. WKD-IBE keys may also be included in DOTs granting publish permission, for anonymous signatures (§4.2). Using our library for JEDI, we implemented a wrapper around the bw2 client library. It transparently encrypts and decrypts messages using WKD-IBE, and includes WKD-IBE parameters and keys in DOTs and principals, as needed for JEDI. bw2 signs each message with a digital signature (first alternative in §4.3).

Table 1: Latency of JEDI's implementation of BLS12-381

Operation	Laptop	Rasp. Pi	Sensor
$\mathbb{G}_1$ Mul. (Chosen Scalar)	109 µs	1.33 ms	509 ms
$\mathbb{G}_2$ Mul. (Chosen Scalar)	343 µs	3.86 ms	1.44 s
$\mathbb{G}_T$ Mul. (Rand. Scalar)	504 µs	5.47 ms	1.90 s
$\mathbb{G}_T$ Mul. (Chosen Scalar)	507 µs	5.48 ms	2.81 s
Pairing	1.29 ms	14.0 ms	3.83 s

The bw2-specific wrapper is less than 900 lines of Go code. Our implementation required no changes to bw2's client library, router, blockchain, or core—it is a separate module. Importantly, it provides the same API as the standard bw2 client library. Thus, it can be used as a drop-in replacement for the standard bw2 client library, to easily add end-to-end encryption to existing bw2 applications with minimal changes.

7 Evaluation

We evaluate JEDI via microbenchmarks, determine its power consumption on a low-power sensor, measure the overhead of applying it to bw2, and compare it to other systems.

7.1 Microbenchmarks

Benchmarks labeled "Laptop" were produced on a Lenovo T470p laptop with an Intel Core i7-7820HQ CPU @ 2.90 GHz. Benchmarks labeled "Raspberry Pi" were produced on a Raspberry Pi 3 Model B+ with an ARM Cortex-A53 @ 1.4 GHz. Benchmarks labeled "Sensor" were produced on a commercially available ultra low-power environmental sensor platform called "Hamilton" with an ARM Cortex-M0+ @ 48 MHz. We describe Hamilton in more detail in §7.3.

7.1.1 Performance of BLS12-381 in JEDI

Table 1 compares the performance of JEDI's BLS12-381 implementation on the three platforms, with our assembly optimizations. As expected from Fig. 1, the Raspberry Pi performance is an order of magnitude slower than Laptop performance, and performance on the Hamilton sensor is an additional two-to-three orders of magnitude slower.

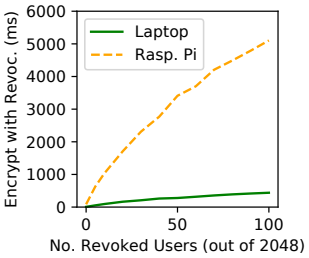
7.1.2 Performance of WKD-IBE in JEDI

Fig. 6 depicts the performance of JEDI's cryptography primitives. Fig. 6 does not include the sensor platform; §7.3 thoroughly treats performance of JEDI on low-power sensors.

In Figure 6a, we used a pattern of length 20 for all operations, which would correspond to, e.g., a URI of length 14 and an Expiry hierarchy of depth 6. To measure decryption and signing time, we measure the time to decrypt the ciphertext or sign the message, plus the time to generate a decryption key for that pattern or ID. For example, if one receives a message

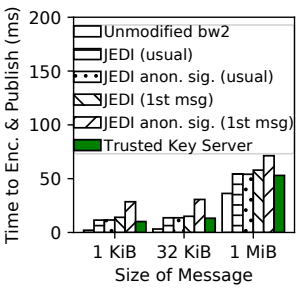
	Laptop	Rasp. Pi
Enc.	3.08 ms	37.3 ms
Dec.	3.61 ms	43.9 ms
KeyD.	4.77 ms	58.5 ms
Sign	4.80 ms	61.2 ms
Verify	4.78 ms	56.3 ms

(a) Latency of **Encrypt**, **Decrypt**, **KeyDer**, **Sign**, and **Verify** with 20 attributes

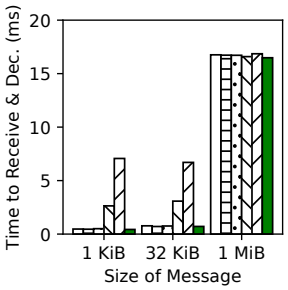


(b) Encryption with Revocation

Figure 6: Performance of JEDI's cryptography



(a) Encrypt/publish message



(b) Receive/decrypt message

Figure 7: Critical-path operations in bw2, with/without JEDI

on `a/b/c/d/e/f`, but has the key for `a/*`, he must generate the key for `a/b/c/d/e/f` to decrypt it.

Figure 6a demonstrates that the JEDI encrypts and signs messages and generates qualified keys for delegation at practical speeds. On a laptop, all WKD-IBE operations take less than 10 ms with up to 20 attributes. On a Raspberry Pi, they are 10x slower (as expected), but still run at interactive speeds.

7.1.3 Performance of Immediate Revocation in JEDI

Figure 6b shows the cost of JEDI's immediate revocation protocol (§5). A private key containing k leaves consists of $O(\log k + \log n)$ WKD-IBE secret keys where n is the total number of leaves. Therefore, the performance of immediate revocation depends primarily on the number of leaves.

To encrypt a message, one WKD-IBE encryption is performed for each subtree needed to cover all unrevoked leaves. In general, encryption is $O(r \log \frac{n}{r})$, where r is the number of revoked leaves. Each key contains a set of *consecutive* leaves, so encryption is also $O(R \log \frac{n}{R})$, where R is the number of revoked JEDI keys. Decryption time remains almost the same, since only one WKD-IBE decryption is needed.

To benchmark revocation, we use a complete binary tree of depth 16 ($n = 65536$). The time to generate a new key for delegation is essentially independent of the number of leaves conveyed in that key, because $\log k \ll \log n$. We empirically confirmed this; the time to generate a key for delegation was constant at 2.4 ms on a laptop and 31 ms on a Raspberry Pi as the number of leaves in the key was varied from 5 to 1,000.

To benchmark encryption with revocation, we assume that there exist 2,048 users in the system each with 32 leaves. We measure encryption time with a pattern with 20 fixed slots (for URI and time) as we vary the number of revoked users. Figure 6b shows that encryption becomes expensive when the revocation list is large (500 milliseconds on laptop and ≈ 5 seconds on Raspberry Pi). However, such an encryption only needs to be performed by a publisher when the URI, time, or revocation list changes; subsequent messages can reuse the underlying symmetric key (§5.5). Furthermore, the revocation list includes only revoked keys that match the (URI, time) pair being used, so it is not expected to grow very large.

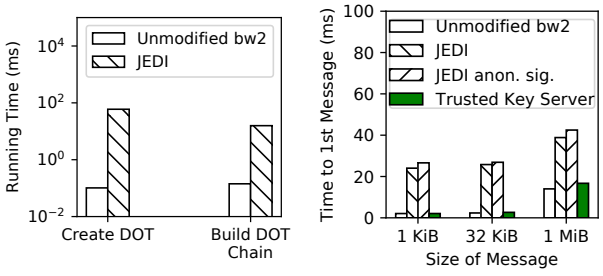
7.2 Performance of JEDI in bw2

In bw2, the two critical-path operations are publishing a message to a URI, and receiving a message as part of a subscription. We measure the overhead of JEDI for these operations because they are core to bw2's functionality and would be used by any messaging application built on bw2. Our methodology is to perform each operation repeatedly in a loop, to measure the sustained performance (operations/second), and report the average time per operation (inverse). To minimize the effect of the network, the router was on the same link as the client, and the link capacity was 1 Gbit/s. In our experiments, we used a URI of length 6 and an Expiry tree of depth 6. We also include measurements from a strawman system with pre-shared AES keys—this represents the critical-path overhead of an approach based on the Trusted Key Server discussed in §2. Our results are in Fig. 7.

We implement the optimizations in §3.6.1, so only symmetric key encryption/decryption must be performed in the common case (labeled "usual" in the diagram). However, the symmetric keys will *not* be cached for the first message sent every hour, when the WKD-IBE pattern changes. A WKD-IBE operation must be performed in this case (labeled "1st message" in the diagram). For large messages, the cost of symmetric key encryption dominates. JEDI has a particularly small overhead for 1 MiB messages in Fig. 7b, perhaps because 1 MiB messages take several milliseconds to transmit over the network, allowing the client to decrypt a message while the router is sending the next message.

We also consider creating DOTs and initiating subscriptions, which are not in the critical path of bw2. These results are in Fig. 8 (note the log scale in Fig. 8a). Creating DOTs is slower with JEDI, because WKD-IBE keys are generated and included in the DOT. Initiating a subscription in bw2 requires forming a DOT chain; in JEDI, one must also derive a private key from the DOT chain. Fig. 8a shows the time to form a short one-hop DOT chain, and in the case of JEDI, includes the time to derive the private key. For JEDI's encryption (§3), these additional costs are incurred only by DOTs that grant permission to subscribe. With anonymous signatures, DOTs granting permission to publish incur this overhead as well, as WKD-IBE keys must be included. Fig. 8b puts this in context by measuring the end-to-end latency from initiating a subscription to receiving the first message (measured using bw2's "query" functionality).

For a DOT to be usable, it must be inserted into bw2's



(a) Create DOT, Build Chain (b) Time to Query/Subscribe

Figure 8: Occasional bw2 operations, with and without JEDI

registry. This requires a blockchain transaction (not included in Fig. 8). An important consideration in this regard is *size*. In the unmodified bw2 system, a DOT that grants permission on `a/b/c/d/e/f` is 198 bytes. With JEDI, each DOT also contains multiple WKD-IBE keys, according to the time range. In the "worst case," where the start time of a DOT is Jan 01 at 01:00, and the end time is Dec 31 at 22:59, a total of 45 keys are needed. Each key is approximately 1 KiB, so the size of this DOT is approximately 45 KiB.

Because bw2's registry of DOTs is implemented using blockchain smart contracts, the bandwidth for inserting DOTs is limited. Using JEDI would increase the size of DOTs as above, resulting in an approximately 100-400x decrease in aggregate bandwidth for creating DOTs. However, this can be mitigated by changing bw2 to not store DOTs directly in the blockchain. DOTs can be stored in untrusted storage, with only their hashes stored in the blockchain-based registry. Such a solution could be based on Swarm [79] or Filecoin [43].

7.3 Feasibility on Ultra Low-Power Devices

We use a commercially available sensor platform called "Hamilton" [4, 49] built around the Atmel SAMR21 system-on-chip (SoC). The SAMR21 costs approximately $2.25 per unit [40] and integrates a low-power microcontroller and radio. The sensor platform we used in this study costs $18 to manufacture [55]. For battery lifetime calculations, we assume that the platform is powered using a CR123A Lithium battery that provides 1400 mAh at 3.0 V (252 J of energy). Such a battery costs $1. The SAMR21 is heavily constrained: it has only a 48 MHz CPU frequency based on the ARM Cortex-M0+ microarchitecture, and a total of only 32 KiB of data memory (RAM). Our goal is to validate that JEDI is practical for an ultra low-power sensor platform like Hamilton, in the context of a "sense-and-send" application in a smart building. Since most of the platform's cost ($18) comes from the on-board transducers and assembly, rather than the SAMR21 SoC, *using an even more resource-constrained SoC would not significantly decrease the platform's cost.* An analogous argument applies to energy consumption, as the transducers account for more than half of Hamilton's idle current [55].

Hamilton/SAMR21 is on the lower end of platforms typically used for sense-and-send applications in buildings. Some older studies [41, 59] use even more constrained hardware like the TelosB; this is because those studies were constrained by hardware available at the time. Modern 32-bit SoCs, like the SAMR21, offer substantially better performance at a similar price/power point to those older platforms [55].

Table 2: CPU and power costs on the Hamilton platform

Operation	Time	Average Current
Sleep (Idle)	N/A	0.0063 mA
WKD-IBE Encrypt	6.50 s	10.2 mA
WKD-IBE Encrypt and Sign	9.89 s	10.2 mA

Table 3: Average current and expected battery life (for 1400 mAh battery) for sense-and-send, with varying sample interval

	AES Only	JEDI (enc)	JEDI (enc & sign)
10 s	32 µA / 5.1 y	50 µA / 3.2 y	60 µA / 2.6 y
20 s	20 µA / 8.1 y	38 µA / 4.2 y	48 µA / 3.3 y
30 s	15 µA / 10 y	34 µA / 4.7 y	44 µA / 3.6 y

7.3.1 CPU Usage

Table 2 shows the time for encryption and anonymous signing in JEDI on Hamilton. The results use the optimizations discussed in §3.6 and §4.3, and include the time to "adjust" precomputed state. They indicate that symmetric keys can be encrypted and anonymously signed in less than 10 seconds. This is feasible given that encryption and anonymous signing occur rarely, once an hour, and need not be produced at interactive speeds in the normal "sense-and-send" use case.

7.3.2 Power Consumption

To calculate the impact on battery lifetime, we consider a "sense-and-send" application, in which the Hamilton device obtains readings from its sensors at regular intervals, and immediately sends the readings encrypted over the wireless network. We measured the average current consumed for varying sample intervals, when each message is encrypted with AES-CCM, without using JEDI ("AES Only" in Table 3). We estimate JEDI's average current based on the current, duration, and frequency (once per hour, for these estimates) of JEDI operations, and add it to the average current of the "AES Only" setup. Our estimates assume that the µTESLA-based technique in §4.3 is used to avoid attaching a digital signature to each message. We divide the battery's energy capacity by the result to compute lifetime. As shown in Table 3, JEDI decreases battery life by about 40-60%. Battery life is several years even with JEDI, acceptable for IoT sensor platforms.

JEDI's overhead depends primarily on the granularity of expiry times (one hour, for these estimates), *not* the sample interval. To improve power consumption, one could use a time tree with larger leaves, allowing principals to perform WKD-IBE encryptions and anonymous signatures less often. This would, of course, make expiry times coarser.

7.3.3 Memory Budget

Performing WKD-IBE operations requires only 6.5 KiB of data memory, which fits comfortably within the 32 KiB of data memory (RAM) available on the SAMR21. The code space required for our implementation of WKD-IBE and BLS12-

381 is about 74 KiB, which fits comfortably in the 256 KiB of code memory (ROM) provided by the SAMR21.

A related question is whether storing a hash chain in memory (as required for authenticated broadcast, §4.3) is practical. If we use a granularity of 1 minute for authenticated broadcast, the length of the hash chain is 60. At the start of an hour, one computes the entire chain, storing 10 hashes equally spaced along the chain, each separated by 5 hashes. As one progresses along the hash chain, one re-computes each set of 5 hashes one additional time. This requires storage for only 15 hashes ($<$ 4 KiB memory) and computation of only 105 hashes *per hour*, which is practical. One could possibly optimize performance further using *hierarchical hash chains* [50].

7.3.4 Impact of JEDI's Optimizations

JEDI's cryptographic optimizations (§3.6.2, §4.2.2, §4.3), which use WKD-IBE in a non-black-box manner, provide a 2-3x performance improvement. Our assembly optimizations (§6) provide an additional 4-5x improvement. Without both of these techniques, JEDI would not be practical on low-power sensors. Hybrid encryption and key reuse (§3.6.1), which let JEDI use WKD-IBE *rarely*, are also crucial.

7.4 Comparison to Other Systems

Table 4 compares JEDI to other systems and cryptographic approaches, particularly those geared toward IoT, in regard to security, expressivity and performance. We treat these existing systems as they would be used in a messaging system for smart buildings (§1). Table 4 contains quantitative comparisons to the cryptography used by these systems; for those schemes based on bilinear groups, we re-implemented them using our JEDI crypto library (§6.1) for a fair comparison.

Security. The owner of a resource is considered *trusted* for that resource, in the sense that an adversary who compromises a principal can read all of that principal's resources. In Table 4, we focus on whether a single component is trusted for *all* resources in the system. Note that, although Trusted Key Server (§2) and PICADOR [23] encrypt data in flight, granting or revoking access to a principal requires participation of an *online trusted party* to generate new keys.

Expressivity. PRE-based approaches, which associate public keys with users and support delegation via proxy re-encryption, are fundamentally coarse-grained—a re-encryption key allows *all* of a user's data to be re-encrypted. PICADOR [23] allows more fine-grained semantics, but does not enforce them cryptographically. ABE-based approaches typically do not support delegation beyond a single hop, whereas JEDI achieves multi-hop delegation. In ABE-based schemes, however, attributes/policies attached to keys can describe more complex sets of resources than JEDI. That said, a hierarchical resource representation is sufficient for JEDI's intended use case, namely smart cities; existing syndication systems for smart cities, which do not encrypt data and are unconstrained by the expressiveness of crypto schemes, choose a hierarchical rather than attribute-based representation (§1).

Performance. The Trusted Key Server (§2) is the most naïve approach, requiring an online trusted party to enforce all policy. Even so, JEDI's performance in the common case is the same as the Trusted Key Server (Fig. 7), because of JEDI's hybrid encryption—JEDI invokes WKD-IBE *rarely*. Even when JEDI invokes WKD-IBE, its performance is not significantly worse than PRE-based approaches. An alternative design for JEDI uses the GPSW KP-ABE construction instead of WKD-IBE, but it is significantly more expensive. Based Table 3, the power cost of a WKD-IBE operation *even when only invoked once per hour* contributes significantly to the overall energy consumption on the low-power IoT device; using KP-ABE instead of WKD-IBE would increase this power consumption by an order of magnitude, reducing battery life significantly.

In summary, existing systems fall into one of three categories. (1) The Trusted Key Server allows access to resources to be managed by arbitrary policies, but relies on a *central trusted party* who must be online whenever a user is granted access or is revoked. (2) PRE-based approaches, which permit sharing via re-encryption, cannot cryptographically enforce fine-grained policies or support multi-hop delegation. (3) ABE-based approaches, if carefully designed, *can* achieve the same expressivity as JEDI, but are substantially less performant and are not suitable for low-power embedded devices.

8 Related Work

We organize related work into the following categories.

Traditional Public-Key Encryption. SiRiUS [47] and Plutus [54] are encrypted filesystems based on traditional public-key cryptography, but they do not support delegable and qualifiable keys like JEDI. Akl et al. [2] and further work [33, 34] propose using key assignment schemes for access control in a hierarchy. A line of work [8, 9, 51, 80] builds on this idea to support both hierarchical structure and temporal access. Key assignment approaches, however, require the full hierarchy to be known at setup time, which is not flexible in the IoT setting. JEDI does not require this, allowing different subtrees of the hierarchy to be managed separately (§1.1, "Delegation").

Identity-Based Encryption. Tariq et al. [78] use Identity-Based Encryption (IBE) [18] to achieve end-to-end encryption in publish-subscribe systems, without the router's participation in the protocol. However, their approach does not support hierarchical resources. Further, encryption and private keys are on a credential-basis, so each message is encrypted multiple times according to the credentials of the recipients.

Wu et al. [87] use a prefix encryption scheme based on IBE for mutual authentication in IoT. Their prefix encryption scheme is different from JEDI, in that users with keys for identity `a/b/c` can decrypt messages encrypted with prefix identity `a`, `a/b` and `a/b/c`, but not identities like `a/b/c/d`.

Hierarchical Identity-Based Encryption. Since the original proposal of Hierarchical Identity-Based Encryption (HIBE) [46], there have been multiple HIBE constructions [16, 17, 45, 46] and variants of HIBE [1, 88]. Although

Table 4: Comparison of JEDI with other crypto-based IoT/cloud systems

Crypto Scheme / System	Avoids Central Trust?	Expressivity	Performance
Trusted Key Server (§2)	– No	+ Supports arbitrary policies (beyond hierarchies) – No delegation	+ ≈ 10 µs to encrypt 1 KiB message (same as JEDI in common case, faster for first message after key rotation) – Trusted party generates one key *per resource*
PRE (Lattice-Based), as used in PICADOR [23]	– No	+ Supports arbitrary policies (beyond hierarchies) – No delegation	+ ≈ 5 ms encrypt, ≈ 3 ms decrypt (similar to JEDI: 3-4 ms) – Trusted party must generate one key per sender-receiver pair
PRE (Pairing-Based), as used in Pilatus [75]	+ Yes	– Delegation is single-hop – Delegation is coarse (all-or-nothing) + Can compute aggregates on encrypted data	+ 0.6 ms encrypt, 1.3 ms re-encrypt, 0.5 ms decrypt (faster than JEDI: 3-4 ms) + Practical on constrained IoT device with crypto accelerator
CP-ABE [12]	+ Yes	+ Good fit for RBAC policies – Cannot support JEDI's hierarchy abstraction with delegation	+ Only symmetric crypto in common case – 14 ms encrypt for first time after key rotation (4-5x slower than JEDI: 3 ms)
KP-ABE, as used in Sieve [83]	+ Yes	+ Succinct delegation based on attributes – Delegation is single-hop	+ Only symmetric crypto in common case – 25 ms encrypt for first time after key rotation (8-9x slower than JEDI: 3 ms)
Delegable Large Univ. KP-ABE [48] (used in Alternative JEDI Design)	+ Yes	+ Generalizes beyond hierarchies and supports multi-hop delegation (subsumes JEDI)	+ Only symmetric crypto in common case – 60 ms encrypt for first time after key rotation (20x slower than JEDI: 3 ms) – Impractical for low-power sense-and-send
This paper: WKD-IBE [1] with Optimizations, as used in JEDI	+ Yes	+ Delegation is multi-hop + Succinct delegation of *subtrees* of resources (or more complex sets, §3.7) + Non-interactive expiry	+ After key rotation (e.g., once per hour), 3 ms encrypt, 4 ms decrypt (Fig. 6a) + Only symmetric crypto in common case + Practical for ultra low-power "sense-and-send" *without crypto accelerator*

seemingly a good match for resource hierarchies, HIBE cannot be used as a black box to efficiently instantiate JEDI. We considered alternative designs of JEDI based on existing variants of HIBE, but as we elaborate in the appendix of our extended paper [57], each resulting design is either less expressive or significantly more expensive than JEDI.

Attribute-Based Encryption. A line of work [83, 90] uses Attribute-Based Encryption (ABE) [12,48] to delegate permission. Our work additionally supports hierarchically-organized resources and decentralized delegation of keys, which [90] and [83] do not address. As discussed in §7.4, WKD-IBE is substantially more efficient than KP-ABE and provides enough functionality for JEDI.

Other approaches prefer Ciphertext-Policy ABE (CP-ABE) [12]. Existing work [84, 85] combines HIBE with CP-ABE to produce Hierarchical ABE (HABE), a solution for sharing data on untrusted cloud servers. The "hierarchical" nature of HABE, however, corresponds to the hierarchical organization of domain managers in an enterprise, not a hierarchical organization of *resources* as in our work.

Proxy Re-Encryption. NuCypher KMS [39] allows a user to store data in the cloud encrypted under her public key, and share it with another user using Proxy Re-Encryption (PRE) [14]. While NuCypher assumes limited collusion among cloud servers and recipients (e.g., m of n secret sharing) to achieve properties such as expiry, JEDI enforces expiry via cryptography, and therefore remains secure against *any* amount of collusion. Furthermore, NuCypher's solution for resource hierarchies requires a keypair for each node in the hierarchy, meaning that the creation of resources is centralized. Finally, keys in NuCypher are not qualifiable.

PICADOR [23], a publish-subscribe system with end-to-end encryption, uses a lattice-based PRE scheme. However, PICADOR requires a central Policy Authority to specify access control, by creating a re-encryption key for every permitted pair of publisher and subscriber. In contrast, JEDI's access control is decentralized.

Revocation Schemes. Broadcast encryption (BE) [19–22, 37, 58, 67] is a mechanism to achieve revocation, by encrypting messages such that they are only decryptable by a specific set of users. However, these existing schemes do not support key qualification and delegation, and therefore, cannot be used in JEDI directly. Another line of work builds revocation directly into the underlying cryptography primitive, achieving

Revocable IBE [15, 62, 72, 86], Revocable HIBE [63, 71, 73] and Revocable KP-ABE [10]. These papers use a notion of revocation in which URIs are revoked. In contrast, JEDI supports revocation at the level of keys. If multiple principals have access to a URI, and one of their keys is revoked, then the other principal can still use its key to access the resource. Some systems [11, 39] rely on the participation of servers or routers to achieve revocation.

Secure Reliable Multicast Protocol. Secure Reliable Multicast [64, 65] also uses a many-to-many communication model, and ensures correct data transfer in the presence of malicious routers. JEDI, as a protocol to *encrypt* messages, is complementary to those systems.

Authorization Services. JEDI is complementary to authorization services for IoT, such as bw2 [5], Vanadium [77], WAVE [6], and AoT [68], which focus on expressing authorization policies and enabling principals to prove they are authorized, rather than on encrypting data. Droplet [74] provides encryption for IoT, but does not support delegation beyond one hop and does not provide hierarchical resources.

An authorization service that provides secure in-band permission exchange, like WAVE [6], can be used for key distribution in JEDI. JEDI can craft keys with various permissions, while WAVE can distribute them without a centralized party by including them in its attestations.

9 Conclusion

In this paper, we presented JEDI, a protocol for end-to-end encryption for IoT. JEDI provides *many-to-many* encrypted communication on complex resource hierarchies, supports decentralized key delegation, and decouples senders from receivers. It provides expiry for access to resources, reconciles anonymity and authorization via anonymous signatures, and allows revocation via tree-based broadcast encryption. Its encryption and integrity solutions are capable of running on embedded devices with strict energy and resource constraints, making it suitable for the Internet of Things.

Availability

The JEDI cryptography library is available at https://github.com/ucbrise/jedi-pairing and our implementation of the JEDI protocol for bw2 is available at https://github.com/ucbrise/jedi-protocol.

Acknowledgments

We thank our anonymous reviewers and our shepherd William Enck for their invaluable feedback. We would also like to thank students from the RISE Security Group and BETS Research Group for giving us feedback on early drafts of this paper. This research was supported by Intel/NSF CPS-Security #1505773 and #20153754, DoE #DE-EE000768, California Energy Commission #EPC-15-057, NSF CISE Expeditions #CCF-1730628, NSF GRFP #DGE-1752814, and gifts from the Sloan Foundation, Hellman Fellows Fund, Alibaba, Amazon, Ant Financial, Arm, Capital One, Ericsson, Facebook, Google, Intel, Microsoft, Scotiabank, Splunk and VMware.

References

[1] M. Abdalla, E. Kiltz, and G. Neven. Generalized key delegation for hierarchical identity-based encryption. Cryptology ePrint Archive, Report 2007/221.

[2] S. G. Akl and P. D. Taylor. Cryptographic solution to a problem of access control in a hierarchy. *TOCS*, 1983.

[3] M. P Andersen, G. Fierro, and D. E. Culler. System design for a synergistic, low power mote/BLE embedded platform. In *IPSN*, 2016.

[4] M. P. Andersen, H.-S. Kim, and D. E. Culler. Hamilton - a cost-effective, low power networked sensor for indoor environment monitoring. In *BuildSys*, 2017.

[5] M. P. Andersen, J. Kolb, K. Chen, D. E. Culler, and R. Katz. Democratizing authority in the built environment. In *BuildSys*, 2017.

[6] M. P Andersen, S. Kumar, M. AbdelBaky, G. Fierro, J. Kolb, H.-S. Kim, D. E. Culler, and R. A. Popa. WAVE: A decentralized authorization framework with transitive delegation. In *USENIX Security*, 2019.

[7] P. Arjunan, N. Batra, H. Choi, A. Singh, P. Singh, and M. B. Srivastava. SensorAct: A privacy and security aware federated middleware for building management. In *BuildSys*, 2012.

[8] M. J. Atallah, M. Blanton, N. Fazio, and K. B. Frikken. Dynamic and efficient key management for access hierarchies. In *TISSEC*, 2009.

[9] M. J. Atallah, M. Blanton, and K. B. Frikken. Incorporating temporal capabilities in existing key management schemes. In *ESORICS*, 2007.

[10] N. Attrapadung and H. Imai. Conjunctive broadcast and attribute-based encryption. In *ICPBC*, 2009.

[11] S. Belguith, S. Cui, M. R. Asghar, and G. Russello. Secure publish and subscribe systems with efficient revocation. In *SAC*, 2018.

[12] J. Bethencourt, A. Sahai, and B. Waters. Ciphertext-policy attribute-based encryption. In *S&P*, 2007.

[13] A. Birgisson, J. G. Politz, Ú. Erlingsson, A. Taly, M. Vrable, and M. Lentczner. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *NDSS*, 2014.

[14] M. Blaze, G. Bleumer, and M. Strauss. Divertible protocols and atomic proxy cryptography. *EUROCRYPT*, 1998.

[15] A. Boldyreva, V. Goyal, and V. Kumar. Identity-based encryption with efficient revocation. In *CCS*, 2008.

[16] D. Boneh and X. Boyen. Efficient selective-ID secure identity-based encryption without random oracles. In *EUROCRYPT*, 2004.

[17] D. Boneh, X. Boyen, and E.-J. Goh. Hierarchical identity based encryption with constant size ciphertext. In

EUROCRYPT and Cryptology ePrint Archive, 2005.

[18] D. Boneh and M. Franklin. Identity-based encryption from the Weil pairing. In *CRYPTO*, 2001.

[19] D. Boneh, C. Gentry, and B. Waters. Collusion resistant broadcast encryption with short ciphertexts and private keys. In *CRYPTO*, 2005.

[20] D. Boneh and B. Waters. A fully collusion resistant broadcast, trace, and revoke system. In *CCS*, 2006.

[21] D. Boneh, B. Waters, and M. Zhandry. Low overhead broadcast encryption from multilinear maps. In *CRYPTO*, 2014.

[22] D. Boneh and M. Zhandry. Multiparty key exchange, efficient traitor tracing, and more from indistinguishability obfuscation. *Algorithmica*, 2017.

[23] C. Borcea, A. B. D. Gupta, Y. Polyakov, K. Rohloff, and G. Ryan. PICADOR: End-to-end encrypted publish-subscribe information distribution with proxy re-encryption. *FGCS*, 2017.

[24] S. Bowe. BLS12-381: New zk-SNARK elliptic curve construction, 2018. https://z.cash/blog/new-snark-curve/.

[25] A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. P. Vasseur, and R. Alexander. RPL: IPv6 routing protocol for low-power and lossy networks. RFC, RFC Editor, 2012.

[26] D. Brunelli, I. Minakov, R. Passerone, and M. Rossi. POVOMON: An ad-hoc wireless sensor network for indoor environmental monitoring. In *EESMS*, 2014.

[27] bw2. https://github.com/immesys/bw2.

[28] B. Campbell. Introducing Hail, 2017. https://www.tockos.org/blog/2017/introducing-hail/.

[29] R. Cheng, W. Scott, B. Parno, I. Zhang, A. Krishnamurthy, and T. Anderson. Talek: A private publish-subscribe protocol. Technical report, University of Washington CSE, 2016.

[30] Cisco. The Internet of things reference model. Technical report, Cisco, 2014.

[31] D. Clarke, J.-E. Elien, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 2001.

[32] H. Corrigan-Gibbs, D. Boneh, and D. Mazières. Riposte: An anonymous messaging system handling millions of users. In *S&P*, 2015.

[33] J. Crampton, N. Farley, G. Gutin, M. Jones, and B. Poettering. Cryptographic enforcement of information flow policies without public information. In *ACNS*, 2015.

[34] J. Crampton, K. Martin, and P. Wild. On key assignment for hierarchical access control. In *CSFW*, 2006.

[35] S. Dawson-Haggerty, X. Jiang, G. Tolle, J. Ortiz, and D. E. Culler. sMAP: A simple measurement and actuation profile for physical information. In *SenSys*, 2010.

[36] S. Dawson-Haggerty, A. Krioukov, J. Taneja, S. Karandikar, G. Fierro, N. Kitaev, and D. E. Culler. BOSS: Building operating system services. In *NSDI*, 2013.

[37] Y. Dodis and N. Fazio. Public key broadcast encryption for stateless receivers. In *DRM*, 2002.

[38] P. Dutta, D. E. Culler, and S. Shenker. Procrastination might lead to a longer and more useful life. In *HotNets*, 2007.

[39] M. Egorov and M. Wilkison. NuCypher KMS: decentralized key management system. *CoRR*, 2017.

[40] DigiKey Electronics. Atsamr21e18a-mu microchip technology. Feb. 8, 2019.

[41] M. C. Feldmeier. *Personalized Building Comfort Control*. PhD thesis, MIT, 2009.

[42] G. Fierro and D. E. Culler. XBOS: An extensible building operating system. Technical report, EECS Department, University of California, Berkeley, 2015.

[43] Filecoin. https://filecoin.io. Jan. 19, 2018.

[44] T. Frosch, C. Mainka, C. Bader, F. Bergsma, J. Schwenk, and T. Holz. How secure is TextSecure? In *EuroS&P*, 2016.

[45] C. Gentry and S. Halevi. Hierarchical identity based encryption with polynomially many levels. In *TCC*, 2009.

[46] C. Gentry and A. Silverberg. Hierarchical ID-based cryptography. In *ASIACRYPT*, 2002.

[47] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In *NDSS*, 2003.

[48] V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *CCS*, 2006.

[49] Hamilton IoT. https://hamiltoniot.com/.

[50] Y.-C. Hu, M. Jakobsson, and A. Perrig. Efficient constructions for one-way hash chains. In *ACNS*, 2005.

[51] H.-F. Huang and C.-C. Chang. A new cryptographic key assignment scheme with time-constraint access control in a hierarchy. *Computer Standards & Interfaces*, 2004.

[52] J. Hviid and M. B. Kjaergaard. Activity-tracking service for building operating systems. In *PerCom*, 2018.

[53] imix: Low-power IoT research platform, 2017. https://github.com/helena-project/imix.

[54] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *FAST*, 2003.

[55] H.-S. Kim, M. P. Andersen, K. Chen, S. Kumar, W. J. Zhao, K. Ma, and D. E. Culler. System architecture directions for post-SoC/32-bit networked sensors. In *SenSys*, 2018.

[56] A. Krioukov, G. Fierro, N. Kitaev, and D. E. Culler.

Building application stack (BAS). In *BuildSys*, 2012.

[57] S. Kumar, Y. Hu, M. P Andersen, R. A. Popa, and D. E. Culler. JEDI: Many-to-many end-to-end encryption and key delegation for IoT. *CoRR*, 2019.

[58] A. Lewko, A. Sahai, and B. Waters. Revocation systems with very small private keys. In *S&P*, 2010.

[59] C. Li, Z. Li, M. Li, F. Meggers, A. Schlueter, and H. B. Lim. Energy efficient HVAC system with distributed sensing and control. In *ICDCS*, 2014.

[60] B. Libert, T. Peters, and M. Yung. Group signatures with almost-for-free revocation. In *CRYPTO*, 2012.

[61] B. Libert, T. Peters, and M. Yung. Scalable group signatures with revocation. In *EUROCRYPT*, 2012.

[62] B. Libert and D. Vergnaud. Adaptive-ID secure revocable identity-based encryption. In *CT-RSA*, 2009.

[63] W. Liu, J. Liu, Q. Wu, B. Qin, D. Naccache, and H. Ferradi. Compact CCA2-secure hierarchical identity-based broadcast encryption for fuzzy-entity data sharing. Cryptology ePrint Archive, Report 2016/634.

[64] D. Malkhi, M. Merritt, and O. Rodeh. Secure reliable multicast protocols in a WAN. *Dist. Computing*, 2000.

[65] D. Malkhi and M. Reiter. A high-throughput secure reliable multicast protocol. *Computer Security*, 1997.

[66] A. Mehanovic, T. H. Rasmussen, and M. B. Kjærgaard. Brume - a horizontally scalable and fault tolerant building operating system. In *IoTDI*, 2018.

[67] D. Naor, M. Naor, and J. Lotspiech. Revocation and tracing schemes for stateless receivers. In *CRYPTO*, 2001.

[68] A. L. M. Neto, A. L. F. Souza, I. Cunha, M. Nogueira, I. O. Nunes, L. Cotta, N. Gentille, A. A. F. Loureiro, D. F. Aranha, H. K. Patil, and L. B. Oliveira. AoT: Authentication and access control for the entire IoT device life-cycle. In *SenSys*, 2016.

[69] Particle Mesh. https://www.particle.io/mesh. Feb. 2, 2019.

[70] A. Perrig, R. Szewczyk, V. Wen, D. E. Culler, and J. D. Tygar. SPINS: Security protocols for sensor networks. In *MobiCom*, 2001.

[71] J. H. Seo and K. Emura. Efficient delegation of key generation and revocation functionalities in identity-based encryption. In *CT-RSA*, 2013.

[72] J. H Seo and K. Emura. Revocable identity-based encryption revisited: Security model and construction. In *PKC*, 2013.

[73] J. H. Seo and K. Emura. Revocable hierarchical identity-based encryption: History-free update, security against insiders, and short ciphertexts. In *CT-RSA*, 2015.

[74] H. Shafagh, L. Burkhalter, S. Duquennoy, A. Hithnawi, and S. Ratnasamy. Droplet: Decentralized authorization for IoT data streams. *CoRR*, 2018.

[75] H. Shafagh, A. Hithnawi, L. Burkhalter, P. Fischli, and S. Duquennoy. Secure sharing of partially homomorphic encrypted IoT data. In *SenSys*, 2017.

[76] Solace cloud. https://solace.com. Jan. 17, 2018.

[77] A. Taly and A. Shankar. Distributed authorization in Vanadium. In *FOSAD VIII*, 2016.

[78] M. A. Tariq, B. Koldehofe, and K. Rothermel. Securing broker-less publish/subscribe systems using identity-based encryption. *TPDS*, 2014.

[79] V. Tron, A. Fischer, and N. Johnson. Smash-proof: Auditable storage for Swarm secured by masked audit secret hash. Technical report, Ethersphere, 2016.

[80] W.-G. Tzeng. A time-bound cryptographic key assignment scheme for access control in a hierarchy. *TKDE*, 2002.

[81] J. van den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *SOSP*, 2015.

[82] VOLTTRON. https://volttron.org/. Jan. 23, 2019.

[83] F. Wang, J. Mickens, N. Zeldovich, and V. Vaikuntanathan. Sieve: Cryptographically enforced access control for user data in untrusted clouds. *NSDI*, 2016.

[84] G. Wang, Q. Liu, and J. Wu. Hierarchical attribute-based encryption for fine-grained access control in cloud storage services. In *CCS*, 2010.

[85] G. Wang, Q. Liu, J. Wu, and M. Guo. Hierarchical attribute-based encryption and scalable user revocation for sharing data in cloud servers. *Computers & Security*, 2011.

[86] Y. Watanabe, K. Emura, and J. H. Seo. New revocable IBE in prime-order groups: Adaptively secure, decryption key exposure resistant, and with short public parameters. In *CT-RSA*, 2017.

[87] D. J. Wu, A. Taly, A. Shankar, and D. Boneh. Privacy, discovery, and authentication for the Internet of things. In *ESORICS*, 2016.

[88] D. Yao, N. Fazio, Y. Dodis, and A. Lysyanskaya. ID-based encryption for complex hierarchies with applications to forward security and broadcast encryption. In *CCS*, 2004.

[89] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient MAC protocol for wireless sensor networks. In *INFOCOM*, 2002.

[90] S. Yu, C. Wang, K. Ren, and W. Lou. Achieving secure, scalable, and fine-grained data access control in cloud computing. In *INFOCOM*, 2010.

[91] T. Zachariah, N. Klugman, B. Campbell, J. Adkins, N. Jackson, and P. Dutta. The Internet of things has a gateway problem. In *HotMobile*, 2015.

[92] Zigbee gateway. https://www.zigbee.org/zigbee-for-developers/zigbee-gateway/. Feb. 13, 2019.

Birthday, Name and Bifacial-security: Understanding Passwords of Chinese Web Users

Ding Wang[†*], Ping Wang[†*], Debiao He[§], Yuan Tian[‡]
[†] *Peking University, Beijing 100871, China; {wangdingg, pwang}@pku.edu.cn*
[*] *Key Lab of High-Condence Software Technology (PKU), Ministry of Education, China*
[§] *School of Cyber Science and Engineering, Wuhan University, China; hedebiao@whu.edu.cn*
[‡] *School of Engineering and Applied Science, University of Virginia; yuant@virginia.edu*

Abstract

Much attention has been paid to passwords chosen by English speaking users, yet only a few studies have examined how *non*-English speaking users select passwords. In this paper, we perform *an extensive, empirical analysis* of 73.1 million real-world Chinese web passwords in comparison with 33.2 million English counterparts. We highlight a number of interesting structural and semantic characteristics in Chinese passwords. We further evaluate the security of these passwords by employing two state-of-the-art cracking techniques. In particular, our cracking results reveal the *bifacial-security* nature of Chinese passwords. They are weaker against online guessing attacks (i.e., when the allowed guess number is small, $1\sim10^4$) than English passwords. But out of the remaining Chinese passwords, they are stronger against offline guessing attacks (i.e., when the guess number is large, $>10^5$) than their English counterparts. This reconciles two conflicting claims about the strength of Chinese passwords made by Bonneau (IEEE S&P'12) and Li et al. (Usenix Security'14 and IEEE TIFS'16). At 10^7 guesses, the success rate of our improved PCFG-based attack against the Chinese datasets is 33.2%$\sim$49.8%, indicating that our attack can crack 92% to 188% *more* passwords than the state of the art. We also discuss the implications of our findings for password policies, strength meters and cracking.

1 Introduction

Textual passwords are the dominant form of access control in almost every web service today. Although their security pitfalls were revealed as early as four decades ago [39] and various alternative authentication methods (e.g., graphical passwords and multi-factor authentication) have been proposed since then, passwords are still widely used. For one reason, passwords offer many advantages, such as low deployment cost, easy recovery, and remarkable simplicity, which cannot always be offered by other authentication methods [6]. For another reason, there is a lack of effective tools to quantify the less obvious costs of replacing passwords [8] because the marginal gains are often insufficient to make up for the significant transition costs. Furthermore, users also favor passwords. A recent survey on 1,119 US users [49] showed that 58% of the participants prefer passwords as their online login credentials, while only 16% prefer biometrics, and 10% prefer other ways. Thus, passwords are likely to persist in the foreseeable future.

Despite its ubiquity, password authentication is confronted with a challenge [62]: truly random passwords are difficult for users to memorize, while easy-to-remember passwords tend to be highly predictable. To eliminate this notorious "security-usability" dilemma, researchers have put a lot of effort [12, 17, 36, 46, 47] into the following two types of studies.

Type-1 research aims at evaluating the strength of a password dataset (distribution) by measuring its statistical properties (e.g., Shannon entropy [10], α-guesswork [7], λ-success-rate [53]) or by gauging its "guessability" [24, 59]. Guessability characterizes the fraction of passwords that, at a given number of guesses, can be cracked by cracking algorithms such as Markov-Chains [36] and probabilistic context-free grammars (PCFG) [58]. As with most of these previous studies, we mainly consider *trawling guessing* [55], while other attacking vectors (e.g., phishing, shoulder-surfing and targeted guessing [56]) are outside of our focus. Hereafter, whenever the term "guessing" is used, it means trawling guessing.

Type-2 research attempts to reduce the use of weak passwords. Two approaches have been mainly utilized: proactive password checking [25, 32] and password strength meter [13, 59]. The former checks the user-selected passwords and only accepts those that comply with the system policy (e.g., at least 8 characters long). The latter is typically a visual feedback of password strength, often presented as a colored bar to help users create stronger passwords [17]. Most of today's leading

sites employ a combination of these two approaches to prevent users from choosing weak passwords. In this work, though we mainly focus on type-1 research, our findings are also helpful for type-2 research.

Existing work (e.g., [14,17,27,37,42]) mainly focuses on passwords chosen by English speaking users. Relatively little attention has been paid to the characteristics and strength of passwords chosen by those who speak other native languages. For instance, "`woaini1314`" is currently deemed "Strong" by password strength meters (PSMs) of many leading services like AOL, Google, IEEE, and Sina weibo. However, this password is highly popular and prone to guessing [56]: "`woaini`" is a Chinese Pinyin phrase that means "I love you", and "`1314`" has a similar pronunciation of "for ever" in Chinese. Failing to catch this would overlook the weaknesses of Chinese passwords, thus posing high risks to the corresponding web accounts.

1.1 Motivations

There have been 802 million Chinese netizens by June, 2018 [1], which account for over 20% (and also the largest fraction) of the world's Internet population. However, to the best of our knowledge, there has been no satisfactory answer to the key questions: *(1) Are there structural or semantic characteristics that differentiate Chinese passwords from English ones? (2) How will Chinese passwords perform against the foremost attacks? (3) Are they weaker or stronger than English ones?* It is imperative to address these questions to provide both security engineers and Chinese users with necessary security guidance. For instance, if the answer to the first question is affirmative, then it indicates that *the password policies (e.g., length-8$^+$ [25] and 2Class12 [44]) and strength meters (e.g., RNN-PSM [38] and Zxcvbn [59]) originally designed for English speaking users cannot be readily applied to Chinese speaking users*.

A few password studies (e.g., [30,36,52,53,56]) have employed some Chinese datasets, yet they mainly deal with the effectiveness of various probabilistic cracking models. Relatively little attention has been given to the above three questions. As far as we know, Li et al.'s work [26, 34] may be the closest to this paper, but our work differs from it in several aspects. First, we explore a number of fundamental characteristics not covered in [26, 34], such as the extent of language dependence, length distribution, frequency distribution and various semantics. Second, our improved PCFG-based algorithm can achieve success rates from 29.41% to 39.47% at *just* 10^7 guesses, while the best success rate of their improved PCFG-based algorithm is only 17.3% at 10^{10} guesses (i.e., significantly underestimate attackers). Third, based on more comprehensive experiments, we outline the need for pairing passwords in terms of site service type when comparing password strength, which is overlooked by Li et al.'s [26, 34] and Bonneau's [7] work. Fourth, as shown in Sec. 3.2, two of Li et al.'s five Chinese datasets are improperly pre-processed when they perform data cleaning,[1] which impairs their results.

1.2 Contributions

We perform a large-scale empirical analysis by leveraging 73.1 million passwords from six popular Chinese sites and 33.2 million passwords from three English sites. Particularly, we seek for fundamental properties of user-generated passwords and systematically measure their structural patterns, semantic characteristics and strength. In summary, we make the following key contributions:

- **An empirical analysis**. By leveraging 73.1 million real-life Chinese passwords, *for the first time*, we: (1) provide a *quantitative* measurement of to what extent user passwords are influenced by their native language; (2) *systematically* explore the common semantics (e.g., date, name, place and phone #) in passwords; and (3) show that passwords of these two distinct user groups follow quite similar Zipf frequency distributions, despite being created under diversified password policies.
- **A reversal principle**. We employ two state-of-the-art password-cracking algorithms (i.e., PCFG-based and Markov-based [36]) to measure the strength of Chinese web passwords. We also improve the PCFG-based algorithm to more accurately capture passwords that are of a monotonically long structure (e.g., "`1qa2ws3ed`"). At 10^7 guesses, our algorithm can crack 92% to 188% more passwords than the best results in [34]. Particularly, we reveal a "reversal principle", i.e. the *bifacial-security* nature of Chinese passwords: when the guess number allowed is small, they are much weaker than their English counterparts, yet this relationship is reversed when the guess number is large, thereby reconciling the contradictory claims made in [7,34].
- **Some insights**. We highlight some insights for password policies, strength meters and cracking. We provide a large-scale empirical evidence that supports the hypothesis raised in the HCI community [17, 46]: users self-reported to rationally choose stronger passwords for accounts associated with a higher value, and knowingly select weaker passwords for a lower-value service even if the latter imposes a stricter policy. Our methodological approaches would also be useful for analyzing passwords of other non-English speaking users.

[1] We reported this issue to the authors of [26, 34], they have acknowledged it. As their journal paper [26] is technically a verbatim of their conference version [34], we mainly use [34] for discussion.

2 Related work

In this section, we briefly review prior research on password characteristics and security.

2.1 Password characteristics

Basic statistics. In 1979, Morris and Thompson [39] analyzed a corpus of 3,000 passwords. They reported that 71% of the passwords are no more than 6 characters long and 14% of the passwords are non-alphanumeric characters. In 1990, Klein [32] collected 13,797 computer accounts from his friends and acquaintances around US and UK. They observed that users tend to choose passwords that can be easily derived from dictionary words: a dictionary of 62,727 words is able to crack 24% of the collected accounts and 52% of the cracked passwords are shorter than 6 characters long. In 2004, Yan et al. [62] found that passwords are likely to be dictionary words since users have difficulty in memorizing random strings. On average, the password length in their user study (288 participants) is 7∼8.

In 2012, Bonneau [7] conducted a systematic analysis of 70 million Yahoo private passwords. This work examined dozens of subpopulations based on demographic factors (e.g., age, gender, and language) and site usage characteristics (e.g., email and retail). They found that even seemingly distant language communities choose the same weak passwords. This research was recently reproduced in [3] by using differential privacy techniques. Particularly, Chinese passwords are found among the most difficult ones to crack [7]. In 2014, however, Li et al. [34] argued that Bonneau's dataset is not representative of general Chinese users, because Yahoo users are familiar with English. Accordingly, Li et al. leveraged a corpus of five datasets from Chinese sites and observed that Chinese users like to use digits when creating passwords, as compared to English speaking users who like to use letters to create passwords. However, as an elementary defect, two of their Chinese datasets have not been cleaned properly (see Section 3.2), which might lead to inaccurate measures and biased comparisons. More importantly, several critical password properties (such as length distributions, frequency distributions and semantics) remain to be explored.

In 2014, Ma et al. [36] investigated password characteristics about the length and the structure of six datasets, three of which are from Chinese websites. Nonetheless, this work mainly focuses on the effectiveness of probabilistic password cracking models and pays little attention to the deeper semantics (e.g., no information is provided about the role of Pinyins, names or dates). In 2017, Pearman et al. [42] reported on an in situ examination of 4057 passwords from 154 English-speaking users over an average of 147 days. They found that the average password is composed of 2.77 character classes and is of length 9.92 characters, including 5.91 lowercase letters, 2.70 digits, 0.84 uppercase letters, and 0.46 symbols.

Semantic patterns. In 1989, Riddle et al. [43] found that birth dates, personal names, nicknames and celebrity names are popular in user-generated passwords. In 2004, Brown et al. [9] confirmed this by conducting a thorough survey that involved 218 participants and 1,783 passwords. They reported that the most frequent entity in passwords is the self (67%), followed by relatives (7%), lovers and friends; Also, names (32%) were found to be the most common information used, followed by dates (7%). Veras et al. [51] examined the 32M RockYou dataset by employing visualization techniques and observed that 15% of passwords contain sequences of 5∼8 consecutive digits, 38% of which could be further classified as dates. They also found that repeated days/months and holidays are popular, and when non-digits are paired with dates, they are most commonly single-characters or names of months.

In 2014, Li et al. [34] showed that Chinese users tend to insert Pinyins and dates into their passwords. However, many other important semantic patterns (e.g., Pinyin name and mobile number) are left unexplored. In addition, we improve upon the processes of data cleaning (see Sec. 3.2) and tuning of cracking algorithms (see Sec. 4.1) to advance beyond Li et al.'s measurement of the strength of Chinese passwords. In 2015, Ji et al. [30] noted that user-IDs and emails have a great impact on password security. For instance, 53% of Dodonew passwords can be guessed by using user-IDs within an average of 706 guesses. This motivates us to investigate to what extent the Pinyin names and Chinese-style dates impact the security of Chinese passwords. In 2018, AlSabah et al. [2] studied 79,760 passwords leaked from the Qatar National Bank, customers of which are mainly Middle Easterners. They observed that over 30% of passwords contain names, over 5% use a 2-digit birth year, and 4% include their own phone number in whole as part of their password.

2.2 Password security

A crucial password research subject is password strength. Instead of using brute-force attacks, earlier works (e.g., [32, 43]) use a combination of ad hoc dictionaries and mangling rules, in order to model the common password generation practice and see whether user passwords can be successfully rebuilt in a period of time. This technique has given rise to automated tools like John the Ripper (JTR), hashcat and L0phtCrack.

Borrowing the idea of Shannon entropy, the NIST-800-63-2 guide [10] attempts to use the concept of *password entropy* for estimating the strength of password creation policy underlying a password system. Password

entropy is calculated mainly according to the length of passwords and augmented with a bonus for special checks. Florencio and Herley [19], and Egelman et al. [17] improved this approach by adding the size of the alphabet into the calculation and called the resulting value $log_2((alpha.size)^{pass.len})$ the bit length of a password.

However, previous ad hoc metrics (e.g., password entropy and bit length) have recently been shown far from accurate by Weir et al. [57]. They suggested that the approach based on simulating password cracking sessions is more promising. They also developed a novel method that first automatically derives word-mangling rules from password datasets by using PCFG, and then instantiates the derived grammars by using string segments from external input dictionaries to generate guesses in decreasing probability order [58]. This PCFG-based cracking approach is able to crack 28% to 129% more passwords than JTR when allowed the same guess number. It is considered as a leading password cracking technique and used in a number of recent works [36, 56].

Differing from the PCFG-based approach, Narayanan and Shmatikov [40] introduced the Markov-Chain theory for assigning probabilities to letter segments, which substantially reduces the password search space. This approach was tested in an experiment against 142 real user passwords and could break 68% of them. In 2014, by utilizing various normalization and smoothing techniques from the natural language processing domain, Ma et al. [36] systematically evaluated the Markov-based model. They found it performs significantly better than the PCFG-based model at large guesses (e.g., 2^{30}) in some cases when parameterized appropriately. In this work, we perform extensive experiments by using both models to evaluate the strength of Chinese passwords.

When these password models are coupled with tools (e.g., AUTOFORGE [63]) that can automatically forge valid online login requests from the client side, server-side mechanisms like rate-limiting (see Sec. 5.2.2 of [25]) and password leakage detection [31] become necessary. However, in reality, few sites have implemented proper countermeasures to thwart online guessing. Among the 182 sites in the Alexa Top 500 sites in the US that Lu et al. [35] were able to examine, 131 sites (72%) allow frequent unsuccessful login attempts, and another 28 sites (15%) can be easily locked out, leading to denial of service attacks. This further suggests the necessity of our work—understanding the strength of Chinese passwords against online guessing.

3 Characteristics of Chinese passwords

We now investigate Chinese password characteristics, most of which are underexplored. In addition, we discuss weaknesses in previous major studies [26, 34].

3.1 Dataset and ethics consideration

Our empirical analysis employs six password datasets from Chinese websites and three password datasets from English websites. In total, these nine datasets consist of 106.3 million real-life passwords. As summarized in Table 1, these nine datasets are different in terms of service, language, culture, and size. The role of each dataset will be specified in Sec. 4 when performing strength comparison. They were hacked and made public on the Internet between 2009 and 2012, and may be a bit old. However, they can represent current passwords due to two reasons. First, Bonneau has shown that "passwords have changed only marginally since then (1990)" [7]. Second, the password ecosystem evolves very slowly. A number of recent researches (see [21, 24, 55]) reveal that password guidance and practices implemented on leading sites have seldom changed over time.

We realize that though publicly available and widely used in the literature [36, 52, 56], these datasets are private data. Thus, we only report the aggregated statistical information, and treat each individual account as confidential so that using it in our research will not increase risk to the corresponding victim, i.e., no personally identifiable information can be learned. Furthermore, these datasets may be exploited by attackers as cracking dictionaries, while our use is both beneficial for the academic community to understand password choices of Chinese netizens and for security administrators to secure user accounts. As our datasets are all publicly available, the results in this work are reproducible.

3.2 Data cleaning

We note that some original datasets (e.g., Rockyou and Tianya) include *un*-necessary headers, descriptions, footnotes, password strings with len>100, etc. Thus, before any exploration, we first launch data cleaning. We remove email addresses and user names from the original data. As with [36], we also remove strings that include symbols beyond the 95 printable ASCII characters. We further remove strings with len>30, because after manually scrutinizing the original datasets, we find that these long strings do *not* seem to be generated by users, but more likely by password managers or simply junk information. Moreover, such unusually long passwords are often beyond the scope of attackers who care about cracking efficiency [4]. In all, the fraction of excluded passwords is *negligible* (see the last column but two in Table 1), yet this cleaning step unifies the input of cracking algorithms and simplifies the later data processing.

We find that either Tianya or 7k7k has been contaminated: there is a *non-negligible* overlap between the Tianya dataset and 7k7k dataset (i.e., 40.85% of 7k7k and 24.62% of Tianya). More specifically, we were first puzzled by the fact that the password "`111222tianya`"

Table 1: Data cleaning of the password datasets leaked from nine web services ("PWs" stands for passwords).

Dataset	Web service	Language	Leaked Time	Original PWs	Miscellany	Length>30	Removed %	After cleaning	Unique PWs
Tianya	Social forum	Chinese	Dec. 2011	31,761,424	860,178	5	2.71%	30,901,241	12,898,437
7k7k	Gaming	Chinese	Dec. 2011	19,138,452	13,705,087	10,078	71.66%*	5,423,287	2,865,573
Dodonew	E-commerce&Gaming	Chinese	Dec. 2011	16,283,140	10,774	13,475	0.15%	16,258,891	10,135,260
178	Gaming	Chinese	Dec. 2011	9,072,966	0	1	0.00%	9,072,965	3,462,283
CSDN	Programmer forum	Chinese	Dec. 2011	6,428,632	355	0	0.01%	6,428,277	4,037,605
Duowan	Gaming	Chinese	Dec. 2011	5,024,764	42,024	10	0.83%	4,982,730	3,119,060
Rockyou	Social forum	English	Dec. 2009	32,603,387	18,377	3140	0.07%	32,581,870	14,326,970
Yahoo	Portal(e.g., E-commerce)	English	July 2012	453,491	10,657	0	2.35%	442,834	342,510
Phpbb	Programmer forum	English	Jan. 2009	255,421	45	3	0.02%	255,373	184,341

*We remove 13M duplicate accounts from 7k7k, because we identify that they are copied from Tianya as we will detail in Section 3.2.

was originally in the top-10 most popular list of both datasets. We manually scrutinize the original datasets (before removing the email addresses and user names) and are surprised to find that there are around 3.91 million (actually 3.91*2 million due to a split representation of 7k7k accounts, as we will discuss later) joint accounts in both datasets. In Appendix A, we provide strong evidence that someone has copied these joint accounts from Tianya to 7k7k, but *not* from 7k7k to Tianya as concluded in previous major studies [26, 34].

3.3 Password characteristics

Language dependence. There is a folklore that user-generated passwords are greatly influenced by their native languages, yet so far no large-scale quantitative measurement has ever been given. To fill this gap, we first illustrate the character distributions of the nine datasets, and then measure the closeness of passwords with their native languages in terms of inversion number of the character distributions (in descending order).

As expected, passwords from different language groups have significantly varied letter distributions (see Fig. 1). What's unexpected is that, even though generated and used in vastly diversified web services, passwords from the same language group have quite similar letter distributions. This suggests that, when given a password dataset, one can largely determine what the native language of its users is by investigating its letter distribution. Arranged in descending order, the letter distribution of all Chinese passwords is `aineo hglwuyszxqcdjmbtfrkpv`, while this distribution for all English passwords is `aeionrlstmcdyhubkgpjvfw zxq`. While some letters (e.g., 'a', 'e' and 'i') occur frequently in both groups, some letters (e.g., 'q' and 'r') only occur frequently in one group. Such information can be exploited by attackers to reduce the search space and optimize their cracking strategies. Note that, here all the percentages are handled case-insensitively.

While users' passwords are greatly affected by their native languages, the letter frequency of general language may be somewhat different from the letter frequency of passwords. *To what extent do they differ?* According to Huang et al.'s work [28], the letter distribution of Chinese language (i.e., written Chinese texts like literary work, newspapers and academic papers), when converted into Chinese Pinyin, is `inauhegoyszdjmxwqbctlpfrkv`. This shows that some letters (e.g., 'l' and 'w'), which are popular in Chinese passwords, appear much less frequently in written Chinese texts. A plausible reason may be that 'l' and 'w' is the first letter of the family names `li` and `wang` (which are the top-2 family names in China), respectively, while Chinese users, as we will show, love to use names to create their passwords.

A similar observation holds for passwords of English speaking users. The letter distribution of English language (i.e., `etaoinshrdlcumwfgypbvkjxqz`) is from www.cryptograms.org/letter-frequencies.php. For example, 't' is common in English texts, but not so common in English passwords. A plausible reason may be that 't' is used in popular words like `the`, `it`, `this`, `that`, `at`, `to`, while such words are rare in passwords.

To further explore the closeness of passwords with their native languages and with the passwords from other datasets, we measure the inversion number of the letter distribution sequences (in descending order) between two password datasets (as well as languages). The results are summarized in Table 2. "Pinyin_fullname" is a dictionary consisting of 2,426,841 unique Chinese full names (e.g., `wanglei` and `zhangwei`), "Pinyin_word" is a dictionary consisting of 127,878 unique Chinese words (e.g., `chang` and `cheng`), and these two dictionaries are detailed in Appendix B. Note that the inversion number of sequence A to sequence B is equal to that of B to A. For instance, the inversion number of `inauh` to `aniuh` is 3, which is equal to that of `aniuh` to `inauh`.

As shown in Table 2, the inversion number of letter distributions between passwords from the same language group is generally much smaller than that of passwords from different language groups. This value is also distinctly smaller than that of the letter distributions between passwords and their native language (see the bold values in Table 2). The latter is less expected. All this indicates that passwords from different languages are intrinsically different from each other in letter distributions, and that passwords are close to their native language yet the distinction is still significant (measurable).

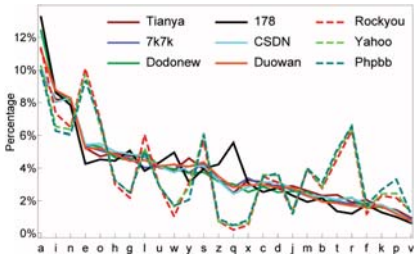


Figure 1: Letter distributions of passwords.

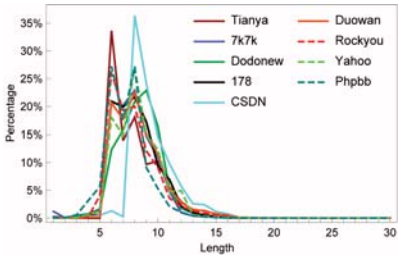


Figure 2: Length distributions of passwords.

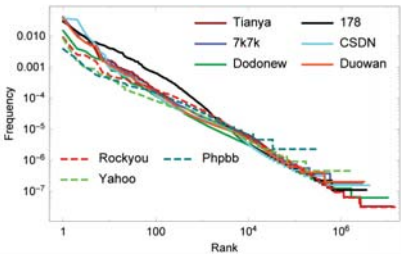


Figure 3: Freq. distributions of passwords.

Table 2: Inversion number of the letter distributions (in descending order) between two datasets.

	Tianya	7k7k	178	CSDN	Dodonew	Duowan	All Chinese PWs	Chinese language	Pinyin fullname	Pinyin word	Rockyou	Yahoo	Phpbb	All English PWs	English language
Tianya	0	15	22	42	15	17	14	40	32	37	100	100	113	100	99
7k7k	15	0	23	31	14	10	13	41	39	38	105	101	112	105	96
Dodonew	22	23	0	42	21	15	12	52	40	49	94	92	105	94	99
178	42	31	42	0	41	35	32	56	48	47	134	130	141	134	125
CSDN	15	14	21	41	0	12	15	45	39	42	95	95	106	95	96
Duowan	17	10	15	35	12	0	9	49	39	44	99	97	110	99	98
All_Chinese_PWs	14	13	12	32	15	**9**	0	44	34	43	104	102	115	104	101
Chinese_language	40	41	52	56	45	49	44	0	38	27	118	114	123	118	113
Pinyin_fullname	32	39	40	48	39	39	34	38	0	31	124	122	135	124	123
Pinyin_word	37	38	49	47	42	44	43	27	31	0	115	113	124	115	112
Rockyou	100	105	94	134	95	99	104	118	124	115	0	12	23	0	47
Yahoo	100	101	92	130	95	97	102	114	122	113	12	0	15	12	39
Phpbb	113	112	105	141	106	110	115	123	135	124	23	15	0	23	44
All_English_PWs	100	105	94	134	95	99	104	118	124	115	**0**	12	23	0	47
English_language	99	96	99	125	96	98	101	113	123	112	47	39	44	47	0

Note that, among all Chinese datasets, Duowan has the least inversion number (i.e., 9 in dark gray) with the dataset "All_Chinese_PWs". This indicates that *Duowan passwords are likely to best represent general Chinese web passwords, and thus Duowan will be selected as the training set for attacking other Chinese datasets (see Sec 5)*. For a similar reason, Rockyou will be selected as the training set when attacking English passwords.

Length distribution. Fig. 2 depicts the length distributions of passwords. Irrespective of the web service, language and culture differences, the most common password lengths of every dataset are between 6 and 10, among which length-6 and 8 take the lead. Merely passwords with lengths of 6 to 10 can account for more than 75% of every entire dataset, and this value will rise to 90% if we consider passwords with lengths of 5 to 12. Very few users prefer passwords longer than 15 characters. Notably, people seem to prefer even lengths over odd ones. Another interesting observation is that, CSDN exhibits only one peak in its length distribution curve and has many fewer passwords (i.e., only 2.16%) with length<8. This might be due to the password policy that requires the length to be no shorter than 8 on this site.

Frequency distribution. Fig. 3 portrays the frequency vs. the rank of passwords from different datasets in a log-log scale. We first sort each dataset according to the password frequency in descending order. Then, each individual password will be associated with a frequency f_r, and its rank in the frequency table is denoted by r. Interestingly, the curve for each dataset closely approximates a straight line, and this trend will be more pronounced if we take all the nine curves as a whole. This well accords with the Zipf's law [53]: f_r and r follow a relationship of the type $f_r = C \cdot r^s - C \cdot (r-1)^s \approx C \cdot s \cdot r^{s-1}$, where $C \in [0.01, 0.06]$ and $s \in [0.15, 0.40]$ are constants. Particularly, $1-s$ is the absolute value of the Zipf linear regression line's slope. The Zipf theory indicates that *the popularity of passwords decreases polynomially with the increase of their rank*. This further implies that a few passwords are overly popular (explaining why online guessing [56] can be effective, even if security mechanisms like rate-limiting and suspicious login detection [16] are implemented at the server), while the least frequent passwords are very sparsely scattered in the password space (explaining why offline guessing attackers need to consider cost-effectiveness [4] and weigh when to stop).

Top popular passwords. Table 3 shows the top-10 most frequent passwords from different services. The most frequent password among all datasets is "123456", with CSDN being the only exception due to its password policy that requires passwords to be of length 8^+ (see Fig. 2). "111111" follows on the heel. Other popular Chinese passwords include "123123", "123321" and "123456789", all composed of digits and in simple patterns such as repetition and palindrome. Love also shows its magic power: "5201314", which has a similar pronunciation of "I love you forever and ever" in Chinese,[2] appears in the top-10 lists of four Chinese

[2] https://ninchanese.com/blog/2016/05/20/520-chinese-love-word-number/

Table 3: Top-10 most popular passwords of each dataset.

Rank	Tianya	7k7k	Dodonew	178	CSDN	Duowan	Rockyou	Yahoo	Phpbb
1	123456	123456	123456	123456	123456789	123456	123456	123456	123456
2	111111	0	a123456	111111	12345678	111111	12345	password	password
3	000000	111111	123456789	zz12369	11111111	123456789	123456789	welcome	phpbb
4	123456789	123456789	111111	qiulaobai	dearbook	123123	password	ninja	qwerty
5	123123	123123	**5201314**	123456aa	00000000	000000	**iloveyou**	abc123	12345
6	123321	**5201314**	123123	wmsxie123	123123123	**5201314**	princess	123456789	12345678
7	**5201314**	123	a321654	123123	1234567890	123321	123321	12345678	letmein
8	12345678	12345678	12345	000000	88888888	a123456	rockyou	sunshine	111111
9	666666	12345678	000000	qq66666	111111111	suibian	12345678	princess	1234
10	111222tianya	wangyut2	123456a	w2w2w2	147258369	12345678	abc123	qwerty	123456789
Sum of top-10	2,297,505	440,300	533,285	793,132	670,881	338,012	669,126	4,476	7,135
Total accounts	30,901,241	5,423,287	16,258,891	9,072,965	6,428,277	4,982,730	32,581,870	442,834	255,373
% of top-10	7.43%	8.12%	3.28%	8.74%	10.44%	**6.78%**	2.05%	1.01%	**2.79%**

Table 4: Top-3 structural patterns in two user groups (each % is taken by dividing the corresponding total accounts).

Top-3 patterns in Chinese PWs	Chinese password datasets: Tianya	7k7k	Dodonew	178	CSDN	Duowan	Average of Chinese PWs	Top-3 patterns in English PWs	English password datasets: Rockyou	Yahoo	Phpbb	Average of English PWs
D(e.g., 123456)	**63.77%**	**59.62%**	**30.76%**	**48.07%**	**45.01%**	**52.84%**	**52.93%**	L(e.g., abcdef)	**41.69%**	33.03%	**50.07%**	**41.59%**
LD(e.g., a12345)	14.71%	17.98%	43.50%	31.12%	26.14%	23.97%	23.72%	LD(e.g., abc123)	27.70%	38.27%	19.14%	28.37%
DL(e.g., 12345a)	4.12%	3.91%	7.55%	6.25%	5.88%	5.83%	5.25%	D(e.g., 123456)	15.94%	5.89%	12.06%	11.30%
Sum of top-3	82.61%	81.51%	81.80%	85.45%	77.03%	82.64%	**81.90%**	Sum of top-3	85.33%	77.19%	81.25%	**81.26%**

datasets. In contrast, popular ones in English datasets tend to be meaningful letter strings (e.g., "sunshine" and "letmein"). The eternal theme of love—frankly, "iloveyou" or perhaps euphemistically, "princess"—also show up in top-10 lists of English datasets. Our results confirm the folklore [50] that "back at the dawn of the Web, the most popular password was 12345. Today, it is one digit longer but hardly safer: 123456."

It is interesting to see that only the top-10 most popular ones account for as high as 6.78%∼10.44% of each entire dataset, with Dodonew being the only exception. However, this figure for Dodonew even achieves 3.24%, while the English datasets are all below 2.80%. This indicates that top-popular Chinese passwords are more concentrated than their English counterparts, which is likely to make Chinese passwords more vulnerable to online guessing. This will be confirmed in Sec. 4.1.

Top popular structures. We have seen that digits are popular in top-10 passwords of Chinese datasets. Are they also popular in the whole datasets? We investigate the frequencies of password patterns that involve *digits*, and show the results of the top 3 most frequent ones in the left hand of Table 4. The first column of the table denotes the pattern of a password as in [58] (i.e., L denotes a lower-case sequence, D for digit sequence, U for upper-case sequence, S for symbol sequence, and the structure pattern of the password "Wanglei123" is ULD). Over 50% of the average Chinese web passwords are only composed of digits, while this value for English datasets is only 11.30%. In contrast to first D then DL, English speaking users prefer the patterns L and LD.

It is somewhat surprising to see that the sum of merely the top-3 digit-based patterns (i.e., D, LD, and DL) accounts for an average of 81.90% for Chinese datasets. In contrast, English speaking users favor letter- related patterns, and on average, their top-3 structures (i.e., L, LD and D) also account for slightly over 80%. This indicates that, unlike English speaking users, Chinese speaking users are inclined to employ digits to build their passwords — digits in Chinese passwords serve the role of letters that play in English passwords, while letters in Chinese passwords mainly come from Pinyin words/ names. This is probably due to that most Chinese users are unfamiliar with English language (and Roman letters on the keyboard). If this is the case, is there any meaningful information in these digit sequences?

Semantics in passwords. As there is little existing work, to gain an insight into the underlying semantic patterns, we have to construct semantic dictionaries from scratch by ourselves. Finally, we construct 22 dictionaries of different semantic categories (see the first column in Table 5). The detailed information about how we construct them is referred to Appendix B. To eliminate ambiguities, we use the "left-most longest match" when matching a password with each item in our dictionaries. Table 5 shows the prevalence of various semantic patterns in passwords. Lots of English speaking users tend to use raw English words as their password building blocks: 25.88% insert a 5^+-letter word into their passwords. Passwords with a 5^+-letter word account for over a third of the total passwords with a 5^+-letter substring. In comparison, fewer Chinese users (2.41%) choose English words to build passwords, yet they prefer Pinyin names (11.50%), especially *full names*.

Particularly, of all the Chinese passwords (22.42%) that include a 5^+-letter substring, more than half

Table 5: Popularity of 22 kinds of semantics in passwords (by matching our 22 semantic dictionaries).*

Semantic dictionary	Tianya	7k7k	Dodonew	178	CSDN	Duowan	Avg Chinese	Rockyou	Yahoo	Phpbb	Avg English
English_word_lower(len ≥ 5)	2.08%	2.05%	3.69%	0.83%	3.41%	2.37%	2.41%	**23.54%**	**29.49%**	**24.60%**	**25.88%**
English_firstname(len ≥ 5)	1.11%	0.93%	2.23%	0.53%	1.47%	1.19%	1.24%	18.80%	15.21%	9.20%	14.40%
English_lastname(len ≥ 5)	2.16%	2.34%	4.48%	1.93%	3.65%	2.77%	2.89%	**20.16%**	**20.82%**	**15.22%**	**18.73%**
English_fullname(len ≥ 5)	4.03%	4.30%	6.14%	4.99%	6.58%	5.07%	5.18%	13.05%	11.35%	8.25%	10.88%
English_name_any(len ≥ 5)	4.60%	4.65%	6.32%	5.20%	6.87%	5.18%	5.35%	**27.67%**	**26.51%**	**18.71%**	**24.30%**
Pinyin_word_lower(len ≥ 5)	7.34%	8.56%	10.82%	10.24%	11.51%	9.92%	9.73%	3.33%	2.99%	2.50%	2.94%
Pinyin_familyname(len ≥ 5)	1.35%	1.64%	2.34%	2.24%	2.47%	1.88%	1.99%	0.05%	0.07%	0.07%	0.06%
Pinyin_fullname(len ≥ 5)	**8.39%**	**9.87%**	**12.91%**	**11.81%**	**13.14%**	**11.29%**	**11.24%**	4.79%	4.17%	3.35%	**4.10%**
Pinyin_name_any(len ≥ 5)	8.56%	10.05%	13.31%	12.11%	13.46%	11.53%	11.50%	4.80%	4.18%	3.36%	4.11%
Pinyin_place(len ≥ 5)	1.24%	1.27%	1.64%	1.58%	2.12%	1.48%	1.55%	0.20%	0.18%	0.16%	0.18%
PW_with_a_5^+-letter_substring	**18.51%**	**19.99%**	**26.95%**	**19.38%**	**28.03%**	**21.70%**	**22.42%**	**71.69%**	**75.93%**	**68.66%**	**72.09%**
Date_YYYY	14.38%	12.82%	12.45%	10.06%	16.91%	14.33%	13.49%	4.34%	4.30%	2.77%	3.80%
Date_YYYYMMDD	6.06%	5.42%	3.93%	3.94%	8.78%	6.17%	**5.72%**	0.10%	0.05%	0.09%	0.08%
Date_MMDD	24.99%	19.97%	17.08%	16.46%	24.45%	22.59%	20.92%	7.53%	4.46%	3.59%	5.20%
Date_YYMMDD	**21.29%**	**15.89%**	**12.70%**	**13.09%**	**20.67%**	**18.28%**	**16.99%**	3.24%	1.23%	1.55%	2.01%
Date_any_above	**36.61%**	**30.39%**	**26.66%**	**27.07%**	**35.30%**	**33.58%**	**31.60%**	11.33%	8.77%	6.45%	8.85%
PW_with_a_digit	89.49%	88.42%	88.52%	90.76%	87.10%	89.26%	88.93%	54.04%	64.74%	46.14%	54.97%
PW_with_a_4^+-digit_substring	81.64%	76.98%	71.90%	78.76%	78.38%	80.60%	78.04%	24.72%	21.85%	19.33%	21.97%
PW_with_a_6^+-digit_substring	**75.59%**	**68.32%**	**61.16%**	**70.02%**	**69.87%**	**73.10%**	**69.68%**	17.77%	8.48%	11.28%	12.51%
PW_with_a_8^+-digit_substring	28.04%	27.56%	26.53%	26.37%	49.73%	31.03%	31.54%	6.88%	2.50%	3.73%	4.37%
Mobile_Phone_Number(11-digit)	**2.90%**	**1.76%**	**2.63%**	**3.97%**	**3.75%**	**2.44%**	***2.91%***	0.07%	0.01%	0.02%	0.03%
PW_with_a_11^+-digit_substring	**4.71%**	**2.09%**	**3.39%**	**5.08%**	**7.57%**	**3.35%**	**4.36%**	0.75%	0.17%	0.18%	0.37%

*Each percentage (%) is counted by the rule of "left-most longest" match and taken by dividing the corresponding password dataset size.

(11.24%) include a 5^+-letter Pinyin full name. There is also 4.10% of English passwords that contain a 5^+-letter full Pinyin name. A reasonable explanation is that many Chinese users have created accounts in these English sites. For instance, the popular Chinese Pinyin name "zhangwei" appears in both Rockyou and Yahoo. We also note that English names are also widely used in English passwords, yet full names are less popular than last names and first names.

Equally interestingly, we find that, on average, 16.99% of Chinese users insert a six-digit date into their passwords. Further considering that users love to include self information into passwords [9, 56], such dates are likely to be users' *birthdays*. Besides, about 30.89% of Chinese speaking users use a 4^+-digit date to create passwords, which is 3.59 times higher than that of English speaking users (i.e. 8.61%). Also, there are 13.49% of Chinese users inserting a four-digit year into their passwords, which is 3.55 times higher than that of English speaking users (3.80%, which is comparable to the results in [14]). We note that there might be some overestimates, for there is no way to definitely tell apart whether some digit sequences are dates or not, e.g., 010101 and 520520. These two sequences may be dates, yet they are also likely to be of other semantic meanings (e.g., 520520 sounds like "I love you I love you"). As discussed later, we have devised reasonable ways to address this issue. In all, dates play a vital role in passwords of Chinese users.

We mainly pay attention to length-4, 6 and 8 digits in passwords, because: 1) Length-4 and 6 are the most widely used lengths of PINs in the West and Asia; and 2) 6&8 are the two most frequent password lengths (see Fig. 2). It is interesting to see that 2.91% of Chinese users are likely to use their 11-digit mobile numbers as passwords, making up 39.59% of all passwords with an 11^+-digit substring. On average, 12.39% of Chinese passwords are longer than 11. Thus, if an attacker can determine (e.g., by shoulder-surfing) that the victim uses a long password, she is likely to succeed with a high chance of 23.48%($= \frac{2.91\%}{12.39\%}$) by just trying the victim's 11-digit mobile number. *This reveals a practical attacking strategy against long Chinese passwords*.

Note that there are some unavoidable ambiguities when determining whether a text/digit sequence belongs to a specific dictionary, and an improper resolution of these ambiguities would lead to an overestimation or underestimation. Here we take "YYMMDD" for illustration. For example, both 111111 and 520521 fall into "YYMMDD" and are highly popular. However, it is more likely that users choose them simply because they are easily memorable repetition numbers or meaningful strings, and counting them as dates would lead to an *overestimation*. Yet they can really be dates (e.g., 111111 stands for "Nov. 11th, 2011" and 520131 for "Jan 31th, 1952") and completely excluding them from "YYMMDD" would lead to *underestimation* of dates.

Thus, we assume that user birthdays are randomly distributed and assign the expectation of the frequency of dates (denoted by E), instead of zero, to the frequency of these abnormal dates. We manually identify 17 abnormal dates in the dictionary "YYMMDD", each of which originally has a frequency$> 10E$ and appears in every top-1000 list of the six Chinese datasets. In this way, the ambiguities can be largely resolved. We similarly tackle 16 abnormal items in "MMDD". The detailed info about these abnormal dates can be found in Appendix B. As for the other 19 dictionaries in Table 5, few abnormal items can be identified, and they are processed as usual.

Summary. We have measured nine password datasets in terms of letter distribution, length distribution, frequency

distribution and semantic patterns. To our knowledge, most of these fundamental characteristics have at most been mentioned/exampled in the literature (see [26, 30, 34, 36, 53]) but never *systematically* examined. We have identified a number of *similarities* (e.g., frequency distribution and the theme of love) and *differences* (e.g., letter distribution, structural patterns, and semantic patterns) between passwords of these two user groups.

4 Strength of Chinese web passwords

Now we employ two state-of-the-art password attacking algorithms (i.e., PCFG-based [58] and Markov-based [36]) to evaluate the strength of Chinese web passwords. We further investigate whether the characteristics identified in Sec. 3.3 (e.g., dates and Pinyin names) can be practically exploited to facilitate password guessing.

Necessity of pairing passwords by service type. There are a number of confounding factors that impact password security, among which language, service type, and password policy are the three most important ones [29, 53, 56]. As shown in [36, 53], except for CSDN that imposes a length 8^+ policy, all our datasets (Table 1) reflect no explicit policy requirements. It has recently been revealed that users often rationally choose robust passwords for accounts perceived to be important [46], while knowingly choose weak passwords for unimportant accounts [17]. Since accounts of the same service would generally have the same level of value for users, we divide datasets into three pairs according to their types of services (i.e., Tianya vs. Rockyou, Dodonew vs. Yahoo, and CSDN vs. Phpbb) for fairer strength comparison, as opposed to existing works [7, 26, 34] that do not take into account the site service type. We emphasize that it is less reasonable if one compares Dodonew passwords (from an e-commerce site) with Phpbb passwords (from a low-value programmer forum): Even if Dodonew passwords are stronger than Phpbb passwords, one can not conclude that Chinese passwords are more secure than English ones, because there is a potential that Dodonew passwords will be weaker than Yahoo e-commerce passwords.

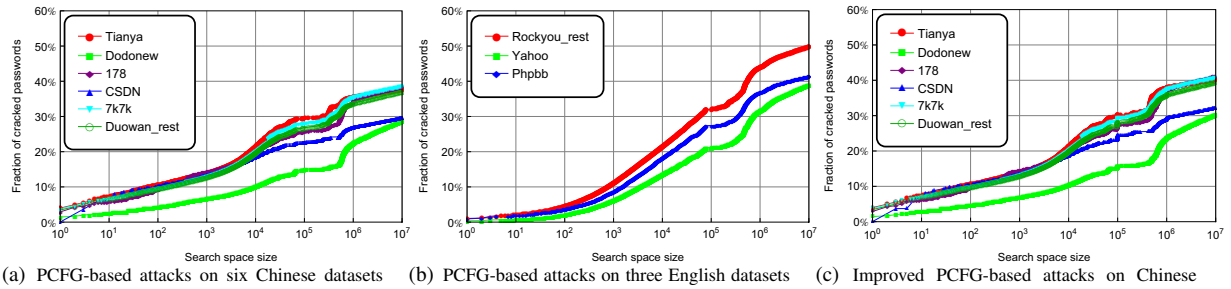
4.1 PCFG-based attacks

The PCFG-based model [58] is one of the state-of-the-art cracking models. Firstly, it divides all the passwords in a training set into segments of similar character sequences and obtains the corresponding base structures and their associated probabilities of occurrence. For example, "`wanglei@123`" is divided into the L segment "wanglei", S segment "@" and D segment "123", resulting in a base structure $L_7S_1D_3$. The probability of $L_7S_1D_3$ is $\frac{\#\text{of } L_7S_1D_3}{\#\text{of base structures}}$. Such information is used to generate the probabilistic context-free grammar.

Then, one can derive password guesses in decreasing order of probability. The probability of each guess is the product of the probabilities of the productions used in its derivation. For instance, the probability of "`liwei@123`" is computed as $P(\text{"liwei@123"}) = P(L_5S_1D_3) \cdot P(L_5 \rightarrow \text{liwei}) \cdot P(S_1 \rightarrow @) \cdot P(D_3 \rightarrow 123)$. In Weir et al.'s original proposal [58], the probabilities for D and S segments are learned from the training set by counting, yet L segments are handled either by learning from the training set or by using an external input dictionary. Ma et al. [36] revealed that PCFG-based attacks with L segments directly learned from the training set generally perform better than using an external input dictionary. Thus, we prefer to instantiate the PCFG L segments of password guesses by directly learning from the training set.

We divide the nine datasets into two groups by language. For the Chinese group of test sets, we randomly select 1M passwords from the Duowan dataset as the training set (denoted by "Duowan_1M"). The reason is that: Duowan has the least inversion number with the dataset "All Chinese PWs" (see Sec. 3.3) and is likely to best represent general Chinese web passwords. Similarly, for the English test sets, we select 1M passwords from Rockyou as the training set. Since we have only used part of Duowan and Rockyou, their remaining passwords and the other 7 datasets are used as the test sets. The attacking results on the Chinese group and English group are depicted in Fig. 4(a) and Fig. 4(b), respectively.

Bifacial-security. When the guess number (i.e., search space size) allowed is below about 3,000, Chinese passwords are generally much *weaker* than English passwords from the same service (i.e., Tianya vs. Rockyou, Dodonew vs. Yahoo, and CSDN vs. Phpbb). For example, at 100 guesses, the success rate against Tianya, Dodonew and CSDN is 10.2%, 4.3% and 9.7%, respectively, while their English counterparts are 4.6%, 1.9% and 3.7%, respectively. However, when the search space size is above 10,000, Chinese web passwords are generally much *stronger* than their English counterparts. For example, at 10 million guesses, the success rate against Tianya, Dodonew and CSDN is 37.5%, 28.8% and 29.9%, respectively, while their English counterparts are 49.7%, 39.0% and 41.4%, respectively. The strength gap will be even wider when the guess number further increases. This reveals a reversal principle, i.e., the bifacial-security nature of Chinese passwords: they are more vulnerable to online guessing attacks (i.e., when the guess number allowed is small) than English passwords; But out of the remaining Chinese passwords, they are more secure against offline guessing. This reconciles two drastically conflicting claims (see Sec. 1.1) made about the strength of Chinese passwords. This bifacial-security is highly due to the bifacial-density nature of digit-based passwords: *Top* digit-based passwords are



(a) PCFG-based attacks on six Chinese datasets (training set: 1M Rockyou passwords)
(b) PCFG-based attacks on three English datasets (training set: 1M Rockyou passwords)
(c) Improved PCFG-based attacks on Chinese datasets (training set: 1M Rockyou passwords)

Figure 4: General and our improved PCFG attacks on different groups of datasets. Our algorithm gains tangible advantages.

more converging (see Table 3), while digits *in general* are more random (and diverging) than letters.

A weakness in PCFG. We observe that, the original PCFG algorithm [36, 58] inherently gives extremely low probabilities to password guesses (e.g., "1q2w3e4r" and "1a2b3c4d") that are of a monotonically long base structure (e.g., $D_1L_1D_1L_1D_1$ $L_1D_1L_1$, or $(D_1L_1)_4$ for short). For example, P("1q2w3e4r") $= P((D_1L_1)_4)$ $\cdot P(D_1 \rightarrow 1)\cdot$ $P(L_1 \rightarrow q)\cdot$ $P(D_1 \rightarrow 2)\cdot$ $P(L_1 \rightarrow w)\cdot$ $P(D_1 \rightarrow 3)\cdot P(L_1 \rightarrow e)\cdot P(D_1 \rightarrow 4)\cdot P(L_1 \rightarrow r)$ can hardly be larger than 10^{-9}, for it is a multiplication of *nine* probabilities. Thus, some guesses (e.g., "1q2w3e4r" and "a12b34c56") will never appear in the top-10^7 guess list generated by the original PCFG algorithm, even if they are popular (e.g, "1q2w3e4r" appears in the top-200 list of every dataset). The essential reason is that the PCFG algorithm simply assumes that each segment in a structure is independent. Yet, in many situations this is *not* true. For instance, the four D_1 segments and L_1 segments in the structure $(D_1L_1)_4$ of password "1q2w3e4r" are evidently interrelated with each other (i.e., D_4: 1234 and L_4: qwer).

Our solution. To address this problem, we specially tackle a few password structures that are long but simple alternations of short segments by treating them as short structures. For instance, $(D_1L_1)_4$ is converted to D_4L_4, and $(D_1L_2)_3$ to D_3L_6. In this way, the probability of "1q2w3e4r" now is computed as P("1q2w3e4r")$=$ $P((D_1L_1)_4)$ $\cdot$ $P((D_1L_1)_4 \rightarrow D_4L_4)\cdot P(D_4 \rightarrow 1234)\cdot$ $P(L_4 \rightarrow \texttt{qwer})$. *Our approach is language-agnostic and constitutes a* general *amendment to the state-of-the -art PCFG-based algorithm in [36].*

To further exploit the characteristics of Chinese passwords, we insert the "Pinyin_name_any" dictionary and the six-digit date dictionary (see Sec. 3.3) into the original PCFG L-segment and D-segment dictionaries, respectively. Details about this insertion process and our improved algorithm for password-guess generation are shown in Algorithm 1. The resulting changes to the original PCFG grammars are given in Table 6.

Fig. 4(c) illustrates that, when the guess number allowed is small (e.g., 10^3), our improved attack exhibits little improvement; As the guess number grows, the improvement increases. For example, at 10^5 guesses, there is 0.09%∼0.85% improvement in success rate; at 10^6 guesses, this figure is 1.32∼4.32%; at 10^7 guesses, this figure reaches 1.70%∼4.29%. This indicates that the vulnerable behaviors of using monotonically long passwords, Pinyin names and birthdays help an attacker reduce her search space, and this issue is more serious when large guesses are allowed.

Comparison. Li et al. [34] reported that using 2M Dodonew passwords as the training set and at 10^{10} guesses, their best success rates ($=\frac{\text{\# of successfully cracked PWs}}{\text{the size of test set}}$) is about 17.30%. However, against the same Chinese test sets, our improved attack can achieve much higher success rates (29.41%∼39.47%) at only 10^7 guesses.

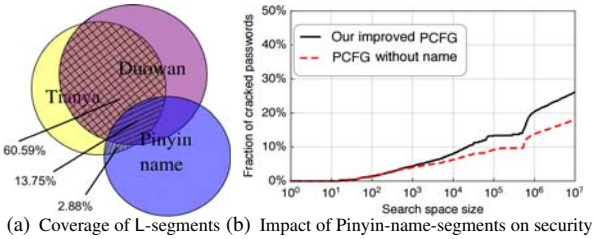
Algorithm 1: Our improved PCFG-based attack

Input: A training set $\mathcal{S}$; A name list *nameList*; A date list *dateList*; A parameter k indicating the desired size of the PW guess list that will be generated (e.g., $k = 10^7$)
Output: A PW guess list L with the top-k items

```
Training (lastly tackle monotonically long PWs:)
    for password ∈ S do
        for segment ∈ splitToSegments(password) do
            segmentSet.insert(segment)
        baseStructure ← getBaseStructure(password)
        if monotonicallyLong(baseStructure) then
            transformStructureSet.insert(baseStructure)
            baseStructure ←
              convertToshort(baseStructure)
        baseStructureSet.insert(baseStructure)
        trainingSet.insert(password)
Append name and date lists to the learned segment list:
    for name ∈ nameList do
        correctedCount =
          totalOverlapNameInSegmentSet *
          nameList.getCount(name)/totalOverlapNameInNameList
        if name ∉ segmentSet and correctedCount ≥ 1 then
            segmentSet.insert(name, correctedCount)
    for date ∈ dateList do
        if date ∉ segementSet then
            segementSet.insert(date)
Produce k guesses: As with [36] and the details are omitted.
```

Table 6: Changes caused to the original PCFG grammars

Training set	Base structures	L segments	D segments	S segments
Duowan_1M	8905+0	155693+24416	465157+20341	865+0
Duowan_All	20961+0	559017+98654	1824404+9744	2417+0



(a) Coverage of L-segments (b) Impact of Pinyin-name-segments on security

Figure 5: Coverage and security impacts of Pinyin-name-segments in the test set Tianya with L-segments involved (Duowan is the training set, Pinyin_name is an extra input dictionary in our improved PCFG attack).

This means that we can crack 70% to 128% more passwords than Li et al.'s best record. Our attacks are better because: 1) Our training-set (i.e., Duowan) is more effective than [34], for we find Duowan represents Chinese password distributions better (see Table 2) than Dodonew as used in [34]; 2) We optimize PCFG not only through adding semantic dictionaries as [34] but also through transforming monotonically long base structures.

The role of Names. In our improved PCFG-based attacks, external name segments are added into the PCFG L-segment dictionary during training, and we get gladsome increases in success rates (see Fig. 4(c)). However, such improvements are still not so prominent as compared to the prevalence of names in Chinese passwords. *To explicate this paradox, we scrutinize the internal process of PCFG-based guess generation and manage to identify its crux*. Here we take the improved PCFG attack against Tianya (trained on Duowan) as an example. During training, we have added 98K name segments (see Table 6) into the L-segment dictionary.

Fig. 5(a) demonstrates that these 98K name segments *only* cover 2.88% of the total L segments of the test set Tianya. However, the original L segments trained from Duowan can cover 13.75% of the name segments and 60.59% of the non-name L segments in Tianya. This suggests that Duowan can *well* cover the name segments in the test set Tianya, and thus the addition of some extra names would have limited impacts. This observation also holds for the other eight test sets. The detailed results are summarized in Table 7, where "Duowan1M" is Duowan_1M for short and "PY_name" is Pinyin_name for short. The fraction of L-segments in the test set y that can be covered by the set x is denoted by CoL(x).

Table 7 shows that no matter x=Duowan_1M or Duowan: 1) CoL(x) is at least 11.12 times (= 65.64%/5.90%) larger than CoL(Pinyin_name)-CoL(x); 2) CoL(Pinyin_name)∩CoL(x) is at least 1.92 times (=11.35%/5.90%) larger than CoL(Pinyin_ name)-CoL(x). This suggests that adding extra names into the PCFG L-segments when training is of limited yields. Note that, this does *not* contradict our observation that Pinyin names are prevalent in Chinese web passwords and pose a serious vulnerability. Actually, this *does* suggest that when the training set is selected properly, the name segments in passwords can be well guessed. Still, when there is no proper training set available, our improved attack would demonstrate its advantages (see Fig. 5(b)). Though our improved PCFG algorithm might not be optimal, its cracking results represent a new benchmark that any future algorithm should aim to decisively clear.

Limitations. We mainly investigate the impacts of names on password cracking, and similar observations and implications are likely to hold for dates (but with no confirmation). We leave it as future work. In addition, as our focus is the overall security of Chinese passwords (and its comparison with English counterparts), we only show the overall effectiveness of our improved PCFG attack. It is also interesting to see to what extent the improved PCFG structure and the usage of Duowan would respectively have impacts on the cracking effectiveness, but it is independent of the presented work.

Table 8: Five Markov-based attacking scenarios

Attacking scenario	Smoothing	Normalization	Markov order
#1	Laplace	End-symbol	3/4/5
#2	Laplace	Distribution	3/4/5
#3	Good-Turing	End-symbol	3/4/5
#4	Good-Turing	Distribution	3/4/5
#5	Backoff	End-symbol	Backoff

4.2 Markov-based attacks

To show the robustness of our findings about password security, we further conduct Markov-based attacks.

4.2.1 Markov-based experimental setups

To make our experiments as reproducible as possible, we now detail the setups. As recommended in [36], we consider two smoothing techniques (i.e., Laplace Smoothing and Good-Turing Smoothing) to deal with the data sparsity problem and two normalization techniques (i.e., distribution-based and end-symbol-based) to deal with the unbalanced length distribution problem of passwords. This brings four attacking scenarios in Table 8. In each scenario we consider three types of Markov order (i.e., order-5, 4 and 3) to investigate which order performs best. It is reported that another scenario (i.e., backoff with end-symbol normalization) performs "slightly better" than the above 4 scenarios, yet it is "approximately 11 times slower, both for guess generation and for probability estimation" [36]. We also investigate this scenario and observe similar results. Thus, attackers, who particularly care about the cost-effectiveness [4], are highly unlikely to exploit this scenario.

Particularly, there is a challenge to be addressed when implementing the Good-Turing (GT) smoothing technique. To our knowledge, we for the first time explicate how to combine GT and simple GT in Markov-based

Table 7: Coverage of letter (CoL) segments in corresponding test sets ("PY" stands for Pinyin).

Test set	CoL (PY_name)	CoL (Duowan1M)	CoL(PY_name)∩ CoL(Duowan1M)	CoL(PY_name)− CoL(Duowan1M)	CoL(Duowan1M) −CoL(PY_name)	CoL (Duowan)	CoL(PY_name) ∩CoL(Duowan)	CoL(PY_name) −CoL(Duowan)	CoL(Duowan)− CoL(PY_name)
Tianya	16.63%	67.53%	11.82%	**4.81%**	55.71%	74.34%	13.75%	**2.88%**	60.59%
7k7k	16.70%	71.60%	12.35%	**4.35%**	59.25%	79.84%	14.49%	**2.20%**	65.35%
Dodonew	15.76%	75.79%	11.79%	**3.97%**	63.99%	81.19%	13.47%	**2.29%**	67.72%
178	20.30%	79.15%	15.42%	**4.88%**	63.73%	83.98%	17.49%	**2.81%**	66.49%
CSDN	17.26%	65.64%	11.35%	**5.90%**	54.28%	72.70%	13.43%	**3.83%**	59.27%
Duowan	18.06%	80.05%	14.38%	**3.68%**	65.67%	100.00%	18.06%	**0.00%**	81.94%
Duowan_rest	18.07%	75.03%	13.46%	**4.61%**	61.57%	100.00%	18.07%	**0.00%**	81.93%

attacks (see details in Appendix C). As with PCFG-based attacks, in our implementation we use a max-heap to store the interim results to maintain efficiency. To produce k=10^7 guesses, we employ the strategy of first setting a lower bound (i.e., 10^{-10}) for the probability of guesses generated, then sorting all the guesses, and finally selecting the top k ones. In this way, we can reduce the time overheads by 170% at the cost of about 110% increase in storage overheads, as compared to the strategy of producing exactly k guesses. In Laplace Smoothing, it is required to add δ to the count of each substring and we set δ=0.01 as suggested in [36].

4.2.2 Markov-based experimental results

The experiment results for these five scenarios are quite similar. Here we mainly show the cracking results of Scenario #1 in Fig. 6, while the experiment results for Scenarios #2∼#5 are omitted due to space constraints.

We can see that, for both Chinese and English test sets: (1) At large guesses (i.e., >2*10^6), order-4 markov-chain evidently performs better than the other two orders, while at small guesses (i.e., <10^6) the larger the order, the better the performance will be; (2) There is little difference in performance between Laplace and GT Smoothing at small guesses, while the advantage of Laplace Smoothing gets greater as the guess number increases; (3) End-symbol normalization always performs better than the distribution-based approach, while at small guesses its advantages will be more obvious. Such observations have not been reported in previous major studies [15, 36]. This suggests that: 1) At large guesses, the attacks with order-4, Laplace Smoothing and end-symbol normalization (see Figs. 6(b) and 6(e)) perform best; and 2) At small guesses, the attacks preferring order-5, Laplace Smoothing and end-symbol normalization (see Figs. 6(a) and 6(d)) perform best.

Results show that *the bifacial-security nature found in our PCFG attacks (see Sec. 5.1) also applies in all the Markov attacks*. For example, in order-4 markov-chain-based experiments (see Fig.6(b) and Fig.6(e)), we can see that, when the guess number is below about 7000, Chinese web passwords are generally much *weaker* than their English counterparts. For example, at 1000 guesses, the success rate against Tianya, Dodonew and CSDN is 11.8%, 6.3% and 11.6%, respectively, while their English counterparts (i.e., Rockyou, Yahoo and Phpbb) is merely 8.1%, 4.3% and 7.1%, respectively. However, when the guess number allowed is over 10^4, Chinese web passwords are generally *stronger* than their English counterparts. For example, at 10^6 guesses, the success rate against Tianya, Dodonew and CSDN is 38.2%, 20.4% and 25.4%, respectively, while their English counterparts is 38.6%, 24.8% and 32.3%, respectively.

Table 9: Bifacial-security nature of Chinese passwords.†

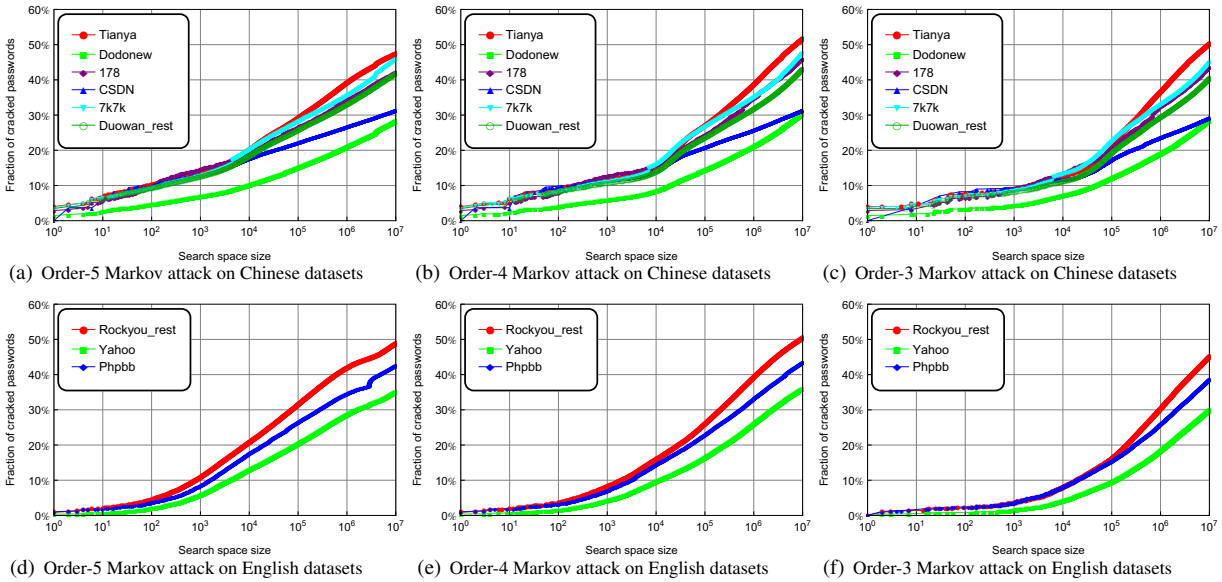
Attacking scenario		Online guessing			Offline guessing			
Algorithm*	Test set	10^1	10^2	10^3	10^4	10^5	10^6	10^7
PCFG	Dodonew	**0.027**	**0.044**	**0.068**	0.103	0.150	0.225	0.288
	Yahoo	0.008	0.022	0.063	**0.136**	**0.212**	**0.316**	**0.390**
	Tianya	**0.073**	**0.105**	**0.138**	0.213	0.295	0.355	0.376
	Rockyou_rest	0.020	0.044	0.110	**0.214**	**0.320**	**0.438**	**0.497**
	CSDN	**0.070**	**0.105**	**0.136**	0.189	0.229	0.272	0.300
	Phpbb	0.021	0.038	0.087	**0.183**	**0.274**	**0.369**	**0.415**
Markov	Dodonew	**0.024**	**0.040**	**0.060**	0.085	0.145	0.212	0.305
	Yahoo	0.007	0.016	0.043	**0.097**	**0.165**	**0.261**	**0.361**
	Tianya	**0.062**	**0.087**	**0.118**	0.154	0.269	0.386	0.516
	Rockyou_rest	0.018	0.035	0.081	**0.159**	**0.259**	**0.392**	**0.503**
	CSDN	**0.037**	**0.098**	**0.116**	0.144	0.211	0.260	0.316
	Phpbb	0.019	0.034	0.071	**0.146**	**0.230**	**0.333**	**0.436**

†A value in bold **green** (e.g., the leftmost 0.027) means that: it is a success-rate under a given guess number (resp. 10^1) against a Chinese dataset (resp. Dodonew) and is *greater* than that of its English counterpart (resp. Yahoo). A value in bold **blue** is on the contrary: it is a guessing success-rate against a English dataset and *greater* than that of its Chinese counterpart.

*For both PCFG- and Markov-based attacks, the training set is Duowan_1M for each Chinese test set and Rockyou_1M for English test sets. Here the Markov setups are from Scenario#1 in Table 8. Other Markov scenarios show the same trends.

As summarized in Table 9, for both PCFG and Markov attacks, the cracking success-rates against Chinese passwords are *always higher* than those of English passwords when the guess number is below 10^4, while this trend is reversed when the guess number is above 10^4. Here we mainly use order-4 Markov attacks (see Figs. 6(b) and 6(e)) as an example, and the other Markov setup scenarios all show the same trends.

Summary. Both PCFG- and Markov-based cracking results reveal the bifacial-security nature of Chinese passwords: They are more prone to online guessing as compared to English passwords; But out of the remaining Chinese passwords, they are more secure against offline guessing. This reconciles the conflicting claims made in [7, 26, 34]. Alarmingly high cracking rates (40%∼50%) highlight the urgency of developing defense-in-depth countermeasures (e.g., cracking-resistant honeywords [31] and password-hardening services [33]) to alleviate the situation. We provide a large-scale empirical evidence for the hypothesis raised by the HCI community [17, 46]: users rationally choose stronger passwords for accounts with higher value.



(a) Order-5 Markov attack on Chinese datasets (b) Order-4 Markov attack on Chinese datasets (c) Order-3 Markov attack on Chinese datasets

(d) Order-5 Markov attack on English datasets (e) Order-4 Markov attack on English datasets (f) Order-3 Markov attack on English datasets

Figure 6: Markov-chain-based attacks on different groups of datasets (scenario #1: *Laplace Smoothing* and *End-Symbol Normalization*). Attacks (a)∼(c) use 1 million Duowan passwords as the training set, while attacks (d)∼(f) use 1 million Rockyou passwords as the training set. The reversal principle also holds. The other four scenarios #2∼#5 show similar cracking results.

5 Some implications

We now elaborate on lessons learned and key takeaways.

5.1 For password creation policies

Interestingly, 2.18% of the passwords in CSDN are of length $len \leq 7$, 97.82% are of length 8-20, no password is of length $len \geq 21$. This means that short passwords (i.e., $len \leq 7$) in the other eight sites are 14∼25 times higher than CSDN. This also suggests that CSDN has changed its password policy at least once before the data breach (i.e., Dec. 2011), but whether the strict policy (i.e., $8 \leq len \leq 20$) is enforced before or later than the weaker policy (i.e., no length requirement) is unknown.[3] Still, what's certain is that most CSDN passwords are generated under the strict policy $8 \leq len \leq 20$. In contrast, no apparent policy can be inferred from the Dodonew data, i.e., neither minimum length (see Fig. 2) nor charset requirement (see Table 3 and Table 2 of [53]).[4] However, Figs. 4 and 6 indicate that, given any guess number below 10^7, passwords from CSDN are significantly weaker than passwords from Dodonew. A plausible reason is that Dodonew provides e-commerce services and users perceive it as more important. As a result, users "rationally" [17, 46] choose more complex passwords for it. As for CSDN, since it is only a technology forum, users knowingly choose weaker passwords for it.

[3] We note that CSDN enforced the policy $6 \leq len \leq 20$ (and no charset requirement) at Jan. 2015 [55], and currently it requires passwords to be $11 \leq len \leq 20$ and consist at least a letter and a digit.

[4] This situation even held at Aug. 2017 (and April 2019): the length-7 letter string "dodonew" is allowed as the default password, see `https://www.5636.com/netbar/money/15886.html`.

In 2012, Bonneau [7] cast doubt on the hypothesis that users rationally select more secure passwords to protect their more important accounts. In 2013, Egelman et al. [17] initiated a field study involving 51 students and confirmed this hypothesis. In 2018, Stobert and Biddle [46] surveyed three groups of English speaking users (i.e., 27 non-experts, 15 experts and 345 MTurk participants), and their results also corroborated this hypothesis. Fortunately, our work provides a large-scale empirical evidence (i.e., on the basis of 6.43M CSDN passwords and 16.26M Dodonew passwords) that confirms this hypothesis.

We also note that though the overall security of Dodonew passwords is higher than that of passwords from the five other Chinese sites, many seemingly complex yet popular passwords (e.g., `5201314`, `321654a` and `love4ever`) dwelling in Dodonew also appear in other less sensitive sites. This can be understood: 1) "Users never see other users' passwords" [47] (and are unaware of how similar their passwords are with other users, and thus they may inadvertently choose popular passwords; 2) Users tend to reuse the same password across multiple sites [27, 42, 56]. What's more, users generally "show a lack of situation awareness" [46] and fail to recognize different categories of accounts [41], and most of them reuse (84% [27]) or simply modify a password from an important site for a non-important site.

Further considering the great password burden already placed on users [8] and the "bounded rationality" [27] and "finite-effort" [20] of users, we outline the need for

HCI research to explore nudges that appropriately frame the importance of accounts and study their impacts on password creation. When designing password creation policies, instead of merely insisting on stringent rules, administrators can employ such nudges to help users gain *more accurate perceptions of the importance* of the accounts to be protected and improve their *ability to recognize different categories* of accounts. Both would help enhance user internal impetus and facilitate users to responsibly allocate passwords (i.e., selecting one candidate from their limited pool of passwords memorized [41, 46]).

In addition, the finding of "bifacial-security nature" suggests that Chinese passwords are more vulnerable to online guessing attacks. This is because top popular Chinese passwords are more concentrated (see Table 3). Thus, a special blacklist that includes a moderate number of most common Chinese passwords (e.g., 10K∼20K as suggested in [61]) would be very helpful for Chinese sites to resist against online guessing. Such a blacklist can be learned from various leaked Chinese datasets (see a concrete list at `http://t.cn/RG88tvF` as built according to [56]). Any password falling into this list would be deemed weak. However, it is well known that if some popular passwords (e.g., `woaini1314`) are banned, new popular ones (e.g., `w0aini1314`) will arise. These new popular passwords may be out of static blacklists and subtle to detect. Hence, password creation policies alone (e.g., length and blacklist rules [25, 55]) are inadequate for preventing such weak passwords. An in-depth defense approach is needed: whenever possible, in addition to password creation policies, password strength meters (e.g., fuzzyPSM [54] and Zxcvbn [59]) can be further employed by security-critical services to detect and prevent weak passwords.

5.2 For password strength meters

Leading password strength meters (PSMs) employ the guess number needed for a password-cracking algorithm (e.g. PCFG) to break that password as an indicator of password strength [24]. In Sec. 1, we have exemplified that the PSMs of four popular services are highly inconsistent in assessing the security of (weak) Chinese passwords. Failing to provide accurate/coherent feedback on user password choices would have negative effects such as user confusion, frustration and distrust [48, 60]. Thus, Carnavalet and Mannan [12] suggested that PSMs "can simplify challenges by limiting their primary goal only to detect *weak* passwords, instead of trying to distinguish a good, very good, or great password."

It follows that an essential step of a PSM would be to identify the characteristics of weak passwords. From our findings in Section 3.3 and Section 4.1, it is evident that for passwords of Chinese users, the incorporation of long Pinyin words or full/family names is an important evidence/weight for a "weak" decision. Other signs of weak Chinese passwords are the incorporation of birthdates and simple patterns like repetition, palindrome and keyboard. As a caveat, even if signs of weak passwords are found, one cannot simply deem such passwords as weak and reject them as is done in many high-profile sites (e.g., Microsoft Azure [18]) and by the "substring blacklist" approach recommended in [44]. Instead, such undesirable/insecure signs should be weighted (see some promising attempts in [54, 59]).

The superiority of our improved PCFG-based attacks over Li et al.'s [34] (see Sec. 4.1) is partly attributed to the proper selection of Duowan (instead of Dodonew as in [34]) as the training set. This indicates that, for a PSM to be accurate, its training set should be representative of the password base of the target site. The distance of letter distributions (see Table 2) would be an effective metric. In addition, the universal "bifacial-security nature" revealed in Sec. 4 implies that, the language factor is more impactful than service type. We also find that CSDN passwords are weaker than Dodonew passwords (see Figs. 4 and 6), but CSDN imposes a stricter policy than Dodonew, and this suggests that the service-type factor might be more impactful than password policy.

Thus, when measuring the letter distributions is infeasible, these con-founding factors underlying a password distribution can be considered for training-set selection: 1) In the order of language, service, and password policy; and 2) The closer the training set to the target password, the better. This suggests that there is no single training set that can fit all PSMs. Thus, PSMs that are originally designed for English speaking users and also do *not* employ a training set (e.g., NIST entropy [10], RNN-PSM [38] and Zxcvbn [59]) cannot be readily applied to Chinese users. This also explains why such PSMs are generally less accurate than those using a training set (e.g., fuzzyPSM [54]) as observed in [24].

5.3 For password cracking

Password cracking algorithms are not only necessary tools for security administrators to measure password strength, but also they can be used to facilitate information forensics (e.g., for law enforcement agencies to recover encrypted data of criminal suspects). Three main lessons for password cracking can be learned from our above results. Firstly, our findings in Sec. 3.3 show that Chinese passwords have a vastly different letter distribution, structure and semantic patterns as compared to English passwords, and thus when targeting a Chinese password, it is crucial for cracking algorithms to be trained on datasets from Chinese sites. Such sites should also have the same password creation policy and the same (or a similar) service type as the target site.

Secondly, for PCFG-based attacks, when the training set is sufficiently large (e.g., over 1M as ours), besides the D and S-segments, it is better to also directly learn the L-segments of guesses from the training set. This can be well established by the fact that, given the same guess numbers and against the same test sets, our PCFG-based attacks can obtain much higher success rates (see Sec. 4.1) than those of the PCFG-based attacks in [34,58] where external dictionaries are used to instantiate the L-segments. This practice has been recommended by Ma et al. [36], but they did not specify when to apply it. Further, one may include some external semantic dictionaries to instantiate the L and D-segments as we do.

Thirdly, as compared to Markov-based attacks, PCFG-based ones are simpler to implement (31% less computation and 70% less memory cost), and they perform equally well, or even better, when the guess number is small (e.g. 10^3, see Figs. 4 and 6). For large guess numbers, order-4 Markov attacks are the best choices. As far as we know, these observations have not been elucidated in previous major studies [15, 36]. Note that, we have only shown the Markov-based cracking results when the guess number is below 10^7. There is potential that order-3 Markov-based attacks will outperform order-4 and 5 ones at larger guess numbers (e.g., 10^{14}).

6 Conclusion

In this paper, we performed a large-scale empirical analysis of 73.1 million real-world Chinese web passwords. In our empirical analysis, we systematically explored several fundamental password properties (e.g., the distance between passwords and languages, and various semantic patterns) and uncovered the bifacial-security nature of Chinese passwords: They are more prone to online guessing than English passwords; But out of the remaining Chinese passwords, they are stronger against offline guessing. This reconciles two conflicting claims in [7, 26, 34]. We hope this work will help both security administrators and individual Chinese users to more informedly secure their password accounts.

Acknowledgment

The authors are grateful to Mary Ellen Zurko for shepherding our paper. We thank Haibo Cheng, Qianchen Gu, and anonymous referees for invaluable help and comments. Ping Wang is the corresponding author. This research was supported by the National Natural Science Foundation of China under Grants No. 61802006 and No. 61572379, and by the National Key Research and Development Plan under Grant No.2017YFB1200700.

References

[1] *China now has 802 million internet users*, July 2018, http://n0.sinaimg.cn/tech/c0a99b19/20180820/CNNIC42.pdf.

[2] M. AlSabah, G. Oligeri, and R. Riley, "Your culture is in your password: An analysis of a demographically-diverse password dataset," *Comput. Secur.*, vol. 77, pp. 427–441, 2018.

[3] J. Blocki, A. Datta, and J. Bonneau, "Differentially private password frequency lists," in *Proc. NDSS 2016*, pp. 1–15.

[4] J. Blocki, B. Harsha, and S. Zhou, "On the economics of offline password cracking," in *Proc. IEEE S&P 2018*, pp. 35–53.

[5] J. Bonneau, "Guessing human-chosen secrets," Ph.D. dissertation, University of Cambridge, 2012.

[6] J. Bonneau, C. Herley, P. Oorschot, and F. Stajano, "The quest to replace passwords: A framework for comparative evaluation of web authentication schemes," in *Proc. IEEE S&P 2012*, pp. 553–567.

[7] J. Bonneau, "The science of guessing: Analyzing an anonymized corpus of 70 million passwords," in *Proc. IEEE S&P 2012*, pp. 538–552.

[8] J. Bonneau, C. Herley, P. van Oorschot, and F. Stajano, "Passwords and the evolution of imperfect authentication," *Comm. ACM*, vol. 58, no. 7, pp. 78–87, 2015.

[9] A. S. Brown, E. Bracken, and S. Zoccoli, "Generating and remembering passwords," *Applied Cogn. Psych.*, vol. 18, no. 6, pp. 641–651, 2004.

[10] W. Burr, D. Dodson, R. Perlner, S. Gupta, and E. Nabbus, "NIST SP800-63-2: Electronic authentication guideline," National Institute of Standards and Technology, Reston, VA, Tech. Rep., 2013.

[11] R. A. Butler, *List of the Most Common Names in the U.S.*, Jan. 2018, http://names.mongabay.com/most_common_surnames.htm.

[12] X. Carnavalet and M. Mannan, "From very weak to very strong: Analyzing password-strength meters," in *Proc. NDSS 2014*.

[13] C. Castelluccia, M. Dürmuth, and D. Perito, "Adaptive password-strength meters from markov models," in *Proc. NDSS 2012*.

[14] A. Das, J. Bonneau, M. Caesar, N. Borisov, and X. Wang, "The tangled web of password reuse," in *Proc. NDSS 2014*, pp. 1–15.

[15] M. Dell'Amico and M. Filippone, "Monte carlo strength evaluation: Fast and reliable password checking," in *Proc. ACM CCS 2015*, pp. 158–169.

[16] M. Dürmuth, D. Freeman, and B. Biggio, "Who are you? A statistical approach to measuring user authenticity," in *Proc. NDSS 2016*, pp. 1–15.

[17] S. Egelman, A. Sotirakopoulos, K. Beznosov, and C. Herley, "Does my password go up to eleven?: the impact of password meters on password selection," in *Proc. ACM CHI 2013*, pp. 2379–2388.

[18] *Eliminate bad passwords in your organization*, July 2018, https://docs.microsoft.com/bs-latn-ba/azure/active-directory/authentication/concept-password-ban-bad.

[19] D. Florêncio and C. Herley, "A large-scale study of web password habits," in *Proc. WWW 2007*, pp. 657–666.

[20] D. Florêncio, C. Herley, and P. C. Van Oorschot, "Password portfolios and the finite-effort user: Sustainably managing large numbers of accounts," in *Proc. USENIX SEC 2014*, pp. 575–590.

[21] S. Furnell and R. Esmael, "Evaluating the effect of guidance and feedback upon password compliance," *Comput. Fraud Secur.*, vol. 2017, no. 1, pp. 5–10, 2017.

[22] W. Gale and G. Sampson, "Good-turing smoothing without tears," *J. Quanti. Linguistics*, vol. 2, no. 3, pp. 217–237, 1995.

[23] J. Goldman, *Chinese Hackers Publish 20 Million Hotel Reservations*, Dec. 2013, http://www.esecurityplanet.com/hackers/chinese-hackers-publish-20-million-hotel-reservations.html.

[24] M. Golla and M. Dürmuth, "On the accuracy of password strength meters," in *Proc. ACM CCS 2018*, pp. 1567–1582.

[25] P. A. Grassi, E. M. Newton, R. A. Perlner, and et al., "NIST 800-63B digital identity guidelines: Authentication and lifecycle management," McLean, VA, Tech. Rep., June 2017.

[26] W. Han, Z. Li, L. Yuan, and W. Xu, "Regional patterns and vulnerability analysis of chinese web passwords," *IEEE Trans. Inform. Foren. Secur.*, vol. 11, no. 2, pp. 258–272, 2016.

[27] A. Hanamsagar, S. S. Woo, C. Kanich, and J. Mirkovic, "Leveraging semantic transformation to investigate password habits and their causes," in *Proc. ACM CHI 2018*, pp. 1–10.

[28] J. Huang, H. Jin, F. Wang, and B. Chen, "Research on keyboard layout for chinese pinyin ime," *J. Chin. Inf. Process.*, vol. 24, no. 6, pp. 108–113, 2010.

[29] M. Jakobsson and M. Dhiman, "The benefits of understanding passwords," in *Proc. HotSec 2012*, pp. 1–6.

[30] S. Ji, S. Yang, X. Hu, and et al., "Zero-sum password cracking game," *IEEE Trans. Depend. Secur. Comput.*, vol. 14, no. 5, pp. 550–564, 2017.

[31] A. Juels and R. L. Rivest, "Honeywords: Making password-cracking detectable," in *Proc. ACM CCS 2013*, pp. 145–160.

[32] D. V. Klein, "Foiling the cracker: A survey of, and improvements to, password security," in *Proc. of USENIX SEC 1990*, pp. 5–14.

[33] R. W. Lai, C. Egger, M. Reinert, S. S. Chow, M. Maffei, and D. Schröder, "Simple password-hardened encryption services," in *Proc. Usenix SEC 2018*, pp. 1405–1421.

[34] Z. Li, W. Han, and W. Xu, "A large-scale empirical analysis on chinese web passwords," in *Proc. USENIX SEC 2014*.

[35] B. Lu, X. Zhang, Z. Ling, Y. Zhang, and Z. Lin, "A measurement study of authentication rate-limiting mechanisms of modern web-sites," in *Proc. ACSAC 2018*, pp. 89–100.

[36] J. Ma, W. Yang, M. Luo, and N. Li, "A study of probabilistic password models," in *IEEE S&P 2014*, 2014, pp. 689–704.

[37] M. L. Mazurek, S. Komanduri, T. Vidas, L. F. Cranor, P. G. Kelley, R. Shay, and B. Ur, "Measuring password guessability for an entire university," in *Proc. ACM CCS 2013*, pp. 173–186.

[38] W. Melicher, B. Ur, S. Segreti, and et al., "Fast, lean and accurate: Modeling password guessability using neural networks," in *Proc. USENIX SEC 2016*, pp. 1–17.

[39] R. Morris and K. Thompson, "Password security: A case history," *Comm. ACM*, vol. 22, no. 11, pp. 594–597, 1979.

[40] A. Narayanan and V. Shmatikov, "Fast dictionary attacks on passwords using time-space tradeoff," in *Proc. ACM CCS 2005*, pp. 364–372.

[41] R. Nithyanand and R. Johnson, "The password allocation problem: strategies for reusing passwords effectively," in *Proc. ACM WPES 2013*, pp. 255–260.

[42] S. Pearman, J. Thomas, P. E. Naeini, and et al., "Let's go in for a closer look: Observing passwords in their natural habitat," in *Proc. ACM CCS 2017*, pp. 295–310.

[43] B. L. Riddle, M. S. Miron, and J. A. Semo, "Passwords in use in a university timesharing environment," *Comput. Secur.*, vol. 8, no. 7, pp. 569–579, 1989.

[44] R. Shay, S. Komanduri, A. L. Durity, and et al., "Designing password policies for strength and usability," *ACM Trans. Inform. Syst. Secur.*, vol. 18, no. 4, pp. 1–34, 2016.

[45] *Sogou Internet thesaurus*, Sogou Labs, April 17 2018, http://www.sogou.com/labs/dl/w.html.

[46] E. Stobert and R. Biddle, "The password life cycle," *ACM Trans. Priv. Secur.*, vol. 21, no. 3, pp. 1–32, 2018.

[47] B. Ur, J. Bees, S. M. Segreti, L. Bauer, N. Christin, L. F. Cranor, and A. Deepak, "Do users' perceptions of password security match reality?" in *Proc. ACM CHI 2016*, pp. 1–10.

[48] B. Ur, P. G. Kelley, S. Komanduri, and et al., "How does your password measure up? the effect of strength meters on password creation," in *Proc. USENIX SEC 2012*, pp. 65–80.

[49] L. Vaas, https://nakedsecurity.sophos.com/2016/08/16/people-like-using-passwords-way-more-than-biometrics/.

[50] A. Vance, *If Your Password Is 123456, Just Make It HackMe*, Jan. 2010, https://www.nytimes.com/2010/01/21/technology/21password.html.

[51] R. Veras, J. Thorpe, and C. Collins, "Visualizing semantics in passwords: The role of dates," in *Proc. ACM VizSec 2012*, pp. 88–95.

[52] C. Wang, S. T. Jan, H. Hu, D. Bossart, and G. Wang, "The next domino to fall: Empirical analysis of user passwords across online services," in *Proc. CODASPY 2018*, pp. 196–203.

[53] D. Wang, H. Cheng, P. Wang, X. Huang, and G. Jian, "Zipf's law in passwords," *IEEE Trans. Inform. Foren. Secur.*, vol. 12, no. 11, pp. 2776–2791, 2017.

[54] D. Wang, D. He, H. Cheng, and P. Wang, "fuzzyPSM: A new password strength meter using fuzzy probabilistic context-free grammars," in *Proc. IEEE/IFIP DSN 2016*, pp. 595–606.

[55] D. Wang and P. Wang, "The emperor's new password creation policies," in *Proc. ESORICS 2015*, pp. 456–477.

[56] D. Wang, Z. Zhang, P. Wang, J. Yan, and X. Huang, "Targeted online password guessing: An underestimated threat," in *Proc. ACM CCS 2016*, pp. 1242–1254.

[57] M. Weir, S. Aggarwal, M. Collins, and H. Stern, "Testing metrics for password creation policies by attacking large sets of revealed passwords," in *Proc. ACM CCS 2010*, pp. 162–175.

[58] M. Weir, S. Aggarwal, B. de Medeiros, and B. Glodek, "Password cracking using probabilistic context-free grammars," in *Proc. IEEE S&P 2009*, pp. 391–405.

[59] D. Wheeler, "zxcvbn: Low-budget password strength estimation," in *Proc. USENIX SEC 2016*, pp. 157–173.

[60] *Why is Gbt3fC79ZmMEFUFJ a weak password?*, Jan. 2019, https://security.stackexchange.com/questions/201210/why-is-gbt3fc79zmmefufj-a-weak-password.

[61] R. Williams, *The UX of a blacklist*, Mar. 2018, https://news.ycombinator.com/item?id=16434266.

[62] J. Yan, A. F. Blackwell, R. J. Anderson, and A. Grant, "Password memorability and security: Empirical results." *IEEE Secur. Priv.*, vol. 2, no. 5, pp. 25–31, 2004.

[63] C. Zuo, W. Wang, R. Wang, and Z. Lin, "Automatic forgery of cryptographically consistent messages to identify security vulnerabilities in mobile services," in *Proc. NDSS 2016*.

APPENDIX

A Justification for our cleaning approach

Contaminated datasets. Interestingly, we observe that there is a non-negligible overlap between the Tianya dataset and 7k7k dataset. We were first puzzled by the fact that the password "`111222tianya`" was originally in the top-10 most popular list of both datasets. We manually scrutinized the original datasets (i.e., before removing the email addresses and user names) and are

surprised to find that there are around 3.91 million (actually 3.91*2 million due to a split representation of 7k7k accounts, as we will discuss later) joint accounts in both datasets. We posit that someone probably has copied these joint accounts from one dataset to the other.

Our cleaning approach. Now, a natural question arises: *From which dataset have these joint accounts been copied?* We conclude that these joint accounts were copied from Tianya to 7k7k, *mainly for two reasons*. Firstly, it is unreasonable for 0.34% users in 7k7k to insert the string "`tianya`" into their 7k7k passwords, while users from `tianya.cn` naturally include the site name "`tianya`" into their passwords for convenience. The following second reason is quite subtle yet convincing. In the original Tianya dataset, the joint accounts are of the form {user name, email address, password}, while in the original 7k7k dataset such joint accounts are divided into two parts: {user name, password} and {email address, password}. The password "`111222tianya`" occurs 64822 times in 7k7k and 48871 times in Tianya, and one gets that $64822/2 < 48871$. Thus, it is more plausible for users to copy *some* (i.e., $64822/2$ of a total of 48871) accounts using "`111222tianya`" as the password from Tianya to 7k7k, rather than to first copy all the accounts (i.e., $64822/2$) using "`111222tianya`" as the password from 7k7k to Tianya and then reproduces $16460 (= 48871 - 64822/2)$ such accounts.

After removing 7.82 million joint accounts from 7k7k, we found that all of the passwords in the remaining 7k7k dataset occur even times (e.g., 2, 4 and 6). This is expected, for we observe that in 7k7k half of the accounts are of the form {user name, password}, while the rest are of the form {email address, password}. It is likely that both forms are directly derived from the form {user name, email address, password}. For instance, both {wanglei, wanglei123} and {wanglei@gmail.com, wanglei123} are actually derived from the single account {wanglei, wanglei@gmail.com, wanglei123}. Consequently, we further divide 7k7k into two equal parts and discard one part. The detailed information on data cleaning is summarized in Table 1.

Previous studies. In 2014, Li et al. [34] has also exploited the datasets Tianya and 7k7k. However, contrary to us, they think that the 3.91M joint accounts are copied from 7k7k to Tianya. Their main reason is that, when dividing these two datasets into the reused passwords group (i.e., the joint accounts) and the not-reused passwords group, they find that "the proportions of various compositions are similar between the reused passwords and the 7k7k's not-reused passwords, but different from Tianya's not-reused passwords". However, they did not explain what the "various compositions" are. Their explanation also does not answer the critical question: why are there so many 7k7k users using "`111222tianya`" as their passwords? We posit they had removed 3.91*2 million joint accounts from 7k7k but the not 3.91 million ones from Tianya In addition, they did not observe the extremely *abnormal* fact that all the passwords in 7k7k occur even times. Such contaminated data would lead to inaccurate results. For example, Li et al. [34] reported that there are 32.41% of passwords in 7k7k containing dates in "YYYMMDD", yet the actual value is 6 times lower: 5.42%.

We have reported this issue to the authors of [34], they responded to us and acknowledged this flaw in their journal version [26]. Unfortunately, Han et al. [26] do not clean the datasets in the journal version in the manner that we outlined.

B Detailed information about our 22 semantic dictionaries

In order to make our work as reproducible as possible and to facilitate the community, we now detail how to construct our 22 semantic-based dictionaries. All dictionaries are built with natural lengths. The length$\geq$5 requirement in the upper-part of Table 5 is set *conservatively* for ensuring accuracy only when we perform matching. Actually, we also performed measurements for length$\geq$3 and length$\geq$4, and got higher figures (percentages) but less accuracy. Thus, we omit them.

The first dictionary "English_word_lower" is from `http://www.mieliestronk.com/wordlist.html` and it contains about 58,000 popular lower-case English words. "English_lastname" is a dictionary consisting of 18,839 last names with over 0.001% frequency in the US population during the 1990 census, according to the US Census Bureau [11]. "English_firstname" contains 5,494 most common first names (including 1,219 male and 4,275 female names) in US [11]. The dictionary "English_fullname" is a cartesian product of "English_firstname" and "English_lastname", consisting of 1.04 million most common English full names.

To get a Chinese full name dictionary, we employ the 20 million hotel reservations dataset [23] leaked in Dec. 2013. The Chinese family name dictionary includes 504 family names which are officially recognized in China. Since the first names of Chinese users are widely distributed and can be almost any combinations of Chinese words, we do not consider them in this work. As the names are originally in Chinese, we transfer them into Pinyin without tones by using a Python procedure from `https://pypinyin.readthedocs.org/en/latest/` and remove the duplicates. We call these two dictionaries "Pinyin_fullname" and "Pinyin_familyname", respectively.

"Pinyin_word_lower" is a Chinese word dictionary known as "SogouLabDic.dic", and "Pinyin_place" is a Chinese place dictionary. Both of them are from [45]

and also originally in Chinese. We translate them into Pinyin in the same way as we tackle the name dictionaries. "Mobile_number" consists of all potential Chinese mobile numbers, which are 11-digit strings with the first seven digits conforming to pre-defined values and the last four digits being random. Since it is almost impossible to build such a dictionary on ourselves, we instead write a Python script and automatically test each 11-digit string against the mobile-number search engine `https://shouji.supfree.net/`.

As for the birthday dictionaries, we use date patterns to match digit strings that might be birthdays. For example, "YYYYMMDD" stands for a birthday pattern that the first four digits indicate years (from 1900 to 2014), the middle two represent months (from 01 to 12) and the last two denote dates (from 01 to 31). Similarly, we build the date dictionaries "YYYY", "MMDD" and "YYMMDD". Note that, "PW with a l^+-letter substring" means a subset of the corresponding dataset and consists of all passwords that include a letter substring *no shorter than* l, and similarly for "PW with a l^+-digit substring".

Though we use the "left-most longest" rule to minimize ambiguities when matching, there are some unavoidable ambiguities when determining whether a text/digit sequence belongs to a semantic dictionary. An improper resolution would lead to an overestimation or underestimation. For instance, `111111` falls into "YYMMDD" and is highly popular, yet it is more likely that users choose it simply because it is easily memorable repetition numbers. To tackle this issue, we manually identify 17 abnormal dates in "YYMMDD", each of which originally has a frequency$> 10E$ and appears in every top-1000 list of the six Chinese datasets: `111111`, `520131`, `111222`, `121212`, `520520`, `110110`, `231231`, `101010`, `110119`, `321123`, `010203`, `110120`, `010101`, `520530`, `000111`, `000123`, `080808`. Similarly, we identify 16 abnormal items in "MMDD": `1111`, `1122`, `1231`, `1212`, `1112`, `1222`, `1010`, `0101`, `1223`, `1123`, `0123`, `1020`, `1230`, `0102`, `0520`, `1110`. Few abnormal items can be identified in the other 19 dictionaries (Table 5), and they are processed as usual.

C A subtlety about Good-Turing smoothing in Markov-based cracking

In 2014, Ma et al. [36] introduced the Good-Turing (GT) smoothing into password cracking, yet little attention has been paid to the unsoundness of GT for popular password segments. We illustrate the following subtlety.

We denote f to be the frequency of an event and N_f to be the frequency of frequency f. According to the basic GT smoothing formula, the probability of a string "$c_1c_2\cdots c_l$" in a Markov model of order n is denoted by

$$P(\text{“}c_1\cdots c_{l-1}c_l\text{”}) = \prod_{i=1}^{l} P(\text{“}c_i|c_{i-n}c_{i-(n-1)}\cdots c_{i-1}\text{”}), \quad (1)$$

where the individual probabilities in the product are computed empirically by using the training sets. More specifically, each empirical probability is given by

$$P(\text{“}c_i|c_{i-n}\cdots c_{i-1}\text{”}) = \frac{S(count(c_{i-n}\cdots c_{i-1}c_i))}{\sum_{c\in\Sigma} S(count(c_{i-n}\cdots c_{i-1}c))}, \quad (2)$$

where the alphabet Σ includes 95 printable ASCII characters on the keyboard (plus one special end-symbol c_E denoting the end of a password), and $S(\cdot)$ is defined as:

$$S(f) = (f+1)\frac{N_{f+1}}{N_f}. \quad (3)$$

This kind of smoothing works well when f is small, but it fails for passwords with a high frequency because the estimates for $S(f)$ are not smooth. For instance, `12345` is the most common 5-character string in Rockyou and occurs $f = 490,044$ times. Since there is no 5-character string that occurs 490,045 times, N_{490045} will be zero, implying the basic GT estimator will set $P(\text{“}12345\text{”})$=0. A similar problem regarding the smoothing of password frequencies is identified in [5].

There have been various improvements suggested in linguistics to tackle this problem, among which is the "simple Good-Turing smoothing" [22]. This improvement (denoted by SGT) is famous for its simplicity and accuracy. SGT takes two steps of smoothing. Firstly, SGT performs a smoothing operation for N_f:

$$SN(f) = \begin{cases} N(1) & \text{if } f = 1 \\ \dfrac{2N(f)}{f^+ - f^-} & \text{if } 1 < f < \max(f) \\ \dfrac{2N(f)}{f - f^-} & \text{if } f = \max(f) \end{cases} \quad (4)$$

where f^+ and f^- stand for the next-largest and next-smallest values of f for which $N_f > 0$. Then, SGT performs a linear regression for all values SN_f and obtains a Zipf distribution: $Z(f) = C\cdot(f)^s$, where C and s are constants resulting from regression. Finally, SGT conducts a second smoothing by replacing the raw count N_f from Eq.3 with $Z(f)$:

$$S(f) = \begin{cases} (f+1)\dfrac{N_{f+1}}{N_f} & \text{if } 0 \le f < f_0 \\ (f+1)\dfrac{Z(f+1)}{Z(f)} & \text{if } f_0 \le f \end{cases} \quad (5)$$

where $t(f) = |(f+1)\cdot\frac{N_{f+1}}{N_f} - (f+1)\cdot\frac{Z(f+1)}{Z(f)}|$ and $f_0 = \min\left\{ f\in\mathbb{Z} \middle| N_f > 0, t(f) > 1.65\sqrt{(f+1)^2\frac{N_{f+1}}{N_f^2}(1+\frac{N_{f+1}}{N_f})} \right\}$.

To the best of our knowledge, we for the first time well explicate how to combine the two smoothing techniques (i.e., GT and SGT) in Markov-based password cracking.

Protecting accounts from credential stuffing with password breach alerting

Kurt Thomas[*] *Jennifer Pullman*[*] *Kevin Yeo*[*] *Ananth Raghunathan*[*]
Patrick Gage Kelley[*] *Luca Invernizzi*[*] *Borbala Benko*[*] *Tadek Pietraszek*[*]
Sarvar Patel[*] *Dan Boneh*[◇] *Elie Bursztein*[*]

Google[*] *Stanford*[◇]

Abstract

Protecting accounts from credential stuffing attacks remains burdensome due to an asymmetry of knowledge: attackers have wide-scale access to billions of stolen usernames and passwords, while users and identity providers remain in the dark as to which accounts require remediation. In this paper, we propose a privacy-preserving protocol whereby a client can query a centralized breach repository to determine whether a specific username and password combination is publicly exposed, but without revealing the information queried. Here, a client can be an end user, a password manager, or an identity provider. To demonstrate the feasibility of our protocol, we implement a cloud service that mediates access to over 4 billion credentials found in breaches and a Chrome extension serving as an initial client. Based on anonymous telemetry from nearly 670,000 users and 21 million logins, we find that 1.5% of logins on the web involve breached credentials. By alerting users to this breach status, 26% of our warnings result in users migrating to a new password, at least as strong as the original. Our study illustrates how secure, democratized access to password breach alerting can help mitigate one dimension of account hijacking.

1 Introduction

The wide-spread availability of usernames and passwords exposed by data breaches has trivialized criminal access to billions of accounts. In the last two years alone, breach compilations like Antipublic (450 million credentials), Exploit.in (600 million credentials), and Collection 1-5 (2.2 billion credentials) have steadily grown as their creators aggregated material shared on underground forums [21, 25]. Despite the public nature of this data, it remains no less potent. Previous studies have shown that 6.9% of breached credentials remain valid due to reuse, even multiple years after their initial exposure [51]. Absent defense in depth techniques that expand authentication to include a user's location and device details [12, 17], hijackers need only conduct a credential stuffing attack—attempting to log in with every breached credential—to isolate vulnerable accounts.

While users (or identity providers) can mitigate this hijacking risk by resetting an account's password, in practice, discovering which accounts require attention remains a critical barrier. This has given rise to breach alerting services like HaveIBeenPwned and PasswordPing that actively source breached credentials to notify affected users [26, 43]. At present, these services make a variety of tradeoffs spanning user privacy, accuracy, and the risks involved with sharing ostensibly private account details through unauthenticated public channels. One consequence of these tradeoffs is that users may receive inaccurate remediation advice due to false positives. For example, both Firefox and LastPass check the breach status of usernames to encourage password resetting [13, 42], but they lack context for whether the user's password was actually exposed for a specific site or whether it was previously reset. Equally problematic, other schemes implicitly trust breach alerting services to properly handle plaintext usernames and passwords provided as part of a lookup. This makes breach alerting services a liability in the event they become compromised (or turn out to be adversarial).

In this paper, we present the design, implementation, and deployment of a new privacy-preserving protocol that allows a client to learn whether their username and password appears in a breach without revealing the information queried. Our protocol offers two main advantages compared to existing schemes. First, our design takes into account the threat of both an adversarial client (e.g., an attacker attempting to steal usernames and passwords from our service) and an adversarial server (e.g., an attacker harvesting usernames and passwords sent to the service). We address these risks using a combination of computationally expensive hashing, k-anonymity, and private set intersection. Second, these privacy requirements allow us to check a client's exact username and password against a database of breached credentials (versus only usernames, or only passwords currently), thus reducing false positives that lead to warning fatigue.

To demonstrate the feasibility of our protocol, we publicly

released a Chrome extension that warns users when they log in to a website using one of over 4 billion breached usernames and passwords. While in theory any identity provider or password manager can integrate with our protocol, we opted for in-browser alerting first as it scales to the long tail of domains. Nearly 670,000 users from around the world installed our extension over a period of February 5–March 4, 2019. During this measurement window, we detected that 1.5% of over 21 million logins were vulnerable due to relying on a breached credential—or one warning for every two users. By alerting users to this breach status, 26% of our warnings resulted in users migrating to a new password. Of these new passwords, 94% were at least as strong as the original.

Anonymous telemetry reported by our extension reveals that users reused breached credentials on over 746,000 distinct domains. The risk of hijacking was highest for video streaming and adult sites, where 3.6–6.3% of logins relied on breached credentials. Conversely, users appeared to internalize password security advice (or were forced to do so via password composition policies) specifically for financial and government sites, where only 0.2–0.3% of logins involved breached credentials. Despite variations across industries, our analysis reveals that the threat of credential stuffing extends well into the long tail of the Internet. Absent new forms of authentication, we believe that it is critical to democratize access to breach alerting so that both users and identity providers can proactively resecure their accounts.

In summary, we frame our key contributions as follows:

- We develop and publicly release a new protocol for detecting whether a username and password pair appears in a data breach without revealing the information queried. Our protocol improves on the privacy of existing schemes while also reducing the risk of false positives.
- We outline the technical challenges of deploying this scheme in practice, including the computational overhead, latency, and cost required to mediate access to over 4 billion breached usernames and passwords.
- Based on a real-world deployment, we find that 1.5% of logins across the web involve breached credentials. We caution this is a lower bound as logins are not unique. Roughly one in two of our 670,000 users received a warning.
- Users responded to 26% of our warnings by resetting their password; 94% of new passwords were as strong or stronger than the original passwords.

2 Background and requirements

To start, we establish the design principles and threat model that underpin our breach alerting protocol. We compare these requirements against existing solutions from HaveIBeenPwned and PasswordPing—as well as related cryptographic protocols like private information retrieval and oblivious transfer—to highlight the tradeoffs that all of these approaches make in terms of privacy, overhead, accuracy, and trust.

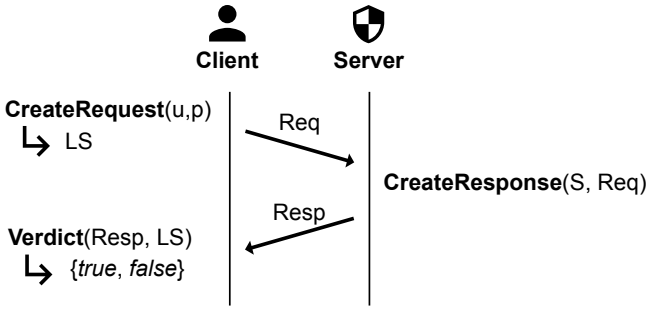


Figure 1: Abstract protocol for a breach alerting service. At a high level, a client generates a request based on some computation over a username and password. The server then returns a response that allows the client to arrive at a verdict for whether their credential is in a breach.

2.1 Abstract protocol

We provide an abstract protocol for our breach alerting service in Figure 1. We reuse these function names and terminology throughout our work. Here, a client with access to a username and password tuple (u,p) executes some computation via $\mathsf{CreateRequest}(\mathsf{u},\mathsf{p})$ that produces a local state LS and request Req that it sends to the breach alerting service. This service stores and regularly updates a database of unsafe credentials $\mathsf{S} = \{(\mathsf{u}_1,\mathsf{p}_1),\ldots,(\mathsf{u}_\mathsf{n},\mathsf{p}_\mathsf{n})\}$. Upon receiving a request, the server accesses its credential store S, runs $\mathsf{CreateResponse}(\mathsf{S},\mathsf{Req})$, and sends the resulting response Resp to the client. Finally, the client arrives at a verdict whether the credential queried was exposed through a breach by calculating $\mathsf{Verdict}(\mathsf{Resp},\mathsf{LS})$. Because new breaches emerge over time, a client should regularly repeat this process as prior verdicts may no longer be valid.

2.2 Design principles

Democratized access: At present, identity providers individually collect breached password data to reset their affected user accounts [4, 58]. This fails to scale to all identity providers, resulting in patchy protection across services and incidents. Any breach alerting service should be accessible to all end users and identity providers, and as such, not require trust between the parties involved. This means we cannot rely on authenticated accounts as a form of rate limiting. We define trust more formally in our threat model in Section 2.3.

Actionable, not informational: Any breach alerting service should provide users with accurate and actionable security ad-

vice such as re-securing an account via a password reset. An alert that warns a client about the mere presence of exposed data such as a client's email address, phone number, or physical address lacks a straightforward recovery step and is thus out of scope for our design. Similarly, an alert merely warning a client that password material was exposed (rather than the specific password involved) may lead to false positives.

Breached, not weak: Alerting should only trigger when all the information necessary to access an account (e.g., a username and password) is exposed. While cracking dictionaries (often composed from breached passwords) may include a client's weak, guessable password, any subsequent attack potentially requires multiple guesses and thus represents a smaller threat than full credential exposure. We assume that most online services employ sufficient throttling to make such bruteforcing impractical. Conversely, attacks against exact username and password pairs are actively deployed in the wild. Indeed, Thomas et al. showed that users with non-stale credentials exposed by third-party breaches were ten times more likely to become hijacked than a random user [51]. Our emphasis on breached credentials helps us prioritize scarce user attention [5] and avoid potential warning fatigue similar to other warning models [2]. While migrating users to stronger passwords in general remains an important task, it is out of scope for our design.

Near real-time: The time that elapses between a client querying a credential and learning its breach status should be near real-time in order to facilitate integration directly with account security flows, password managers, or upon password entry. This potentially constrains the level of privacy protections provided by any protocol due to computational overhead and network latency of any cryptographic primitives involved.

2.3 Threat model

Democratized access hinges on mutual distrust between a client and the server involved in our breach alerting protocol. We develop our threat model with both an adversarial client and adversarial server in mind. In the case of an adversarial client with access to their own breach dataset $\mathsf{D} = \{(\mathsf{u}_1, \mathsf{p}_1), \ldots, (\mathsf{u}_n, \mathsf{p}_n)\}$, the attacker seeks to learn $\mathsf{u} \in \mathsf{S} - \mathsf{D}$ (e.g., a new email to spam), $\mathsf{p} \in \mathsf{S} - \mathsf{D}$ (e.g., a new password to add to a cracking dictionary), or a new credential $(\mathsf{u}, \mathsf{p}) \in \mathsf{S} - \mathsf{D}$. In the case of an adversarial server where a client has access to (u, p), the threat landscape is larger. An adversarial server may learn the client's identity u (even if $\mathsf{u} \in \mathsf{S}$, this enables tracking), a client's password p (even if $\mathsf{p} \in \mathsf{S}$, this identifies active usage), or the credential (u, p) (even if $(\mathsf{u}, \mathsf{p}) \in \mathsf{S}$).

To address these threats, we outline the minimum security and privacy requirements any implementation of the abstract protocol previously outlined in Figure 1 must satisfy. In the security notions discussed below, we work with *anonymity sets* (denoted K) that describe a set of values (in our case, user credentials) that are large enough to give clients plausible deniability about their data even if their membership in K is revealed. These sets must be carefully defined to avoid trivial constructions that are insecure. At a high-level, they must have a sufficiently large support jointly over usernames and passwords (to aid in plausible deniability regarding both), should "partition" the space of credentials in a somewhat uniform manner independent of any actual usernames or passwords, and roughly all values in an anonymity set should be equally likely to be the client's credentials. (A full discussion is deferred to Appendix A.)

Requester credential anonymity: A protocol provides requester credential anonymity if for every credential $(\mathsf{u}_1, \mathsf{p}_1)$, there exists a sufficiently large anonymity set K containing $(\mathsf{u}_1, \mathsf{p}_1)$ such that $\forall (\mathsf{u}_2, \mathsf{p}_2) \in \mathsf{K}$:

$$\mathsf{CreateRequest}(\mathsf{u}_1, \mathsf{p}_1) \approx \mathsf{CreateRequest}(\mathsf{u}_2, \mathsf{p}_2). \quad (1)$$

For two distributions A and B, we let $A \approx B$ denote the computational indistinguishablity of the two distributions—that no efficient adversary given samples from A and B can distinguish them apart much better than randomly guessing. Thus, clients with credentials from the same anonymity set create requests that are indistinguishable to the server. While a minimum $|\mathsf{K}|$ likely depends on the sensitivities of the client involved, we set an initial threshold at $|\mathsf{K}| > 50{,}000$. While the IP address tied to a client's request reduces $|\mathsf{K}|$, a client can rely on a mix network such as Tor to prevent this leakage. IP address anonymity is out of scope of our threat model.

Responses with bounded leakage: Given a request for (u, p), the response from a breach alerting service should bound the information leaked, denoted L, about the membership of other credentials in S. To do this, we require an efficient simulator Sim that given only L can act as the server without being noticed by the client:

$$\mathsf{CreateResponse}(\mathsf{S}, \mathsf{Req}) \approx \mathsf{Sim}(\mathsf{L}, \mathsf{Req}). \quad (2)$$

The presence of a successful simulator shows that the client may learn at most L by looking at responses from the server. Ideally, we want leakage to consist of only the membership of the queried credential and the anonymity set:

$$\mathsf{L} = ([(\mathsf{u}, \mathsf{p}) \in \mathsf{S}], \mathsf{K}). \quad (3)$$

We can rephrase this security notion as follows. For any $(\mathsf{u}_1, \mathsf{p}_1), (\mathsf{u}_2, \mathsf{p}_2) \in \mathsf{K}$ such that their membership in S is identical, i.e., $[(\mathsf{u}_1, \mathsf{p}_1) \in \mathsf{S}] = [(\mathsf{u}_2, \mathsf{p}_2) \in \mathsf{S}]$:

$$\begin{aligned} &\mathsf{CreateResponse}(\mathsf{S}, \mathsf{CreateRequest}(\mathsf{u}_1, \mathsf{p}_1)) \\ &\approx \mathsf{CreateResponse}(\mathsf{S}, \mathsf{CreateRequest}(\mathsf{u}_2, \mathsf{p}_2)). \end{aligned} \quad (4)$$

In other words, our security notion implies that the responses to credentials with identical leakage will be computationally indistinguishable.

Inefficient oracle: Learning u, p, or $(\mathsf{u},\mathsf{p}) \in \mathsf{S}$ via the breach alerting service should be equally or less efficient compared to guessing attempts performed on the login portal where the account originates from. Alternatively, a pragmatic attacker should be better off finding a plaintext copy of the breach. Let $\mathsf{t}(\mathsf{f})$ denote the running time of the function f. We capture this for a remote attacker as there being a time period T such that:

$$\mathsf{t}(\mathsf{CreateRequest}(\mathsf{u_i},\mathsf{p_i})) > T, \tag{5}$$

for every $(\mathsf{u}_i,\mathsf{p}_i)$. This requirement extends to an attacker with direct access to S due to an insider risk, a court order, or a breach of the alerting service's database. We frame this as merely checking the membership of a credential:

$$\mathsf{t}([(\mathsf{u},\mathsf{p}) \in \mathsf{S}]) > T'. \tag{6}$$

Ideally, $T = T'$, such that local access to S provides no advantage compared to the access mediated by the protocol. We consider a protocol where $T > 1$ second to satisfy this requirement.

Resistance to Denial of Service: A response from the server should not require significantly more computation than a request by a client (including bogus requests). As such, it should be difficult for an attacker to find a sequence $(\mathsf{u}_1,\mathsf{p}_1),\ldots,(\mathsf{u}_\mathsf{n},\mathsf{p}_\mathsf{n})$ such that:

$$\sum_i \mathsf{t}(\mathsf{CreateRequest}(\mathsf{u_i},\mathsf{p_i})) \ll \sum_i \mathsf{t}(\mathsf{CreateResponse}(\mathsf{Req_i})) \tag{7}$$

Where $\mathsf{t}(\mathsf{f}(\cdot))$ denotes the running time of the function f.

Non-threats: Some threats are explicitly outside our threat model. These include an attacker attempting to confirm whether a breach they have access to is known to the alerting service (e.g., $\mathsf{D} \subseteq \mathsf{S}$), as well as an attacker learning $|\mathsf{S}|$. Such information may instead be beneficial to have public, allowing the service to publicly communicate which breaches it covers.

2.4 Tradeoffs of existing schemes

Existing breach alerting services include HaveIBeenPwned and PasswordPing, both of which have publicly documented APIs [26, 43]. Clients for each service include the 1Password [48] and LastPass [42] password managers. GitHub relies on a local mirror of HaveIBeenPwned's password dictionary for detection [36]. Firefox uses HaveIBeenPwned to warn users when they browse to a site that previously suffered a data breach, or if users supply their email address to Firefox [13]. We examine the tradeoffs these protocols make in terms of our design principles and threat model, with Table 1 serving as summary.

Query by username: HaveIBeenPwned and PasswordPing both support querying a specific plaintext username u. PasswordPing also supports querying $\mathsf{H}(\mathsf{u})$, the SHA256 hash of a username. In response, both services provide a list of breaches that the specified user was affected by and the class of data exposed (e.g., password, physical address). Lastpass currently relies on the username-only protocol from PasswordPing for breach alerting (after user consent).

Query by	Setup	Actionable, not informational	Breached, not weak	Near real-time	Requester credential anonymity	Inefficient oracle	Bounded leakage response	Resistant to Denial of Service
Username	Plaintext			●			●	●
	Hash			●			●	●
Password	Plaintext	●		●			●	●
	Hash	●		●			●	●
	Hash prefix	●		●	●			●
Domain	Plaintext			●	●	●	●	●
Username, then password	Plaintext, hash	●	●	●		●		●
	Hash, hash	●	●	●		●		●

Table 1: Summary of protocols supported by HaveIBeenPwned and PasswordPing and their tradeoffs according to our design principles and threat model.

In terms of our threat model (see Table 1), $\mathsf{H}(\mathsf{u})$ creates a unique, stable identifier of the user that is possibly reversible via a dictionary attack. This fails our requirement of requester credential anonymity. Likewise, querying u directly leaks the user's identity. Neither $\mathsf{H}(\mathsf{u})$ or u provides a computational hurdle, thus providing an efficient oracle for performing reconnaissance on victims. Knowledge of which breaches a victim is involved in can expose the victim to extortion, similar to recent scams that include breached data to coerce victims into paying the attacker by misrepresenting wider access [31].

Revisiting our design principles, we find that username-only protocols fail to satisfy our requirement of actionable rather than informational breach warnings. Users may have changed their password, or no longer use the account involved. Likewise, isolating responses solely to the types of data exposed fails to alert users to breached passwords that they reuse across multiple sites, where just one of the sites involved might be breached.

Query by password: PasswordPing allows clients to send a plaintext password p, or $\mathsf{H}(\mathsf{p})$ using SHA1, SHA256, or MD5. Both PasswordPing and HaveIBeenPwned provide a more secure alternative, whereby clients supply an N-bit prefix

$H(p)_{[0:N]}$. The server then returns all known breached passwords with that prefix, with the client performing the final exactness check locally. PasswordPing uses a 10-hex character prefix of a SHA1, SHA256, or MD5 hash; HaveIBeenPwned uses a 5-hex character prefix of a SHA1 hash. 1Password currently relies on HaveIBeenPwned and the password-prefix approach for breach alerting.

As detailed in Table 1, while supplying p explicitly exposes a client's non-breached password, revealing even $H(p)$ leads to a potential pre-computed dictionary attack by an adversarial server. This threat is simplified by the lack of salt. As such, both schemes fail to provide requester credential anonymity. In the prefix-based variant, the same attack reduces the search space necessary by 2^N, with the attacker prioritizing guesses based on a password's popularity. With a sufficiently small N, this meets our criteria for anonymity—though weakly. We provide a deeper treatment of our rationale in Appendix A. However, as the response contains multiple passwords per lookup, this does not satisfy our requirement for bounded leakage. An adversarial client can enumerate each bucket to acquire a local copy of all $H(p)$ for offline cracking to rebuild the underlying password dictionary.[1] While there is a legitimate argument that an attacker could more easily acquire a plaintext copy of the data breach, ideally any such protocol should also work for more sensitive breach data that is not widely accessible.

From a design perspective, we find that password-only protocols run the risk of alerting users to merely weak passwords. If u_1 in a breach shares the same password as u_2 who was not in any breach, there is no way to curate the security advice to both users' circumstances.

Query by domain: Both HaveIBeenPwned and PasswordPing provide a protocol for determining whether a domain was part of a breach. Firefox currently uses HaveIBeenPwned to warn users when they visit a domain that's previously suffered a breach [9]. This alert specifies that if they had an account, their data may no longer be secure. While these domain-only protocols satisfy every requirement laid out in our threat model (assuming the list of insecure domains is locally cached rather than queried), they provide neither actionable advice nor specific insights into breached rather than weak passwords. For example, a site visitor may have registered an account after the breach date. Likewise, domain-only protocols cannot capture the risk of password re-use across breached and non-breached sites.

Query by username, then password: PasswordPing provides a protocol whereby a client first queries u or $H(u)$ using SHA-256, in turn receiving a salt s associated with that account. The client uses this to calculate $H(u,p,s)$ via Argon2, sending only the N-bit prefix $H(u,p,s)_{[0:N]}$. PasswordPing relies on a 10-hex character prefix. The server responds with all known matching credentials, allowing a client to perform the confirmation locally. This approach satisfies all of our design principles. Additionally, due to the use of Argon2, the hash complexity involved compared to SHA or MD5 satisfies our requirement of an inefficient oracle. While we can bound the leakage of this protocol, it leaks information about both a requester's identity as well as multiple $H(u,p,s)$ per response enabling offline attacks. (The s prevents pre-computed dictionary attacks.) This protocol bears a close resemblance to ours, but we satisfy all the criteria laid out in Table 1 and show in Section 3.2 how to further protect users' password information when querying by username.

[1] HaveIBeenPwned provides a direct download to every password in its corpus (hashed via SHA1), so this enumeration step is unnecessary and something the service argues is outside their threat model.

2.5 Alternative cryptographic protocols

Our threat model is closely related to several well-studied cryptographic primitives. These protocols offer stricter privacy guarantees, but are computationally burdensome for a network setting in practice. As such, our threat model uses a relaxed requirement of anonymity. Secure hardware enclaves would also enable stricter privacy guarantees, but current enclaves have been shown to be vulnerable to side-channel and speculative execution attacks [53, 54].

Private Information Retrieval (PIR): PIR protocols, introduced by Chor et al. [6], require that a user be able to query an item from a server without revealing which item was queried. While PIR protocols which are secure against computationally-bounded adversaries [32] exceed our requester anonymity and password secrecy requirements, their security guarantees are one-sided—they allow the server to leak arbitrary information about the database to the clients. Additionally, single-server PIR protocols require communication that is effectively comparable to the size of the database [22]. Multi-server PIR protocols reduce this overhead, and even offer security guarantees against adversarial clients [19], but require that users trust that there is no risk of collusion amongst servers.

Oblivious Transfer (OT): 1-out-of-N OT protocols [8, 46] extend the PIR threat model to also require that a client learns no information about unaccessed elements of the server's database during the query. (Here N refers to the number of database entries.) While OT appears to capture the ideal requirements for a breach alerting protocol, we note that without weakening its security requirements, OT turns out to be a powerful crypto primitive [29] and requires communication overhead proportional to N.

Private Set Intersection (PSI): PSI protocols allow two parties with sets S_1 and S_2 respectively to compute some functions each of $S_1 \cap S_2$ and learn nothing more about each other's sets. We can model our use case as PSI where the client has a singleton set and the server learns nothing (an additional requirement needed in our work not typically seen in PSI).

Early works leading to PSI [24, 38] are based off of the Diffie-Hellman assumption which we also leverage in our protocol. While PSI protocols based on OT have been shown to be the fastest in practice, they require significant communication overhead that is unsuitable for a network setting [45]. Additionally, they are designed for settings where both parties have large, balanced sets which does not map to our scenario.

2.6 Ethics

Providing a breach alerting service necessitates access to credentials that were illicitly obtained and then released. For our work, we exclusively rely on credential breaches that are now publicly accessible, which any sophisticated attacker is likely to already have access to. As such, we argue that making this information accessible to users and identity providers does not materially increase the potential for harm—but that any protocol should have measures in place to protect against abuse. Passwords exposed by breaches have a history of research applications including improving password strength meters [11, 39, 57] and studying password use in the wild [10]. Surveyed users have also expressed a positive attitude towards breach alerting services, particularly in the context of password resetting [28]. We believe the potential to reduce account hijacking outweighs any risk of collating already public credential data.

3 Breach alerting protocol

Our design for a data breach alerting protocol relies on a combination of k-anonymity, private set intersection, and computationally expensive hashing to address all the risks outlined in our threat model. Here, we detail the cryptographic primitives we use to implement our protocol and the data exchanged between a client and server. We consider two variants: one that leaks some bits of password material that is secure against a resource-constrained attacker (e.g., the attacker is unable to circumvent k-anonymity and expensive hashing); and one that leaks zero bits of password material, but where clients must spend twice as much time hashing and receive weaker bounds on requester anonymity.

3.1 Resource-constrained attacker variant

CreateDatabase: Prior to any client lookup, the server must construct a secure database containing all known breached credentials. We outline this process in Algorithm 1. The server first canonicalizes the username associated with a credential by removing any capitalization and stripping information related to email providers (e.g., `user@gmail.com` becomes `user`). This step aids in de-duplication while also enabling us to detect reuse across sites that exclusively use usernames rather than email addresses. Post-canonicalization, the server calculates a computationally expensive hash of both the canonical username and credential password. We rely on Argon2 with a configuration that uses a single thread, 256 MB of memory, and a time cost of three.[2]

The server then blinds the 16-byte hash output with a 224-bit secret key b by mapping the hash to the elliptic curve NID_secp224r1 and raising the resulting point to the power b.[3] The server saves only a 2-byte prefix of hash unblinded which it uses for partitioning the entire database, where we denote a partition as S'. Here, hashing satisfies our requirement for an inefficient oracle even in the event that an attacker gains direct access to the underlying database. Blinding serves as an additional layer of defense in the event of a breach, but also to prevent information leakage and ensure requester anonymity and password secrecy via private set intersection (detailed shortly). As the key b has no external dependencies, the server can rotate it regularly by first decrypting old records and then re-blinding with a new key b'.

CreateRequest: When generating a request, a client repeats the same hashing and blinding strategy as the server. We outline this process in Algorithm 2. In contrast to the server, the client adopts its own secret key a which it initializes per request. The resulting request includes the 2-byte hash prefix and the blinded full hash. This 2-byte prefix—while leaking some bits of password material—provides the client with k-anonymity over the universe of all username and password pairs (not just those in breaches). Previous investigations of password usage estimate that users have roughly 6–8 unique passwords [16, 44, 56]. With an estimated 3.9 billion Internet users in the world [52], if we assume each user has just one unique username, this amounts to an estimated 23.4–31.2 billion unique credential pairs. As a rough approximation then, a user will share their credential prefix with 357,000–476,000 other credentials. Even if an adversarial server were to precompute a dictionary of the most popular passwords, they would have to repeat this process for each individual username. As such, our protocol satisfies our computational requirement for requester anonymity and password secrecy. In the case of an adversarial client, any request for a guessed credential is gated on the successful computation of an expensive hash, thus satisfying our requirement for an inefficient oracle.

CreateResponse: A server responds to a request according to Algorithm 3. Given a hash prefix, the server returns all known unsafe credentials S' tied to the prefix. While ideally we could provide the entire blinded contents of S to a client, in practice this is too computationally expensive as $|S|$ scales to billions of records. By partitioning S, we can limit the data downloaded to a client while ensuring membership correctness, at the cost of working with anonymity sets rather than perfect se-

[2] According to libsodium, this amounts to roughly 0.7 seconds on a 2.8 Ghz Core i7 CPU [34].

[3] We use multiplicative notation to refer to elliptic-curve group operations in the paper.

Algorithm 1 CreateDatabase: Store a blinded and strongly hashed copy of all known breached credentials.

Require: $S = \{(u_1, p_1), \ldots, (u_n, p_n)\}$, $b = rand()$, and $n = 2$, a prefix length

function CREATEDATABASE(S, b, n)
  for $(u_i, p_i) \in S$ **do**
    $u_i' \leftarrow$ CANONCIALIZE(u_i)
    $H \leftarrow$ HASH(u_i', p_i)
    $H^b \leftarrow$ BLIND(H, b)
    $H_{[0:n]} \leftarrow$ BYTESUBSTRING(H, n)
    PARTITIONSTORE($H_{[0:n]}, H^b$)
  end for
end function

Algorithm 2 CreateRequest: Client query to determine whether a blinded username and password with a cleartext hash prefix was exposed in a breach.

Require: n, a prefix length

function CREATEREQUEST(u, p, n)
  $a \leftarrow$ RAND()
  $u_i' \leftarrow$ CANONCIALIZE(u)
  $H \leftarrow$ HASH(u', p)
  $H^a \leftarrow$ BLIND(H, a)
  $H_{[0:n]} \leftarrow$ BYTESUBSTRING(H, n)
  LOCALSTORE(a)
  return HSTSREQUEST($H_{[0:n]}, H^a$)
end function

Algorithm 3 CreateResponse: Server response for all information known about the cleartext hash prefix.

Require: $b = rand()$

function CREATERESPONSE($H_{[0:n]}, H^a$)
  $H^{ab} \leftarrow$ BLIND(H^a, b)
  $S' \leftarrow$ PARTITIONLOOKUP($H_{[0:n]}$)
  return HSTSRESPONSE(H^{ab}, S')
end function

crecy. As noted in Section 2.5, the best current constructions dictate that without partitioning S, we cannot hope to deploy a scheme with reasonable limits on data downloaded by clients and computation performed by the server. By avoiding any client nonce or salt for hashing, retrieval is entirely static for the server apart from inexpensive blinding (at least compared to hashing). This satisfies our requirement for resistance to denial of service.

Providing S' absent blinding would leak information about other exposed credentials. Instead, we rely on Diffie-Hellman private set intersection [24] which is relatively efficient for a network setting on non-mobile devices [45]. The server returns all known breached credentials blinded with b while providing a client with an index into the doubly-blinded list H^{ab}. This requires the commutative properties of elliptic curve Diffie–Hellman (ECDH) such that the client can decrypt this result to recover H^b during verification, while the remaining contents of S' remain hidden.

More formally, under the random oracle model [3], with Argon2 modeled as a perfect hash function, our hash-and-blind scheme implements an oblivious pseudorandom function (OPRF) against honest-but-curious adversaries under the decisional Diffie-Hellman assumption. When b is kept secret, outputs of the hash-and-blind scheme on any user inputs $(\mathsf{u}_i, \mathsf{p}_i)$ reveal no information about the hashed and blinded output on *any other* $(\mathsf{u}', \mathsf{p}')$. A more technical and detailed note is laid out in Appendix B. This achieves bounded leakage and given only the leakage L as defined in Section 2.3, we can construct a Simulator to simulate the entire response of the server.

Verdict: Finally, a client determines whether their credential was exposed in a breach by finishing the private set intersection protocol as detailed in Algorithm 4. This process is entirely local and, absent independent telemetry, never reveals the verdict of a match to the server.

3.2 Zero-password leakage variant

Our previous approach makes a practical tradeoff between client hashing overhead and revealing some bits of a client's password. (While still protected by a computationally expensive hash and anonymity sets spanning both usernames and passwords, this information can be leaked if an attacker has auxiliary information about the username.) As an alternative, we outline a zero-password leakage variant. In Algorithm 2, a client now calculates a hash prefix of only the username $\mathsf{H}(\mathsf{u}')_{[0:n]}$ along with a blinded hash of the entire credential. Algorithm 1 is modified to create a mapping between $\mathsf{H}(\mathsf{u}_i')_{[0:n]}$ to $\mathsf{H}(\mathsf{u}_i', \mathsf{p}_i')$ and to use it to partition the database by $\mathsf{H}(\mathsf{u}_i')_{[0:n]}$. This variant still provides the same protection with bounded leakage, denial of service resistance, and an inefficient oracle, and modifies (and reduces) the anonymity set of credentials to only usernames. For an estimated 3.9 billion unique usernames, this amounts to $|\mathsf{K}| = 60{,}000$.[4] However, this variant ensures that all password material from the client is protected by blinding. In practice, given near real-time constraints, this requires that a client spend twice as much time hashing which is non-negligible.[5] For the purposes of our initial deployment (detailed in Section 5), we opted for the first variant to understand the computational bounds of clients. We now plan to migrate to the zero-password leakage variant.

[4] With no password guessing required, it also enables an attacker to reasonably pre-compute the Argon2 hash of all possible usernames.

[5] This expense can be amortized if the client reuses their username for multiple sites with distinct passwords, or if the client regularly polls the server for the same username to obtain the most recent breach status.

Algorithm 4 Verdict: Final client-side verdict for whether a username or password was exposed in a breach.

Require: a, secret key for original request
1: **function** VERDICT(H^{ab}, S', a)
2: $H^b \leftarrow$ UNBLIND(H^{ab}, a)
3: **return** $H^b \in S'$
4: **end function**

3.3 Expansion to metadata

Our protocol currently does not include information on the origin of an exposed credential as metadata (e.g., which service was compromised). In practice, we believe this is the best strategy as origin information is both untrustworthy and often unavailable. For example, large composite breaches such as Collection 1-5 and Antipublic include hundreds of millions of credentials, all of which are unattributed [21, 25]. Moreover, metadata expands the size of data downloaded as part of S'.

For completeness, our protocol can be extended to include origin information, or any metadata, by encrypting it with the output of a cryptographically secure key-derivation function such as HKDF [30] applied to $\mathsf{H}(\mathsf{u}, \mathsf{p})$. This approach limits access strictly to clients that prove knowledge of the associated, strongly hashed username and password. This is easy to observe; as outlined in Appendix B, the hashed-and-blinded outputs still hide information about other usernames and passwords and hence the derived keys are cryptographically strong and hide the contents of encrypted metadata. This is only done once when creating the database and adds very little overhead to the system. We note that it is crucially important that this metadata not include sensitive personally identifying information as it is not hidden from a compromised service.

3.4 Limitations

Our protocol requires that clients are capable of computing an expensive hash with 256MB of memory. This is a necessary requirement to hamper attackers, but it may also prove untenable for resource-constrained devices. Additionally, our approach requires that clients download a non-negligible amount of data. For context, with 1 billion credentials uniformly split into 2^{16} prefixes, this equates to roughly 15,000 blinded hashes per request. At 29-bytes per item, that amounts to roughly 435KB on average. This grows linearly with the volume of newly discovered credentials.

4 Implementation

We implemented our protocol as a publicly accessible API hosted on Google Cloud. The API mediates access to over 4 billion unique usernames and passwords collected using an approach previously documented by Thomas et al. [51]. Canonicalization further reduces this set to 3.36 billion credentials. We also developed a Chrome extension as a proof of concept client that we could share among early testers to gather telemetry on the frequency and impact of breach notifications in the wild. In practice, other applications that handles credentials can integrate with our service by implementing the client half of our protocol.

4.1 Client

Our Chrome extension monitors when users submit their username and password on a login page and generates a browser warning for breached credentials detected by our API. We rely on a JavaScript implementation of Argon2 from libsodium for all hashing and a web assembly compilation of OpenSSL for the elliptic curve computation required for private set intersection. Both libraries are open source with multiple years of vetting. Here, we discuss the details behind our extension, the design of our warning dialogues, and the telemetry the extension collects.

Detecting login events: At present, Chrome does not export an API for detecting login events. Instead, our extension registers a callback function to interpose on all `webRequests` that contain form data. When triggered, the extension relies on heuristics to detect whether the form contains a username or password field, such as matching on field names like `password` and `passwd`. If the heuristic fails to detect both a username and password, nothing is sent to our API. We manually tested our detection on the Alexa US Top 50: we successfully captured login events for 40 pages and failed for 4, while the remaining 6 did not have login forms. For the failures, login information was either obfuscated (e.g., a byte blob of all field data), or part of the payload body rather than form data. We are thus cautious when discussing data from our real world deployment in Section 6 that not every domain will be covered by our technique.

Warning design: Our extension modifies the DOM of the page where a user entered their breached credential to show a warning similar to Figure 2. In the browser tray, users can reach an extension popup that displays a stateful warning –similar to Figure 3. This gives users a way to see past warnings, in the event that they closed their browser tab before reviewing the warning (or due to a DOM refresh that overwrites our modifications). Additionally, this serves as a secure UI element that runs in isolation of other extensions and pages. Both styles of warning never reveal information about the username or password found in a breach. This design decision limits the context we can provide users, but allows us to avoid storing sensitive credential material that might make persistent local storage a target for attacks.

In designing our warning, we followed emerging advice about data breach notifications [20], proven terminology around data breaches [1, 28], and historical studies of browser

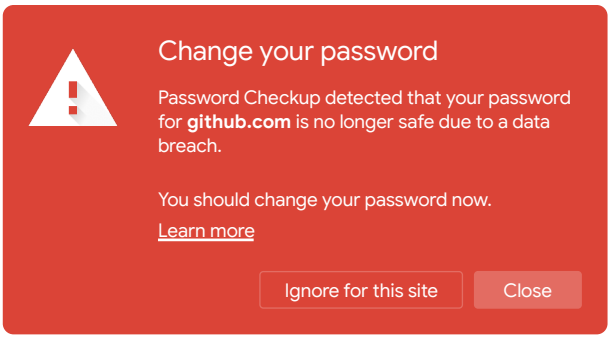


Figure 2: In-page warning generated by our extension when we detect that a credential is no longer secure due to a breach.

warnings related to phishing and unsafe network connections [2, 15]. In particular, we provided a clear action—"Change your password"—along with context for the danger behind the event. At the same time, we minimized unnecessary or overly technical information. We also provided a "Learn More" link that explained in greater detail the root cause of the warning and security best practices. In particular, users should (1) reset their password for the affected page; (2) reset their password wherever it was reused; (3) consider a password manager; and (4) consider adopting two-factor authentication. We collected feedback from 550 early testers from our organization before settling on the final design and language of our dialogue.

Compared to other browser warnings where the safest action is to close the tab, breached passwords require users to follow a series of unguided, proactive next steps. We emphasize unguided as there is no canonical account security page for every site to simplify password resetting. While there are industry initiatives to create common reset paths [41], these have yet to materialize. As such, we consider a more formal usability study of the warning experience—and automating the password change process—as future work. We provide a deeper treatment of the effectiveness of our warnings in terms of successful password resets later in Section 6 (in short, a quarter of warnings result in a reset during our observation period).

Identifying user actions: By default, our extension continuously triggers a warning each time the user authenticates with a breached credential. Given the computation and network overhead involved for each API query, if the extension detects a breached credential, it caches a 12-byte prefix of the Argon2 credential hash to avoid generating a new API query for the same credential. This also reduces the latency between a user entering a credential and observing a warning on all subsequent logins to the same domain. Conversely, if a credential was previously not present in a breach, we avoid caching any verdict and perform a new API query on each login. In the future, caching here is also possible if the cache were invalidated upon the server announcing the arrival of a new breach.

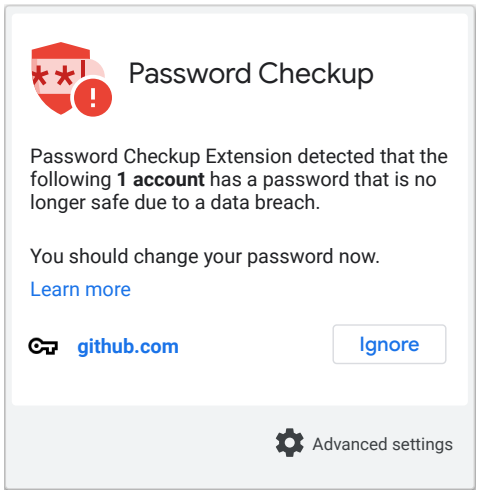


Figure 3: Stateful icon tray warning message to remind users which accounts need their attention. This avoids the transient nature of in-page warnings, which we use to provide better context to users.

For low-value accounts that a user might deem unnecessary to secure, we provide an option to ignore our warning on a per-domain basis as shown in Figure 2 and Figure 3. The extension manages this state by caching a local copy of the domain involved and a 12-byte prefix of the Argon2 hash of the credential that the user ignored (which is necessary in the event the user has multiple accounts on the domain).

We detect when a user resets their exposed password in order to provide a positive feedback signal to the user that their account is no longer at risk. We also purge all cached information about the now stale credential. To do this, we cache a 12-byte Argon2 prefix of an account's username (with only an 8MB memory requirement)—used only locally— along with a 12-byte prefix of the Argon2 credential hash. If the credential hash changes for the same username, this indicates the user signed in with a new password and that all local state for the credential should be reset. In the event a user merely mistyped their breached password, correct password entry will trigger a new warning and refresh the cache.

Telemetry: We instrument our extension to report anonymous telemetry pertaining to the volume of lookups against our API that result in a breach warning, along with whether users ignore our warnings or reset their passwords. All of these events lack any form of user identifier, precluding the possibility of correlating events or understanding per-user experiences. Each event also includes the domain of the login page involved, which we use to estimate our compatibility with popular sites and to estimate the prevalence of breached passwords across the Internet. For password changes related to breached credentials, we also report the strength of the old and new password to understand whether users as a whole migrate to stronger passwords. We use zxcvbn [57] for strength

estimation as it is entirely client-side and open source. This telemetry forms the basis of our analysis of the impact of password breach warnings in the wild, discussed in Section 6. We disclose the data we collect upfront to users in the description of our Chrome Webstore listing.[6] We had all of our telemetry reviewed by a group of internal experts and followed our organization's ethics review process.

4.2 Storage

We partitioned our pre-computed, blinded and hashed credential corpus (totaling roughly 110GB) into 2^{16} slices. We stored each slice as a static file in Google Cloud Storage. We restricted access to these files so that only the server handling requests could fetch content from storage. We also stored the key material necessary to re-blind client-blinded hashes in the same storage system.

4.3 Server

The stateless nature of our credential breach protocol allowed us to implement our serving using Google Cloud Functions. The primary benefit of this approach is that we could scale arbitrarily to the volume of incoming requests while also avoiding dormant compute cycles on pre-requisitioned cloud instances. This design also allowed us to avoid having to reason about the side-effects across requests. We implemented our Cloud Function using the same JavaScript elliptic curve library as our Chrome extension (recall that hashing is not part of the server protocol). We avoid application-layer denial of service attacks—such as sending an arbitrary length string for the server to blind—by blocking malformed requests that do not adhere to the fixed-length blinded hash we expect from a client.

5 Deployment

We made our extension publicly available via the Chrome Web Store and announced it through major media channels. In total, 667,716 users installed our extension over a measurement period of February 5, 2019–March 4, 2019 (UTC).

User demographics: Based on aggregate statistics provided by the Chrome Web Store, 48% of the users who installed our extension were from North America, 29% from Europe, 17% from Asia, and the remaining 6% from around the world. In terms of operating systems, 71% of users who installed the extension used Windows, 14% used MacOS, 13% ChromeOS, and 2% Linux. We note that extensions are unavailable on mobile devices and thus are not present in our device breakdown.

[6] *https://chrome.google.com/webstore/detail/password-checkup/pncabnpcffmalkkjpajodfhijclecjno*

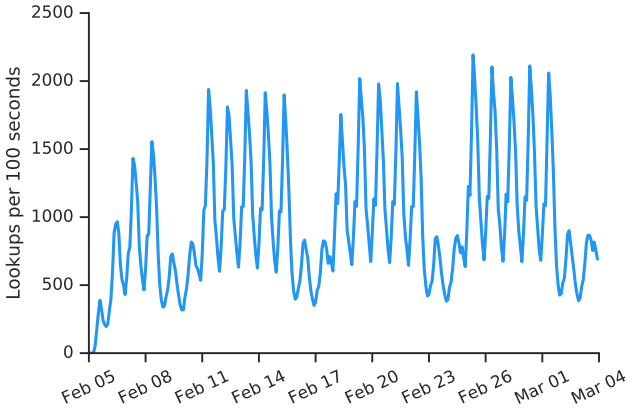


Figure 4: Volume of logins scanned by our extension every 100 seconds. Requests to our API scaled from 0.11 queries per 100 seconds in early testing, to a peak of 2,192 queries per 100 seconds at the end of our measurement window. The dips in the graph reflect lower activity during weekends.

Scaling to requests: Over the course of our measurement window, the lookup volume to our API scaled gracefully from 0.11 lookups per 100 seconds during early testing to a peak of 2,043 lookups per 100 seconds as shown in Figure 4. The diurnal pattern present reflects the geographic concentration of users in North America and Europe. The periodic dips reflect lower login activity over the weekend. By comparing query volume with active user metrics provided by the Chrome Web store, we estimate that an average user generates 3 API requests (e.g., logins) per weekday, and 1.5 requests per weekend. Critically, the diurnal cadence and lack of bursty behavior indicates a lack of large-scale abuse during our measurement window which might otherwise pollute our analysis later in Section 6.

Client overhead: We present a breakdown of the computational overhead and network latency incurred by clients that query our API in Table 2. Overall, a median query took 8.5 seconds to return a verdict, during which a user would continue browsing uninterrupted. Roughly half of this time was spent strongly hashing the user's credential, while the remaining time was spent downloading potential credential matches. Our username hash (used for locally caching state) took a median of 100ms and was a negligible part of this delay. For 10% of users, the overall query time exceeded 18 seconds, half of which was spent in network latency. While part of this lookup overhead can be optimized—credential hashing in native code takes an average of 0.7 seconds—the only way to reduce network latency would be to download fewer breached records, thus reducing the k-anonymity set of our protocol. As such, our current privacy constraints likely remain out of reach for resource-constrained devices, at least for near real time detection.

Duration	Median	90%	95%
Argon2 username hash	0.1s	0.3s	0.3s
Argon2 credential hash	4.4s	9.8s	12.7s
End-to-end API query	8.5s	18.8s	26.9s

Table 2: Time spent performing API operations including hashing and downloading potentially matching breached credentials.

Cost modeling: A practical reality of running a breach detection service is cost. In our case, cost is intrinsically tied to the k-anonymity privacy that we provide. Every 1,000 invocations of our API costs approximately $0.19 at the current volume of credentials in our storage and for a 2-byte k-anonymity prefix. Data serving makes up 94% of this cost, while the CPU and memory necessary to field requests and to re-encrypt client credentials makes up only 5%. Based on our query volume per user, operating our service for an estimated 500,000 users would cost $85,500 a year. Caching the status of negative breach verdicts would substantially reduce expenses. Our goal in documenting these details is to provide other members of the community a benchmark for the costs of any improved privacy scheme. For our protocol, adding a single bit of privacy nearly doubles our operating expenses while also doubling the network latency for clients.

6 Analysis

We analyzed the anonymous telemetry reported during our measurement window to understand the state of breached passwords across the Internet. Facets we consider include the frequency that users log in with a breached password, the types of sites where reuse is most common, and ultimately whether displaying warnings helps users to address the risk of credential stuffing. We provide a high-level statistical summary of our telemetry in Table 3 . We note that our telemetry is biased towards the users who installed our extension, which is a non-random sample of the Internet population.

6.1 Credential stuffing risk and remediation

Frequency of breached credential reuse: Overall, our API fielded 21,177,237 lookup requests, where a lookup maps to a single login attempt performed by an anonymous user. We detected that 316,531 logins involved breached credentials—roughly 1.5% of all logins. We caution this is a lower bound as we only generate telemetry for breached credentials once before caching the result locally, whereas lookups to non-breached credentials generate telemetry upon each new login. Our detection rate is lower than the 6.9% reported by Thomas et al. [51] for 751 million Google accounts and 1.9 billion breached credentials. Possible reasons include the user population that adopted our extension is more security conscious—thus avoiding reuse as a behavior—or that dormant accounts have a higher reuse rate, which by nature our extension cannot observe as we perform checks at login time. During our 28 day measurement window, if we assume that logins and warnings are uniformly distributed across users, 47.3% of our users received a warning. Our anonymous reporting precludes more detailed per-user statistics. Taken as a whole, our results reveal that global Internet users regularly access accounts that are vulnerable to credential stuffing.

Metric	Value
Extension users	667,716
Logins analyzed	21,177,237
Domains covered	746,853
Breached credentials found	316,531
Warnings ignored	81,368 (26%)
Passwords reset	82,761 (26%)

Table 3: Summary of the anonymous telemetry data reported over the course of our analysis window from February 5–March 4, 2019.

Ignoring breached credentials: Users opted to ignore 81,368—or 25.7%—of the breach warnings we surfaced. We consider three possible explanations. Users may be making an explicit risk assessment that the value of their account is not worth the effort of adopting a new password. Alternatively, users may not be in full control of the account (e.g., a shared household account) [37]. Finally, as our extension does not automate the process of password resetting, users may ignore our warning out of frustration due to a lack of guidance. Regardless of the underlying cause, ignored warnings leave accounts vulnerable to credential stuffing. That said, there is an opportunity here for identity providers to take action and guide users through the password resetting process.

Remediation of breached passwords: Our warnings resulted in users resetting 82,761—or 26.1%—of their breached passwords. Critically, we find that users used this opportunity to migrate to stronger passwords. On average, the passwords we detected as breached had a zxcvbn strength of 1.6. After remediation, this score increased to an average of 2.9. We present a more detailed summary of strength before and after resetting in Figure 5. For context, a score of one indicates a "weak password" that an attacker can guess in under 10^6 attempts. A score of two reflects a password that an attacker can guess in under 10^8 attempts, and a score of three 10^{10} attempts and is considered "strong".

Overall, 94% of password changes led to a stronger or equal zxcvbn score, while just 6% of changes resulted in a regression to a weaker password. Our results indicate that users of our extension understand stronger password compo-

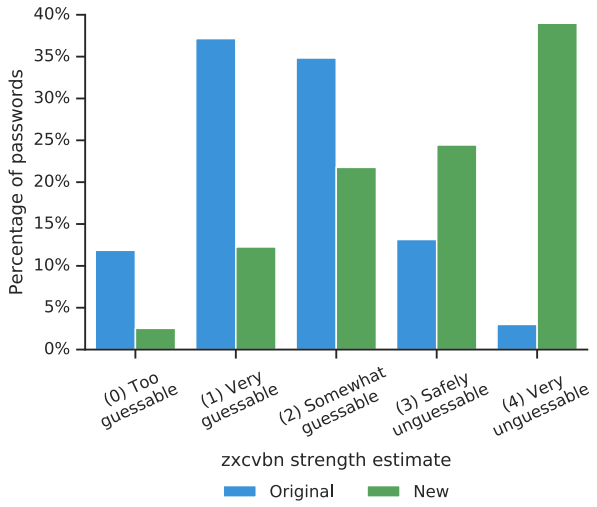


Figure 5: Histogram of zxcvbn password strength for passwords detected as breached and the password adopted by users after remediation. Users migrated towards stronger passwords overall as a result of our warnings.

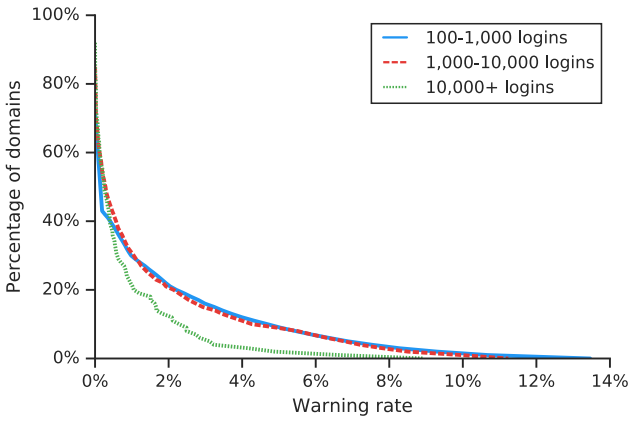


Figure 6: CCDF of the percentage of logins per domain that result in a warning across. We group domains by the volume of logins we observed, including 100-1,000, 1,000–10,000, and 10,000+. Popular sites tend to face less of a threat from credential stuffing, while the long tail of domains remain at risk.

sition strategies. Equally important, 39% of new passwords achieved the highest possible strength score (up from 3% for the original passwords), a potential sign of the growing prevalence of password managers that automatically compose strong passwords. Our results highlight how surfacing actionable security information can help mitigate the risk of account hijacking.

6.2 Influence of domains on account security

Category: We examine whether the perceived value of an account influences the rate that users rely on reused, breached credentials. To do this, we manually labeled the top 332 domains that received more than 5,000 logins during our measurement period into one of thirteen categories (e.g., finance, email and messaging, and social networking). We used a catch all "Other" category for domains that fell outside this categorization. Combined, logins to these domains accounted for 41% of lookups against our API.

We present a breakdown of the aggregate warning rates and ignore rates across all domains per category in Table 4. Domains that we categorized as related to finance or governments exhibited the lowest rate of reused, breached credentials (0.2–0.3%). Possible explanations include the password composition policies of these domains, the fact that users adhere to popular security advice to have one strong password for their bank, or that the sites actively identify breaches and previously forced password resets. In contrast, entertainment sites like streaming video platforms and adult websites had the highest warning rate for breached credentials (3.6–6.3%). Users may adopt disposable passwords due to perceived lack of risk, or in the case of streaming sites, they may use shared accounts. Surprisingly, users ignored our breach warnings nearly uniformly across categories, with the exception of adult websites. For the latter, users ignored nearly twice as many of our warnings—potentially to hide the domain from our persistent warning tray (see Figure 3 earlier).

Popularity: We also consider whether more popular sites are less vulnerable to credential stuffing. We present a CCDF of the frequency of warnings per domain versus the volume of logins to the domain during our analysis window in Figure 6. We find that just 6% of domains with 10,000+ logins have a warning rate higher than 3%, compared to 15% of domains with fewer than 10,000+ logins. We believe this gap in security results from larger security investments on the part of popular domains towards proactively resetting passwords and helping users avoid "weak" passwords. While large identity providers can equally take advantage of our API, addressing the long tail of domains affected by credential stuffing likely requires relying on in-browser warnings.

7 Related Work

Account hijacking threats: Credential stuffing represents just one dimension of account hijacking threats. Other risks include large-scale phishing [7, 51], credential or token theft from local machines [50], and even targeted attacks [35, 40]. Users have internalized these risks and adopted a security model of joint responsibility between themselves and identity providers [47]. The most prominent solutions to these threats include users adopting two-factor authentication, or identity providers expanding authentication to include other passive

Category	Domains	Total visits	Breakdown	Warning rate	Ignore rate
Finance	90	1,684,851	8.0%	0.3%	18.6%
Email, messaging	47	1,519,795	7.2%	0.5%	14.0%
Social networking	15	1,191,546	5.6%	0.8%	17.8%
Shopping	29	1,007,103	4.8%	1.2%	16.4%
Technology	34	624,702	2.9%	0.7%	16.9%
Business	12	585,797	2.8%	0.7%	20.3%
Education	16	261,563	1.2%	0.9%	26.5%
Gaming	11	201,646	1.0%	0.5%	18.6%
Entertainment	9	168,565	0.8%	6.3%	27.1%
Travel	14	138,968	0.7%	1.8%	19.6%
Government	5	60,967	0.3%	0.2%	16.9%
News	5	54,864	0.3%	1.9%	20.7%
Adult	3	50,408	0.2%	3.6%	38.5%
Other	42	429,786	2.0%	1.0%	17.8%

Table 4: Breakdown of reused, breached passwords for domains receiving more than 5,000 logins, aggregated by business sector. Finance and govt. domains had the lowest usage of breached passwords, compared to entertainment and adult-related domains.

factors such as a user's device and location [12, 17]. The protections we propose in this work are complementary to a defense in depth authentication model, where breach detection represents one additional factor in risk modeling.

Password reuse behaviors: Text passwords continue to be the prevailing mechanism for online authentication. Given the human constraints of memorizing a large number of unique text strings, people have adopted various strategies—including reuse and weak patterns—for managing their growing number of online identities [18, 23, 49, 55]. Florencio and Herley published the first large-scale study of password behavior, where they found both weak and reused passwords were a frequent flaw [16]. More recently, Wash et al. [56] and Pearman et al. [44] observed the password usage behaviors of hundreds of participants over multiple weeks. They estimated that 32% of all entered passwords involved exact reuse. Wash et al. found that users reused their most popular password on an average of 9 sites. Examining breach data directly, Das et al. found that 43–51% of users reused the same password on multiple sites [10]. While automated password filling has become more commonplace—participants used these means 57% of the time [44]—both Pearman et al. and Wash et al. found password managers have yet to be adopted as a tool for password generation. All of these factors compound the threat of credential stuffing, where inverting a single weak password hash can grant an attacker access to multiple sites.

Improving breach alerting protocols: In a contemporaneous work, Li et al. presented a framework for reasoning about leakages resulting from the password-based prefixes used by our protocol and HaveIBeenPwned [33]. The authors show how a password-only prefix (or an attacker with access to the plaintext username in a username-password prefix) can leverage a partition's underlying password distribution to reduce the number of guesses necessary to potentially learn a user's password. To address this, the authors outline a zero-password leakage variant that relies on private set membership in conjunction with a username hash prefix for partitioning, akin to our own model from Section 3.2. Their work provides further motivation for a zero-password leakage protocol, despite its additional computational complexity as we outlined.

8 Conclusion

In this paper, we demonstrated the feasibility of a privacy-preserving protocol that allows a client to query whether their login credentials were exposed in a breach, without revealing the information queried. Our protocol relies on a combination of computationally expensive hashing, k-anonymity, and private set intersection. Our approach improves on existing protocols by taking into account both an adversarial client and server, while also minimizing the chance of false positives. We envision this service being used by end users, password managers, and by identity providers. As a proof-of-concept, we created a cloud service that mediates access to 4 billion usernames and passwords publicly exposed by breaches. We then released a Chrome extension that would query credentials entered at login time against our service. Based on telemetry produced by nearly 670,000 users, we estimated that 1.5% of credentials used across the web are vulnerable to credential stuffing (based on a sample of 21 million logins).

Addressing this problem requires action from both users and identity providers. In the context of our study, 26% of the warnings we generated for breached passwords resulted in users adopting a new password—94% of which were stronger or as strong as the original. Both the volume of user interest and response rate surfaced during our study demonstrate that there is an appetite on the part of users to secure their accounts from credential stuffing. We hope that by making

our protocol public, other researchers can improve on the privacy protections, computational bounds, and cost models that we establish. Our protocol is a first step in democratizing access to breach alerting in order to mitigate one dimension of account hijacking.

9 Acknowledgements

We would like to thank Oxana Comanescu, Sunny Consolvo, Ali Zand, and our anonymous reviewers for their feedback and support in designing our breach alerting protocol. This work was partially supported by funding from the NSF.

References

[1] Lillian Ablon, Paul Heaton, Diana Catherine Lavery, and Sasha Romanosky. Consumer attitudes toward data breach notifications and loss of personal information. In *Proceedings of the Workshop on the Economics of Information Security*, 2016.

[2] Devdatta Akhawe and Adrienne Porter Felt. Alice in warningland: A large-scale field study of browser security warning effectiveness. In *Proceedings of the USENIX Security Symposium*, 2013.

[3] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the ACM Conference on Computer and Communications Security*, 1993.

[4] Borbala Benko, Elie Bursztein, Tadek Pietraszek, and Mark Risher. Cleaning up after password dumps. *https://security.googleblog.com/2014/09/cleaning-up-after-password-dumps.html*, 2014.

[5] Rainer Böhme and Jens Grossklags. The security cost of cheap user interaction. In *Proceedings of the New Security Paradigms Workshop*, 2011.

[6] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *Proceedings of the Annual Symposium on Foundations of Computer Science*, 1995.

[7] Marco Cova, Christopher Kruegel, and Giovanni Vigna. There is no free phish: an analysis of "free" and live phishing kits. In *Proceedings of the Workshop on Offensive Technologies*, 2008.

[8] Claude Crépeau. Equivalence between two flavours of oblivious transfers. In *Conference on the Theory and Application of Cryptographic Techniques*, 1987.

[9] Luke Crouch. When does firefox alert for breached sites? *https://blog.mozilla.org/security/2018/11/14/when-does-firefox-alert-for-breached-sites/*, 2018.

[10] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. The tangled web of password reuse. In *Proceedings of the Network and Distributed System Security Symposium*, 2014.

[11] Xavier De Carné De Carnavalet, Mohammad Mannan, et al. From very weak to very strong: Analyzing password-strength meters. In *Proceedings of the Network and Distributed System Security Symposium*, 2014.

[12] Periwinkle Doerfler, Maija Marincenko, Juri Ranieri, Angelika Moscicki Yu Jiang, Damon McCoy, and Kurt Thomas. Evaluating login challenges as a defense against account takeover. In *Proceedings of the Web Conference*, 2019.

[13] Peter Dolanjski. Testing firefox monitor, a new security tool. *https://blog.mozilla.org/futurereleases/2018/06/25/testing-firefox-monitor-a-new-security-tool/*, 2018.

[14] Adam Everspaugh, Rahul Chaterjee, Samuel Scott, Ari Juels, and Thomas Ristenpart. The pythia PRF service. In *Proceedings of the USENIX Security Symposium*, 2015.

[15] Adrienne Porter Felt, Alex Ainslie, Robert W Reeder, Sunny Consolvo, Somas Thyagaraja, Alan Bettes, Helen Harris, and Jeff Grimes. Improving ssl warnings: Comprehension and adherence. In *Proceedings of the Conference on Human Factors in Computing Systems*, 2015.

[16] Dinei Florencio and Cormac Herley. A large scale study of web password habits. In *Proceedings of the International World Wide Web Conference*, 2006.

[17] David Mandell Freeman, Sakshi Jain, Markus Dürmuth, Battista Biggio, and Giorgio Giacinto. Who are you? a statistical approach to measuring user authenticity. In *Proceedings of the Symposium on Network and Distributed System Security*, 2016.

[18] Shirley Gaw and Edward W. Felten. Password management strategies for online accounts. In *Proceedings of the Symposium on Usable Privacy and Security*, 2006.

[19] Yael Gertner, Yuval Ishai, Eyal Kushilevitz, and Tal Malkin. Protecting data privacy in private information retrieval schemes. *Journal of Computer and System Sciences*, 2000.

[20] Maximilian Golla, Miranda Wei, Juliette Hainline, Lydia Filipe, Markus Dürmuth, Elissa Redmiles, and Blase Ur. What was that site doing with my facebook password?: Designing password-reuse notifications. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2018.

[21] Andy Greenberg. Hackers are passing around a megaleak of 2.2 billion records. *https://www.wired.com/story/collection-leak-usernames-passwords-billions/*, 2019.

[22] Iftach Haitner, Jonathan J Hoch, and Gil Segev. A linear lower bound on the communication complexity of single-server private information retrieval. In *Proceedings of the Theory of Cryptography Conference*, 2008.

[23] Eiji Hayashi and Jason Hong. A diary study of password usage in daily life. In *Proceedings of the Conference on Human Factors in Computing Systems*, 2011.

[24] Bernardo A Huberman, Matt Franklin, and Tad Hogg. Enhancing privacy and trust in electronic communities. In *Proceedings of the ACM Conference on Electronic Commerce*, 1999.

[25] Troy Hunt. Password reuse, credential stuffing and another billion records in Have I been pwned. *https://www.troyhunt.com/password-reuse-credential-stuffing-and-another-1-billion-records-in-have-i-been-pwned/*, 2017.

[26] Troy Hunt. Have i been pwned? *https://haveibeenpwned.com/*, 2019.

[27] Stanisław Jarecki and Xiaomin Liu. Fast secure computation of set intersection. In *Proceedings of the International Conference on Security and Cryptography for Networks*, 2010.

[28] Sowmya Karunakaran, Kurt Thomas, Elie Bursztein, and Oxana Comanescu. Data breaches: user comprehension, expectations, and concerns with handling exposed data. In *Proceedings of the Symposium on Usable Privacy and Security*, 2018.

[29] Joe Kilian. Founding crytpography on oblivious transfer. In *Proceedings of the Symposium on Theory of Computing*, 1988.

[30] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *Proceedings of the Annual Cryptology Conference*, 2010.

[31] Brian Krebs. Sextortion scam uses recipient's hacked passwords. *https://krebsonsecurity.com/2018/07/sextortion-scam-uses-recipients-hacked-passwords/*, 2018.

[32] Eyal Kushilevitz and Rafail Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, pages 364–373. IEEE, 1997.

[33] Lucy Li, Bijeeta Pal, Junade Ali, Nick Sullivan, Rahul Chatterjee, and Thomas Ristenpart. Protocols for checking compromised credentials. *https://rist.tech.cornell.edu/papers/c3.pdf*, 2019.

[34] libsodium. The Argon2 function. *https://libsodium.gitbook.io/doc/password_hashing/the_argon2i_function*, 2019.

[35] William R Marczak, John Scott-Railton, Morgan Marquis-Boire, and Vern Paxson. When governments hack opponents: a look at actors and technology. In *Proceedings of the USENIX Security Symposium*, 2014.

[36] Neil Matatall. New improvements and best practices for account security and recoverability. *https://github.blog/2018-07-31-new-improvements-and-best-practices-for-account-security-and-recoverability/*, 2018.

[37] Tara Matthews, Kerwell Liao, Anna Turner, Marianne Berkovich, Robert Reeder, and Sunny Consolvo. She'll just grab any device that's closer: A study of everyday device & account sharing in households. In *Proceedings of the Conference on Human Factors in Computing Systems*, 2016.

[38] Catherine Meadows. A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1986.

[39] William Melicher, Blase Ur, Sean M Segreti, Saranga Komanduri, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Fast, lean, and accurate: Modeling password guessability using neural networks. In *Proceedings of the USENIX Security Symposium*, 2016.

[40] Ariana Mirian, Joe DeBlasio, Stefan Savage, Geoffrey M. Voelker, , and Kurt Thomas. Hack for hire: Exploring the emerging market for account hijacking. In *Proceedings of The Web Conf*, 2019.

[41] Theresa O'Connor. A well-known url for changing passwords. *https://wicg.github.io/change-password-url/index.html*, 2018.

[42] Password Ping. LastPass selects PasswordPing for compromised credential screening. *https://www.passwordping.com/lastpass-selects-passwordping-for-compromised-credential-screening/*, 2017.

[43] Password Ping. Block attacks from compromised credentials. *https://www.passwordping.com/*, 2019.

[44] Sarah Pearman, Jeremy Thomas, Pardis Emani Naeini, Hana Habib, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Serge Egelman, and Alain Forget. Let's go in for a closer look: Observing passwords in their natural

habitat. In *Proceedings of the 2017 ACM Conference on Computer and Communications Security*, 2017.

[45] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on ot extension. In *Proceedings of the USENIX Security Symposium*, 2014.

[46] Michael O. Rabin. How to exchange secrets by oblivious transfer. Technical report, Tech. rep. TR-81, AikenComputation Laboratory, Harvard University, Cambridge, MA, 1981.

[47] Richard Shay, Iulia Ion, Robert W Reeder, and Sunny Consolvo. "My religious aunt asked why I was trying to sell her viagra": experiences with account hijacking. In *Proceedings of ACM Conference on Human Factors in Computing Systems*, 2014.

[48] Jeff Shiner. Finding pwned passwords with 1password. *https://blog.1password.com/finding-pwned-passwords-with-1password/*, 2019.

[49] Elizabeth Stobert and Robert Biddle. The password life cycle: User behaviour in managing passwords. In *Proceedings of the Symposium on Usable Privacy and Security*, 2014.

[50] Brett Stone-Gross, Marco Cova, Lorenzo Cavallaro, Bob Gilbert, Martin Szydlowski, Richard Kemmerer, Christopher Kruegel, and Giovanni Vigna. Your botnet is my botnet: Analysis of a botnet takeover. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2009.

[51] Kurt Thomas, Frank Li, Ali Zand, Jacob Barrett, Juri Ranieri, Luca Invernizzi, Yarik Markov, Oxana Comanescu, Vijay Eranti, Angelika Moscicki, et al. Data breaches, phishing, or malware?: Understanding the risks of stolen credentials. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2017.

[52] International Telecommunications Union. Statistics. *https://www.itu.int/en/ITU-D/Statistics/Pages/stat/default.aspx*, 2019.

[53] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *Proceedings of the USENIX Security Symposium*, 2018.

[54] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proceedings of the USENIX Security Symposium*, 2017.

[55] Emanuel von Zezschwitz, Alexander De Luca, and Heinrich Hussmann. Survival of the shortest: A retrospective analysis of influencing factors on password composition. In *Proceedings of the International Conference on Human-Computer Interaction*, 2013.

[56] Rick Wash, Emilee Rader, Ruthie Berman, and Zac Wellmer. Understanding password choices: How frequently entered passwords are re-used across websites. In *Proceedings of the Symposium on Usable Privacy and Security*, 2016.

[57] Daniel Lowe Wheeler. zxcvbn: Low-budget password strength estimation. In *Proceedings of the USENIX Security Symposium*, 2016.

[58] Victoria Woollaston. Facebook and netflix reset passwords after data breaches. *http://www.wired.co.uk/article/facebook-netflix-password-reset*, 2016.

A Anonymity Sets

In this section, we describe properties of anonymity sets (from Section 2.3) in more detail. Recall that anonymity sets are large sets of user credentials that provide plausible deniability about client data even if information about their membership in this set is revealed. Defining and arguing with anonymity sets is challenging and must be done carefully so as to avoid some trivialities. To avoid constructions with vacuous security, we require anonymity sets to have the following properties.

Large marginal supports: Our anonymity sets containing tuples (u,p) must additionally have sufficiently large marginal supports over both usernames and passwords. This ensures that despite there being several possible tuples (u,p), there is sufficiently large ambiguity about whether membership implies a specific username or password. A trivial anonymity set, for example, might have several possible credentials with different passwords all tied to the same username.

More mathematically, given an anonymity set K of size $|\mathsf{K}|$, we require that the size of the following sets:

$$\mathsf{SuppUser}(\mathsf{K}) := \{\mathsf{u} : (\mathsf{u},\mathsf{p}) \in \mathsf{K}\},$$
$$\mathsf{SuppPwd}(\mathsf{K}) := \{\mathsf{p} : (\mathsf{u},\mathsf{p}) \in \mathsf{K}\},$$

both have large cardinalities comparable to that of $|\mathsf{K}|$. Observe that $|\mathsf{SuppUser}(\mathsf{K})|$ indicates how many bits of information about the username is leaked (smaller sets narrow the set of possible users and leak a lot of information). This is similarly true for $|\mathsf{SuppPwd}(\mathsf{K})|$ for passwords.

Hashing both usernames and passwords with cryptographically strong hash functions satisfies these requirements. In fact, it is possible that both sets have cardinalities as large $|\mathsf{K}|$ itself which would imply that for every possible common password there might be a username such that $(\mathsf{u},\mathsf{p}) \in \mathsf{K}$. This is true in our scheme modeling Argon2 as a random oracle.

Schemes that only hash passwords, as noted previously in Section 2.4, might still satisfy a weaker anonymity property. They hide usernames, but depending on how they truncate hashes, password-only schemes might allow for small or large SuppPwd sets. Thus, they might satisfy these requirements, but only weakly at least with respect to passwords. Furthermore, it is not true that for every password there is a username which might be part of the client's credentials, unlike our scheme. We also note that our trivial example, of having several passwords all tied to the same username, violates our requirement by having $|\mathsf{SuppUser}(\mathsf{K})| = 1$.

Uniformity requirement: This is a more challenging requirement to model mathematically. Intuitively, however, it states that anonymity sets should partition the space of usernames and passwords in a somewhat uniform manner. In other words, over random choices of the system parameters, it should be equally likely for any (u,p) to end up in any anonymity set. A trivial anonymity set violating this requirement would, as an example, only truncate usernames, thereby trivially leaking some information about the username. Truncation does not make it equally likely that any (u,p) can end up in any anonymity set.

Under the reasonable assumption that our hash function is independent of the domain of typical usernames and passwords and does not have any "weak inputs"—domains of inputs where it does not behave like an ideal hash function—this condition is easily satisfied. It is highly improbable that related credentials such as `username`, `username0`, `username123`, will all end up in the same anonymity set.

B Security of the Hash-and-Blind operation

In this section, we outline the security properties satisfied by the hash-and-blind operation, which is an important part of our protocol. Consider a keyed function $F(k,x) := \mathsf{H}(x)^k$ where $\mathsf{H} : \{0,1\}^* \to \mathbb{G}$ is a hash function mapping strings to a group element. In our construction, H is Argon2, and $\mathbb{G}$ is the elliptic curve NID_secp224r1.

The work of Jarecki et al. [27] shows that $F(k,\cdot)$ implements an *oblivious* pseudorandom function in the random oracle model assuming the hardness of the decision Diffie-Hellman assumption in $\mathbb{G}$. In this section, we do not elaborate on the details of the proof, but we state what is meant by a pseudorandom function and how $F(k,\cdot)$ can be evaluated obliviously—without the secret key k holder knowing which input they're evaluating on. Pseudorandomness helps us achieve bounded leakage and protects the credentials not queried by the user; obliviousness enables us to implement the Diffie-Hellman blinding based private set intersection within our protocol.

Pseudorandomness: Informally, a function $F(k,\cdot)$ with outputs in $\mathbb{Y}$ is said to be pseudorandom if the function behaves like a random function when evaluated on new inputs. More formally, given outputs $F(k,x_1),\ldots,F(k,x_Q)$ for Q queries $x_1,\ldots,x_Q$ of an adversary's choice, for any other $x' \notin \{x_1,\ldots,x_Q\}$, we require that $F(k,x')$ be computationally indistinguishable from a random element in $\mathbb{Y}$ as long as k is chosen uniformly at random and remains hidden.

When applied to our construction, it implies that a client that sees several possible $\mathsf{H}(\mathsf{u}_i,\mathsf{p}_i)^b$ still cannot distinguish $\mathsf{H}(\mathsf{u}',\mathsf{p}')^b$ from a random element in $\mathbb{G}$ if b is hidden. Hash-and-blind therefore protects the contents of the server database when interacting with clients. For the sake of completeness, we add that this protocol is only secure against honest-but-curious adversaries which assumes that a client might be curious to learn more than it is allowed to, but chooses to honestly follow the protocol.

Obliviousness: A function is said to be evaluated in an oblivious manner if there is a protocol between a client holding an input x and a server holding a function f such that at the end of the protocol, the client learns $f(x)$ and the server learns nothing. In our construction, $f(x) = \mathsf{H}(x)^b$ for some value b. The protocol between the client and server is fairly straightforward (and somewhat implicit in our construction): the client chooses a uniform random value a, sends $\mathsf{H}(x)^a$, receives $\mathsf{H}(x)^{ab}$, and reconstructs

$$f(x) = \left(\mathsf{H}(x)^{ab}\right)^{1/a} = \mathsf{H}(x)^b.$$

Correctness is fairly straightforward. To see why this is oblivious, observe that for any two inputs x_1 and x_2, the distributions $\mathsf{H}(x_1)^r$ and $\mathsf{H}(x_2)^s$ for uniformly drawn values r and s are identical. This implies that the server learns nothing about the client's input x.

When applied to our construction, it states that the component $\mathsf{H}(\mathsf{u},\mathsf{p})^a$ computed in Algorithm 2 allows the client to obliviously evaluate the PRF without revealing to the server information about the credential (u,p).

We end this section with a couple of notes. First, a caveat noting that some information about (u,p) does end up being leaked via the anonymity set, which we capture through our notion of leakage. A direct composition of proofs of security involving anonymity sets and obliviousness might be tricky and will require careful work. Deriving keys from these PRF outputs will additionally require careful applications of KDFs with the right domain separators to avoid re-use of crypto components.

Second, Everspaugh et al. [14] propose an OPRF service that is closely related to our construction here. Our requirements out of an OPRF differs on a couple of key points which does not enable us to use such a service directly: 1) we do not require the notion of *partial* obliviousness in their construction which adds significant computational overheads to their service, and 2) our clients use of an OPRF does not require an immediate evaluation of the PRF, but rather its application to a database to obliviously evaluate its inputs and return potential matches.

Probability Model Transforming Encoders Against Encoding Attacks

Haibo Cheng[†,‡], Zhixiong Zheng[†,‡], Wenting Li[†,‡], Ping Wang[†,‡,⋆], Chao-Hsien Chu[§]
[†]*Peking University,* {hbcheng,zxzheng,wentingli,pwang}@pku.edu.cn
[‡] *Key Laboratory of High Confidence Software Technologies (PKU), Ministry of Education, China*
[§]*Pennsylvania State University,* chu@ist.psu.edu
[⋆]*Corresponding author*

Abstract

Honey encryption (HE) is a novel encryption scheme for resisting brute-force attacks even using low-entropy keys (e.g., passwords). HE introduces a *distribution transforming encoder (DTE)* to yield plausible-looking decoy messages for incorrect keys. Several HE applications were proposed for specific messages with specially designed *probability model transforming encoders (PMTEs)*, DTEs transformed from probability models which are used to characterize the intricate message distributions.

We propose attacks against three typical PMTE schemes. Using a simple machine learning algorithm, we propose a distribution difference attack against genomic data PMTEs, achieving 76.54%–100.00% accuracy in distinguishing real data from decoy one. We then propose a new type of attack—*encoding attacks*—against two password vault PMTEs, achieving 98.56%–99.52% accuracy. Different from distribution difference attacks, encoding attacks do not require any knowledge (statistics) about the real message distribution.

We also introduce a generic conceptual probability model—*generative probability model (GPM)*—to formalize probability models and design a generic method for transforming an arbitrary GPM to a PMTE. We prove that our PMTEs are information-theoretically indistinguishable from the corresponding GPMs. Accordingly, they can resist encoding attacks. For our PMTEs transformed from existing password vault models, encoding attacks cannot achieve more than 52.56% accuracy, which is slightly better than the randomly guessing attack (50% accuracy).

1 Introduction

Password-based encryption (PBE) is a fundamental scheme in many real-world systems for file encryption or authentication. However, due to the limitations of human memory, users often use weak passwords [26, 43] and reuse them [11, 30]. This leads to the vulnerability of traditional PBEs (e.g., PKCS #5 [22]) against brute-force attacks (so-called password guessing attacks), including trawling guessing attacks [25, 42] and targeted guessing attacks [29, 41].

Several methods were proposed to address this threat. We summarize these countermeasures into three types. The first type is to increase the complexity of decryption for attackers, including: 1) salting, which pressurizes attackers into enumerating passwords for every user (salt); 2) using special password-hashing functions (e.g., iterated hash functions [22, 33] and memory-hard functions [6, 31]) as the key derivation function (KDF) in PBE, which increases attackers' cost of computing and memory by a constant factor but also consumes legitimate users' extra cost by the same factor. For example, LastPass, a password vault software, utilizes these methods, including salting, 5,000 rounds of PBKDF2-SHA256 on clients and 100,000 rounds on servers [38].

The second type of countermeasures is to harden passwords with other factors (e.g., servers [12, 23], devices [19, 35, 37], biometrics [9, 28]) to generate high-entropy keys. These methods are widely used in authentication protocols, for example, two-factor authentication [20, 36]. Note that LastPass also supports YubiKey devices to secure password vaults [2]. However, these methods need additional devices (servers, biometric readers) and do worse than single password methods on deployability [8]. Besides, if the additional factor gets stolen or lost (without a backup), the message encrypted cannot be recovered (e.g., [23]).

The last type of countermeasures is to generate plausible-looking decoy messages for wrong keys to confuse attackers. Several specific encryption schemes for specific data used this method [5, 17], and Juels and Ristenpart proposed a generic method called Honey Encryption (HE) [21]. HE introduces a distribution-transforming encoder (DTE) and encodes a message following a known distribution to a uniform seed before encrypting. Therefore, plausible-looking decoy messages are generated by DTE decoding incorrect seeds when decrypting a ciphertext under wrong keys. If the DTE is perfectly secure, i.e., decoy messages are indistinguishable from real ones, then attackers enumerating all passwords only get many messages and cannot distinguish the right one. This countermeasure

achieves information-theoretic security without declining on deployability and bringing legitimate users' extra cost.

Owing to the security of HE, several applications of HE [10,14,18] were proposed. In this paper, we focus on three typical ones, including two password vault schemes [10, 14] and one genomic data protection scheme [18]. A password vault contains an individual user's multiple passwords on websites or services and is usually encrypted under a user-chosen password, called master password. Passwords stored in password vaults are of great value (e.g., PINs of credit cards, passwords of virtual currency accounts) and hence greatly attract attackers' attention. Similar to password vaults, genomic data is sensitive and needs long-term protection, as it is unchangeable during one's lifetime and correlated with his relatives.

The key of a HE scheme is to design a secure DTE. It is easy for messages following a simple distribution, for example, a uniform distribution, a normal distribution. Juels and Ristenpart [21] designed a generic purpose DTE *IS-DTE* and several specific DTEs for RSA secret keys. Notwithstanding, it is still a great challenge to design a secure DTE for messages following intricate distributions, e.g., natural language texts, passwords, password vaults, and genomic data. Probability models are usually needed to characterize the message distributions. We call these DTEs *probability model transforming encoders (PMTEs)*, which are transformed from probability models instead of distributions. Note that Chatterjee et al. [10] named DTEs for natural language texts as *natural language encoders (NLEs)*, which are a subset of PMTEs. Though all the existing PMTE schemes [10, 14, 18] are designed for specific messages, it is still of necessity to propose a generic PMTE designing method.

In addition, the security evaluations of PMTEs are not comprehensive. The designers of password vault PMTEs [10, 14] tried to use machine learning algorithms and Kullback-Leibler divergence to distinguish real and decoy vaults, without considering the difference between the real and decoy seeds. For the PMTEs in [18], it evaluated the goodness of probability models with chi-square goodness-of-fit tests, but did not study the influence of their goodness on the security of PMTEs. These issues on PMTE study hinder the widespread use of HE.

1.1 Our Contribution

In order to evaluate the security of PMTEs, we propose a framework with encoding attacks and distribution difference attacks. We show that password vault PMTEs [10, 14] suffer from encoding attacks while genomic data PMTEs [18] cannot resist distribution difference attacks. *Encoding attacks*, which are a new type of attack we propose, do not require any knowledge of real message distributions. The strong encoding attack achieves 98.56%–99.52% accuracy (in distinguishing a real vault from a decoy one) against password vault PMTEs. Meanwhile, using a principal component analysis (PCA) and a support vector machine (SVM) with a radial basis function (RBF) kernel, a distribution difference attack achieves 76.54%–100.00% accuracy against genomic data PMTEs.

We also propose a generic PMTE designing method for arbitrary probability models, by introducing a generic conceptual probability model—*generative probability model (GPM)*—to formalize probability models. We prove that our PMTEs are information-theoretically indistinguishable from corresponding GPMs, which means that they can resist encoding attacks. For our proposed PMTEs of existing password vault models, encoding attacks cannot capture more than 52.56% accuracy, compared with the randomly guessing attack (50% accuracy).

2 Background and Related Works

We introduce the basic concepts of HE as well as three typical HE applications with their specific PMTEs.

2.1 Honey Encryption

Honey Encryption (HE) [21], proposed by Juels and Ristenpart, is a novel encryption scheme using low-entropy keys (e.g., passwords) which resists brute-force attack through generating a plausible decoy message for every incorrect key. To produce decoy messages, HE introduces a randomized encoder, called *distribution transforming encoder (DTE)*.

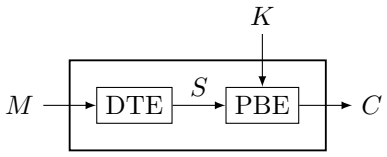


Figure 1: Honey Encryption

Figure 1 shows the encryption progress for a message M. The encryption first encodes M into a seed S by DTE, then encrypts S using PBE with the key K and finally outputs the ciphertext C. The PBE used in HE is a traditional PBE but must satisfy that decrypting any ciphertext under any key yields a valid seed (e.g., AES in CTR-mode with PBKDF). Therefore, decrypting C under an incorrect key K' will yield a wrong seed S' and a decoy message M' by decoding S'. The key of HE is designing a secure DTE which generates indistinguishable decoy messages. Juels and Ristenpart [21] proposed a general purpose DTE *IS-DTE*, for the messages following a simple distribution, such as a uniform distribution.

2.2 Password Vault Schemes

Two HE-based password vault schemes [10, 14] were proposed to resist brute-force attack in the literature. A password vault contains several passwords encrypted under a master

password and hence is a rich target for attackers due to the value of passwords. Using HE, attackers who have stolen an encrypted password vault will get many vaults by enumerating master passwords offline and need to verify the correctness of these vaults online. In contrast to offline guessing, online guessing is more resource-consuming, because it is easily blocked by remote servers with diversities of methods (e.g., login rate limiting [16,32] and malicious login detection [13]). Hence, HE-based password vault schemes have great improvements in security. Moreover, user surveys [11, 24, 30] and empirical experiments on real data [7, 11, 41, 42] showed that users often use weak passwords and reuse passwords on different services and websites. Therefore, designing a PMTE for password vaults needs to characterize the single-password distribution and the similarity between passwords in a vault.

Chatterjee et al. [10] proposed the first HE-based password vault scheme *NoCrack*. They improved a kind of password model—PCFG model [42]—and put forward a sub-grammar approach for the password similarity based on PCFG models. We denote this improved PCFG model as *Chatterjee-PCFG* in this paper. By designing PMTEs for PCFG models and sub-grammars respectively, they presented a PMTE for password vaults. Also, Chatterjee et al. [10] designed PMTEs for another kind of password models—Markov models.

Golla et al. [14] put forward a new PMTE for password vaults. In contrast to Chatterjee et al. [10], Golla et al. used a Markov model [25] for the single-password distribution and a reuse-rate approach for the password similarity. Their PMTE for password vaults combines Chatterjee et al.'s PMTEs for Markov models and IS-DTEs for normal distributions (assuming reuse-rates follow normal distributions). In addition, Golla et al. [14] brought in the concept of adaptive PMTEs (Golla et al. used the word *adaptive NLEs*). By adjusting the Markov model according to the real vault, an adaptive PMTE can generate decoy vaults which are more similar to the real vault. Therefore, these decoy vaults are more difficult to be distinguished from the real one. In this paper, we call adjusted probability models according to the real message *adaptive probability models*, in contrast to *static probability models*.

Chatterjee et al.'s [10] and Golla et al.'s [14] PMTEs for password vaults have the same form named *encode-then-concatenate*. Taking the Chatterjee et al.'s PMTE for a PCFG model as an example, when encoding a password, this PMTE: 1) parses the derivation of the password; 2) encodes each production rule in the derivation to a seed respectively; 3) concatenates these seeds of production rules in order and pads the concatenation to a fix length. Other PMTEs are similar, which encode each character or reuse-rate and then concatenate these seeds.

2.3 Genomic Data Protection Scheme

Genomic data is more sensitive than password vault and needs long-term protection. Once a person's genomic data is compromised, it will affect him during his lifetime and even his relatives, because of the correlation between relatives' genomic data. Huang et al. [18] proposed a genomic data protection scheme called *GenoGuard* based on HE. The genomic data protected by GenoGuard is represented by a sequence of single nucleotide variants (SNVs), which can be viewed as a string of the alphabet $\{0,1,2\}$.

To fit genomic data, Huang et al. [18] evaluated four types of models with chi-square goodness-of-fit tests, including a uniform distribution model, a public LD (linkage disequilibrium) model, three Markov models, and a recombination model. Since the recombination model delivers the best performance, they chose it for GenoGuard. Furthermore, Huang et al. [18] proposed a novel PMTE for these sequences with a different form named *shrink-then-encode*. When encoding, this PMTE shrinks the seed interval for each character in the string according to the probability of the character and randomly picks a seed in the final seed interval.

3 Attacks Against Typical PMTEs

A PMTE is secure (i.e., decoy messages generated by a PMTE are indistinguishable from real ones), if and only if the probability model is accurate for the real message distribution and the PMTE is secure for the probability model. Based on that, we propose a framework to evaluate the security of PMTEs with two types of attacks: 1) *distribution difference attacks* exploiting the difference between the real message distribution and the message probability model (i.e., the decoy message distribution); 2) *encoding attacks* exploiting the difference between the probability model and the PMTE.

3.1 Attacker Model

Attackers that we study in this paper have stolen ciphertext of a message and further want to recover it. Based on the Kerckhoffs's principle, we assume that attackers know the HE algorithm, including DTEs, but do not know the key or any information of the message. It is reasonable because the program shipped to users usually contains the encryption/decryption module. Moreover, this is an essential assumption for an attacker to carry out decrypting. More advanced attackers (e.g., attackers in [10]) may equip themselves with some knowledge about the real message distribution (e.g., the character distribution of messages). However, encoding attackers we employed do not need any information about the real message distribution. Merely relying on the DTEs, such attackers can distinguish real and decoy messages with high accuracy.

To recover the message, attackers: 1) decrypt the ciphertext under N keys $\{k_i\}_{i=1}^{N}$ and get N messages $\{M_i\}_{i=1}^{N}$; 2) choose the most likely message. For some special types of messages which can be verified online, for example, authentication certificates (passwords, password vaults or authentication keys),

Algorithm 1: The attack process to recover a stolen ciphertext.

Input: a stolen ciphertext c, N keys/passwords $\{k_i\}_{i=1}^{N}$ for decryption, and a weight function p.
Output: a guessing list for messages (in decreasing order of p).

```
for i ← 1 to N do
    S_i ← decrypt_{k_i}(c)
    M_i ← decode(S_i)
end
Sort {M_i}_{i=1}^N in decreasing order of p(M_i) (or p(S_i)), then output
the list. /* Different attacks are equipped with
different weight functions p, where p(M_i) usually
reflects the probability that M_i is real.       */
```

attackers need to sort these N messages to minimize the number of online verifications. To characterize attackers in a unified form, we consider an attacker only picking one message also as a sorting attacker who picks the first one in his order. Assuming an attacker sorts the messages in decreasing order of a weight function p, the attack process can be represented as Algorithm 1.

The efficiency of an attacker depends on 1) the guessing order of keys and 2) the sorted order of messages. These two orders correspond to two factors—keys and DTEs, affecting the security of HE schemes. The stronger the keys are, the harder they are to be cracked. Keys used by HE are usually human-memorable passwords. Password researches have attracted great attention recently, such as password guessing [25,41,42], password strength meter [15,39,40], password generation policy [3,34]. However, same as previous literature [10,14], we ignore the influence of keys on the security of HE schemes and only focus on the security of DTEs, i.e., the indistinguishability of decoy messages.

3.2 Analyses of Password Vault PMTEs

Chatterjee et al.'s PMTE [10] for password vaults uses a sub-grammar approach to model the similarity of passwords in one vault. Specifically, the sub-grammar (based on Chatterjee-PCFG) of vault $V = (\text{password}, \text{password1})$ is $\{\texttt{S} \to \texttt{W},\ \texttt{S} \to \texttt{WD},\ \texttt{W} \to \text{password},\ \texttt{D} \to 1\}$, where `W` represents an English word and `D` represents a digit string. In fact, Chatterjee-PCFG is more comprehensive. We simplify it for ease of explanation. To encode a vault, this PMTE 1) first parses the sub-grammar of the vault, 2) then encodes the sub-grammar, and 3) finally encodes the passwords in the vault according to the sub-grammar. Decoding is in the opposite direction.

Because sub-grammars are parsed from the real vaults when encoding, all production rules in sub-grammars are used by passwords in the real vaults. Unfortunately, it may not hold when decoding a random seed. For example, decoding a random seed, the sub-grammar may be $SG = \{\texttt{S} \to \texttt{W},\ \texttt{S} \to \texttt{WD},\ \texttt{W} \to \text{password},\ \texttt{D} \to 1\}$, and the vault may be $V = (\text{password}, \text{password})$. As passwords are generated independently based on sub-grammars when decoding, production rules (e.g., $\texttt{S} \to \texttt{WD}$) in the sub-grammar may not be used by any password in the vault. In addition, decoded sub-grammars may contain identical rules, but encoded ones do not, because the rules are also independently generated when decoding a random seed.

Similar phenomena also appear in Golla et al.'s PMTEs [14]. They used a reuse-rate approach to model password similarity. Given $V = (\text{password1}, \text{password1}, \text{password@})$, Golla et al.'s PMTEs take "password1" as the base password of V and "password@" as a password modified from the base password. When encoding, they 1) encode the base password ("password1") and the reuse-rate of the base password ($\frac{2}{3}$), 2) encode reuse-rates of modified passwords ($\frac{1}{3}$) and the modified characters ("@"). More specifically, Golla et al.'s PMTEs divide the vault into six subsets $\{V_i\}_{i=0}^{5}$: passwords with an edit distance of i to the base passwords V_i ($0 \le i \le 4$) and the remaining passwords V_5. Assuming the proportion (reuse-rate) of V_i in V follow a normal distribution with a small variance, $|V_i|$ (the cardinality of V_i) is encoded by the DTE of the normal distribution, for $0 \le i \le 4$. In addition, the base password, modified characters (of passwords in V_i for $1 \le i \le 4$) and remaining passwords in V_5 are encoded by PMTEs of Markov models.

The sum of $|V_i|$ for $0 \le i \le 4$ (without $|V_5|$) is less than or equal to $|V|$ when encoding. However, it may not hold when decoding a random seed, because proportions of V_i are generated independently. Further, the modified character of password pw in V_i may be the same as the original character of the base password when decoding a random seed, which means pw actually belongs to V_j with $j < i$. But this is not possible when encoding a real vault.

3.3 Attacks Against Password Vault PMTEs

In the above analyses of password vault PMTEs, we dig out some features that real seeds (encoding from real vaults) must have but decoy seeds (random seeds) may not have. Therefore, an attacker is able to exclude some decoy seeds if they do not have these features. Let p_F denote the weight function based on the feature F:

$$p_F(S) = \begin{cases} 1, & \text{if the seed } S \text{ has feature } F, \\ 0, & \text{otherwise.} \end{cases}$$

We now present four features for exploration, the first two features for the Chatterjee et al.'s PMTE [10] and the last two features for Golla et al.'s PMTEs [14]:

1. Feature UR (unused rule): there is no unused rule in the sub-grammar decoded from the seed.
2. Feature DR (duplicate rule): there is no duplicate rule in the sub-grammar decoded from the seed.
3. Feature ED (edit distance): every password in the vault has the same value of i as the one decoded from the seed.

Algorithm 2: The weight function $p_{\text{PCA+SVM}}$ of the PCA+SVM attack

training:
  Input: a dataset *snvsList* containing the same number of real and decoy SNV sequences with the label (0 for decoy and 1 for real) list *labelList*.
  Output: a PCA model *pca* and a SVM model *svm*.
  `/* The classes SVC and PCA we use are svm.SVC and decomposition.PCA in Scikit-learn, a machine learning library for Python. */`
  $pca \leftarrow PCA(n_components = 10)$ `/* We use the default parameters except n_components as 10. */`
  $pca.fit(snvsList)$
  $reducedSNVsList \leftarrow pca.transform(snvsList)$
  $svm \leftarrow SVC(probability = True)$
  $svm.fit(reducedSNVsList, labelList)$
end
function $p_{\text{PCA+SVM}}(s)$
  Input: an SNV sequence s.
  Output: the SVM-estimated probability that s is real.
  $reducedSNVs \leftarrow pca.transform([s])[0]$
  $p \leftarrow svm.predict_proba([reducedSNVs])[0,1]$
  return p
end

4. Feature PN (password number): the sum of $|V_i|$ $(0 \le i \le 4)$ is no larger than V.

To evaluate the security of PMTEs, Chatterjee et al. [10] used a Support Vector Machine (SVM) to distinguish the real and decoy vaults, and Golla et al. [14] used Kullback-Leibler (KL) divergence. These attacks only exploit the difference between the real and decoy vault distributions but neglect the seeds. We call this type of attack *distribution difference attack*. These attacks cannot exploit the features discussed above. In contrast, our proposed feature attacks only exploit seeds with PMTEs and do not require any knowledge of the real vault distribution. We call this new type of attack *encoding attack*.

3.4 Attacks Against Genomic Data PMTEs

Huang et al. [18] provided a formal proof for the security of their PMTEs. They proved that their PMTEs are indistinguishable from probability models, but did not consider the difference between the real message distribution and probability models. This means their PMTEs resist encoding attacks but have not been evaluated by distribution difference attacks. Although Huang et al. evaluated six probability models with chi-square goodness-of-fit tests, they did not study the influence of their goodness on the security of PMTEs.

In order to evaluate the security, we propose a simple machine learning algorithm to distinguish the real and decoy data (i.e., SNV sequences). As shown in Algorithm 2, we use a training set to train a principal component analysis (PCA) model and a support vector machine (SVM) with a radial basis function (RBF) kernel, where the training set contains the same number of real and decoy SNV sequences, the real sequences are randomly picked from the real dataset, and the decoy sequences are generated by decoding random seeds with the corresponding PMTEs. Specifically, the PCA model is trained and used to reduce the 1000-dimensional sequences in training set to 10 dimensions, and the SVM is trained with the 10-dimensional sequences and the "real/decoy" labels. To estimate the probability that a test sequence s is real, we first use the trained PCA model to reduce s to 10 dimensions, then resort to the trained SVM to classify the reduced sequence and output the probability of it being real. All parameters of the PCA and the SVM are default except "n_components" as 10. Since the default parameters deliver good performance, we do not adjust them. We denote the SVM-estimated probability of s as $p_{\text{PCA+SVM}}(s)$ and propose a PCA+SVM attack with the weight function $p_{\text{PCA+SVM}}$.

4 Generative Probability Models and Generic Encoding Attacks

In this section, we propose a generic conceptual probability model—*Generative Probability Model (GPM)*—to formalize all the existing probability models. This formalization uncovers the principle of encoding attacks. Based on this principle, we propose two generic encoding attacks—a *weak encoding attack* and a *strong encoding attack*.

4.1 Definition

Simple models (e.g., uniform distribution models) assign every message a probability directly, but other complex models cannot. Most complex models (e.g., PCFG models [42]) design a generative method for messages and assign every message a probability with the generated probability of the message. By assigning probabilities to the generating rules, one can get a probability model for the messages. From this point of view, we give a formal definition of *Generative Probability Model*.

Definition 1. A Generative Probability Model (GPM) *is a 5-tuple* $(\mathcal{M}, \mathcal{R}, \mathcal{RS}, G, P)$*, where* $\mathcal{M}$ *is the message space,* $\mathcal{R}$ *is the set of generating rules,* $\mathcal{RS} \subset \mathcal{R}^*$ *is the set of valid generating sequences, G is the generating function mapping a generating sequence RS in* $\mathcal{RS}$ *to a message M in* $\mathcal{M}$*, and P is the probability density function on* $\mathcal{RS}$*. Here* $\mathcal{M}, \mathcal{R}, \mathcal{RS}$ *are finite sets and G is surjective. Then the GPM gives* $\mathcal{M}$ *a probability distribution by*

$$P(M) = \sum_{RS \in G^{-1}(M)} P(RS). \tag{1}$$

In addition, if G is bijective (i.e., for every message in $\mathcal{M}$*, there is only one generating sequence which can generate it), the GPM is* unambiguous*, and otherwise, it is* ambiguous.

Usually, the probability density function P on $\mathcal{RS}$ is given by the conditional probability distribution as follows:

$$P(RS) = \prod_{i=1}^{n} P(r_i | r_1 r_2 \dots r_{i-1}), \tag{2}$$

where $RS = (r_1, r_2, \dots, r_n)$. The conditional probability $P(r_i | r_1 r_2 \dots r_{i-1})$ is usually given in a simple form. Note that the generating sequences in $\mathcal{RS}$ have variable lengths, therefore, the above equation requires that $\mathcal{RS}$ is *prefix-free*, i.e., no sequence in $\mathcal{RS}$ is a prefix of another sequence. Otherwise, the function P defined by Equation 2 is not a probability density function on $\mathcal{RS}$, because $\sum_{RS \in \mathcal{RS}} P(RS) > 1$. Fortunately, if $\mathcal{RS}$ is not prefix-free, it can easily be converted to a prefix-free sequence space $\mathcal{RS}'$ by two simple methods: 1) add a special rule at the beginning of the sequence to represent the length of the sequence; 2) add a special rule at the end of the sequence to represent the end of the sequence. Therefore, without loss of generality, we assume generating sequence spaces of GPMs are all prefix-free.

4.2 Formalization of Existing Models

For a Markov model of order n, a generating rule is a character, and a valid generating sequence is a string. The conditional probability of a rule only depends on last n rules, formally

$$P(a_i | a_1 a_2 \dots a_{i-1}) = P(a_i | a_{i-n} a_{i-n+1} \dots a_{i-1}),$$

where $i > n$ and $P(a_i | a_{i-n} a_{i-n+1} \dots a_{i-1})$ is trained on a training set (RockYou for password vault schemes). The Markov model with distribution-based normalization adds some extra rules $\{L = l\}_{l=1}^{l_{\max}}$ to $\mathcal{R}$, $L = l$ represents that the password length is equal to l, where $1 \le l \le l_{\max}$ and $l_{\max}$ is the max password length (e.g., 30). A valid generating sequence has the form $(L = l, a_1, a_2, \dots, a_l)$ which means generating the length first and then generating the characters. $P(L = l, a_1, a_2, \dots, a_l) = P(L = l) P(a_1, a_2, \dots, a_l)$, where $P(a_1, a_2, \dots, a_l)$ can be calculated as the ordinary Markov model and $P(L = l)$ represents the probability that the length of a password is l. Note that $l_{\max} < \infty$, because the message space $\mathcal{M}$ is finite (the seed space $\mathcal{S}$ is finite).

For a PCFG model, a generating rule is a production rule of the PCFG, a valid generating sequence is a leftmost derivation of a string. The conditional probability of a rule does not depend on any previous rule, formally

$$P(r_i | r_1 r_2 \dots r_{i-1}) = P(r_i),$$

where $P(r_i)$ is also trained on a training set.

For the Golla et al.'s model [14] of password vaults, a generating rule is a character or a value of $|V_i|$ for $0 \le i \le 4$. A valid generating sequence of a vault consists of the following rules: 1) characters of the base password, 2) $|V_i|$, 3) modified characters of passwords in V_i and 4) characters of passwords in V_5. In this case, the conditional probabilities of characters are calculated as the Markov model and $|V_i|$ is calculated by normal distributions.

For the Chatterjee et al.'s model [10] of password vaults, a generating rule is a production rule of the PCFG or a number of production rules with a certain lefthand-side in a vault, a valid generating sequence contains a generating sequence of a sub-grammar and leftmost derivations of passwords based on the sub-grammar. More specifically, a valid generating sequence of the sub-grammar $\{\mathtt{S} \to \mathtt{D}, \mathtt{S} \to \mathtt{W}, \mathtt{D} \to 123456, \mathtt{W} \to \text{password}\}$ is $(\#\mathtt{S} = 2, \mathtt{S} \to \mathtt{D}, \mathtt{S} \to \mathtt{W}, \#\mathtt{D} = 1, \mathtt{D} \to 123456, \#\mathtt{W} = 1, \mathtt{W} \to \text{password})$. The rule $\#\mathtt{X} = i$ represents that there are i rules with the lefthand-side $\mathtt{X}$ in sub-grammar, it is for the sake of the prefix-free property of $\mathcal{RS}$. The conditional probability of the rule $\#\mathtt{X} = i$ only depends on the rule itself, denoted as $P(\#\mathtt{X} = i)$, which is trained on a password vault dataset (Pastebin). The conditional probability of the rule $\mathtt{X} \to str$ is the same as that in PCFG models.

For Huang et al.'s models [18] for genomic data, a generating rule is a character of $\{0, 1, 2\}$ (representing an SNV), a valid generating sequence is a string. The conditional probability of a rule relies on the genomic data model: for the uniform distribution model, it is equal to $\frac{1}{3}$ for each rule; for the public LD model (as discussed above), it depends on the last rule; for Markov model of order n, it depends on the last n rules; for the recombination model, it is calculated by the forward-backward algorithm with a hidden Markov model.

Up to this point, the existing models are all formalized with our proposed GPMs and the distributions of generating sequences are defined by the conditional distributions of generating rules. Beyond that, more probability models can be formalized. For example, neural networks for passwords [27] can be formalized as the same as Markov models except that condition probabilities are calculated by neural networks.

4.3 Generating Graphs

To represent a GPM visually, we propose a *generating graph*, which is a connected directed acyclic graph with a single source and with edges labeled by generating rules. In a generating graph, a generating sequence is illustrated by a path whose edges denote the corresponding generating rules in order. Moreover, a message is figured by a sink (because the generating sequence space is prefix-free) and a path from the source to the sink illustrates one generating sequence of the message. Hence, the path is called a *generating path* of the message. Note that the generating graph of a model is an arborescence, if and only if the model is unambiguous. (Note an arborescence is a directed graph in which there is only one single source and each other vertex has only one directed path from the source.)

As shown in Figure 2, in Chatterjee-PCFG model, there are two generating paths for "password". These two generating paths correspond to two generating sequences: $\{\mathtt{S} \to \mathtt{W}, \mathtt{W} \to$

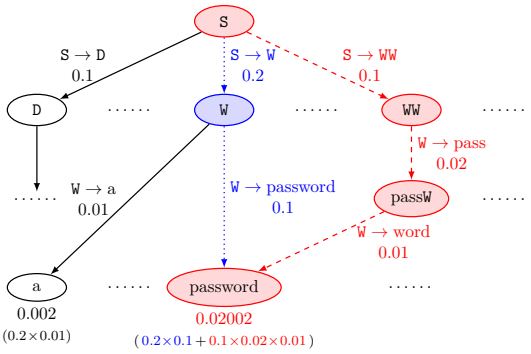


Figure 2: Generating graph of Chatterjee-PCFG

Algorithm 3: The weight function p_{WEA} ($= p_{\text{EC}}$) of the weak encoding attack

```
function p_WEA(S)
    Input: a seed S.
    Output: the weight of S (for sorting in Algorithm 1).
    Obtain the generating sequence RS and the message M of S by
      decoding S
    S' ← encode(M) /* Since encode is a randomized
       algorithm, S' is probably not equal to S.    */
    Obtain the generating sequence RS' of S' by decoding S'
    if RS = RS' then return 1 /* S may be a real seed. */
    else return 0    /* S is definitely a decoy seed. */
end
```

password} and {S → WW, W → pass, W → word}. Further, the probability of the first sequence is $0.2 \times 0.1 = 0.02$ and that of the second one is $0.1 \times 0.02 \times 0.01 = 0.00002$. This makes the probability of "password" be $0.02 + 0.00002 = 0.02002$. Since "password" has two generating sequences, Chatterjee-PCFG model is ambiguous.

4.4 The Principle of Encoding Attacks

The features used by encoding attacks in Section 3 are all based on heuristic analyses of specific PMTEs. Some other features are still neglected due to the lack of a systematic analysis. For example, on Chatterjee et al.'s password vault PMTE [10], the order of rules in the sub-grammar is deterministic for real vaults, but not for decoy seeds. When encoding the vault $V = (123456, \text{password})$, the first two rules in the sub-grammar are S → D, S → W in order. But if the vault V is decoded by a decoy seed, the first two rules may be S → W, S → D in a different order from the real vault.

Fortunately, with the formalizations by GPMs and the visual representations by generating graphs, the principle of encoding attacks is uncovered: *existing PMTEs neglect the ambiguity of GPMs. More specifically, in an ambiguous GPM, there may exist multiple generating paths for a message, but the existing PMTEs only select one deterministic path when encoding.* We name these paths *encoding paths* which can be selected when encoding and meanwhile name these corresponding generating sequences *encoding sequences*. The generating sequence of a seed can be obtained by decoding the seed. Due to the determinacy of encoding paths, encoding attacks can exclude some decoy seeds by checking if the generating path of a seed is an encoding path, without any information of the real message distribution.

We then take Chatterjee et al.'s PMTE [10] for Chatterjee-PCFG as an example. As shown in Figure 2, this PMTE only uses the blue dotted path when encoding "password", but the generating path of a decoy seed may be the red dashed one. In fact, Chatterjee et al. [10] noticed the ambiguity of Chatterjee-PCFG and briefly mentioned that the PMTE needs to choose one parse tree randomly in all parse trees when encoding. However, in Chatterjee et al.'s code [10], they have not implemented the random selection method until now (June 1, 2019) and only one parse tree is selected when encoding. Moreover, Chatterjee et al. [10] completely neglected the ambiguity of the sub-grammar approach. For example, a vault $V = (123456, \text{password})$ is encoded only with the sub-grammar $SG = \{$S → D, S → W, D → 123456, W → password$\}$, but V can be generated by multiple sub-grammars as long as they contain SG. Therefore, the encoding paths definitely have feature UR while other generating paths may not.

Similarly, Golla et al. [14] also did not consider the ambiguity of the reuse-rate approach. For example, $V = (\text{password1}, \text{password1}, \text{password@})$ can be generated by "password1" as the base password with reuse-rates $|V_0| = \frac{2}{3}$ and $|V_1| = \frac{1}{3}$. It also can be generated by "password1" as the base password with reuse-rates $|V_0| = \frac{1}{3}$ and $|V_1| = \frac{2}{3}$. In addition, Golla et al.'s GPMs [14] allow modifying the character of the base password to the same character. Therefore, "password@" may be in V_2 (with "@" modified from "1" and "d" modified from itself). This brings ambiguity to the GPM, i.e., a huge number of generating paths for a vault. Only one deterministic path (the first one for V) is chosen when encoding. Therefore, the encoding paths definitely have feature ED while other generating paths may not.

Any feature utilized by any encoding attack, including features proposed in Section 3.3, the rule-order feature or the base-password feature discussed above, can be seen as a feature of encoding paths.

4.5 Generic Encoding Attacks

Due to the determinacy of encoding paths, we further propose two generic encoding attacks—a *weak encoding attack* and a *strong encoding attack*.

The weak encoding attack is accordance with feature EC (encoding consistency) that the generating path is an encoding path, i.e., the weight function $p_{\text{WEA}} = p_{\text{EC}}$. We use the abbreviation of the attack as the subscript of p for convenience. More specifically, p_{WEA} (i.e., whether a seed S has feature

EC) can be calculated as Algorithm 3.

In contrast to the feature attacks (proposed in Section 3.3) based on some features of encoding path, the weak encoding attack is based on feature EC. Therefore, the seeds having feature EC certainly have other features proposed in Section 3.3. In other words, the weak encoding attack excludes all decoy vaults which are excluded by any feature attack.

As the seeds with feature EC are sorted randomly by the weak encoding attack, we propose a *strong encoding attack* to sort them. Let RS denote the generating sequence of the seed S, then the weight function p_{SEA} is defined as

$$p_{\mathrm{SEA}}(S) = \frac{1}{P(RS)} \times p_{\mathrm{WEA}}(S).$$

4.6 Efficiency of Encoding Attacks

These two generic encoding attacks are efficient for PMTEs with significantly ambiguous GPMs and deterministic encoding paths, such as all existing PMTEs for password vaults. In other words, these attacks recover the encrypted real vaults with a high probability but a small number of online verifications. To make it clear, the weak encoding attack excludes the seeds whose generating paths are not encoding paths, e.g., the red dashed path in Figure 2. Namely, the excluded proportion of the weak encoding attack is equal to the total probability of all generating paths except encoding paths. This means that *the more ambiguous the GPM is, the more efficiency the weak encoding attack can achieve*. As discussed in Section 4.4, in the existing GPMs for password vaults [10, 14], every vault has countless generating paths. Due to the great ambiguity of these GPMs, the weak encoding attack is efficient for the corresponding existing PMTEs with deterministic encoding paths. On the other hand, if a GPM is unambiguous (e.g., the models of genomic data [18]), the PMTE for it can resist encoding attacks naturally. Besides, the strong encoding attack excludes all decoy seeds which are excluded by the weak encoding attacks. Therefore, the strong encoding attack is always more efficient than the weak encoding attack.

5 Probability Model Transforming Encoders

We propose a *generic transforming method* which transforms an arbitrary GPM to a secure PMTE. Further, we give a formal proof that the PMTE transformed by our method is indistinguishable from the GPM.

5.1 Conditional DTEs

Inspired by the way Chatterjee et al.'s PMTEs [10] encoding password character by character or rule by rule, we propose a fundamental concept of PMTE—*conditional distribution transforming encoder (CDTE)*—to encode message rule by rule. A DTE is an encoder transformed from a probability distribution, while a CDTE is an encoder transformed from a conditional probability distribution. Unlike a DTE, a CDTE needs not only the message M but also the condition X to encode M (denoted as $\mathsf{encode}(M|X)$) by the conditional probability distribution $P(\cdot|X)$. It also needs the condition X to decode the seed S (denoted as $\mathsf{decode}(S|X)$). In this aspect, for every condition X, the CDTE $(\mathsf{encode}(\cdot|X), \mathsf{decode}(\cdot|X))$ is a DTE. Interestingly, if the condition X and the message M are mutually independent (i.e., the conditional probability distribution $P(\cdot|X)$ is the same for every condition X), a CDTE degenerates into a DTE. Therefore, we state that DTEs can be seen as a special case of CDTEs. Juels and Ristenpart [21] proposed a generic method to transform a distribution to a DTE and named the DTE *IS-DTE*. For the general conditional distribution, we get a DTE $\mathbf{IS\text{-}DTE}_X$ for each condition X by means of Juels-Ristenpart method and thus we give a general CDTE scheme *IS-CDTE* by the combination $\{\mathbf{IS\text{-}DTE}_X\}_X$.

In the following, we give the details of our IS-CDTE. Let X denote the condition, $\mathcal{X}$ denote the condition space, and $\mathcal{M}_X = \{M_i\}_i$ denote the message space under the condition X. The corresponding conditional probability is $P(M_i|X)$, and the cumulative distribution function is $F_i = \sum_{i'=1}^{i} P(M_{i'}|X)$. When encoding the message M under the condition X, the IS-CDTE randomly generates a real number S in the interval $[F_{i-1}, F_i)$ as a seed of M. When decoding the seed S under condition X, the IS-CDTE searches the interval $[F_{i-1}, F_i)$ containing S and then outputs the corresponding message M_i. Encoding or decoding only requires a binary search of the corresponding CDF (cumulative distribution function) table $\{(M_i, F_i)\}_i$ under the condition. Therefore, the space complexity and the time complexity of the IS-CDTE are $O(|\mathcal{X}| \cdot |\mathcal{M}|)$ and $O(\log(|\mathcal{M}|))$, respectively.

For implementing with encryption, real-number seeds are usually represented as bit strings of length l, i.e., integers in $[0, 2^l)$, where l is a storage overhead parameter. IS-DTEs use the function $\mathsf{round}_l(x)$ converting a real-number seed to an integer seed, where $\mathsf{round}_l(x) = \mathsf{round}(2^l x)$ and round represents rounding function. We use the same method for IS-CDTEs. In such case, the integer seed interval of M_i is $[\mathsf{round}(2^l F_{i-1}), \mathsf{round}(2^l F_i))$. Hence, to ensure that each message has at least one integer seed, l must be greater than or equal to $-\log_2(\min_i P(M_i|X))$. The loss of precision by the discretization with round_l causes a slight difference between these two conditional distributions $\Pr_{\mathbf{IS\text{-}CDTE}}(M|X) = \Pr[M = M' : S \leftarrow_{\$} \mathcal{S}; M' \leftarrow \mathsf{decode}(S|X)]$ and $P(M|X)$, where $\mathbf{IS\text{-}CDTE} = (\mathsf{encode}(\cdot|\cdot), \mathsf{decode}(\cdot|\cdot))$. Fortunately, the difference is negligible in l (see Theorem 4). For convenience, we let $P^{(d)}$ denote the discretization $\Pr_{\mathbf{IS\text{-}CDTE}}$ of P.

5.2 Probability Model Transforming Encoder

Combining IS-CDTEs for the conditional distributions of generating rules, we present a PMTE for the messages, which we call an *IS-PMTE*. Let l denote the storage overhead parameter,

then the IS-PMTE encodes the message M as follows:

1. Parse M and get all generating sequences $G^{-1}(M)$.
2. Calculate the probability $P^{(d)}(RS)$ for each generating sequence RS in $G^{-1}(M)$, where $P^{(d)}(r_1 r_2 \dots r_n) = \prod_{i=1}^{n} P^{(d)}(r_i | r_1 r_2 \dots r_{i-1})$ and $P^{(d)}(r_i | r_1 r_2 \dots r_{i-1})$ is the discretization of $P(r_i | r_1 r_2 \dots r_{i-1})$.
3. Choose a generating sequence RS in $G^{-1}(M)$ with the probability $P^{(d)}(RS|M)$, where $P^{(d)}(RS|M) = \frac{P^{(d)}(RS)}{\sum_{RS' \in G^{-1}(M)} P^{(d)}(RS')}$.
4. Encode each rule r_i in $RS = (r_i)_i$ by the IS-CDTE $\mathsf{encode}(\cdot | r_1 r_2 \dots r_{i-1})$ to a l-bit string S_i.
5. Concatenate $(S_i)_i$, pad the concatenation to a string S of length $ln_{\max}$ with random bits and then output S as a seed for M, where $n_{\max}$ is the maximum length of generating sequences in $\mathcal{RS}$ (i.e., the depth of the generating graph).

In opposite, the IS-PMTE decodes the seed S as follows:

1. Split S into $n_{\max}$ l-bit strings $(S_i)_{i=1}^{n_{\max}}$.
2. Decode S_i to the rule r_i by $\mathsf{decode}(\cdot | r_1 r_2 \dots r_{i-1})$ in turn and ignore the padding bits.
3. Generate the message M from the generating sequence $RS = (r_i)_{i=1}^{n}$ by $M = G(RS)$, then output M as the message of S.

Note that generating sequences vary in length. Because seeds in $\mathcal{S}$ are of fixed length, padding is necessary for some sequences when encoding. Furthermore, as the sequence space $\mathcal{RS}$ is prefix-free, padding bits can be ignored unambiguously when decoding. In addition, note that in Step 2) of encoding the probabilities of sequences are calculated as the discretization $P^{(d)}$ of P, which is necessary to guarantee the uniformity of seeds (see Theorem 3).

Due to the generality of GPMs, IS-PMTEs not only apply to probability models discussed in this paper, but also apply to general probability models, such as neural networks for passwords [27].

Figure 3 depicts how "password" is encoded by our IS-PMTE for the Chatterjee-PCFG model. First, parse all generating sequences of "password". Corresponding to Figure 2, "password" has two generating sequences $\{\mathtt{S} \to \mathtt{W}, \mathtt{W} \to \text{password}\}$ and $\{\mathtt{S} \to \mathtt{WW}, \mathtt{W} \to \text{pass}, \mathtt{W} \to \text{word}\}$. Second, choose a sequence with the probability ($0.02/0.02002 \approx 0.999$ for the first one and 0.001 for the second one). Here we take the second one as an example. Third, encode each generating rule in the sequence by searching the CDF table and translate real-number seeds to bit-string seeds with round_l. Note that in the PCFG models, the conditional probabilities of generating rules do not depend on the previous rules and the rules with the same lefthand-side have the same CDF table. Therefore, the same CDF table is searched for generating rules $\mathtt{W} \to$ pass and $\mathtt{W} \to$ word. Finally, concatenate seeds of rules, pad the concatenation to a fixed length with random bits and get a seed for "password".

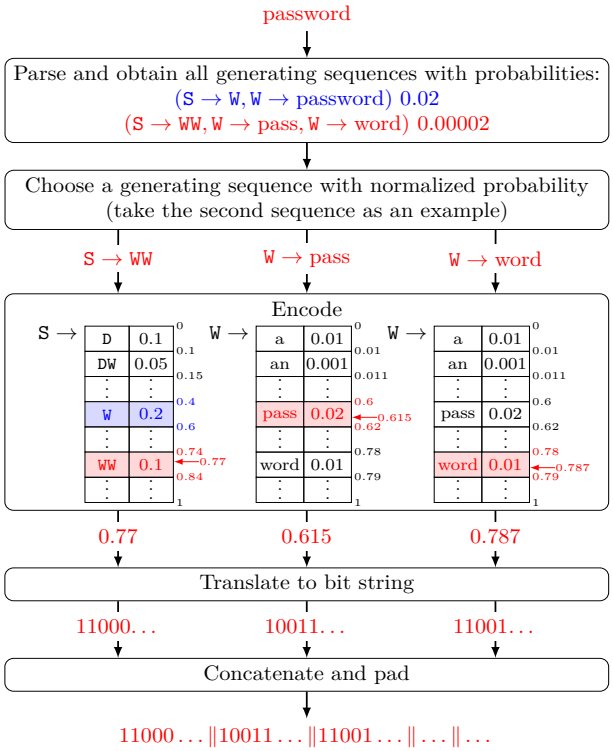


Figure 3: Encode "password" by our IS-PMTE for the Chatterjee-PCFG model

5.3 Difference Between IS-PMTEs and Existing PMTEs

It is easy to get IS-PMTEs from existing GPMs of password vaults and genomic data by our proposed generic transforming method. The following are the differences between the IS-PMTEs and the existing PMTEs [10, 14] for password vaults:

1. IS-PMTEs randomly choose a generating sequence, while the existing PMTEs only choose a deterministic generating sequence. This is the key to resist encoding attacks. Note that the random selection may have high time complexity, fortunately there is a method to reduce it. We leave the details in Appendix C.
2. IS-PMTEs use $\mathsf{round}_l(x)$ to convert a real-number seed to an integer seed, while Chatterjee et al. [10] designed another method to convert a rational-number seed to an integer seed. Unfortunately, Chatterjee et al.'s method cannot be applied to some distributions, e.g., normal distribution. This is because probabilities may be irrational numbers. The method we use (proposed by Juels and Ristenpart [21]) is applicable to arbitrary distributions.

In addition, IS-PMTEs have the same form as the existing PMTEs for password vaults, which is *encode-then-concatenate*. At the same time, the existing PMTEs [18] for genomic data use another *shrink-then-encode* form. When

encoding a string, these genomic data PMTEs shrink the seed interval for each character in the string and further pick a random seed in the final seed interval as the seed for the string. Unfortunately, each interval-shrinking needs to complete large integer arithmetic of length ln to calculate the interval boundary, where l is the storage overhead parameter, and n is the length of the string. This arithmetic costs $\Omega(ln)$ time for each character and $\Omega(ln^2)$ time for the string. In contrast, our IS-PMTEs only need to do integer arithmetic of length l for each character with lower time complexity $\Theta(ln)$ for a string.

5.4 Security of IS-PMTEs

The weak and strong encoding attacks have more generic forms for the PMTEs such as IS-PMTEs who may randomly choose a generating path when encoding. If the PMTE chooses a deterministic generating path when encoding, these generic forms will degenerate to the given forms in Section 4.5. For the weak encoding attack, the more generic form of feature EC is

$$S \in \mathsf{encode}(\mathsf{decode}(S)),$$

where $\mathsf{encode}(M)$ represents all encoded seeds from M. If the seed S does not have feature EC, then S can be decoded to the message $M = \mathsf{decode}(S)$ but cannot be encoded from the message M. Therefore, S is a decoy seed. In order to resist weak encoding attack, it is necessary to ensure that $\mathsf{encode}(M) = \mathsf{decode}^{-1}(M)$ for every message $M \in \mathcal{M}$, where $\mathsf{decode}^{-1}(M)$ represents all seeds which can be decoded into M. In PMTEs with deterministic encoding paths, the generating paths for all seeds in $\mathsf{encode}(M)$ are the same one. In this case, the weak encoding attack degenerates to the given form in Section 4.5.

For the strong encoding attack, the more generic form of the weight function is

$$\Pr_{\mathsf{encode}}(S|\mathsf{decode}(S)),$$

where $\Pr_{\mathsf{encode}}(S|M)$ represents the probability that M is encoded as S under the condition of message M. We denote it as $p_{\mathrm{GSEA}}(S)$. In order to resist strong encoding attack, it is necessary to ensure that $\Pr_{\mathsf{encode}}(S|M)$ are equal for every $S \in \mathsf{decode}^{-1}(S)$, i.e., all valid seeds are uniformly chosen when encoding. We call this property *seed uniformity*. Further, if a DTE has this property, attackers cannot get any useful information except the message from a seed (see Theorem 2). This well explains why our IS-PMTEs choose a generating sequence RS in $G^{-1}(M)$ with the probability $P^{(d)}(RS|M)$ when encoding—it precisely guarantees that seeds are uniform (see Theorem 3). In addition, for PMTEs with deterministic encoding path, the strong encoding attack degenerates to the form in Section 4.5, because $p_{\mathrm{GSEA}} \propto p_{\mathrm{SEA}}$. Let M denote the message, $RS = (r_i)_i$ denote the deterministic generating sequence of M, S denote the seed of M, then we have: 1) if $S \in \mathsf{encode}(M)$, $p_{\mathrm{GSEA}}(S) = \frac{1}{|\mathsf{encode}(M)|} = \frac{1}{|\mathsf{encode}(RS)|} = \frac{1}{|\mathcal{S}|P(RS)} = \frac{1}{|\mathcal{S}|}p_{\mathrm{SEA}}(S)$; 2) otherwise, $p_{\mathrm{GSEA}}(S) = 0 = p_{\mathrm{SEA}}(S)$.

In the following, we prove the security of IS-PMTEs, i.e., decoy seeds/messages are indistinguishable from real ones by any adversary. Let $\mathcal{M}$ denote the message space, $\Pr_{\mathrm{real}}$ denote the probability density function of real messages, $\mathcal{S}$ denote the seed space, and $\mathbf{DTE} = (\mathsf{encode}, \mathsf{decode})$ denote the DTE. Juels and Ristenpart [21] used the advantage of an attacker $\mathcal{A}$ who distinguishes between the real and decoy message-seed pairs to evaluate the security of a DTE, where the advantage is $\mathsf{Adv}^{\mathrm{dte}}_{\mathbf{DTE},\mathrm{real}}(\mathcal{A}) = |\Pr[\mathcal{A}(S,M) = 1 : M \leftarrow_{\Pr_{\mathrm{real}}} \mathcal{M};\ S \leftarrow_{\$} \mathsf{encode}(M)] - \Pr[\mathcal{A}(S,M) = 1 : S \leftarrow_{\$} \mathcal{S};\ M \leftarrow \mathsf{decode}(S)]|$. This advantage can be simplified, if **DTE** has some properties. *Correctness* is the most basic property of a DTE, which means seeds encoded from the message M can be decoded to M correctly for every message M, i.e., $\mathsf{encode}(M) \subseteq \mathsf{decode}^{-1}(M)$ for every $M \in \mathcal{M}$. If **DTE** is correct, attackers can get the message M from the seed S. Therefore, $\mathsf{Adv}^{\mathrm{dte}}_{\mathbf{DTE},\mathrm{real}}(\mathcal{A})$ can be simplified to the advantage of attacker $\mathcal{B}$, who distinguishes between the real and decoy seeds, where the advantage is $\mathsf{Adv}^{\mathrm{dte},\mathcal{S}}_{\mathbf{DTE},\mathrm{real}}(\mathcal{B}) = |\Pr[\mathcal{B}(S) = 1 : M \leftarrow_{\Pr_{\mathrm{real}}} \mathcal{M};\ S \leftarrow_{\$} \mathsf{encode}(M)] - \Pr[\mathcal{B}(S) = 1 : S \leftarrow_{\$} \mathcal{S}]|$ (see Theorem 1). Moreover, if **DTE** is correct and seed-uniform, $\mathsf{Adv}^{\mathrm{dte}}_{\mathbf{DTE},\mathrm{real}}(\mathcal{A})$ can be further simplified to the advantage of an attacker $\mathcal{B}$, who distinguishes between the real and decoy messages, where the advantage is $\mathsf{Adv}^{\mathrm{dte},\mathcal{M}}_{\mathbf{DTE},\mathrm{real}}(\mathcal{B}) = |\Pr[\mathcal{B}(M) = 1 : M \leftarrow_{\Pr_{\mathrm{real}}} \mathcal{M}] - \Pr[\mathcal{B}(M) = 1 : S \leftarrow_{\$} \mathcal{S};\ M \leftarrow \mathsf{decode}(S)]|$ (see Theorem 2). The proof details are given in Appendix A.

Theorem 1. *If **DTE** is correct, then for any attacker $\mathcal{A}$, who distinguishes between the real and decoy message-seed pairs, there exists an attacker $\mathcal{B}$ (as follows), who distinguishes between the real and decoy seeds with* $\mathsf{Adv}^{\mathrm{dte},\mathcal{S}}_{\boldsymbol{DTE},\mathrm{real}}(\mathcal{B}) = \mathsf{Adv}^{\mathrm{dte}}_{\boldsymbol{DTE},\mathrm{real}}(\mathcal{A})$.

$\mathcal{B}(S)$
$M \leftarrow \mathsf{decode}(S)$
return $\mathcal{A}(S,M)$

Theorem 2. *If **DTE** is correct and seed-uniform, for any attacker $\mathcal{A}$, who distinguishes between the real and decoy seeds, there exists an attacker $\mathcal{B}$ (as follows), who distinguishes between the real and decoy messages with* $\mathsf{Adv}^{\mathrm{dte},\mathcal{M}}_{\boldsymbol{DTE},\mathrm{real}}(\mathcal{B}) = \mathsf{Adv}^{\mathrm{dte},\mathcal{S}}_{\boldsymbol{DTE},\mathrm{real}}(\mathcal{A})$.

$\mathcal{B}(M)$
$S \leftarrow_{\$} \mathsf{encode}(M)$
return $\mathcal{A}(S)$

Our proposed IS-PMTEs have the above two properties, thus we neglect the difference between these three types of attackers. Let **GPM** denote the GPM and **IS-PMTE** denote the IS-PMTE of **GPM**. The message generated by **IS-PMTE** (decoding random seed) is indistinguishable from the message generated by **GPM**. Formally, the advantage $\max_{\mathcal{A}} \mathsf{Adv}^{\mathrm{gpm}}_{\mathbf{IS\text{-}PMTE},\mathbf{GPM}}(\mathcal{A})$ is negligible in l (Theorem 5), where $\mathsf{Adv}^{\mathrm{gpm}}_{\mathbf{IS\text{-}PMTE},\mathbf{GPM}}(\mathcal{A}) = |\Pr[\mathcal{A}(M) = 1 : M \leftarrow_{\Pr_{\mathbf{IS\text{-}PMTE}}} \mathcal{M}] - \Pr[\mathcal{A}(M) = 1 : M \leftarrow_{\Pr_{\mathbf{GPM}}} \mathcal{M}]|$, $\Pr_{\mathbf{GPM}}$ is the probability density function P of **GPM** and $\Pr_{\mathbf{IS\text{-}PMTE}}(M) = P^{(d)}(M) = \Pr[M = M' : S \leftarrow_{\$} \mathcal{S}; M' \leftarrow \mathsf{decode}(S)]$. This means that we design a secure PMTE for a GPM. In addition, $\mathsf{Adv}^{\mathrm{dte}}_{\mathbf{IS\text{-}PMTE},\mathrm{real}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathrm{gpm}}_{\mathbf{IS\text{-}PMTE},\mathbf{GPM}}(\mathcal{A}) + \mathsf{Adv}^{\mathrm{gpm}}_{\mathbf{GPM},\mathrm{real}}(\mathcal{A})$. If **GPM** is an accurate probability model for real messages, i.e., $\mathsf{Adv}^{\mathrm{gpm}}_{\mathbf{GPM},\mathrm{real}}(\mathcal{A})$ is negligible, then $\mathsf{Adv}^{\mathrm{dte}}_{\mathbf{IS\text{-}PMTE},\mathrm{real}}$ is negligible, i.e., **IS-PMTE** is secure for the real message distribution.

Theorem 3. ***IS-PMTE*** *is correct and seed-uniform.*

Theorem 4. ***IS-CDTE*** *is transformed from the conditional probability* $\Pr_{\mathrm{real}}(M|X)$*, the seed length is* l *and* $m = |\mathcal{M}|$*. Then for any condition* X *and any distinguishing attacker* $\mathcal{A}$*,* $\mathsf{Adv}^{\mathrm{dte}}_{\boldsymbol{IS\text{-}CDTE}_X,\mathrm{real}_X}(\mathcal{A}) \leq \frac{m}{2^l}$*, where* $\Pr_{\boldsymbol{IS\text{-}CDTE}_X}(M) = \Pr_{\boldsymbol{IS\text{-}CDTE}}(M|X)$ *and* $\Pr_{\mathrm{real}_X}(M) = \Pr_{\mathrm{real}}(M|X)$*.*

Theorem 5. *Assume the maximum length of generating paths is* n *and each vertex has at most* m *children in the generating graph of* ***GPM****, then* $\mathsf{Adv}^{\mathrm{gpm}}_{\boldsymbol{IS\text{-}PMTE},\boldsymbol{GPM}}(\mathcal{A}) \leq \frac{nm}{2^l}$ *for any attacker* $\mathcal{A}$*. Further,* $\mathsf{Adv}^{\mathrm{dte}}_{\boldsymbol{IS\text{-}PMTE},\mathrm{real}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathrm{gpm}}_{\boldsymbol{GPM},\mathrm{real}}(\mathcal{A}) + \mathsf{Adv}^{\mathrm{gpm}}_{\boldsymbol{IS\text{-}PMTE},\boldsymbol{GPM}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathrm{gpm}}_{\boldsymbol{GPM},\mathrm{real}}(\mathcal{A}) + \frac{nm}{2^l}$.

In summary, we propose a generic method for transforming a GPM to a PMTE. The PMTE is secure for the GPM, which means the PMTE is able to resist encoding attacks. To resist distribution difference attacks, an appropriate GPM is needed, for example, statistical language models for natural language texts. Designing such a GPM, however, needs professional knowledge of the real messages, we leave it to experts in related fields.

6 Experimental Results

In this section, we evaluate the security of the existing PMTEs on real datasets under the attacks we propose. In the literature, none of the PMTEs for password vaults can resist encoding attacks as well as none of the PMTEs for genomic data can resist the PCA+SVM attack. But here, we show that our proposed IS-PMTEs for existing password vault models [10, 14] achieve the expected security against encoding attacks as stated in Section 5.4.

6.1 Security Metrics

The ranks of real messages in the order sorted by attackers reflect the security of DTEs. If a DTE is perfectly secure, the real message ranks are evenly distributed under any attack. Accordingly, we use the real message rank distribution as a security metric like [10, 14].

More specifically, we calculate the rank of the message M as follows: 1) generate N decoy messages $\{M_i\}_{i=1}^N$ (by decoding random seeds); 2) calculate the proportion $\hat{r}^-(M)$ (resp. $\hat{r}^+(M)$) of decoy messages with greater (resp. greater or equal) weight than M in $\{M_i\}_{i=1}^N$; 3) pick a random real number in $[\hat{r}^-(M), \hat{r}^+(M)]$ as the rank $\hat{r}(M)$. Same as [10, 14], we set $N = 999$. But different from [10, 14] using average rank $\bar{r}$ (of real messages) and accuracy α (of distinguishing a real message from a decoy one), we use rank cumulative distribution functions (RCDFs) $F(x)$ of real messages to represent attack results. This presentation is more comprehensive than $\bar{r}$ and α. For example, $F^{-1}(1)$ indicates the max rank of real messages, and $F(0)$ indicates the proportion of real messages of rank 0 (i.e., ranking the first). In other words, the attacker excludes $1 - F^{-1}(1)$ proportion decoy messages for all real messages and excludes all decoy messages for $F(0)$ proportion of real messages. In addition, $\bar{r}$ and α can be calculated from $F(x)$ as:

$$\bar{r} = 1 - \int_0^1 F(x)\,dx, \tag{3}$$

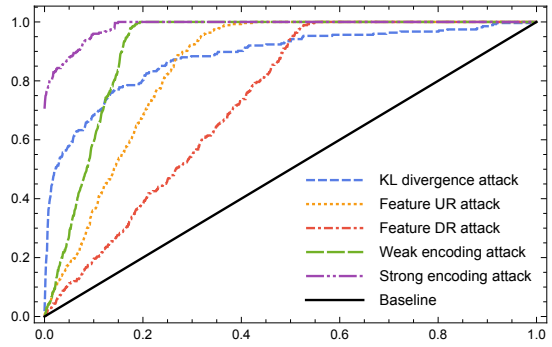
$$\alpha = 1 - \bar{r}. \tag{4}$$

6.2 Datasets

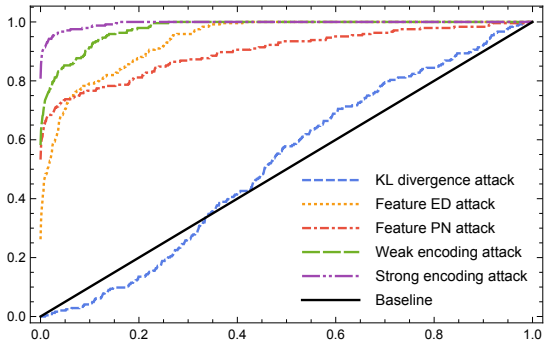
For a fair comparison, we use the same datasets as the previous literature [10, 14, 18]: a password dataset RockYou and a password vault dataset Pastebin for password vault schemes [10, 14], real genomic datasets from HapMap [1] for the genomic data protection scheme [18]. RockYou is a password dataset widely used in password security research, some notable ones like [4, 25, 27, 41], which includes 32.6 million passwords. To the best of our knowledge, Pastebin is the only publicly available dataset for real password vaults so far, and it contains 276 real vaults. Because RockYou and Pastebin are already public and no further harm will be caused, we believe *it is ethical to use them for experiments*. Multiple types of genomic datasets from HapMap are used, including a diploid genotype dataset, a haploid genotype dataset, allele frequency (AF) and linkage disequilibrium (LD) datasets, and recombination rates. The diploid genotype dataset contains 165 persons' SNV sequences. For other details of the above datasets, please refer to [10, 18].

6.3 Evaluating Password Vault PMTEs

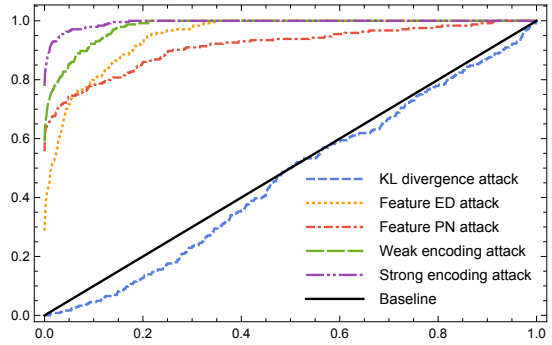
As shown in Figure 4a and Table 1, in Chatterjee et al.'s PMTE [10], the average ranks $\bar{r}$ of real vaults under the feature



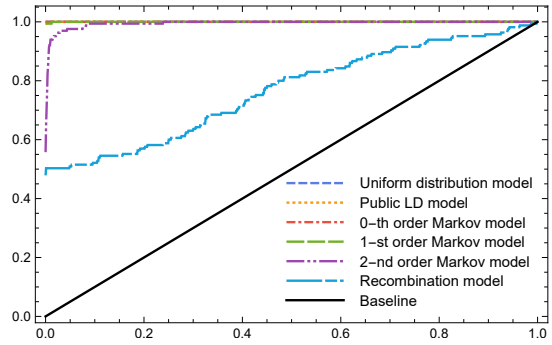
(a) Chatterjee et al.'s PMTE [10]



(b) Golla et al.'s static PMTE [14]



(c) Golla et al.'s adaptive PMTE [14]



(d) Huang et al.'s PMTEs [18] under the PCA+SVM attack

Figure 4: Rank cumulative distribution functions (RCDFs) $F(x)$ of the existing PMTEs

Table 1: The existing PMTEs under encoding attacks or distribution difference attacks

Application	PMTE/Probability model	Attack	$\bar{r}$	α	$F(0)$	$F^{-1}(1)$
Password vault	Chatterjee et al.'s PMTE [10]	KL divergence attack	11.83%	88.17%	1.82%	98.80%
		Feature UR attack	15.14%	84.86%	0.36%	42.24%
		Feature DR attack	26.96%	73.04%	0.00%	54.95%
		Weak encoding attack	8.74%	91.26%	0.36%	19.42%
		Strong encoding attack	1.44%	98.56%	70.55%	15.02%
	Golla et al.'s static PMTE [14]	KL divergence attack	48.26%	51.74%	0.00%	98.70%
		Feature ED attack	6.04%	93.96%	26.23%	41.14%
		Feature PN attack	10.03%	89.97%	53.28%	99.20%
		Weak encoding attack	2.25%	97.75%	58.20%	26.03%
		Strong encoding attack	0.48%	99.52%	80.74%	16.12%
	Golla et al.'s adaptive PMTE [14]	KL divergence attack	53.58%	46.42%	0.00%	100.00%
		Feature ED attack	5.18%	94.82%	28.69%	35.44%
		Feature PN attack	8.60%	91.40%	55.74%	91.79%
		Weak encoding attack	2.01%	97.99%	59.02%	21.22%
		Strong encoding attack	0.58%	99.42%	77.87%	17.22%
Genomic data protection [18]	Uniform distribution model	PCA+SVM attack	0.00%	100.00%	100.00%	0.00%
	Public LD model		0.00%	100.00%	99.39%	0.20%
	0-th order Markov model		0.00%	100.00%	100.00%	0.00%
	1-st order Markov model		0.01%	99.99%	99.39%	1.30%
	2-nd order Markov model		0.53%	99.47%	55.76%	23.92%
	Recombination model		23.46%	76.54%	47.88%	99.90%

UR attack and the feature DR attack are 15.14% and 26.96% respectively, the accuracies α are 84.86% and 73.04%. Moreover, under the feature UR attack, the max rank (i.e., $F^{-1}(1)$) is 42.24%; under the feature DR attack, this number is 54.95%. This means the feature UR attack can exclude at least 57.76% (i.e., $1-F^{-1}(1)$) decoy vaults for every real vault and the feature DR attack can exclude at least 45.05%. Figures 4b, 4c and Table 1 show the performance of Golla et al.'s static PMTE and adaptive PMTE [14], the average ranks under the feature ED attack are 6.04% and 5.18%, while under the feature PN attack are 10.03% and 8.60%. Further, in Golla et al.'s static PMTE, the feature ED attack excludes all decoy vaults for 26.23% (i.e., $F(0)$) real vaults and meanwhile, it excludes at least 58.86% decoy vaults for each real vault. $F(0)$ and $1-F^{-1}(1)$ under the feature PN attack are 53.28% and 0.8% respectively. In Golla et al.'s adaptive PMTE, these numbers are 28.69%, 64.56% under the feature ED attack, and 55.74%, 8.21% under the feature PN attack.

Compared to the above feature attacks, the weak encoding attack has a significant improvement, where the average ranks $\bar{r}$ of Chatterjee et al.'s PMTE [10] and Golla et al.'s (static and adaptive) PMTEs [14] are 8.74%, 2.25%, and 2.01% respectively. The excluded proportions $1-F^{-1}(1)$ are 80.58%, 78.78%, and 73.97%. The strong encoding attack has a further significant improvement compared to the weak encoding attack. The average ranks $\bar{r}$ of these three PMTEs are 1.44%, 0.48%, and 0.57% respectively, which decrease by 84.99%, 83.88%, and 82.78% . Excluded proportions $1-F^{-1}(1)$ are 84.99%, 83.88%, and 82.78% respectively, which also slightly increase by 5.47%, 13.40%, and 5.08%.

Because the KL divergence attack performs better than SVM attacks on all existing PMTEs for password vaults [14], we use it for comparison. As shown in Figures 4a, 4b, 4c and Table 1, the KL divergence attack performs well on the Chatterjee et al.'s PMTE [10], achieving 88.17% accuracy, but it performs almost the same as the randomly guessing attack on Golla et al.'s PMTEs [14], only achieving 46.42%–51.74% accuracy. Further, the RCDFs on Golla et al.'s PMTEs under the KL divergence attack are close to the baseline (the RCDFs under the randomly guessing attack).

For all the existing PMTEs, the curves of RCDFs under the strong encoding attack are all above those under the KL divergence attack. This means that every metric in Table 1 under the strong encoding attack is better than that of the KL divergence attack. More specifically, the average ranks of these three PMTEs under the KL divergence attack are 11.83%, 48.26%, and 53.58%, the accuracies α are 88.17%, 51.74%, and 46.42%. In contrast, the accuracies of the strong encoding attack are 98.56%, 99.52%, and 99.43%, which are 11.78%, 92.35%, and 114.20% higher than those of the KL divergence attack.

In addition, metric values in Table 1 under the KL divergence attack are different from those given in [14], owing to a couple of reasons: 1) for Chatterjee et al.'s PMTE [10], the version of NoCrack used by Golla et al. [14] cannot decode some seeds correctly, therefore have to remedy and reimplement it in the experiments; 2) for Golla et al.'s PMTEs [14], we set the pseudocount of Markov for Laplace smoothing as 1, because under this setting the PMTEs achieve the best security (see Appendix B).

To conclude, the Chatterjee et al.'s PMTE [10] and Golla et al.'s PMTEs [14] are all vulnerable to encoding attacks; meanwhile, Golla et al.'s PMTEs [14] are perfectly secure against the best-reported distribution difference attack.

6.4 Evaluating Genomic Data PMTEs

Different from encoding attacks, the PCA+SVM attack is a distribution difference attack which needs a training set consisting of real and decoy data. We randomly pick 83 individual's SNV sequences in the real dataset[1], generate a decoy sequence for each real sequence, and use them to train our PCA and SVM in the PCA+SVM attack. Then we use remaining 82 individual's sequences in the real dataset and generate N (= 999) decoy sequences for each of them as the test set to compute the RCDF $F(x)$ with the weight function $p_{\text{PCA+SVM}}$. To avoid the impact of randomness on results, we repeat the attack 10 times with different random divisions of the real SNV sequences and newly generated decoy sequences for training/testing, and calculate the average of $F(x)$.

As shown in Figure 4d and Table 1, the PCA+SVM attack achieves more than 99.47% accuracy for all probability models except the recombination model. Even for the recombination model, this attack achieves 76.54% accuracy. This is consistent with Huang et al.'s result [18] that the recombination model performs best. However, it still falls short of the desired security, as our attack excludes all decoy data for 47.88% persons.

To summarize, Huang et al.'s PMTEs for all six models [18] resist encoding attacks but none of them can resist distribution difference attacks. Even the recombination model cannot be rejected at the significance level of 0.2. This means the chi-square goodness-of-fit test is unable to correctly evaluate the security of probability models for generating decoy data.

6.5 Evaluating IS-PMTEs

As stated in Section 5.4, IS-PMTEs resist any encoding attack in theory, we confirm that in practice with IS-PMTEs transformed from existing password vault models. Formally, Theorem 5 demonstrates that the IS-PMTE of an accurate GPM resists arbitrary attacks including encoding attacks. In fact, the IS-PMTE for an arbitrary GPM resists the weak encoding attack. The weight function of the weak encoding attack is constant because every generating path has a chance to be chosen when encoding. This means the weak encoding

[1]We use the small dataset published with the code of GenoGuard on GitHub, which includes 165 persons' SNV sequences of length 1000.

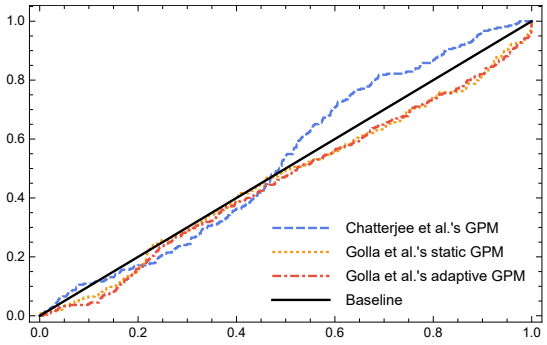


Figure 5: RCDFs of our proposed IS-PMTEs under the strong encoding attack. Note that RCDFs of IS-PMTEs under the weak encoding attack are all equal to the baseline and these under the KL divergence attack are the same as those of the corresponding existing PMTEs in Figure 4.

Table 2: Our IS-PMTEs under the strong encoding attack

Probability model	$\bar{r}$	α	$F(0)$	$F^{-1}(1)$
Chatterjee et al.'s GPM	47.44%	52.56%	0.00%	97.60%
Golla et al.'s static GPM	53.62%	46.38%	0.41%	100.00%
Golla et al.'s adaptive GPM	54.25%	45.75%	0.00%	100.00%

Note: RCDFs of IS-PMTEs under the KL divergence attack are the same as those of existing PMTEs, therefore these metrics under this attack are the same as those in Table 1. RCDFs of IS-PMTEs under the weak encoding attack are the same as those under the randomly guessing attack, therefore these metrics are trivial (50% for $\bar{r}$, 50% for α, 0% for $F(0)$ and 100% for $F^{-1}(1)$).

attack degenerates to the randomly guessing attack (with a constant weight function). In contrast, the weight function p_{GSEA} of the strong encoding attack is inconstant, therefore, RCDFs under the strong encoding attack depend on GPMs.

To evaluate the security of IS-PMTEs for existing vault models under the strong encoding attack, it is necessary to implement the random selection method for generating paths with the parsing function G^{-1}. However, in existing GPMs for password vaults, there are numerous generating paths for messages (as discussed in Section 4.4), therefore, it has high time complexity to parse all generating paths (see the discussion in Appendix C). For example, in Chatterjee et al.'s GPM [10], a vault $V = (123456, \text{password})$ can be generated by any sub-grammar containing $SG = \{\mathtt{S} \to \mathtt{D},\ \mathtt{S} \to \mathtt{W},\ \mathtt{D} \to 123456,\ \mathtt{W} \to \text{password}\}$. It has high time complexity to enumerate all these sub-grammars and calculate the probabilities of generating V by them. Instead, we carry out simulation experiments under the degenerated form of the strong encoding attack with the weight function p_{SEA}. Because all generating paths are encoding paths, there is no seed S with $p_{\text{SEA}}(S) = 0$, i.e., $p_{\text{SEA}}(S) = \frac{1}{P(RS)}$ for every seed S. Accordingly, we use this weight function to sort the seeds in simulation experiments.

Compared to the existing PMTEs, IS-PMTEs transformed from the existing GPMs have a significant improvement on security. As shown in Figure 5 and Table 2, all RCDFs of the IS-PMTEs under the strong encoding attack are approaching to the baseline, i.e., the RCDF under the randomly guessing attack. Average ranks $\bar{r}$ are all near to the expected value of 50%, which are 47.44%, 53.62%, and 54.25%, respectively. Meanwhile, the accuracies are 52.56%, 46.38%, and 45.75%, respectively. Recall that accuracies of existing PMTEs under the strong encoding attacks are 98.56%, 99.52%, and 99.42%, respectively.

Note that our IS-PMTEs have the same decoy message distributions with the corresponding GPMs. This means our IS-PMTEs achieve the same security as the existing PMTEs for the same GPMs under distribution difference attacks. Due to the good performance of Golla et al.'s PMTEs [14] against the best-reported distribution difference attack, our IS-PMTEs for Golla et al.'s GPMs achieve the expected security under both encoding attacks and distribution difference attacks.

7 Conclusion

With encoding attacks and distribution difference attacks, we evaluate three typical existing PMTEs, including two for password vaults and one for genomic data. Using a PCA and an SVM, a distribution difference attack can distinguish real and decoy genomic data with high accuracy. Different from distribution difference attacks exploiting the difference between real and decoy message distributions, encoding attacks are a new type of attack we propose, which exploit the difference between probability models and PMTEs. Encoding attacks can exclude most decoy password vaults/seeds, without any knowledge of real vault distributions.

Further, we introduce a generic conceptual probability model—*generative probability model (GPM)*—to formalize probability models. With the formalization by GPMs, the principle of encoding attacks is uncovered. Based on this principle, we propose two generic and more efficient encoding attacks. In addition, we propose a generic method for transforming an arbitrary GPM to a PMTE. We prove that PMTEs transformed by this method are information-theoretically indistinguishable from the corresponding GPMs, thus can resist encoding attacks. Using this transforming method, we simplify the task of designing a secure PMTE to the task of designing an accurate GPM. Designing such a GPM needs professional knowledge of real messages, we leave it to experts in related fields for future work.

Acknowledgment

The authors are grateful to the anonymous reviewers and the shepherd, Prof. Vincent Bindschaedler, for their invaluable comments that highly improve the completeness of the paper. We also give our special thanks to Prof. Kaitai Liang and

Qianchen Gu for their insightful suggestions and invaluable help. This research was supported by the National Key R&D Program of China under Grant No.2017YFB1200700, and by the National Natural Science Foundation of China (NSFC) under Grant No.61672059.

References

[1] Hapmap. http://hapmap.ncbi.nlm.nih.gov/downloads/index.html.en.

[2] LastPass and YubiKey. https://lastpass.com/yubico/.

[3] Ingolf Becker, Simon Parkin, and M Angela Sasse. The rewards and costs of stronger passwords in a university: linking password lifetime to strength. In *Proc. USENIX Security 2018*, pages 239–253, 2018.

[4] Jeremiah Blocki, Ben Harsha, and Samson Zhou. On the economics of offline password cracking. In *Proc. IEEE S&P 2018*, pages 35–53.

[5] Hristo Bojinov, Elie Bursztein, Xavier Boyen, and Dan Boneh. Kamouflage: Loss-resistant password management. In *Proc. ESORICS 2010*, pages 286–302. Springer.

[6] Dan Boneh, Henry Corrigan-Gibbs, and Stuart Schechter. Balloon hashing: A memory-hard function providing provable protection against sequential attacks. In *Proc. ASIACRYPT 2016*, pages 220–248. Springer.

[7] Joseph Bonneau. The science of guessing: Analyzing an anonymized corpus of 70 million passwords. In *Proc. IEEE S&P 2012*, pages 538–552, 2012.

[8] Joseph Bonneau, Cormac Herley, Paul C Oorschot, and Frank Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *Proc. IEEE S&P 2012*, pages 553–567.

[9] Daniel Buschek, Alexander De Luca, and Florian Alt. Improving accuracy, applicability and usability of keystroke biometrics on mobile touchscreen devices. In *Proc. ACM CHI 2015*, pages 1393–1402.

[10] Rahul Chatterjee, Joseph Bonneau, Ari Juels, and Thomas Ristenpart. Cracking-resistant password vaults using natural language encoders. In *Proc. IEEE S&P 2015*, pages 481–498.

[11] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. The tangled web of password reuse. In *Proc. NDSS 2014*.

[12] Warwick Ford and Burton S Kaliski. Server-assisted generation of a strong secret from a password. In *Proc. WETICE 2000*, pages 176–180.

[13] David Freeman, Sakshi Jain, Markus Dürmuth, Battista Biggio, and Giorgio Giacinto. Who are you? a statistical approach to measuring user authenticity. In *Proc. NDSS 2016*, pages 1–15.

[14] Maximilian Golla, Benedict Beuscher, and Markus Dürmuth. On the security of cracking-resistant password vaults. In *Proc. ACM CCS 2016*, pages 1230–1241.

[15] Maximilian Golla and Markus Dürmuth. On the accuracy of password strength meters. In *Proc. ACM CCS 2018*, pages 1567–1582.

[16] Paul A Grassi, James L Fenton, Elaine M Newton, Ray A Perlner, Andrew R Regenscheid, William E Burr, and Justin P Richer. Nist special publication 800-63b. Digital identity guidelines: Authentication and lifecycle management. *Bericht, NIST*, 2017.

[17] Douglas N Hoover and BN Kausik. Software smart cards via cryptographic camouflage. In *Proc. IEEE S&P 1999*, pages 208–215.

[18] Zhicong Huang, Erman Ayday, Jacques Fellay, Jean-Pierre Hubaux, and Ari Juels. Genoguard: Protecting genomic data against brute-force attacks. In *Proc. IEEE S&P 2015*, pages 447–462.

[19] Stanislaw Jarecki, Hugo Krawczyk, Maliheh Shirvanian, and Nitesh Saxena. Device-enhanced password protocols with optimal online-offline protection. In *Proc. ACM CCS 2016*, pages 177–188.

[20] Stanislaw Jarecki, Hugo Krawczyk, Maliheh Shirvanian, and Nitesh Saxena. Two-factor authentication with end-to-end password security. In *Proc. PKC 2018*, pages 431–461. Springer.

[21] Ari Juels and Thomas Ristenpart. Honey encryption: Security beyond the brute-force bound. In *Proc. EUROCRYPT 2014*, pages 293–310. Springer.

[22] Burt Kaliski. PKCS #5: Password-based cryptography specification version 2.0. 2000.

[23] Russell WF Lai, Christoph Egger, Manuel Reinert, Sherman SM Chow, Matteo Maffei, and Dominique Schröder. Simple password-hardened encryption services. In *Proc. USENIX Security 2018*, pages 1405–1421.

[24] Sanam Ghorbani Lyastani, Michael Schilling, Sascha Fahl, Sven Bugiel, and Michael Backes. Better managed than memorized? studying the impact of managers on password strength and reuse. In *Proc. USENIX Security 2018*, pages 203–220.

[25] Jerry Ma, Weining Yang, Min Luo, and Ninghui Li. A study of probabilistic password models. In *Proc. IEEE S&P 2014*, pages 538–552.

[26] Michelle L Mazurek, Saranga Komanduri, Timothy Vidas, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Patrick Gage Kelley, Richard Shay, and Blase Ur. Measuring password guessability for an entire university. In *Proc. ACM CCS 2013*, pages 173–186.

[27] William Melicher, Blase Ur, Sean M Segreti, Saranga

Komanduri, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Fast, lean, and accurate: Modeling password guessability using neural networks. In *Proc. USENIX Security 2016*, pages 175–191.

[28] Fabian Monrose, Michael K Reiter, and Susanne Wetzel. Password hardening based on keystroke dynamics. *Int. J. Netw. Secur.*, 1(2):69–83, 2002.

[29] Bijeeta Pal, Tal Daniel, Rahul Chatterjee, and Thomas Ristenpart. Beyond credential stuffing: Password similarity models using neural networks. In *Proc. IEEE S&P 2019*, pages 814–831.

[30] Sarah Pearman, Jeremy Thomas, Pardis Emami Naeini, Hana Habib, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Serge Egelman, and Alain Forget. Let's go in for a closer look: Observing passwords in their natural habitat. In *Proc. ACM CCS 2017*, pages 295–310.

[31] Colin Percival. Stronger key derivation via sequential memory-hard functions. *Self-published*, pages 1–16, 2009.

[32] Benny Pinkas and Tomas Sander. Securing passwords against dictionary attacks. In *Proc. ACM CCS 2002*, pages 161–170.

[33] Niels Provos and David Mazieres. A future-adaptable password scheme. In *Proc. USENIX ATC 1999*, pages 81–91.

[34] Richard Shay, Saranga Komanduri, Adam L Durity, Phillip Seyoung Huh, Michelle L Mazurek, Sean M Segreti, Blase Ur, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Designing password policies for strength and usability. *ACM Trans. Inform. Syst. Secur.*, 18(4):13, 2016.

[35] Maliheh Shirvanian, Stanislaw Jarecki, Nitesh Saxena, and Naveen Nathan. Two-factor authentication resilient to server compromise using mix-bandwidth devices. In *Proc. NDSS 2014*.

[36] Maliheh Shirvanian, Stanislaw Jarecki, Nitesh Saxena, and Naveen Nathan. Two-factor authentication resilient to server compromise using mix-bandwidth devices. In *Proc. NDSS 2014*, pages 1–16. The Internet Society.

[37] Maliheh Shirvanian, Stanislaw Jareckiy, Hugo Krawczykz, and Nitesh Saxena. Sphinx: A password store that perfectly hides passwords from itself. In *Proc. ICDCS 2017*, pages 1094–1104.

[38] Joe Siegrist. LastPass security notification, July 2015. https://blog.lastpass.com/2015/06/lastpass-security-notice.html/.

[39] Blase Ur, Felicia Alfieri, Maung Aung, Lujo Bauer, Nicolas Christin, Jessica Colnago, Lorrie Faith Cranor, Henry Dixon, Pardis Emami Naeini, Hana Habib, et al. Design and evaluation of a data-driven password meter. In *Proc. ACM CHI 2017*, pages 3775–3786.

[40] Ding Wang, Debiao He, Haibo Cheng, and Ping Wang. fuzzypsm: A new password strength meter using fuzzy probabilistic context-free grammars. In *Proc. IEEE DSN 2016*, pages 595–606.

[41] Ding Wang, Zijian Zhang, Ping Wang, Jeff Yan, and Xinyi Huang. Targeted online password guessing: An underestimated threat. In *Proc. ACM CCS 2016*, pages 1242–1254.

[42] Matt Weir, Sudhir Aggarwal, Breno de Medeiros, and Bill Glodek. Password cracking using probabilistic context-free grammars. In *Proc. IEEE S&P 2009*, pages 391–405.

[43] Jeff Yan, Alan Blackwell, Ross Anderson, and Alasdair Grant. Password memorability and security: Empirical results. *IEEE Secur. & Priv.*, 2(5):25–31, 2004.

A Proofs in Section 5

Proof of Theorem 1.

$$
\begin{aligned}
&\mathsf{Adv}^{\mathrm{dte},\mathcal{S}}_{\mathbf{DTE},\mathrm{real}}(\mathcal{B})\\
=&|\Pr[\mathcal{B}(S)=1:M\leftarrow_{\Pr_{\mathrm{real}}}\mathcal{M};S\leftarrow_{\$}\mathsf{encode}(M)]\\
&-\Pr[\mathcal{B}(S)=1:S\leftarrow_{\$}\mathcal{S}]|\\
=&|\Pr[\mathcal{A}(S,M')=1:M\leftarrow_{\Pr_{\mathrm{real}}}\mathcal{M};S\leftarrow_{\$}\mathsf{encode}(M);\\
&M'\leftarrow\mathsf{decode}(S)]\\
&-\Pr[\mathcal{A}(S,M')=1:S\leftarrow_{\$}\mathcal{S};M'\leftarrow\mathsf{decode}(S)]|\\
=&|\Pr[\mathcal{A}(S,M)=1:M\leftarrow_{\Pr_{\mathrm{real}}}\mathcal{M};S\leftarrow_{\$}\mathsf{encode}(M)]\\
&-\Pr[\mathcal{A}(S,M)=1:S\leftarrow_{\$}\mathcal{S};M\leftarrow\mathsf{decode}(S)]|\\
=&\mathsf{Adv}^{\mathrm{dte}}_{\mathbf{DTE},\mathrm{real}}(\mathcal{A}).
\end{aligned}
$$

□

Proof of Theorem 2.

$$
\begin{aligned}
&\mathsf{Adv}^{\mathrm{dte},\mathcal{M}}_{\mathbf{DTE},\mathrm{real}}(\mathcal{B})\\
=&|\Pr[\mathcal{B}(M)=1:M\leftarrow_{\Pr_{\mathrm{real}}}\mathcal{M}]\\
&-\Pr[\mathcal{B}(M)=1:S\leftarrow_{\$}\mathcal{S};M\leftarrow\mathsf{decode}(S)]|\\
=&|\Pr[\mathcal{A}(S')=1:M\leftarrow_{\Pr_{\mathrm{real}}}\mathcal{M};S'\leftarrow_{\$}\mathsf{encode}(M)]\\
&-\Pr[\mathcal{A}(S')=1:S\leftarrow_{\$}\mathcal{S};M\leftarrow\mathsf{decode}(S);\\
&S'\leftarrow_{\$}\mathsf{encode}(M)]|\\
=&|\Pr[\mathcal{A}(S)=1:M\leftarrow_{\Pr_{\mathrm{real}}}\mathcal{M};S\leftarrow_{\$}\mathsf{encode}(M)]\\
&-\Pr[\mathcal{A}(S)=1:S\leftarrow_{\$}\mathcal{S}]|\\
=&\mathsf{Adv}^{\mathrm{dte},\mathcal{S}}_{\mathbf{DTE},\mathrm{real}}(\mathcal{A}).
\end{aligned}
$$

□

Proof of Theorem 3. **IS-DTE** is correct, therefore, the combination $\mathbf{IS\text{-}CDTE}=\{\mathbf{IS\text{-}DTE}_X\}_{X\in\mathcal{X}}$ is correct. In addition, because $\mathcal{RS}$ is prefix-free, the padding bits can be ignored unambiguously when decoding. Thus, **IS-PMTE** is correct.

Let S be a seed of the message M, $RS=(r_i)_{i=1}^{n}$ be the generating sequence of S, then the length of padding bits is $ln_{\max}-ln$ and

$$\Pr_{\mathsf{encode}}(S|M)$$

$$
\begin{aligned}
&= \frac{P^{(d)}(RS)}{P^{(d)}(M)} \cdot \frac{1}{2^{ln_{\max}-ln}} \prod_{i=1}^{n} \frac{1}{|\mathsf{encode}(r_i|r_1 r_2 \dots r_{i-1})|} \\
&= \frac{P^{(d)}(RS)}{P^{(d)}(M)} \cdot \frac{1}{2^{ln_{\max}-ln}} \prod_{i=1}^{n} \frac{1}{2^{l} P^{(d)}(r_i|r_1 r_2 \dots r_{i-1})} \\
&= \frac{P^{(d)}(RS)}{P^{(d)}(M)} \cdot \frac{1}{2^{ln_{\max}}} \prod_{i=1}^{n} \frac{1}{P^{(d)}(r_i|r_1 r_2 \dots r_{i-1})} \\
&= \frac{P^{(d)}(RS)}{P^{(d)}(M)} \cdot \frac{1}{2^{ln_{\max}} P^{(d)}(RS)} \\
&= \frac{1}{2^{ln_{\max}} P^{(d)}(M)}.
\end{aligned}
$$

Therefore, **IS-PMTE** is seed-uniform. □

Proof of Theorem 4. According to the definition of $\mathbf{IS\text{-}CDTE}_X$, $\Pr_{\mathbf{IS\text{-}CDTE}_X}(M_i) = \Pr^{(d)}_{\mathrm{real}_X}(M_i) = \mathsf{round}_l(F_i) - \mathsf{round}_l(F_{i-1})$ and $\Pr_{\mathrm{real}_X}(M_i) = F_i - F_{i-1}$, so that $|\Pr_{\mathbf{IS\text{-}CDTE}_X}(M_i) - \Pr_{\mathrm{real}_X}(M_i)| \le \frac{1}{2^l}$. To summarize, $\mathsf{Adv}^{\mathrm{dte}}_{\mathbf{IS\text{-}CDTE}_X,\mathrm{real}_X}(\mathcal{A}) \le \sum_{M\in\mathcal{M}} |\Pr_{\mathbf{IS\text{-}CDTE}_X}(M) - \Pr_{\mathrm{real}_X}(M)| \le \frac{m}{2^l}$. □

Proof of Theorem 5. $\Pr_{\mathbf{IS\text{-}PMTE}}$ is the discretization of $\Pr_{\mathbf{GPM}}$. Similarly, discretizing the first i levels of the generating graph (and keeping the rest levels unchanged) gets a GPM, denoted as $\mathbf{GPM}_i$. Therefore, $\Pr_{\mathbf{GPM}_i}(r_j|r_1 r_2 \dots r_{j-1}) = \Pr_{\mathbf{GPM}_{i-1}}(r_j|r_1 r_2 \dots r_{j-1})$ for $j \neq i$ and by Theorem 4 $|\Pr_{\mathbf{GPM}_i}(r_i|r_1 r_2 \dots r_{i-1}) - \Pr_{\mathbf{GPM}_{i-1}}(r_i|r_1 r_2 \dots r_{i-1})| \le \frac{1}{2^l}$, then

$$
\begin{aligned}
&\mathsf{Adv}^{\mathrm{gpm}}_{\mathbf{GPM}_i,\mathbf{GPM}_{i-1}}(\mathcal{A}) \\
&\le \sum_{M\in\mathcal{M}} |\Pr_{\mathbf{GPM}_i}(M) - \Pr_{\mathbf{GPM}_{i-1}}(M)| \\
&\le \sum_{RS\in\mathcal{RS}} |\Pr_{\mathbf{GPM}_i}(RS) - \Pr_{\mathbf{GPM}_{i-1}}(RS)| \\
&= \sum_{(r_j)_j\in\mathcal{RS}} \Big|\prod_j \Pr_{\mathbf{GPM}_i}(r_j|r_1 r_2 \dots r_{j-1}) \\
&\qquad\qquad - \prod_j \Pr_{\mathbf{GPM}_{i-1}}(r_j|r_1 r_2 \dots r_{j-1})\Big| \\
&= \sum_{(r_j)_j\in\mathcal{RS}} \prod_{j\neq i} \Pr_{\mathbf{GPM}_i}(r_j|r_1 r_2 \dots r_{j-1}) \times \\
&\quad |\Pr_{\mathbf{GPM}_i}(r_i|r_1 r_2 \dots r_{i-1}) - \Pr_{\mathbf{GPM}_{i-1}}(r_i|r_1 r_2 \dots r_{i-1})| \\
&\le \sum_{(r_j)_j\in\mathcal{RS}} \prod_{j\neq i} \Pr_{\mathbf{GPM}_i}(r_j|r_1 r_2 \dots r_{j-1}) \frac{1}{2^l} \\
&= \frac{m}{2^l}.
\end{aligned}
$$

Because $\Pr_{\mathbf{GPM}_0} = \Pr_{\mathbf{GPM}}$ and $\Pr_{\mathbf{GPM}_n} = \Pr_{\mathbf{IS\text{-}PMTE}}$, $\mathsf{Adv}^{\mathrm{gpm}}_{\mathbf{IS\text{-}PMTE},\mathbf{GPM}}(\mathcal{A}) \le \sum_{i=1}^{n} \mathsf{Adv}^{\mathrm{gpm}}_{\mathbf{GPM}_i,\mathbf{GPM}_{i-1}}(\mathcal{A}) \le \frac{nm}{2^l}$. Moreover, $\mathsf{Adv}^{\mathrm{dte}}_{\mathbf{IS\text{-}PMTE},\mathrm{real}}(\mathcal{A}) \le \mathsf{Adv}^{\mathrm{gpm}}_{\mathbf{IS\text{-}PMTE},\mathbf{GPM}}(\mathcal{A}) + \mathsf{Adv}^{\mathrm{gpm}}_{\mathbf{GPM},\mathrm{real}}(\mathcal{A}) \le \mathsf{Adv}^{\mathrm{gpm}}_{\mathbf{GPM},\mathrm{real}}(\mathcal{A}) + \frac{nm}{2^l}$. □

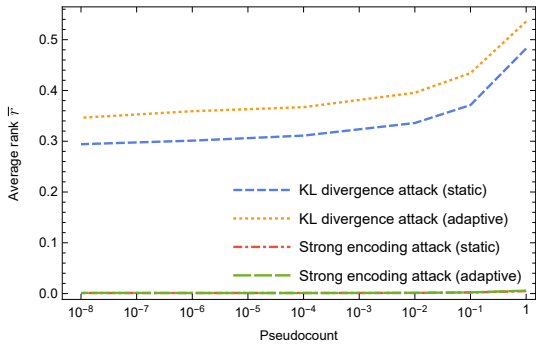


Figure 6: Average rank vs. pseudocount for Golla et al.'s PMTEs [14]

B The Security of Golla et al.'s PMTEs [14] with Different Pseudocounts

We find out that pseudocounts (smoothing parameter) of Markov models with Laplace smoothing in Golla et al.'s PMTEs [14] have a significant influence on the security of PMTEs. As shown in Figure 6 and Table 3, the average rank $\bar{r}$ increases as pseudocount increases under both the KL divergence attack and the strong encoding attacks, and meanwhile, the accuracy α decreases. This means that Golla et al.'s PMTEs [14] achieve the best security when pseudocount is 1. Other metrics in Table 3 also support this conclusion.

Further, when pseudocount is 1, α of the KL divergence attacks are 51.74% and 46.42% on Golla et al.'s static and adaptive PMTEs, respectively. This means Golla et al.'s PMTEs/GPMs almost achieve the expected security ($\alpha = 50\%$) under the best-reported distribution difference attacks.

C The Complexity of IS-PMTEs and Optimization for Encoding

The complexity of an IS-PMTE is of the same order as that of the corresponding GPM. The IS-PMTE stores the CDF table as well as the GPM stores the PDF (probability density function) table. These two tables are of the same size, which means the PMTE and the GPM have the same order of space complexity. When encoding a message, the IS-PMTE needs to obtain all generating sequences for the message and calculate the probability of each sequence, which also needs to be done when the GPM calculates the (total) probability of the message. Moreover, encoding/decoding a sequence needs to do binary search on CDF tables, and meanwhile, calculating the probability of the sequence needs to do binary search on PDF tables. Therefore, the IS-PMTE and the GPM have the same order of time complexity (for encoding messages and calculating message probabilities, respectively).

However, it suffers from high time complexity to obtain all generating paths for some GPMs with great ambiguity. As

Table 3: Golla et al.'s PMTEs [14] with different pseudocounts

Pseudo-count	Attack	Golla et al.'s static PMTE [14]				Golla et al.'s adaptive PMTE [14]			
		$\bar{r}$	α	$F(0)$	$F^{-1}(1)$	$\bar{r}$	α	$F(0)$	$F^{-1}(1)$
1	KL divergence attack	48.26%	51.74%	0.00%	98.70%	53.58%	46.42%	0.00%	100.00%
10^{-1}		37.13%	62.87%	0.00%	99.50%	43.42%	56.58%	0.00%	100.00%
10^{-2}		33.59%	66.41%	2.46%	99.40%	39.55%	60.45%	2.87%	100.00%
10^{-4}		31.11%	68.89%	11.89%	99.20%	36.71%	63.29%	11.89%	100.00%
10^{-6}		30.11%	69.89%	14.75%	99.00%	35.91%	64.09%	14.75%	100.00%
10^{-8}		29.42%	70.58%	17.21%	99.50%	34.62%	65.38%	16.80%	100.00%
1	Strong encoding attack	0.48%	99.52%	80.74%	16.12%	0.58%	99.42%	77.87%	17.22%
10^{-1}		0.22%	99.78%	87.30%	10.11%	0.23%	99.77%	82.38%	9.81%
10^{-2}		0.12%	99.88%	90.57%	9.51%	0.14%	99.86%	90.16%	7.81%
10^{-4}		0.12%	99.88%	92.21%	10.51%	0.10%	99.90%	92.21%	6.41%
10^{-6}		0.11%	99.89%	91.80%	7.91%	0.11%	99.89%	90.98%	8.41%
10^{-8}		0.11%	99.89%	91.80%	8.61%	0.13%	99.87%	92.62%	9.51%

discussed in Section 4.4, in Chatterjee et al.'s GPM [10], a vault can be generated by numerous sub-grammars in Chatterjee et al.'s GPMs. In Golla et al.'s [14] GPMs, a vault can be generated by different base passwords, different cardinalities of subsets and different modified characters. Fortunately, some generating paths can be pruned to reduce the time complexity of encoding. In some models, the dependency of some rules is ignored (by assuming the rules are independent). This triggers some unnecessary paths which can be pruned. For example, in Golla et al.'s GPMs [14], the modified character b_i of passwords in V_i ($1 \leq i \leq 4$) and the corresponding character a_i of the base password are assumed to be independent. In other words, the character of the base password can be modified to itself, i.e., $a_i = b_i$. This yields significant ambiguity. By prohibiting this, we can prune the branch of the original character a_i when generating the modified character b_i. More specifically, the steps of the pruned encoding are as follows: 1) copy a CDF table and delete a_i in the new table; 2) renormalize remaining characters; 3) encode b_i through the renormalized CDF table; 4) abandon the copied CDF table (use the original table for encoding other characters). From the view of the generating graph, the branch of a_i on the node of generating b_i are pruned, resulting in a decrease of time complexity. Besides, the following branches can also be pruned: 1) the character of passwords in V_5 which is the same as the corresponding character of the base password; 2) the cardinality of V_i which is larger than the number of rest passwords. By pruning unnecessary branches on some nodes in the generating graph, we greatly reduce the ambiguity of Golla et al.'s GPMs [14]. For the vaults V, there are only n' generating paths left, where n' is the number of unique passwords in V. Each path corresponds to a different password for generating the vault as the base password.

In Chatterjee et al.'s GPM [10], some unnecessary branches can also be pruned efficiently, e.g., the branches of duplicate rules. However, the branches of unused rules are difficult to be pruned. For example, a vault V of size 2 is generated by the sub-grammar $SG = \{\texttt{S} \to \texttt{D},\ \texttt{S} \to \texttt{W},\ \texttt{D} \to 123456,\ \texttt{W} \to \text{password}\}$. If the first password in V is "123456", then the second one must be "password" to avoid unused rules, i.e., the branch of the rule $\texttt{S} \to \texttt{D}$ should be pruned when generating the second password. In addition, some sub-grammars cannot generate a vault of size 2 without unused rules, for example, the sub-grammars consist of three rules with the lefthand-side S. It also needs to be pruned the branches of all these sub-grammars and renormalize the rest branches. Therefore, in order to prune the branches of unused rules, it is necessary to prune and renormalize branches on almost all nodes in the generating graph. This pruning is difficult because of the high time complexity, especially for the vaults of large sizes. Another simple and straightforward method is to add extra rules in the sub-grammar randomly when encoding. It seems to address this problem. However, the Chatterjee et al.'s GPM [10] with this rule-adding method resists the weak encoding attack but still suffers from the strong encoding attack unless the probability of adding extra rules is equal to the probability of the generating path. This is because the DTE must be seed-uniform in order to resist the strong encoding attack. Moreover, calculating the probability of adding extra rules has the same order of time complexity as calculating the probability of the generating path. Therefore, if this rule-adding method guarantees the property of seed-uniformity, it is equivalent to our method which randomly chooses a generating path with its probability. In other words, the rule-adding method does not perform efficiently in resisting the strong encoding attack. To conclude, we state that a secure DTE of the sub-grammar approach does have high time complexity.

To get rid of the high time complexity of encoding sub-grammars, we propose a design principle for GPMs—*minimizing the ambiguity of the GPM*—to reduce the time complexity of encoding in the corresponding PMTEs. Instead of optimizing the encoding algorithm after designing a GPM with great ambiguity, it may be better to minimize the ambiguity when designing the GPM.

The Art of The Scam: Demystifying Honeypots in Ethereum Smart Contracts

Christof Ferreira Torres
SnT, University of Luxembourg

Mathis Steichen
SnT, University of Luxembourg

Radu State
SnT, University of Luxembourg

Abstract

Modern blockchains, such as Ethereum, enable the execution of so-called *smart contracts* – programs that are executed across a decentralised network of nodes. As smart contracts become more popular and carry more value, they become more of an interesting target for attackers. In the past few years, several smart contracts have been exploited by attackers. However, a new trend towards a more proactive approach seems to be on the rise, where attackers do not search for vulnerable contracts anymore. Instead, they try to lure their victims into traps by deploying seemingly vulnerable contracts that contain hidden traps. This new type of contracts is commonly referred to as *honeypots*. In this paper, we present the first systematic analysis of honeypot smart contracts, by investigating their prevalence, behaviour and impact on the Ethereum blockchain. We develop a taxonomy of honeypot techniques and use this to build HONEYBADGER – a tool that employs symbolic execution and well defined heuristics to expose honeypots. We perform a large-scale analysis on more than 2 million smart contracts and show that our tool not only achieves high precision, but is also highly efficient. We identify 690 honeypot smart contracts as well as 240 victims in the wild, with an accumulated profit of more than $90,000 for the honeypot creators. Our manual validation shows that 87% of the reported contracts are indeed honeypots.

1 Introduction

The concept of blockchain has been introduced in 2009 with the release of Satoshi Nakamoto's Bitcoin [26] and has greatly evolved since then. It is regarded as one of the most disruptive technologies since the invention of the Internet itself. In recent years, companies across the globe have poured value into blockchain research, examining how it can make their existing business more efficient and secure. A blockchain is essentially a verifiable, append-only list of records in which all transactions are recorded in so-called *blocks*. Every block is linked to its previous block via a cryptographic hash, thus forming a chain of blocks or a so-called *"blockchain"*. This list is maintained by a distributed peer-to-peer network of untrusted nodes, which follow a consensus protocol that dictates the appending of new blocks. Trust is obtained via the assumption that the majority acts faithfully and going against the protocol is too costly.

A broad range of different blockchain implementations have emerged since the inception of Bitcoin. However, all of these implementations pursue a common goal, namely, the decentralisation of control over a particular asset. Bitcoin's asset is its cryptocurrency and the trusted centralised entities it attempts to decentralise are traditional banks. Modern blockchains such as Ethereum [46] aim to decentralise the computer as a whole through so-called *smart contracts*. Smart contracts are programs that are stored and executed across the Ethereum blockchain via the Ethereum Virtual Machine (EVM). The EVM is a purely stack-based virtual machine that supports a Turing-complete instruction set of opcodes. Smart contracts are deployed, invoked and removed from the blockchain via transactions. Each operation on the EVM costs a specified amount of *gas*. When the total amount of gas assigned to a transaction is exceeded, program execution is terminated and its effects are reversed. In contrast to traditional programs, smart contracts are immutable. Thus, programming mistakes that were never intended by the developer, become now irreversible. Developers usually write smart contract code in a high-level language which compiles into EVM bytecode. At the time of writing, Solidity [47] is the most prevalent high-level language for developing smart contracts in Ethereum.

In 2018, Ethereum reached a market capitalisation of over $133 billion [9]. As it becomes more and more valuable, attackers become more and more incentivised to find and exploit vulnerable contracts. In fact, Ethereum already faced several devastating attacks on vulnerable smart contracts. The most prominent ones being the DAO hack in 2016 [34] and the Parity Wallet hack in 2017 [29], together causing a loss of over $400 million. In response to these attacks,

academia proposed a plethora of different tools that allow to scan contracts for vulnerabilities, prior to deploying them on the blockchain (see e.g. [21, 25, 38]). Unfortunately, these tools may also be used by attackers in order to easily find vulnerable contracts and exploit them. This potentially enables attackers to follow a reactive approach by actively scanning the blockchain for vulnerable contracts.

Alternatively, attackers could follow a more proactive approach by luring their victims into traps. In other words: *Why should I spend time on looking for victims, if I can just let the victims come to me?* This new type of fraud has been introduced by the community as *"honeypots"* (see e.g. [32, 33]). Honeypots are smart contracts that appear to have an obvious flaw in their design, which allows an arbitrary user to drain *ether* (Ethereum's cryptocurrency) from the contract, given that the user transfers a priori a certain amount of ether to the contract. However, once the user tries to exploit this apparent vulnerability, a second, yet unknown, trapdoor unfolds which prevents the draining of ether to succeed. The idea is that the user solely focuses on the apparent vulnerability and does not consider the possibility that a second vulnerability might be hidden in the contract. Similar to other types of fraud, honeypots work because human beings are often easily manipulated. People are not always capable of quantifying risk against their own greed and presumptions.

In this paper, we investigate the prevalence of such honeypot smart contracts in Ethereum. To the best of our knowledge this is the first work to provide an in depth analysis on the inner workings of this new type of fraud. Moreover, we introduce HONEYBADGER – a tool that uses a combination of symbolic execution and precise heuristics to automatically detect various types of honeypots. Using HONEYBADGER, we are able to provide interesting insights on the plethora, anatomy and popularity of honeypots that are currently deployed on the Ethereum blockchain. Finally, we investigate whether this new type of scam is profitable and we discuss the effectiveness of such honeypots. In summary, we present the following main contributions:

- We conduct the first systematic analysis of an emerging new type of fraud in Ethereum: *honeypots*.
- We identify common techniques used by honeypots and organise them in a taxonomy.
- We present HONEYBADGER, a tool that automatically detects honeypots in Ethereum smart contracts.
- We run HONEYBADGER on 151,935 unique smart contracts and confirm the prevalence of at least 282 unique honeypots.

2 Background

In this section, we provide the required background for understanding the setting of our work, including a description of smart contracts, the Ethereum virtual machine, and the Etherscan blockchain explorer.

2.1 Smart Contracts

The notion of smart contracts has been introduced by Nick Szabo in 1997 [35]. He described the concept of a trustless system consisting of self-executing computer programs that would facilitate the digital verification and enforcement of contract clauses contained in legal contracts. However, this concept only became a reality with the release of Ethereum in 2015. Ethereum smart contracts are different from traditional programs in several aspects. For example, as the code is stored on the blockchain, it becomes immutable and its execution is guaranteed by the blockchain. Nevertheless, smart contracts may be destroyed, if they contain the necessary code to handle their destruction. Once destroyed, a contract can no longer be invoked and its funds are transferred to another address. Smart contracts are usually developed using a dedicated high-level programming language that compiles into low-level bytecode. The bytecode of a smart contract is then deployed to the blockchain through a transaction. Once successfully deployed, a smart contract is identified by a 160-bit address. Despite a large variety of programming languages (e.g. Vyper [44], LLL [19] and Bamboo [6]), Solidity [47] remains the most prominent programming language for developing smart contracts in Ethereum. Solidity's syntax resembles a mixture of C and JavaScript. It comes with a multitude of unique concepts that are specific to smart contracts, such as the transfer of funds or the capability to call other contracts.

2.2 Ethereum Virtual Machine

The Ethereum blockchain consists of a network of mutually distrusting nodes that together form a decentralised public ledger. This ledger allows users to create and invoke smart contracts by submitting transactions to the network. These transactions are processed by so-called *miners*. Miners execute smart contracts during the verification of blocks, using a dedicated virtual machine denoted as the Ethereum Virtual Machine [46]. The EVM is a stack-based, register-less virtual machine, running low-level bytecode, that is represented by an instruction set of opcodes. To guarantee termination of a contract and thus prevent miners to be stuck in endless loops of execution, the concept of *gas* has been introduced. It associates costs to the execution of every single instruction. When issuing a transaction, the sender has to specify the amount of gas that he or she is willing to pay to the miner for the execution of the smart contract. The execution of a smart contract results in a modification of the world state σ, a data structure stored on the blockchain mapping an address a to an account state $\sigma[a]$. The account state of a smart contract consists of two main parts: a balance $\sigma[a]_b$, that holds

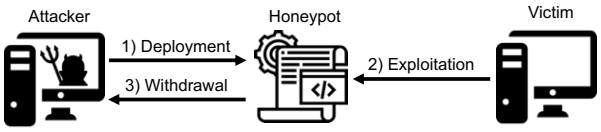


Figure 1: Actors and phases of a honeypot.

the amount of ether owned by the contract, and storage $\sigma[a]_s$, which holds the persistent data of the contract. Storage is organised as a key-value store and is the only way for a smart contract to retain state across executions. Besides the world state σ, the EVM also holds a transaction execution environment I, which contains the address of the smart contract that is being executed I_a, the transaction input data I_d, the transaction sender I_s and the transaction value I_v. The EVM can essentially be seen as a transaction-based state machine, that takes as input σ and I, and outputs a modified world state σ'.

2.3 Etherscan Blockchain Explorer

Etherscan[1] is an online platform that collects and displays blockchain specific information. It acts as a blockchain navigator allowing users to easily lookup the contents of individual blocks, transactions and smart contracts on Ethereum. It offers multiple services on top of its exploring capabilities. One of these services is the possibility for smart contract creators to publish their source code and confirm that the bytecode stored under a specific address is the result of compilation of the specified source code. It also offers users the possibility to leave comments on smart contracts.

3 Ethereum Honeypots

In this section, we provide a general definition of a honeypot and introduce our taxonomy of honeypots.

3.1 Honeypots

Definition 1 (Honeypot) *A honeypot is a smart contract that pretends to leak its funds to an arbitrary user (victim), provided that the user sends additional funds to it. However, the funds provided by the user will be trapped and at most the honeypot creator (attacker) will be able to retrieve them.*

Figure 1 depicts the different actors and phases of a honeypot. A honeypot generally operates in three phases:

1. The attacker deploys a seemingly vulnerable contract and places a bait in the form of funds;
2. The victim attempts to exploit the contract by transferring at least the required amount of funds and fails;
3. The attacker withdraws the bait together with the funds that the victim lost in the attempt of exploitation.

An attacker does not require special capabilities to set up a honeypot. In fact, an attacker has the same capabilities as a regular Ethereum user. He or she solely requires the necessary funds to deploy the smart contract and place a bait.

3.2 Taxonomy of Honeypots

We grasped public sources available on the Internet, in order to have a first glimpse at the inner workings of honeypots [45, 22, 32, 31, 33]. We were able to collect a total of 24 honeypots (see Table 5 in Appendix A) and distill 8 different honeypot techniques. We organise the different techniques in a taxonomy (see Table 1), whose purpose is twofold: (i) as a reference for users in order to avoid common honeypots in Ethereum; (ii) as a guide for researchers to foster the development of methods for the detection of fraudulent smart contracts. We group the different techniques into three different classes, according to the level on which they operate:

1. *Ethereum Virtual Machine*
2. *Solidity Compiler*
3. *Etherscan Blockchain Explorer*

The first class tricks users by making use of the unusual behaviour of the EVM. Although the EVM follows a strict and publicly known set of rules, users can still be misled or confused by devious smart contract implementations that suggest a non-conforming behaviour. The second class relates to honeypots that benefit from issues that are introduced by the Solidity compiler. While some compiler issues are well known, others still remain undocumented and might go unnoticed if a user does not analyse the smart contract carefully or does not test it under real-world conditions. The final and third class takes advantage of issues that are related to the limited information displayed on Etherscan's website. Etherscan is perhaps the most prominent Ethereum

Level	Technique
Ethereum Virtual Machine	Balance Disorder
Solidity Compiler	Inheritance Disorder
	Skip Empty String Literal
	Type Deduction Overflow
	Uninitialised Struct
Etherscan Blockchain Explorer	Hidden State Update
	Hidden Transfer
	Straw Man Contract

Table 1: A taxonomy of honeypot techniques in Ethereum smart contracts.

[1] https://etherscan.io/

```
1  contract MultiplicatorX3 {
2    ...
3    function multiplicate(address adr) payable {
4      if (msg.value >= this.balance)
5        adr.transfer(this.balance+msg.value);
6    }
7  }
```

Figure 2: An example of a balance disorder honeypot.

blockchain explorer and many users fully trust the data displayed therein. In the following, we explain each honeypot technique through a simplified example. We also assume that: 1) the attacker has placed a bait in form of ether into the smart contract, as an incentive for users to try to exploit the contract; 2) the attacker has a way of retrieving the amount of ether contained in the honeypot.

3.2.1 Ethereum Virtual Machine

Balance Disorder. Every smart contract in Ethereum possesses a balance. The contract in Figure 2 depicts an example of a honeypot that makes use of a technique that we denote as *balance disorder*. The function `multiplicate` suggests that the balance of the contract (`this.balance`) and the value included in the transaction to this function call (`msg.value`) are transferred to an arbitrary address, if the caller of this function includes a value that is higher than or equal to the current balance of the smart contract. Hence, a naive user will believe that all that he or she needs to do, is to call this function with a value that is higher or equal to the current balance, and that in return he or she will obtain the "invested" value plus the balance contained in the contract. However, if a user tries to do so, he or she will quickly realise that line 5 is not executed because the condition at line 4 does not hold. The reason for this is that the balance is already incremented with the transaction value, before the actual execution of the smart contract takes place. It is worth noting that: 1) the condition at line 4 can be satisfied if the current balance of the contract is zero, but then the user does not have an incentive to exploit the contract; 2) the addition `this.balance+msg.value` at line 5, solely serves the purpose of making the user further believe that the balance is updated only after the execution.

3.2.2 Solidity Compiler

Inheritance Disorder. Solidity supports inheritance via the `is` keyword. When a contract inherits from multiple contracts, only a single contract is created on the blockchain, and the code from all the base contracts is copied into the created contract. Figure 3 shows an example of a honeypot that makes use of a technique that we denote as *inheritance disorder*. At first glance, there seems to be nothing special

```
1  contract Ownable {
2    address owner = msg.sender;
3    modifier onlyOwner {
4      require(msg.sender == owner);
5      _;
6    }
7  }
8  contract KingOfTheHill is Ownable {
9    address public owner;
10   ...
11   function() public payable {
12     if(msg.value>jackpot)owner=msg.sender;
13     jackpot += msg.value;
14   }
15   function takeAll() public onlyOwner {
16     msg.sender.transfer(this.balance);
17     jackpot = 0;
18   }
19 }
```

Figure 3: An example of an inheritance disorder honeypot.

about this code, we have a contract `KingOfTheHill` that inherits from the contract `Ownable`. We notice two things though: 1) the function `takeAll` solely allows the address stored in variable `owner` to withdraw the contract's balance; 2) the `owner` variable can be modified by calling the fallback function with a message value that is greater than the current jackpot (line 12). Now, if a user tries to call the function in order to set themself as the owner, the transaction succeeds. However, if he or she afterwards tries to withdraw the balance, the transaction fails. The reason for this is that the variable `owner`, declared at line 9, is not the same as the variable that is declared at line 2. We would assume that the `owner` at line 9 would be overwritten by the one at line 2, but this is not the case. The Solidity compiler will treat the two variables as distinct variables and thus writing to `owner` at line 9 will not result in modifying the owner defined in the contract `Ownable`.

Skip Empty String Literal. The contract illustrated in Figure 4 allows a user to place an investment by sending a minimum amount of ether to the contract's function `invest`. Investors may withdraw their investment by calling the function `divest`. Now, if we have a closer look at the code, we realise that there is nothing that prohibits the investor from divesting an amount that is greater than the originally invested amount. Thus a naive user is led to believe that the function `divest` can be exploited. However, this contract contains a bug known as *skip empty string literal*[2]. The empty string literal that is given as an argument to the function `loggedTransfer` (line 14), is skipped by the encoder of the Solidity compiler. This has the effect that the encoding of all arguments following this argument are shifted to the left by 32 bytes and thus the function call argument

[2] https://github.com/ethereum/solidity/blob/develop/docs/bugs.json

```
contract DividendDistributorv3 {
  ...
  function loggedTransfer(uint amount,bytes32
      msg,address target,address currentOwner){
    if (!target.call.value(amount)()) throw;
    Transfer(amount,msg,target,currentOwner);
  }
  function invest() public payable {
    if (msg.value >= minInvestment)
      investors[msg.sender].investment+=msg.
          value;
  }
  function divest(uint amount) public {
    if (investors[msg.sender].investment == 0
        || amount == 0) throw;
      investors[msg.sender].investment-=amount;
      this.loggedTransfer(amount,"",msg.sender,
          owner);
  }
}
```

Figure 4: An example of a skip empty string literal honeypot.

```
contract For_Test {
  ...
  function Test() payable public {
    if (msg.value > 0.1 ether) {
      uint256 multi = 0;
      uint256 amountToTransfer = 0;
      for (var i = 0; i < 2*msg.value; i++) {
        multi = i*2;
        if (multi < amountToTransfer) {
          break;
        amountToTransfer = multi;
      }
      msg.sender.transfer(amountToTransfer);
    }
  }
}
```

Figure 5: An example of a type deduction overflow honeypot.

`msg` receives the value of `target`, whereas `target` is given the value of `currentOwner`, and finally `currentOwner` receives the default value zero. Thus, in the end the function `loggedTransfer` performs a transfer to *currentOwner* instead of *target*, essentially diverting all attempts to divest from the contract to transfers to the owner. A user trying to use the smart contract's apparent vulnerability thereby effectively just transfers the investment to the contract owner.

Type Deduction Overflow. In Solidity, when declaring a variable as type `var`, the compiler uses type deduction to automatically infer the smallest possible type from the first expression that is assigned to the variable. The contract in Figure 5 depicts an example of a honeypot that makes use of a technique that we denote as *type deduction overflow*. At first, the contract suggests that a user will be able to double the investment. However, since the type is only deduced from the first assignment, the loop at line 7 will be infinite. Variable `i` will have the type `uint8` and the highest value of this type is 255, which is smaller than 2 * `msg.value`[3]. Therefore, the loop's halting condition will never be reached. Nevertheless, the loop can still be stopped, if the variable `multi` is smaller than `amountToTransfer`. This is possible, since `amountToTransfer` is assigned the value of `multi`, which eventually will be smaller than `amountToTransfer` due to an integer overflow happening at line 8, where `i` is multiplied by 2. Once the loop exits, the contract performs a value transfer back to the caller, although with an amount that will be at most 255 wei (smallest sub-denomination of ether, where 1 ether $= 10^{18}$ wei) and therefore far less than the value the user originally invested.

```
contract GuessNumber {
  uint private randomNumber=uint256(keccak256(
      now))%10+1;
  uint public lastPlayed;
  uint public minBet=0.1ether;
  struct GuessHistory {
    address player;
    uint256 number;
  }
  function guessNumber(uint256 _number)payable{
    require(msg.value>=minBet&&_number<=10);
    GuessHistory guessHistory;
    guessHistory.player = msg.sender;
    guessHistory.number = _number;
    if (_number == randomNumber)
      msg.sender.transfer(this.balance);
    lastPlayed = now;
  }
}
```

Figure 6: An example of an uninitialised struct honeypot.

Uninitialised Struct. Solidity provides means to define new data types in the form of structs. They combine several named variables under one variable and are the basic foundation for more complex data structures in Solidity. An example of an *uninitialised struct* honeypot is given in Figure 6. In order to withdraw the contract's balance, the contract requires a user to place a minimum bet and guess a random number that is stored in the contract. However, any user can easily obtain the value of the random number, since every data stored on the blockchain is publicly available. The first thought suggests that the contract creator simply made a common mistake by assuming that variables declared as `private` are secret. An innocent user simply reads the random number from the blockchain and calls the function `guessNumber` by placing a bet and providing the correct number. Afterwards, the contract creates a struct that seems to track the participation of the user. However, the struct

[3] 2 * 0.1 ether = 2 * 10^{17} wei

```
1  contract Gift_1_ETH {
2    bool passHasBeenSet = false;
3    ...
4    function SetPass(bytes32 hash) payable {
5      if (!passHasBeenSet&&(msg.value>=1ether))
6        hashPass = hash;
7    }
8    function GetGift(bytes pass)returns(bytes32){
9      if (hashPass == sha3(pass))
10       msg.sender.transfer(this.balance);
11     return sha3(pass);
12   }
13   function PassHasBeenSet(bytes32 hash) {
14     if (hash==hashPass) passHasBeenSet=true;
15   }
16 }
```

Figure 7: An example of a hidden state update honeypot.

is not properly initialised via the `new` keyword. As a result, the Solidity compiler maps the storage location of the first variable contained in the struct (`player`) to the storage location of the first variable contained in the contract (`randomNumber`), thereby overwriting the random number with the address of the caller and thus making the condition at line 14 fail. It is worth noting that the honeypot creator is aware that a user might try to guess the overwritten value. The creator therefore limits the number to be between 1 and 10 (line 10), which drastically reduces the chances of the user generating an address that fulfils this condition.

3.2.3 Etherscan Blockchain Explorer

Hidden State Update. In addition to normal transactions, Etherscan also displays so-called *internal messages*, which are transactions that originate from other contracts and not from user accounts. However, for usability purposes, Etherscan does not display internal messages that include an empty transaction value. The contract in Figure 7 is an example of a honeypot technique that we denote as *hidden state update*. In this example, the balance is transferred to whoever can guess the correct value that has been used to compute the stored hash. A naive user will assume that `passHasBeenSet` is set to `false` and will try to call the unprotected `SetPass` function, which allows to rewrite the hash with a known value, given that least 1 ether is transferred to the contract. When analysing the internal messages on Etherscan, the user will not find any evidence of a call to the `PassHasBeenSet` function and therefore assume that `passHasBeenSet` is set to `false`. However, the filtering performed by Etherscan can be misused by the honeypot creator in order to silently update the state of the variable `passHasBeenSet`, by calling the function `PassHasBeenSet` from another contract and using an empty transaction value. Thus, by just looking at the internal messages displayed on Etherscan, unaware users will believe that the variable is set to `false` and confidently transfer ether to the `SetPass` function.

```
1  contract TestToken {
2    ...
3    function withdrawAll() payable {
4      require(0.5 ether < total);
                                                    if (block.number > 5040270 ) {if (
                                                    _owner == msg.sender ){_owner.transfer(
                                                    this.balance);} else {throw;}}
5      msg.sender.transfer(this.balance);
6    }
7  }
```

Figure 8: An example of a hidden transfer honeypot.

Hidden Transfer. Etherscan provides a web interface that displays the source code of a validated smart contract. Validated means that the provided source code has successfully been compiled to the associated bytecode. For quite a while, Etherscan presented the source code within an HTML `textarea` element, where larger lines of code would only be displayed up to a certain width. Thus, the rest of the line of code would be hidden and solely visible by scrolling horizontally. The contract in Figure 8 takes advantage of this "feature" by introducing, at line 4 in function `withdrawAll`, a long sequence of white spaces, effectively hiding the code that follows. The hidden code throws, if the caller of the function is not the owner and thereby prevents the subsequent balance transfer to any caller of the function. Also note the check at line 4, where the block number must be greater than 5,040,270. This ensures that the honeypot solely steals funds when deployed on the main network. Since the block numbers on the test networks are smaller, testing this contract on a such a network would transfer all the funds to the victim, making him or her believe that the contract is not a honeypot. We label this type of honeypot as *hidden transfer*.

Straw Man Contract. In Figure 9 we provide an example of a honeypot technique that we denote as *straw man contract*. At first sight, it seems that the contract's `CashOut` function is vulnerable to a reentrancy attack [2] (line 14). In order to be able to mount the reentrancy attack, the user is required to first call the `Deposit` function and transfer a minimum amount of ether. Eventually, the user calls the `CashOut` function, which performs a call to the contract address stored in `TransferLog`. As shown in the Figure 9, the contract called `Log` is supposed to act as a logger. However, the honeypot creator did not initialise the contract with an address containing the bytecode of the shown logger contract. Instead it has been initialised with another address pointing to a contract that implements the same interface, but throws an exception if the function `AddMessage` is called with the string "CashOut" and the caller is not the honeypot creator.

```
contract Private_Bank {
  ...
  function Private_Bank(address _log) {
    TransferLog = Log(_log);
  }
  function Deposit() public payable {
    if (msg.value >= MinDeposit) {
      balances[msg.sender]+=msg.value;
      TransferLog.AddMessage("Deposit");
    }
  }
  function CashOut(uint _am) {
    if(_am<=balances[msg.sender]){
      if(msg.sender.call.value(_am)()){
        balances[msg.sender]-=_am;
        TransferLog.AddMessage("CashOut");
      }
    }
  }
}
contract Log {
  ...
  function AddMessage(string _data) public {
    LastMsg.Time = now;
    LastMsg.Data = _data;
    History.push(LastMsg);
  }
}
```

Figure 9: An example of a straw man contract honeypot.

Thus, the reentrancy attack performed by the user will always fail. Another alternative, is to use a `delegatecall` right before the transfer of the balance. Delegatecall allows a callee contract to modify the stack of the caller contract. Thus, the attacker would simply swap the address of the user contained on the stack with his or her own address and when returning from the delegatecall, the balance would be transferred to the attacker instead of the user.

4 HONEYBADGER

In this section, we provide an overview on the design and implementation of HONEYBADGER[4].

4.1 Design Overview

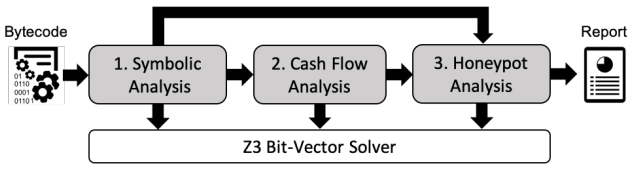


Figure 10: An overview of the analysis pipeline of HONEYBADGER. The shaded boxes represent the main components.

[4]https://github.com/christoftorres/HoneyBadger

Figure 10 depicts the overall architecture and analysis pipeline of HONEYBADGER. HONEYBADGER takes as input EVM bytecode and returns as output a detailed report regarding the different honeypot techniques it detected. HONEYBADGER consists of three main components: *symbolic analysis*, *cash flow analysis* and *honeypot analysis*. The symbolic analysis component constructs the control flow graph (CFG) and symbolically executes its different paths. The result of the symbolic analysis is afterwards propagated to the cash flow analysis component as well as the honeypot analysis component. The cash flow analysis component uses the result of the symbolic analysis to detect whether the contract is capable to receive as well as transfer funds. Finally, the honeypot analysis component aims at detecting the different honeypots techniques studied in this paper using a combination of heuristics and the results of the symbolic analysis. Each of the three components uses the Z3 SMT solver [10] to check for the satisfiability of constraints.

4.2 Implementation

HONEYBADGER is implemented in Python, with roughly 4,000 lines of code. We briefly describe the implementation details of each main component below.

4.2.1 Symbolic Analysis

The symbolic analysis component starts by constructing a CFG from the bytecode, where every node in the CFG corresponds to a basic block and every edge corresponds to a jump between individual basic blocks. A basic block is a sequence of instructions with no jumps going in or out of the middle of the block. The CFG captures all possible program paths that are required for symbolic execution. Symbolic execution represents the values of program variables as symbolic expressions. Each program path consists of a list of path conditions (a formula of symbolic expressions), that must be satisfied for execution to follow that path.

We reused and modified the symbolic execution engine proposed by Luu et al. [21, 20]. The engine consists of an interpreter loop that receives a basic block and symbolically executes every single instruction within that block. The loop continues until all basic blocks of the CFG have been executed or a timeout is reached. Loops are terminated once they exceed a globally defined loop limit. The engine follows a depth first search approach when exploring branches and queries Z3 to determine their feasibility. A path is denoted as feasible if its path conditions are satisfiable. Otherwise, it is denoted as infeasible. Usually, symbolic execution tries to detect and ignore infeasible paths in order to improve their performance. However, our symbolic execution does not ignore infeasible paths, but executes them nevertheless, as they can be useful for detecting honeypots (see Section 4.2.3).

The purpose of the symbolic analysis is to collect all kinds of information that might be useful for later analysis. This information includes a list of storage writes, a list of execution paths P, a list of infeasible as well as feasible basic blocks, a list of performed multiplications and additions, and a list of calls C. Calls are extracted through the opcodes CALL and DELEGATECALL, and either represent a function call, a contract call or a transfer of Ether. A call consists of the tuple $(c_r, c_v, c_f, c_a, c_t, c_g)$, where c_r is the recipient, c_v is the call value, c_f is the called contract function, c_a is the list of function arguments, c_t is the type of call (i.e. CALL or DELEGATECALL) and c_g is the available gas for the call.

4.2.2 Cash Flow Analysis

Given our definition in Section 3.1, a honeypot must be able to *receive* funds (e.g. the investment of a victim) and *transfer* funds (e.g. the loot of the attacker). The purpose of our *cash flow* analysis is to improve the performance of our tool, by safely discarding contracts that cannot receive or transfer funds.

Receiving Funds. There are multiple ways to receive funds besides direct transfers: as a recipient of a block reward, as a destination of a selfdestruct or through the call of a payable function. Receiving funds through a block reward or a selfdestruct makes little sense for a honeypot as this would not execute any harmful code. Also, the compiler adds a check during compilation time, that reverts a transaction if a non-payable function receives a transaction value that is larger than zero. Based on these observations, we verify that a contract is able to receive funds, by first iterating over all possible execution paths contained in P and checking whether there exists an execution path p, that does not terminate in a REVERT. Afterwards, we use Z3 to verify if the constraint $I_v > 0$ can be satisfied under the given path conditions of the execution path p. If p satisfies the constraint, we know that funds can flow into the contract.

Transferring Funds. There are two different ways to transfer funds: either explicit via a *transfer* or implicit via a *selfdestruct*. We verify the former by iterating over all calls contained in C and checking whether there exists a call c, where c_v is either symbolic or $c_v > 0$. We verify the latter by iterating over all execution paths contained in P and checking whether there exists an execution path p that terminates in a SELFDESTRUCT. Finally, we know that funds can flow out of the contract, if we find at least one call c or execution path p, that satisfies the aforementioned conditions.

4.2.3 Honeypot Analysis

Our honeypot analysis consists of several sub-components. Each sub-component is responsible for the detection of a particular honeypot technique. Every honeypot technique is identified via heuristics. We describe the implementation of each sub-component below. The honeypot analysis can easily be extended to detect future honeypots by simply implementing new sub-components.

- **Balance Disorder.** Detecting a balance disorder is straightforward. We iterate over all calls contained in C and report a balance disorder, if we find a call c within an infeasible basic block, where $c_v = I_v + \sigma[I_a]_b$.

- **Inheritance Disorder.** Detecting an inheritance disorder at the bytecode level is rather difficult since bytecode does not include information about inheritance. Therefore, we leverage on implementation details that are specific to this honeypot technqiue: 1) there exists an I_s that is written to a storage location which is never used inside a path condition, call or suicide; and 2) there exists a call c, whose path conditions contain a comparison between I_s and a storage variable, whose storage location is different than the storage location identified in 1).

- **Skip Empty String Literal.** We start by iterating over all calls contained in C and checking whether there exists a call c, where the number of arguments in c_a is smaller than the number of arguments expected by c_f. We report a skip empty string literal, if we can find another call c', that is called within function c_f and where c'_r originates from an argument in c_a.

- **Type Deduction Overflow.** We detect a type deduction overflow by iterating over all calls contained in C and checking whether there exists a call c, where c_v contains the result of a multiplication or an addition that has been truncated via an AND mask with the value `0xff`, which represents the maximum value of an 8-bit integer.

- **Uninitialised Struct.** We use a regular expression to extract the storage location of structs, whose first element is pointing at storage location zero within a basic block. Eventually, we report an uninitialised struct, if there exists a call $c \in C$, where either c_v contains a value from a storage location of a struct or the path condition of c depends on a storage location of a struct.

- **Hidden State Update.** We detect a hidden state update by iterating over all calls contained in C and checking whether there exists a call c, whose path conditions depend on a storage value that can be modified via another function, without the transfer of funds.

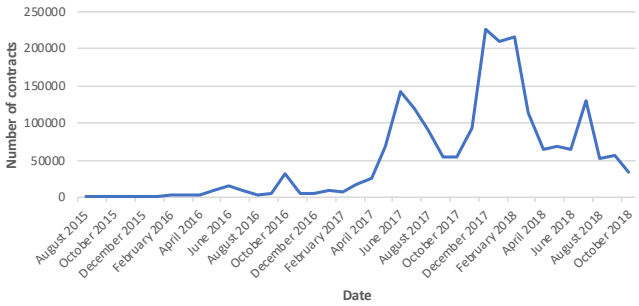


Figure 11: Number of monthly deployed smart contracts in Ethereum.

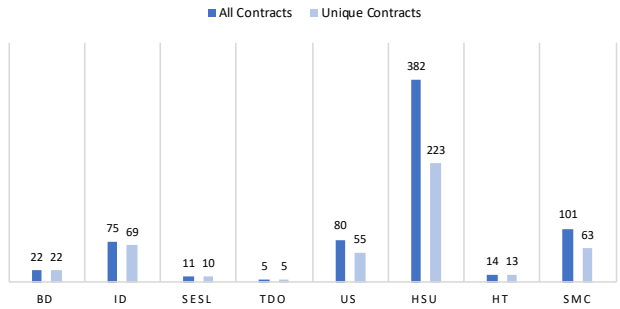


Figure 12: Number of detected honeypots per technique.

- **Hidden Transfer.** We report a hidden transfer, if two consecutive calls c and c' exist along the same execution path p, where $c_r \in \sigma[I_a]_s \wedge c_v = \sigma[I_a]_b$ and $c'_r = I_s \wedge c'_v = \sigma[I_a]_b$.
- **Straw Man Contract.** First, we verify if two consecutive calls c and c' exist along the same execution path p, where $c_r \neq c'_r$. Finally, we report a straw man contract if one of the two cases is satisfied: 1) c is executed after c', where $c'_t = DELEGATECALL \wedge c_v = \sigma[I_a]_b \wedge c_r = I_s$; or 2) c is executed before c', where $c'_t = CALL \wedge I_s \in c'_a$.

5 Evaluation

In this section, we assess the correctness and effectiveness of HONEYBADGER. We aim to determine the reliability of our tool and measure the overall prevalence of honeypots currently deployed on the Ethereum blockchain.

Dataset. We downloaded the bytecode of 2,019,434 smart contracts, by scanning the first 6,500,000 blocks of the Ethereum blockchain. The timestamps of the collected contracts range from August 7, 2015 to October 12, 2018. Figure 11 depicts the number of smart contracts deployed on Ethereum per month. We state a sudden increase in the number of smart contracts deployed between December 2017 and February 2018. We suspect that this inflation is related to the increase of the price of ether and other cryptocurrencies such as Bitcoin [9]. In 2016, 50,980 contracts were deployed on average per month, whereas in 2017 this number increased almost tenfold, with 447,306 contracts on average per month. Interestingly, a lot of contracts share the same bytecode. Out of the 2,019,434 contracts, solely 151,935 are unique in terms of exact bytecode match. In other words, 92.48% of the contracts deployed on the Ethereum blockchain are duplicates.

Experimental Setup. All experiments were conducted on our high-performance computing cluster using 10 nodes with 960 GB of memory, where every node has 2 Intel Xeon L5640 CPUs with 12 cores each and clocked at 2,26 GHz, running 64-bit Debian Jessie 8.10. We used version 1.8.16 of Geth's EVM as our disassembler and Solidity version 0.4.25 as our source-code-to-bytecode compiler. As our constraint solver we used Z3 version 4.7.1. We set a timeout of 1 second per Z3 request for the symbolic execution. The symbolic execution's global timeout was set to 30 minutes per contract. The loop limit, depth limit (for DFS) and gas limit for the symbolic execution were set to 10, 50 and 4 million, respectively.

5.1 Results

We run HONEYBADGER on our set of 151,935 unique smart contracts. Our tool took an average of 142 seconds to analyse a contract, with a median of 31 seconds and a mode of less than 1 second. Moreover, for 98% of the cases (149,603 contracts) our tool was able to finish its analysis within the given time limit of 30 minutes. The number of explored paths ranges from 1 to 8,037, with an average of 179 paths per contract and a median of 105 paths. Finally, during our experiments, HONEYBADGER achieved a code coverage of about 91% on average.

Out of the 151,935 analysed contracts, 48,487 have been flagged as cash flow contracts. In other words, only 32% of the analysed contracts are capable of receiving as well as sending funds. Figure 12 depicts for each honeypot technique the number of contracts that have been flagged by HONEYBADGER. Our tool detected a total of 460 unique honeypots. It is worth mentioning that 24 out of the 460 honeypots were part of our initial dataset (see Table 5 in Appendix A) and that our tool thus managed to find 436 new honeypots. Moreover, as mentioned earlier, many contracts share the same bytecode. Thus, after correlating the results with the bytecode of the 2 million contracts currently deployed on the blockchain, a total of 690 contracts were identified as honeypots[5]. Our tool therefore discovered a total of 22 balance disorders (BD), 75 inheritance disorders (ID), 11

[5] https://honeybadger.uni.lu/

	Balance Disorder	Inheritance Disorder	Skip Empty String Literal	Type Deduction Overflow	Uninitialised Struct	Hidden State Update	Hidden Transfer	Straw Man Contract
TP	20	41	9	4	32	134	12	30
FP	0	7	0	0	0	30	0	4
p	100	85	100	100	100	82	100	88

Table 2: Number of true positives (TP), false positives (FP) and precision p (in %) per detected honeypot technique for contracts with source code.

skip empty string literal (SESL), 5 type deduction overflows (TDO), 80 uninitialised structs (US), 382 hidden state updates (HSU), 14 hidden transfers (HT) and finally 101 straw man contracts (SMC). While many contracts were found to be HSU, SMC and US honeypots, only a small number were found to be TDO honeypots.

5.2 Validation

In order to confirm the correctness of HONEYBADGER, we performed a manual inspection of the source code of the contracts that have been flagged as honeypots. We were able to collect through Etherscan the source code for 323 (70%) of the flagged contracts. We verified the flagged contracts by manually scanning the source code for characteristics of the detected honeypot technique. For example, in case a contract has been flagged as a balance disorder, we checked whether the source code contains a function that transfers the contract's balance to the caller if and only if the value sent to the function is greater than or equal to the contract's balance.

Table 2 summarises our manual verification in terms of true positives (TP), false positives (FP) and precision p, where p is computed as $p = TP/(TP+FP)$. A true positive means that the contract is indeed a honeypot with respect to the reported technique and a false positive means that the contract is *not* a honeypot with respect to the reported technique. Overall our tool shows a very high precision and a very low false positive rate. Our tool achieves a false positive rate of 0% for 5 out of the 8 analysed honeypot techniques. For the remaining 3 techniques, our tool achieves a decent false positive rate, where the highest false positive rate is roughly 18% for the detection of hidden state updates, followed by 15% false positive rate for the detection of inheritance disorder and finally 12% false positive rate for the detection of straw man contracts.

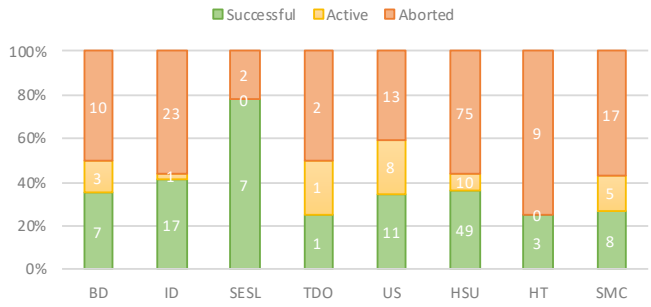


Figure 13: Number of successful, active and aborted honeypots per honeypot technique.

6 Analysis

In this section, we analyse the true positives obtained in Section 5, in order to acquire insights on the *effectiveness*, *liveness*, *behaviour*, *diversity* and *profitability* of honeypots.

6.1 Methodology

We crawled all the transactions of the 282 true positives using Etherchain's[6] API, in order to collect various information about the honeypots, such as the amount of spent and received ether per address, the deployment date and the balance. Afterwards, we used simple heuristics to label every address as either an *attacker* or a *victim*. An address is labeled as an attacker if it either: 1) created the honeypot; 2) was the first address to send ether to the honeypot; or 3) received more ether than it actually spent on the honeypot. An address is labeled as a victim if it has not been labeled as an attacker and if it received less ether than it actually spent on the honeypot. Finally, using this information we were able to tell if a honeypot, was either *successful*, *aborted* or still *active*. A honeypot is marked as successful if a victim has been detected, as aborted if the balance is zero and no victim has been detected or as active if the balance is larger than zero and no victim has been detected.

6.2 Results

Effectiveness. Figure 13 shows the number of successful, aborted and active honeypots per honeypot technique. Our results show that *skip empty string literal* is the most effective honeypot technique with roughly 78% success rate, whereas *hidden transfer* is the least effective technique with solely 33% success rate. The overall success rate of honeypots seems rather low with roughly 37%, whereas the overall abortion rate seems quite high with about 54%. At the time of writing, solely 10% of the analysed honeypots are still active. Figure 14 illustrates the number of monthly deployed

[6] https ://www.etherchain.org/

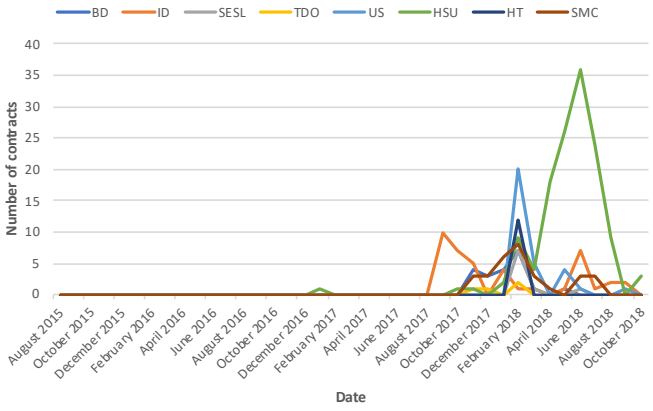


Figure 14: Number of monthly deployed honeypots per honeypot technique.

honeypots per honeypot technique. The very first honeypot technique that has been deployed was a hidden state update in January 2017. February 2018 has been the peak in terms of honeypots being deployed, with a total of 66. The highest number of monthly honeypots that have been deployed per technique are hidden state updates with a total of 36 in June 2018. 7 honeypots have been deployed on average per month. In our analysis, the quickest first attempt of exploitation happened just 7 minutes and 37 seconds after a honeypot had been deployed, whereas the longest happened not until 142 days after deployment. A honeypot takes an average of 9 days and a median of 16 hours before it gets exploited. Interestingly, most honeypots (roughly 55%) are exploited during the first 24 hours after being deployed.

Liveness. We define the lifespan of a honeypot as the period of time between the deployment of a honeypot and the moment when a honeypot was aborted. We found that the shortest lifespan of a honeypot was 5 minutes and 25 seconds and the longest lifespan was about 322 days. The average lifespan of a honeypot is roughly 28 days, whereas the median is roughly 3 days. However, in around 32% of the cases the lifespan of a honeypot is solely 1 day. We also analysed how long an attacker keeps the funds inside a honeypot, by measuring the period of time between the first attempt of exploitation by a victim and the withdrawal of all the funds by the attacker. The shortest period was just 4 minutes and 28 seconds after a victim fell for the honeypot. The longest period was roughly 100 days. On average attackers withdraw all their funds within 7 days after a victim fell for the honeypot. However, in most cases the attackers keep the funds in the honeypot for a maximum of 1 day. Interestingly, only 37 out of 282 honeypots got destroyed, where destroyed means that the attacker called a function within the honeypot that calls the SELFDESTRUCT opcode. In other words, 171 honeypots are in some kind of "zombie" state, where they



Figure 15: A word cloud generated from the comments on Etherscan.

are still alive (i.e. not destroyed), but not active (i.e. their balance is zero). Analysing the 37 destroyed honeypots, we found that 19 got destroyed after being successful and 18 after never having been successful.

Behaviour. Our methodology classified a total of 240 addresses as victims. In 71% of the cases a honeypot managed to trap solely one victim. In one case though, 97 victims have been trapped by just a single honeypot. Interestingly, 8 out of the 240 addresses fell for more than one honeypot, where one address even became a victim to four different honeypots. We also found that 53 attackers deployed at least two honeypots, whereas a sole attacker deployed eight different honeypots. It is worth noting that 42 of the 53 attackers simply deployed copies of one particular honeypot type, whereas the remaining 11 deployed honeypots of varying types. 87 out of the 282 detected and manually confirmed honeypots (about 31%) contained comments on Etherscan. We manually analysed these comments and found that the majority of the comments were indeed warnings stating that the contract might be a honeypot. Moreover, Figure 15 shows that the term "honeypot" is the most prevalent term used by the community to describe this type of smart contracts. Surprisingly, 20 out of the 87 commented honeypots were successful. 16 were successful before a comment had been placed and 4 have been successful even after a comment had been placed. Interestingly, 21 honeypots aborted after a comment was placed. The quickest abort was performed just 33 minutes and 57 seconds after the comment, whereas the longest abort was performed 37 days after the comment. Finally, attackers took an average of 6 days and a median of 22 hours to abort their honeypot after a user had placed a comment.

Diversity. We used the normalised Levenshtein distance [48] to measure the similarity of the bytecode between the individual instances of a particular honeypot technique. Table 3 outlines the similarity in terms of minimum, maximum, mean and mode per honeypot technique. We observe that for almost every technique, except TDO, the bytecode similarity varies tremendously. For example, in case of hidden state update honeypots, we measure a minimum similarity of 11% and a maximum similarity of 98%. This indicates that even though two honeypots share the same technique,

	BD	ID	SESL	TDO	US	HSU	HT	SMC
Min.	27	14	22	88	25	11	28	26
Max.	97	96	98	95	98	98	98	98
Mean	50	40	47	90	52	49	71	53
Mode	35	35	28	89	45	36	95	49

Table 3: Bytecode similarity (in %) per honeypot technique.

their bytecode might still be very diverse.

Profitability. Table 4 lists the profitability per honeypot technique. The profitability is computed as *received amount* - (*spent amount* + *transaction fees*). No values are provided for TDO, because for the single true positive that we analysed, the transaction fees spent by the attacker were higher than the amount that the attacker gained from the victim. The smallest and largest profit were made using a hidden state update honeypot, with 0.00002 ether being the smallest and 11.96 ether being the largest. The most profitable honeypots are straw man contract honeypots, with an average value of 1.76 ether, whereas the least profitable honeypots are uninitialised struct honeypots, with an average value of 0.46 ether. A total profit of 257.25 ether has been made through honeypots, of which 171.22 ether were solely made through hidden state update honeypots. However, the exchange rate of cryptocurrencies is very volatile and thus their value in USD may vary greatly on a day-to-day basis. For example, although 11.96 ether is the largest profit made in ether, its actual value in USD was solely 500 at the point of withdrawal. Thus, we found that the largest profit in terms of USD, was actually a honeypot with 3.10987 ether, as it was worth 2,609 USD at the time of withdrawal. Applying this method across the 282 honeypots, results in a total profit of 90,118 USD.

7 Discussion

In this section we summarise the key insights gained through our analysis and we discuss the ethical considerations as well as the challenges and limitations of our work.

	Min.	Max.	Mean	Mode	Median	Sum
BD	0.01	1.13	0.5	0.11	0.11	3.5
ID	0.004	6.41	1.06	0.1	0.33	17.02
SESL	0.584	4.24	1.59	1.0	1.23	9.57
TDO	-	-	-	-	-	-
US	0.009	1.1	0.46	0.1	0.38	6.44
HSU	0.00002	11.96	1.44	0.1	1.02	171.22
HT	1.009	1.1	1.05	1.0	1.05	2.11
SMC	0.399	4.94	1.76	2.0	1.99	47.39
Overall	0.00002	11.96	1.35	1.0	1.01	257.25

Table 4: Statistics on the profitability of each honeypot technique in ether.

7.1 Honeypot Insights

Although honeypots are capable of trapping multiple users, we have found that most honeypots managed to take the funds of only one victim. This indicates that users potentially look at the transactions of other users before they submit theirs. Moreover, the low success rate of honeypots with comments, suggests that users also check the comments on Etherscan before submitting any funds. We also found that the bytecode of honeypots can be vastly different even if using the same honeypot technique. This suggests that the usage of signature-based detection methods would be rather ineffective. HONEYBADGER is capable of recognising a variety of implementations, as it specifically targets the functional characteristics of each honeypot technique. More than half of the honeypots were successful within the first 24 hours. This suggests that honeypots become less effective the older they become. This is interesting, as it means that users seem to target rather recently deployed honeypots than older ones. We also note that most honeypot creators withdraw their loot within 24 hours or abort their honeypots if they are not successful within the first 24 hours. We therefore conclude that honeypots have in general a short lifespan and only a small fraction remain active for a period longer than one day.

7.2 Challenges and Limitations

The amount of smart contracts with source code available is rather small. At the time of writing, there are only 50,000 contracts with source code available on Etherscan. This highlights the necessity of being able to detect honeypots at the bytecode level. Unfortunately, this turns out to be extremely challenging when detecting certain honeypot techniques. For example, while detecting inheritance disorder at the source code level is rather trivial, detecting it at the bytecode level is rather difficult since all information about the inheritance is lost during compilation and not available anymore at the bytecode level. The fact that certain information is solely available at the source code level and not at the bytecode level, obliges us to make use of other less precise information that is available in the bytecode in order to detect honeypot techniques such as inheritance disorder. However, as Section 5 shows, this approach reduces the precision of our detection and introduces some false positives. Finally, another limitation of our tool is that it is currently limited to the detection of the eight honeypot techniques described in this paper. Thus other honeypot techniques are not detected. Nevertheless, we designed HONEYBADGER with modularity in mind, such that one can easily extend the honeypot analysis component with new heuristics in order to detect more honeypot techniques.

7.3 Ethical Considerations

In general, honeypots have two participants, the creator of the honeypot, and the user whose funds are trapped by the honeypot. However, the ethical intentions of both participants are not always clear. For instance, a honeypot creator might deploy a honeypot with the intention to scam users and make profit. In this case we clearly have a malicious intention. However, one could also argue that a honeypot creator is just attempting to punish users that behave maliciously. Similarly, the intentions of a honeypot user can either be malicious or benign. For example, if a user tries to intentionally exploit a reentrancy vulnerability, then he or she needs to be knowledgable and mischievous enough to prepare and attempt the attack, and thus clearly showing malicious behaviour. However, if we take the example of an uninitialised struct honeypot that is disguised as a simple lottery, then we might have the case of a benign user who loses his funds under the assumption that he or she is participating in a fair lottery. Thus, both honeypot creators and users cannot always be clearly classified as either malicious or benign, this depends on the case at hand. Nevertheless, we are aware that our methodology may serve malicious attackers to protect themselves from other malicious attackers. However, with HONEYBADGER, we hope to raise the awareness of honeypots and save benign users from potential financial losses.

8 Related Work

Honeypots are a new type of fraud that combine security issues with scams. They either rely on the blockchain itself or on related services such as Etherscan. With growing interest within the blockchain community, they have been discussed online [31, 32, 33] and collected within public user repositories [22, 45]. Frauds and security issues are nothing new within the blockchain ecosystem. Blockchains have been used for money laundering [24] and been the target of several scams [42], including mining scams, wallet scams and Ponzi schemes, which are further discussed in [4, 43]. In particular, smart contracts have been shown to contain security issues [2]. Although not performed directly on the blockchain, exchanges have also been the target of fraud [23].

Several different methods have been proposed to discover fraud as well as security issues. Manual analysis is performed on publicly available source code to detect Ponzi schemes [3]. [49] introduces ERAYS, a tool that aims to produce easy to analyse pseudocode from bytecode where the source code is not available. However, manual analysis is particularly laborious, especially considering the number of contracts on the blockchain. Machine learning has been used to detect Ponzi schemes [8] and to find vulnerabilities [36]. The latter relies on [27] to obtain a ground truth of vulnerable smart contracts for training their model. Fuzzing techniques have been employed to detect security vulnerabilities in smart contracts [15] and in combination with symbolic execution to discover issues related to the ordering of events or function calls [17]. However, fuzzing often fails to create inputs to enter specific execution paths and therefore might ignore them [40]. Static analysis has been used to find security [7, 39, 37] and gas-focused [11] vulnerabilities in smart contracts. [7] requires manual interaction, while [39] requires both the definition of violation and compliance patterns. [37] requires Solidity code and therefore cannot be used to analyse the large majority of the smart contracts deployed on the Ethereum blockchain. [11] considers gas-related issues which is not necessary for the purpose of this work. In order to use formal verification, smart contracts can, to some extent, be translated from source code or bytecode into F* [5, 12] where the verification can more easily be performed. Other work operates on high-level source code available for Ethereum or Hyperledger [16]. [13, 14] propose a formal definition of the EVM, that is extended in [1] towards more automated smart contract verification and the consideration of gas. Formal verification often requires (incomplete) translations into other languages or manual user interaction (e.g.: [30]). Both of these reasons make formal verification unsuitable to be used on a large number of contracts, as it is required in this work.

Symbolic execution has been used on smart contracts to detect common [28, 25, 21, 38] vulnerabilities. This technique also allows to find specific kinds of misbehaving contracts [27]. It can further provide values that can serve to generate automated exploits that trigger vulnerabilities [18]. The same technique is used in this paper. Symbolic execution has the advantage of being capable to reason about all possible execution paths and states in a smart contract. This allows for the implementation of precise heuristics while achieving a low false positive rate. Another advantage is that symbolic execution can be applied directly to bytecode, thus making it well suited for our purpose of analysing more than 2 million smart contracts for which source code is largely not available. The disadvantage is the large number of possible paths that need to be analysed. However, in the case of smart contracts this is not an issue, as most are not very complex and very short. Moreover, smart contract bytecode cannot grow arbitrarily large due to the gas limit enforced by the Ethereum blockchain.

To the best of the authors' knowledge, this paper is the first to consider and discuss honeypot smart contracts, a new type of fraud, and to propose a taxonomy as well as an automated tool using symbolic execution for their detection.

9 Conclusion

In this work, we investigated an emerging new type of fraud in Ethereum: *honeypots*. We presented a taxonomy of honeypot techniques and introduced a methodology that uses symbolic execution and heuristics for the automated detec-

tion of honeypots. We showed that HONEYBADGER can effectively detect honeypots in the wild with a very low false positive rate. In a large-scale analysis of 151,935 unique Ethereum smart contracts, HONEYBADGER identified 460 honeypots. Moreover, an analysis on the transactions performed by a subset of 282 honeypots, revealed that 240 users already became victims of honeypots and that attackers already made more than 90,000 USD profit with honeypots. It is worth noting that these numbers solely provide a lower bound and thus might only reflect the tip of the iceberg. Nonetheless, tools such as HONEYBADGER may already help users in detecting honeypots before they can cause any harm. In future work, we plan to further generalise our detection mechanism through the use of machine learning techniques. We also plan to extend our analysis with a larger subset and eventually detect new honeypots by looking at other contracts that are linked to the newly discovered honeypot contracts.

Acknowledgments

We would like to thank Hugo Jonker and Sjouke Mauw as well as the anonymous reviewers for their valuable feedback and comments. The experiments presented in this paper were carried out using the HPC facilities of the University of Luxembourg [41] – see `https://hpc.uni.lu`. This work is partly supported by the Luxembourg National Research Fund (FNR) under grant 13192291.

References

[1] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. Towards verifying ethereum smart contract bytecode in isabelle/hol. *CPP. ACM. To appear*, 2018.

[2] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*, pages 164–186. Springer-Verlag New York, Inc., 2017.

[3] Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, and Roberto Saia. Dissecting ponzi schemes on ethereum: identification, analysis, and impact. *arXiv preprint arXiv:1703.03779*, 2017.

[4] Massimo Bartoletti, Barbara Pes, and Sergio Serusi. Data mining for detecting bitcoin ponzi schemes. *arXiv preprint arXiv:1803.00646*, 2018.

[5] Karthikeyan Bhargavan, Nikhil Swamy, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, and Thomas Sibut-Pinote. Formal Verification of Smart Contracts. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security - PLAS'16*, pages 91–96, New York, New York, USA, 2016. ACM Press.

[6] Cornell Blockchain. Bamboo: a language for morphing smart contracts, May 2018. https://github.com/CornellBlockchain/bamboo.

[7] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981*, 2018.

[8] Weili Chen, Zibin Zheng, Jiahui Cui, Edith Ngai, Peilin Zheng, and Yuren Zhou. Detecting ponzi schemes on ethereum: Towards healthier blockchain technology. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*, pages 1409–1418. International World Wide Web Conferences Steering Committee, 2018.

[9] CoinMarketCap. Ethereum (ETH) price, charts, market cap, and other metrics — CoinMarketCap, January 2018. https://coinmarketcap.com/currencies/ethereum/.

[10] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[11] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):116, 2018.

[12] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. In *International Conference on Principles of Security and Trust*, pages 243–269. Springer, 2018.

[13] Yoichi Hirai. Defining the ethereum virtual machine for interactive theorem provers. In *International Conference on Financial Cryptography and Data Security*, pages 520–535. Springer, 2017.

[14] Yoichi Hirai. Ethereum virtual machine for coq (v0.0.2), June 2017. https://medium.com/@pirapira/ethereum-virtual-machine-for-coq-v0-0-2-d2568e068b18.

[15] Bo Jiang, Ye Liu, and W. K. Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 259–269, New York, NY, USA, 2018. ACM.

[16] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: Analyzing safety of smart contracts. In *NDSS*, 2018.

[17] Aashish Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. Exploiting the laws of order in smart contracts. *arXiv preprint arXiv:1810.11605*, 2018.

[18] Johannes Krupp and Christian Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1317–1333, 2018.

[19] LLL. Ethereum low-level lisp-like language, January 2019. https://lll-docs.readthedocs.io/en/latest/lll_introduction.html.

[20] Loi Luu. Oyente - An Analysis Tool for Smart Contracts v0.2.7 (Commonwealth), February 2017. https://github.com/melonproject/oyente.

[21] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 254–269, New York, NY, USA, 2016. ACM.

[22] misterch0c. Solidity vulnerable honeypots, April 2018. https://github.com/misterch0c/Solidlity-Vulnerable/tree/master/honeypots.

[23] Tyler Moore and Nicolas Christin. Beware the middleman: Empirical analysis of bitcoin-exchange risk. In *International Conference on Financial Cryptography and Data Security*, pages 25–33. Springer, 2013.

[24] Malte Moser, Rainer Bohme, and Dominic Breuker. An inquiry into money laundering tools in the bitcoin ecosystem. In *eCrime Researchers Summit (eCRS), 2013*, pages 1–14. IEEE, 2013.

[25] Bernhard Mueller. Smashing ethereum smart contracts for fun and real profit. In *9th annual HITB Security Conference*, 2018.

[26] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Cryptography Mailing list at https://metzdowd.com*, 03 2009.

[27] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. *arXiv preprint arXiv:1802.06038*, 2018.

[28] Trail of Bits. Manticore - symbolic execution tool, jun 2018. https://github.com/trailofbits/manticore.

[29] Sergey Petrov. Another parity wallet hack explained, nov 2017. https://medium.com/@Pr0Ger/another-parity-wallet-hack-explained-847ca46a2e1c.

[30] Christian Reitwiessner. Formal verification for solidity contracts, June 2018. https://forum.ethereum.org/discussion/3779/formal-verification-for-solidity-contracts.

[31] Josep Sanjuas. An analysis of a couple ethereum honeypot contracts, December 2018. https://medium.com/coinmonks/an-analysis-of-a-couple-ethereum-honeypot-contracts-5c07c95b0a8d.

[32] Alex Sherbachev. Hacking the hackers: Honeypots on ethereum network, December 2018. https://hackernoon.com/hacking-the-hackers-honeypots-on-ethereum-network-5baa35a13577.

[33] Alex Sherbuck. Dissecting an ethereum honey pot, December 2018. https://medium.com/coinmonks/dissecting-an-ethereum-honey-pot-7102d7def5e0.

[34] David Siegel. Understanding the dao attack, jun 2016. https://www.coindesk.com/understanding-dao-hack-journalists/.

[35] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.

[36] A Tann, Xing Jie Han, Sourav Sen Gupta, and Yew-Soon Ong. Towards safer smart contracts: A sequence learning approach to detecting vulnerabilities. *arXiv preprint arXiv:1811.06632*, 2018.

[37] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 9–16, May 2018.

[38] Christof Ferreira Torres, Julian Schütte, and Radu State. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, pages 664–676, New York, NY, USA, 2018. ACM.

[39] Petar Tsankov, Andrei Dan, Dana Drachsler Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. *arXiv preprint arXiv:1806.01143*, 2018.

[40] Mathy Vanhoef and Frank Piessens. Symbolic execution of security protocol implementations: Handling cryptographic primitives. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, Baltimore, MD, 2018. USENIX Association.

[41] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos. Management of an academic hpc cluster: The ul experience. In *Proc. of the 2014 Intl. Conf. on High Performance Computing & Simulation (HPCS 2014)*, pages 959–967, Bologna, Italy, July 2014. IEEE.

[42] Marie Vasek and Tyler Moore. Theres no free lunch, even using bitcoin: Tracking the popularity and profits of virtual currency scams. In *International conference on financial cryptography and data security*, pages 44–61. Springer, 2015.

[43] Marie Vasek and Tyler Moore. Analyzing the bitcoin ponzi scheme ecosystem. In *Bitcoin Workshop*, 2018.

[44] Vyper. Pythonic smart contract language for the evm, January 2019. https://github.com/ethereum/vyper.

[45] Gerhard Wagner. Smart contract honeypots, April 2018. https://github.com/thec00n/smart-contract-honeypots.

[46] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151:1–32, 2014.

[47] Gavin Wood. Solidity 0.5.1 documentation, December 2018. https://solidity.readthedocs.io/en/v0.5.1/.

[48] Li Yujian and Liu Bo. A normalized levenshtein distance metric. *IEEE transactions on pattern analysis and machine intelligence*, 29(6):1091–1095, 2007.

[49] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. Erays: Reverse engineering ethereum's opaque smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1371–1385, 2018.

A List of Honeypots

Table 5 presents the list of 24 honeypots that have been collected from public sources available on the Internet.

Contract Name	Contract Address	Technique
Ethereum Virtual Machine		
MultiplicatorX3	0x5aa88d2901c68fda244f1d0584400368d2c8e739	Balance Disorder
PinCodeEtherStorage	0x35c3034556b81132e682db2f879e6f30721b847c	Balance Disorder
Solidity Compiler		
TestBank	0x70c01853e4430cae353c9a7ae232a6a95f6cafd9	Inheritance Disorder
KingOfTheHill	0x4dc76cfc65b14b3fd83c8bc8b895482f3cbc150a	Inheritance Disorder
RichestTakeAll	0xe65c53087e1a40b7c53b9a0ea3c2562ae2dfeb24	Inheritance Disorder
ICO_Hold	0x4ba0d338a7c41cc12778e0a2fa6df2361e8d8465	Inheritance Disorder
TerrionFund	0x33685492a20234101b553d2a429ae8a6bf202e18	Inheritance Disorder
DividendDistributorv3	0x858c9eaf3ace37d2bedb4a1eb6b8805ffe801bba	Skip Empty String Literal
For_Test	0x2ecf8d1f46dd3c2098de9352683444a0b69eb229	Type Deduction Overflow
Test1	0x791d0463b8813b827807a36852e4778be01b704e	Type Deduction Overflow
CryptoRoulette	0x94602b0e2512ddad62a935763bf1277c973b2758	Uninitialised Struct
OpenAddressLottery	0xd1915a2bcc4b77794d64c4e483e43444193373fa	Uninitialised Struct
GuessNumber	0x559cc6564ef51bd1ad9fbe752c9455cb6fb7feb1	Uninitialised Struct
Etherscan Blockchain Explorer		
TestToken	0x3d8a10ce3228cb428cb56baa058d4432464ea25d	Hidden Transfer
WhaleGiveaway1	0x7a4349a749e59a5736efb7826ee3496a2dfd5489	Hidden Transfer
Gift_1_ETH	0xd8993f49f372bb014fb088eabec95cfdc795cbf6	Hidden State Update
NEW_YEARS_GIFT	0x13c547ff0888a0a876e6f1304eaefe9e6e06fc4b	Hidden State Update
G_GAME	0x3caf97b4d97276d75185aaf1dcf3a2a8755afe27	Hidden State Update
IFYKRYGE	0x1237b26652eebf1cb8f59e07e07101c0df4f60f6	Hidden State Update
EtherBet	0x3c3f481950fa627bb9f39a04bccdc88f4130795b	Hidden State Update
Private_Bank	0xd116d1349c1382b0b302086a4e4219ae4f8634ff	Straw Man Contract
firstTest	0x42db5bfe8828f12f164586af8a992b3a7b038164	Straw Man Contract
TransferReg	0x62d5c4a317b93085697cfb1c775be4398df0678c	Straw Man Contract
testBank	0x477d1ee2f953a2f85dbecbcb371c2613809ea452	Straw Man Contract

Table 5: List of publicly available honeypots on the Internet [45, 22, 32, 31, 33].

The Anatomy of a Cryptocurrency Pump-and-Dump Scheme

Jiahua Xu
École Polytechnique Fédérale de Lausanne (EPFL)
Imperial College London
Harvard University

Benjamin Livshits
Imperial College London
UCL Centre for Blockchain Technologies
Brave Software

Abstract

While pump-and-dump schemes have attracted the attention of cryptocurrency observers and regulators alike, this paper represents the first detailed empirical query of pump-and-dump activities in cryptocurrency markets. We present a case study of a recent pump-and-dump event, investigate 412 pump-and-dump activities organized in Telegram channels from June 17, 2018 to February 26, 2019, and discover patterns in crypto-markets associated with pump-and-dump schemes. We then build a model that predicts the pump likelihood of all coins listed in a crypto-exchange prior to a pump. The model exhibits high precision as well as robustness, and can be used to create a simple, yet very effective trading strategy, which we empirically demonstrate can generate a return as high as 60% on small retail investments within a span of two and half months. The study provides a proof of concept for strategic crypto-trading and sheds light on the application of machine learning for crime detection.

1 Introduction

While pump-and-dump schemes are a well-trodden ruse in conventional financial markets, the old-fashioned ploy has found a new playground to thrive — cryptocurrency exchanges.

The relative anonymity of the crypto space has led to it becoming a fertile ground for unlawful activities, such as currency theft (e.g. the DAO hack [1]), Ponzi schemes [26], and pump-and-dump schemes that have each risen in popularity in cryptocurrency markets over the last few years. Due to their end-to-end encryption, programmability, and relative anonymity, new social media tools such as Telegram[1] and Discord have become cryptocurrency enthusiasts' preferred communication vehicles. While pump-and-dump schemes have been discussed in the press [29], we are not aware of a comprehensive study of this phenomenon to date.

[1] Note that not all Telegram traffic is end-to-end encrypted.

Regulation: In February 2018, the CFTC (Commodity Futures Trading Commission) issued warnings to consumers [8] about the possibility of cryptocurrency pump-and-dump schemes. It also offered a substantial reward to whistleblowers around the same time [12].

In October 2018, the SEC (Securities and Exchange Commission) filed a subpoena enforcement against an investment company trust and trustee for an alleged pump-and-dump ICO scheme [27].

Clearly, regulators are aiming to find perpetrators of pump-and-dump schemes and to actively prosecute them.

This paper: In this paper, we trace the message history of over 300 Telegram channels from June 17, 2018 to February 26, 2019, and identify 412 *pump events* orchestrated through those channels. We analyze features of pumped coins and market movements of coins before, during, and after pump-and-dump. We develop a predictive random forest model that provides the likelihood of each possible coin being pumped *prior* to the actual pump event. With an AUC (area under curve) of the ROC (receiver operating characteristic) curve of over 0.9, the model exhibits high accuracy in predicting pump-and-dump target coins.

Contributions: This paper makes the following contributions:

- **Longitudinal study:** This paper is the *first* research study that examines routinely organized pump-and-dump events in the cryptocurrency space. We use a unique dataset of pump-and-dump records from June 17, 2018 to February 26, 2019 across multiple crypto-exchanges and analyze crypto-market movements associated with those pump-and-dump events.

- **Analysis:** Our analysis shows that pump-and-dump activities are a lot more prevalent than previously believed. Specifically, around 100 organized Telegram pump-and-dump channels coordinate on average 2 pumps a day, which generates an aggregate artificial trading volume of 6 million USD a month. We discover that some ex-



Figure 1: A successfully organized pump event. On the right hand side of the screenshot is the message history of a Telegram channel. The first message is the final countdown; the second message is the coin announcement; the last message presents the pump result. On the left hand side is the market movement of the corresponding coin around the pump time.

changes are also active participants in pump-and-dump schemes.

- **Prediction:** We develop machine learning models that, given pre-pump market movements, can predict the likelihood of each coin being pumped with an AUC (Area Under Curve) of over 0.9 both in-sample and out-of-sample. The models confirm that market movements contain hidden information that can be utilized for monetary purposes.

- **Trading strategy:** We formulate a simple trading strategy which, when used in combination with a calibrated prediction model, demonstrates a return of 60% over a period of three weeks, even under strict assumptions.

Paper organization: The paper is structured as follows. In Section 2 we provide background information on pump-and-dump activities organized by Telegram channels. In Section 3 we present a pump-and-dump case study. In Section 4 we investigate a range of coin features. In Section 5 we build a prediction model that estimates the pump likelihood of each coin for each pump, and propose a trading strategy along with the model. In Section 6 we summarize the related literature. In Section 7 we outline our conclusions. Finally, the Appendix specifies parameters of the models we have used in this paper.

2 Background

A pump is a coordinated, intentional, short-term increase in the demand of a market instrument — in our study, a cryptocurrency — which leads to a price hike. With today's chat applications such as Telegram and Discord offering features of encryption and anonymity, various forms of misconduct in cryptocurrency trading are thriving on those platforms.

2.1 Pump-and-Dump Actors

Pump organizer: Pump organizers can be individuals, or, more likely, organized groups, typically who use encrypted chat applications to coordinate pump-and-dump events. They have the advantage of having insider information and are the ultimate beneficiaries of the pump-and-dump scheme.

Pump participants: Pump participants are cryptocurrency traders who collectively buy a certain coin immediately after receiving the instruction from the pump organizer on which coin to buy, causing the price of the coin to be "pumped". Many of them end up buying coins at an already inflated price and are the ultimate victim of the pump-and-dump scheme.

Pump target exchange: A pump target exchange is the exchange selected by the pump organizer where a pump-and-dump event takes place. Some exchanges are themselves directly associated with pump-and-dump. Yobit, for example, has openly organized pumps multiple times (see Figure 2). The benefits for an exchange to be a pump organizer are threefold:

1. With coins acquired before a pump, it can profit by dumping those coins at a higher, pumped price;

2. It earns high transaction fees due to increased trading volume driven by a pump-and-dump;

3. Exchanges are able to utilize their first access to users' order information for front-running during a frenzied pump-and-dump.

2.2 A Typical Pump-and-Dump Process

Set-up: The organizer creates a publicly accessible group or channel, and recruits as many group members or channel subscribers as possible by advertising and posting invitation links on major forums such as Bitcointalk, Steemit, and Reddit.

Telegram *channels* only allow subscribers to receive messages from the channel admin, but not post discussions in the channel. In a Telegram *group*, members can by default post messages, but this function is usually disabled by the group admin to prohibit members' interference. We use the terms *channel* and *group* interchangeably in this paper.

Pre-pump announcement: The group is ready to pump once it obtains enough members (typically above 1,000). The pump organizer, who is now the group or channel admin, announces details of the next pump a few days ahead. The admins broadcast the exact time and date of the announcement of a coin which would then precipitate a pump of that coin. Other information disclosed in advance includes the exchange where the pump will take place and the pairing coin[2]. The admins advise members to transfer sufficient funds (in the form of the pairing coin) into the named exchange beforehand.

While the named pump time is approaching, the admin sends out countdowns, and repeats the pump "rules" such as: 1) buy fast, 2) "shill"[3] the pumped coin on the exchange chat box and social media to attract outsiders, 3) "HODL"[4] the coin at least for several minutes to give outsiders time to join in, 4) sell in pieces and not in a single chunk, 5) only sell at a profit and never sell below the current price. The admin also gives members a pep talk, quoting historical pump profits, to boost members' confidence and encourage their participation.

Pump: At the pre-arranged pump time, the admin announces the coin, typically in the format of an OCR (optical character recognition)-proof image to hinder machine reading (Figure 1). Immediately afterwards, the admin urges members to buy and hold the coin in order to inflate the coin price. During the first minute of the pump, the coin price surges, sometimes increasing several fold.

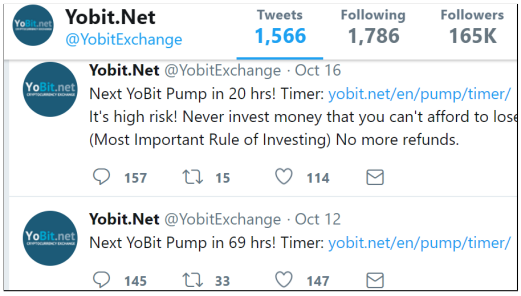
Dump: A few minutes (sometimes tens of seconds) after the pump starts, the coin price will reach its peak. While the admin might shout "buy buy buy" and "hold hold hold" in the channel, the coin price keeps dropping. As soon as the first fall in price appears, pump-and-dump participants start to panic-sell. While the price might be re-boosted by the second wave of purchasers who buy the dips (as encouraged by channel admins), chances are the price will rapidly bounce back to the start price, sometimes even lower. The coin price declining to the pre-pump proximity also signifies the end of the dump, since most investors would rather hold the coin than sell at a loss.

Post-pump review: Within half an hour, after the coin price and trading volume recover to approximately the pre-pump levels, the admin posts a review on coin price change, typically including only two price points – start price (or low price) and peak price, and touts how much the coin price increased by

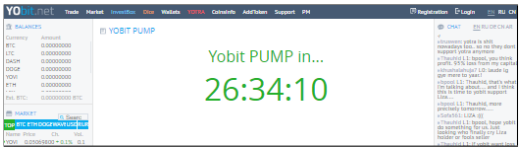
[2]A pairing coin is a coin that is used to trade against other coins. Bitcoin (BTC) is a typical pairing coin.

[3]Crypto jargon for "advertise", "promote".

[4]Crypto jargon for "hold".



(a) Tweets from `@YobitExchange`.



(b) Pump timer from the Yobit website.

Figure 2: The screen-shots demonstrate that the exchange Yobit was actively involved in pump-and-dump activities.

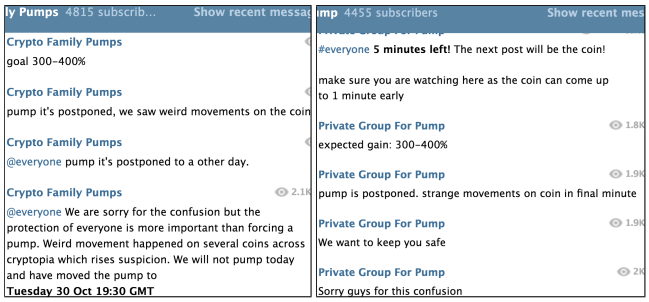


Figure 3: A pump attempt coordinated by multiple channels not executed due to unanticipated price movement of the to-be-pumped coin.

the pump (Section 2). Information such as trading volume and timescale is only selectively revealed: if the volume is high, and the pump-and-dump lasts a long time (over 10 minutes, say, would be considered "long"), then those stats will be "proudly" announced; if the volume is low or the time between coin announcement and price peak is too short (which is often the case), then the information is glossed over. Such posts give newcomers, who can access channel history, the illusion that pump-and-dumps are highly profitable.

Failed pump-and-dump attempts: Note that not every pump attempt is successful. Figure 3 shows that the admins decided not to carry through a pre-announced pump due to unanticipated price movements of the to-be-pumped coin.

While it is unknown what caused these movements, the case evidences that the admin is aware of the coin choice before the pump (as opposed to the coin being randomly selected and immediately announced at the pump time purely by algorithm), and hence has the time advantage of hoarding the coin at a low price before the coin announcement, whereas group members only purchase the coin after the coin announcement and slow buyers risk acquiring the coin at an

already (hyper)inflated price. It is generally known to pump participants that admins benefit the most from a pump. So why are there still people enthusiastic about partaking a pump, given the risk of being ripped off by the admins? Because people believe that they can sell those coins at an even higher price to other "greater fools". The greater fool theory also forms the foundation of many other schemes, such as pyramid scams or Ponzi games [5].

One may also hypothesize that in this case, someone might have worked out the pattern of the coin selection and pre-purchased a basket of coins with high pump likelihood that happens to contain the actual to-be-pumped coin, which might explain why the admin observed peculiar movements of the coin. In the next section, we study the features of pumped coins and their price movements to understand if it is indeed possible to predict the to-be-pumped coin.

2.3 Regulatory and Ethical Considerations

Pump-and-dumps in the stock market nowadays typically involve penny stock manipulation employing deceptive campaigns on social media to amass gains and are deemed criminal [27]. However, since many cryptocurrencies cannot be neatly classified as investment or consumer products [22], the applicability of certain securities laws might be ambiguous, and to date, regulation of pump-and-dumps in the cryptocurrency market is still weak [23].

Yet, the crypto-market is likely to be considered subject to common law and general-purpose statues even though it has not been clearly regulated as either a securities market or a currency market. While offenses of market manipulation can depend on a defined market, outright fraud and deception do not. As pump-and-dump admins create information asymmetry by not showing investors the full picture of their scheme, they intentionally mislead investors for their own financial benefit. As a consequence, when it comes to US legislation, for instance, admins might be committing false advertising under the FTC (Federal Trade Commission) Act (15 USC §45) [15] or fraudulent misrepresentation. Of course, practically speaking, these admins are frequently outside of the US jurisdiction.

Pump-and-dump admins, aiming to profit from price manipulation, are certainly unethical. Nevertheless, other pump-and-dump participants are also culpable since their behaviour enables and reinforces the existence of such schemes; ironically, most participants become the victim of their own choices.

3 A Pump-and-Dump Case Study

We further study in depth the pump-and-dump event associated with Figure 1. The pump-and-dump was organized by at least four Telegram channels, the largest one being **Official McAfee Pump Signals**, with a startling 12,333 members. Prior to the coin announcement, the members were notified

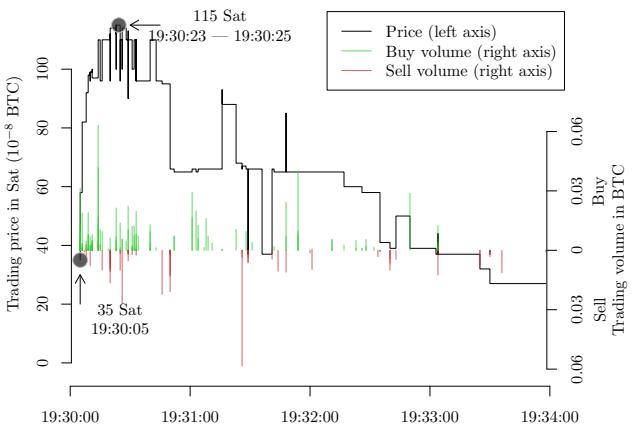


Figure 4: Tick-by-tick movement of the BVB/ BTC market during the first four minutes after the coin announcement.

that the pump-and-dump would take place on one of the Cryptopia's BTC markets (i.e., BTC is the pairing coin).

Announcement: At 19:30 GMT, on November 14, 2018, the channels announced the target coin in the form of a OCR-proof picture, but not quite simultaneously. **Official McAfee Pump Signals** was the fastest announcer, having the announcement message sent out at 19:30:04. **Bomba bitcoin "cryptopia"** was the last channel that broadcast the coin, at 19:30:23.

The target coin was BVB, a dormant coin that is not listed on CoinMarketCap. The launch of the coin was announced on Bitcointalk on August 25, 2016.[5] The coin was claimed to be have been made by and for supporters of a popular German football club, Borussia Dortmund (a.k.a. BVB). The last commit on the associated project's source code on GitHub was on August 10, 2017.[6]

Although it has an official Twitter account, `@bvbcoin`, its last Tweet dates back to 31 August, 2016. The coin's rating on Cryptopia is a low 1 out of possible 5. This choice highlights the preference of pump-and-dump organizers for coins associated with unserious projects.

During the first 15 minutes of the pump, BVB's trading volume exploded from virtually zero to 1.41 BTC (illustrated by the tall grey bar towards the right end of the price/volume chart), and the coin price increased from 35 Sat[7] to its three-fold, 115 Sat (illustrated by the thin grey vertical line inside the tall grey bar).

Price fluctuations: Further dissecting the tick by tick transactions (Figure 4), we note that the first buy order was placed and completed within 1 second after the first coin announcement. With this lightning speed, we conjecture that such an order might have been executed by automation. After a mere 18

[5] https://bitcointalk.org/index.php?topic=1596932.0
[6] https://github.com/bvbcoin/bvbcoin-source
[7] One Satoshi (Sat) equals 10^{-8} Bitcoin (BTC).

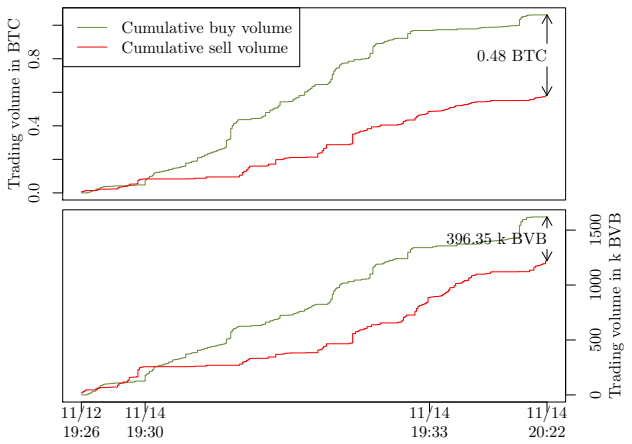


Figure 5: Gap between buy volume and sell volume caused by the BVB pump-and-dump. The figure shows a timeline from 48 hours before up to 1 hour after the pump-and-dump. For the illustration purposes, the timeline is scaled with non-linear transformation to better display the development of volume gaps during the pump-and-dump.

seconds of a manic buying wave, the coin price already skyrocketed to its peak. Note that **Bomba bitcoin "cryptopia"** only announced the coin at the time when the coin price was already at its peak, making it impossible for investors who solely relied on the announcement from the channel to make any money.

Not being able to remain at this high level for more than a few seconds, the coin price began to decrease, with some resistance in between, and then plummeted. Three and half minutes after the start of the pump-and-dump, the coin price had dropped below its open price. Afterwards, transactions only occurred sporadically.

Volume: Figure 5 shows that the pump-and-dump induces fake demand and inflates buy volume. While every pump-and-dump participant would hope for a quick windfall gain during a minute-long pump, the majority would not manage to act fast enough to sell at a high price. Those investors would either end up selling coins at a loss, or, if reluctant to sell low, would hold the virtually worthless coins. This is demonstrated by Figure 5, which shows that the buy volume exceeds the sell volume, whether measured by the target coin BVB or by BTC. The figure also shows small volume movements shortly before the pump-and-dump, also observable in Figure 4(a), which can be indicative of organizers' pre-purchase conduct. As the BVB blockchain is not being actively maintained and the coin itself is extremely illiquid, any market movement may be deemed unusual.

Figure 5 illustrates that the total buy volume (also including the pre-purchased volume, though negligible) in BTC associated with the pump-and-dump amounts to 1.06 BTC, the sell volume only 0.58 BTC; the total buy volume measured in BVB is 1,619.81 thousand BVB, the sell amount 1,223.36 thousand BVB. This volume discrepancy between the sell and

Exchange	Volume (30d)	No. markets	Launch	Country
Binance	$21,687,544,416	385	Jul 2017	China
Bittrex	$1,168,276,090	281	Feb 2014	U.S.A.
Cryptopia	$107,891,577	852	May 2014	New Zealand
YoBit	$797,593,680	485	Aug 2014	Russia

Figure 6: Exchanges involved in pump-and-dump schemes, sorted by 30-day volume: No. markets is the number of trading pairs (eg. DASH/BTC, ETC/USDT) in the exchange. Volume and No. markets were extracted from CoinMarketCap on November 5, 2018.

the buy sides indicates a higher trading aggressiveness on the buy side.[8] This further suggests that many investors may be "stuck" with BVB which they are unwilling to liquidate at the low market price after the pump-and-dump. Those coin holders can only expect to reverse the position in the next pump, which might never come.

Low participation ratio: It is worth noting that the total count of trading transactions associated with this pump-and-dump is merely 322. That number appears very low compared to the 1,376 views of the coin announcement message, let alone the over 10,000 channel members. This indicates that the majority of group members are either observers, who want no skin in the game, or have become aware of the difficulty in securing profit from a pump-and-dump.

4 Analyzing Pump-and-Dump Schemes

In this section we explain how we obtain data from both Telegram and the various exchanges, which allows us to analyze and model pump-and-dump schemes.

4.1 Collecting Pump-and-Dump Events

In this study, we examine routinely organized pump-and-dump events that follow the pattern of "set-up → pre-pump announcement → pump → dump → post-pump review" as described in Section 2. This type of pump-and-dump involves live instructions from organizers (see Figure 1 and Figure 3), so encrypted chat applications such as Telegram and Discord are ideal for broadcasting those events.

We are confident that it suffices to focus solely on pump-and-dump events orchestrated on Telegram as every active pump-and-dump group we found on Discord was also on Telegram.[9] Telegram is among the primary media for pump-and-dump activities and announcements, and it would be both unreasonable and unlikely for any pump-and-dump organizer

[8] Note that Cryptopia is a peer-to-peer trading platform which lets users trade directly with each other; the exchange takes no risk position and only profits from charging trading fees. Therefore, buying volume implies that the trade is initiated by the buyer, which typically drives the market price up; similarly, sale volume is initiated by the sell side and would drive the price down.

[9] This observation has also been confirmed by the PumpOlymp team, an online information provider specialized in cryptocurrency pump-and-dump.

to restrict the platform to only Discord, since the key to the success of a pump-and-dump is the number of participants.

Telegram channels: Our primary source on pump-and-dump Telegram channels and events is provided by PumpOlymp,[10] a website that hosts a comprehensive directory of hundreds of pump-and-dump channels.

PumpOlymp discovers those channels by searching pump-related keywords — e.g. "pump", "whales", "vip" and "coin" — on Telegram aggregators such as https://tgstat.com/ and https://telegramcryptogroups.com/. Another source for new pump-and-dump channels is cross-promotion on the known channels.[11] To validate the incoming data from PumpOlymp, we conduct an independent manual search for pump-and-dump channels. We are not able to add new channels to the existing channel list from PumpOlymp, and we are not aware of any other, more comprehensive pump-and-dump channel list. Therefore, we believe the channel list from PumpOlymp is a good starting point.

Next, we use the official Telegram API to retrieve message history from those channels, in total 358, to check their status and activity. Among those channels, 43 have been deleted from the Telegram sever, possibly due to inactivity for an extended period of time. Among the existing ones, over half (168/315) have not been active for a month, possibly because cautious admins delete pump-and-dump messages to eviscerate their traces. This might also imply that the Telegram channels have a "hit-and-run" characteristic. As described in the section above, one learns from participation in pump-and-dump activities that quick bucks are not easy to make. Therefore, curious newcomers might be fooled by pump-and-dump organizers' advertising and lured into the activity. After losing money a few times, participants may lose faith and interest, and cease partaking. This forms a vicious circle, since with fewer participants, it would be more difficult to pump a coin. Therefore, channel admins might desert their channel when the performance declines, and start new ones to attract the inexperienced.

Pump-and-dump history: Starting June 2018, PumpOlymp has been gleaning pump-and-dump events organized on Telegram. Using their API,[12] we acquire an initial list of historical pump-and-dump activities over the period of June 17, 2018 and February 26, 2019. For each listed pump-and-dump event, the data set contains the pumped coin, the target exchange, the organizing Telegram channel, the coin announcement time, plus the price and volume data on the tick-by-tick level from coin announcement up to 15 minutes afterwards.

We run plausibility checks to validate each record's qualification as a pump-and-dump. For example, if an alleged pump-and-dump is recorded to have started at a time that is far from a full hour (6:00, 7:00, etc.) or a half hour, then we would be suspicious, because an organizer would normally not choose a random time for a pump-and-dump. If there is no significant increase in volume or high price around the pump time, we would also be skeptical. In such a circumstance, we manually check the message history to make a final judgment. In most cases, the message either discusses the potential of a coin or the record is simply a mistake. Note that we exclusively consider message series with count-downs (e.g. "3 hours left", "5 mins left") and coin announcement; messages on pump signal detection are eliminated from our sample.

In the end, we trace 429 pump-and-dump coin announcements from June 17, 2018 to February 26, 2019, each of which is characterized by a series of messages similar to those presented in Figure 1. One pump-and-dump can be co-organized by multiple channels; if two coin announcements were broadcast within 3 minutes apart from each other and they target the same coin at the same exchange, then we consider them to be one pump-and-dump event. In total, we collected 412 unique pump-and-dump events.

Excluded data points: All the pumped coins in our sample were paired with BTC. We also observed and manually collected a few ETH-paired pumps, most of which took place in other exchanges.[13] Inclusion of those cases would require data collection with other methods and resources. Due to their rarity, we do not consider ETH-paired pump-and-dumps in our study.

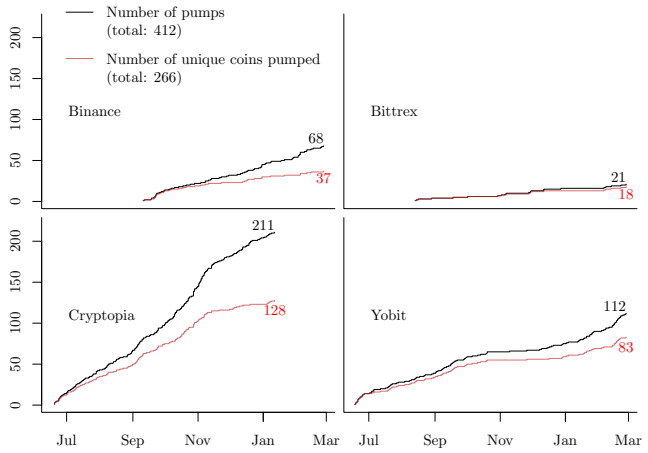


Figure 7: Cumulative counts of pumps and pumped coins on four exchanges from June 2018 to February 2019.

4.2 Obtaining Coin Data

Apart from consulting the online pump-and-dump information center PumpOlymp, we retrieve additional information on features and price movements of coins from other sources,

[10] https://pumpolymp.com

[11] This is based on a conversation with a PumpOlymp staff member.

[12] https://pumpolymp.com:5001/api/allPumps and https://pumpolymp.com:5001/api/PumpMarketHistory/raw, only available for premium users.

[13] For example, PLX on October 10, 2018 in CoinExchange, ETC on April 22, 2018 in Bibox.

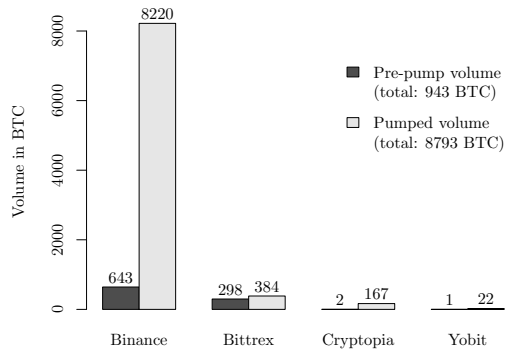


Figure 8: Aggregate trading volume of pumped coins before and during a pump.

in order to establish a connection between the information and the pump-and-dump pattern.

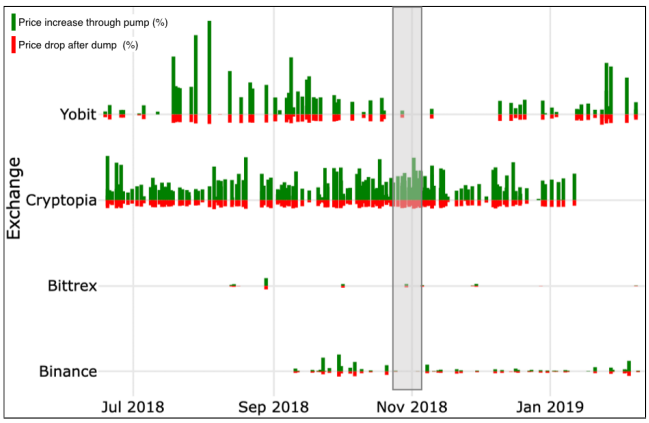
Specifically, we use the public API from CryptoCompare[14] for coins' hourly OHLC (open, high, low, close) and volume data on 189 exchanges, including Binance, Bittrex, Cryptopia and Yobit. The API provides live data, which means users are able to obtain price information up to the time point of data retrieval. While historical minute-level data are also available on CryptoCompare, they are restricted to a 7-day time window and thus not utilized.

In the conventional stock market, pump-and-dump operators favor microcap stocks due to high manipulability of their price [3]; we expect to observe a similar phenomenon in the crypto-market. To collect coins' market cap data, we use the public API from CoinMarketCap. Because we are interested in coins' "true" market cap that is uninfluenced by any maneuver, we purposefully chose to retrieve the data at 08:42 GMT, November 5. We believe the market cap data retrieved are not contaminated by Telegram organized pump-and-dumps, since they typically start on the hour or the half hour and last only a few minutes.

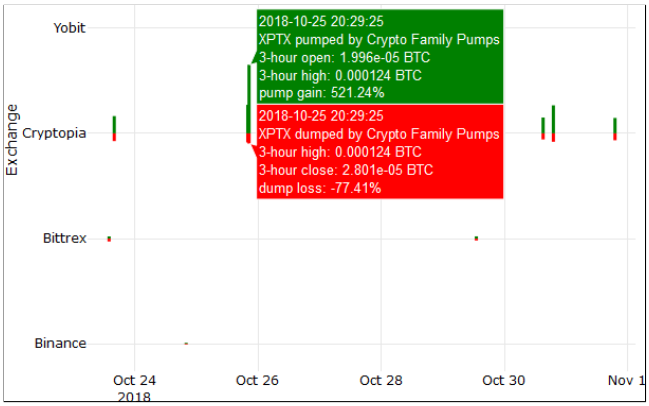
In addition to market trading data, we also retrieve coins' non-financial features. Specifically, we use exchanges' public API[15] to collect information on coins' listing status, algorithm, and total supply. We also collect coins' launch dates using CryptoCompare's public API. For information that is not contained in the API but viewable online (such as coins' rating data on Cryptocurrency), we use either page source scraping or screen scraping, depending on the design of the desired webpage. All our data on coin features are from publicly accessible sources.

[14] https://min-api.cryptocompare.com/

[15] https://api.binance.com/api/v1/ticker/allPrices for Binance, https://bittrex.com/api/v1.1/public/getcurrencies for Bittrex, https://www.cryptopia.co.nz/api/GetCurrencies for Cryptopia, and https://yobit.net/api/3/info for Yobit.



(a) Pump and dump activities from June 2018 to February 2019



(b) Enlarged section of the highlighted area in (a) that shows one of the most recent pump-and-dump

Figure 9: Pump and dump timeline. A green bar represents price increase through pump, calculated as $\frac{\text{high price} - \text{open price}}{\text{open price}}$; a red bar represents price drop after pump, calculated as $\frac{\text{close price} - \text{high price}}{\text{close price}}$. All prices are denominated in BTC, and from a 3-hour window around pump activities. Visit http://rpubs.com/xujiahuayz/pd for the full, interactive chart.

4.3 Role of Exchanges

Pump-and-dump schemes take place within the walled gardens of crypto-exchanges. Binance, Bittrex, Cryptopia, and Yobit are among the most popular exchanges used by pumpers (see Figure 6). While those exchanges differ vastly in terms of their volume, markets, and user base, each of them has its own appeal to pumpers. Large exchanges such as Binance and Bittrex have a large user base, and abnormal price hype caused by pump activities can quickly attract a large number of other users to the exchange. Smaller exchanges such as Cryptopia and Yobit tend to host esoteric coins with low liquidity, whose price can be more easily manipulated compared to mainstream coins such as Ether (ETH) or Litecoin (LTC).

In general, larger exchanges are more reliable than smaller ones. While both Binance and Cryptopia were hacked recently,[16] the former managed to remain operative, while the

[16] https://www.bloomberg.com/news/articles/2019-05-08/crypto-exchange-giant-binance-reports-a-hack-of-7-000-bitcoin

Exchange	Number of PD's	Admins' profit (BTC), aggregated	Admins' return, aggregated
Binance	51	148.97	15%
Bittrex	15	0.92	7%
Cryptopia	180	44.09	57%
Yobit	102	5.54	52%
Total	**348**	**199.52**	**18%**

Table 1: Number of pump-and-dumps (348) considered in this analysis deviates from the total number of pump-and-dumps (412) due to lack of price data for some events.

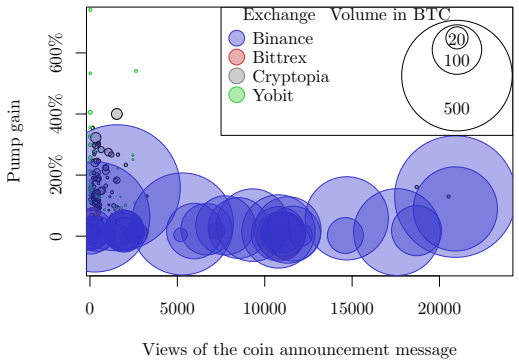


Figure 10: Views of coin announcement message versus coin price increase during the pump. The figure illustrates the relationships between coin price increase through pump, views of coin announcement message, pump volume, and pump exchange.

latter halted trading and fell into liquidation.

Activity distribution by exchange: Among the 412 pump-and-dump activities, 68 (17%) took place in Binance, 21 (5%) in Bittrex, 211 (51%) in Cryptopia and 112 (27%) in Yobit. In aggregate, 35% (146/412) of the time, the selected coin had previously been pumped in the same exchange (see Figure 7).

Figure 8 compares the aggregate three-hour trading volume in BTC of pumped coins before and during a pump-and-dump, and the artificial trading volume generated by those pump-and-dump activities is astonishing: 8,793 BTC (93% from Binance), roughly equivalent to 50 million USD,[17] of trading volume during the pump hours, 9 times as much as the pre-pump volume (943 BTC), and that only over a period of eight months.

Figure 9 illustrates the occurrence and the effectiveness of individual pump-and-dump activities. In terms of frequency, Bittrex is most rarely chosen; Binance started to gain traction only since September, but still witnesses far less pump-and-dump occurrence than Yobit and Cryptopia. Turning to Yobit with Cryptopia, we find that the two exchanges have complemented each other: when Yobit was inactive (most notably October 2018 to January 2019), Cryptopia experienced more traffic; when Cryptopia went silent (since the hack in mid-January 2019), Yobit regained popularity. In terms of percentage of coin price increase, pumps in both Yobit and Cryptopia appear to be more powerful than those in Bittrex and Binance. What goes hand-in-hand with price surge is price dip: coin prices also drop more dramatically during the dump in Yobit and Cryptopia compared to their peer exchanges.

Profit for admins: Even with tick-by-tick data for each pumped coin during their respective pump-and-dump period, due to lack of trader ID we cannot precisely match individuals' buy and sell transactions. Therefore, to estimate profit for admins, we need to make a few assumptions:

1. Admins purchase coins and enter sell orders only prior to the pump.
2. Admins purchase coins at the price immediately before the pump begins.
3. During the pump period — before the price reaches the peak, investors lift the admin's offers and push the price higher; during the dump period — when the price drops, investors transact with each other.

With those assumptions, we arrive at the estimation as presented in Table 1. We estimate that admins made a net profit of 199.52 BTC, equivalent to 1.1 million USD, through 348 pump and dump events during our sample period. The estimated return of insiders averages 18%, which aligns perfectly with Li et al. [23].

So, what is the investors' payout? Some investors win; others lose. Since trading is a zero-sum game, the aggregate investor loss would be on the equivalent scale as the aggregate admin win.

Coin announcement views: While investigating the degree of exposure in coin announcement messages distributed by Telegram channels, we find a negative correlation (-0.162) between number of views of coin announcement and pump gain, which is rather counter-intuitive, because one would think that more views would indicate more participation, which would result in higher pump gain. Two extreme examples: the coin announcement of the pump on MST had 325 views and the pump gain was 12.6%; another coin announcement of the pump on PARTY had only 18 views, and the pump gain was a whopping 533.3%.

This finding suggests that the number of views cannot accurately proxy number of participants, possibly because: (1) only a fraction of message viewers would actually participate in a pump-and-dump; (2) if a user reads the message history after the pump, his/her view would still be counted; (3) if a user re-views a message 24 hours after his/her first view, the user's view would be counted twice;[18] (4) some participants

and https://www.nzherald.co.nz/business/news/article.cfm?c_id=3&objectid=12231209.

[17] This is calculated based on the unit BTC price of 5,715 USD, which is the mean of the high price of 8,250 USD and the low price 3,180 USD during the data period.

[18] https://stackoverflow.com/questions/42585314/telegram-channels-post-view-count

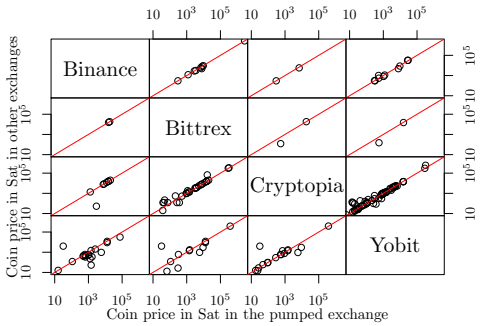


Figure 11: Arbitrage opportunities: coin price (highest during the pump hour) in pumped exchange versus price in other exchanges

might have retrieved messages via bots, which would not be counted in number of views.[19]

Price increase: We further notice that although pump-and-dumps in Binance generate more trading volume during the pump hour (Figure 8),[20] thanks to its large user base, coin price increase through pumps is generally at a much smaller scale than that in Cryptopia and Yobit (Figure 9 and Figure 10). This is possibly caused by high bid and sell walls on the order book that are typical for large crypto exchanges like Binance, which prevent the price from fluctuating significantly even at coordinated pump-and-dump events.

Arbitrage: Pump-and-dump activities not only engender abnormal returns within the pumped exchange, but also arbitrage opportunities across different exchanges. Figure 11 shows the presence of a price discrepancy of the same coin during the pump hour across different exchanges. Interestingly, coin price can sometimes be higher in exchanges other than the pumped one. It is also worth noting that most coins pumped in Cryptopia are also listed in Yobit but not in Bittrex or Binance, and vice versa. This is because the former two have more conservative coin listing strategies, which results in a different, more mainstream portfolio of listed coins compared to the latter two. While there may be trading strategies resulting from these arbitrage opportunities, they are outside the scope of this work.

4.4 Capturing Features

Market cap: Figure 12 presents the market cap distribution of coins pumped in different exchanges. Pumped coins' market cap ranges from 1 BTC (Royal Kingdom Coin (RKC), pumped in Cryptopia) to 27,600 BTC (TrueUSD (TUSD), pumped in Yobit). Half of those coins have a market cap below 100 BTC, most of which were pumped in Cryptopia.

[19] https://stackoverflow.com/questions/49704911/is-it-possible-for-a-telegram-bot-increase-post-view-count

[20] A pump hour refers to the clock hour during which a pump occurs.

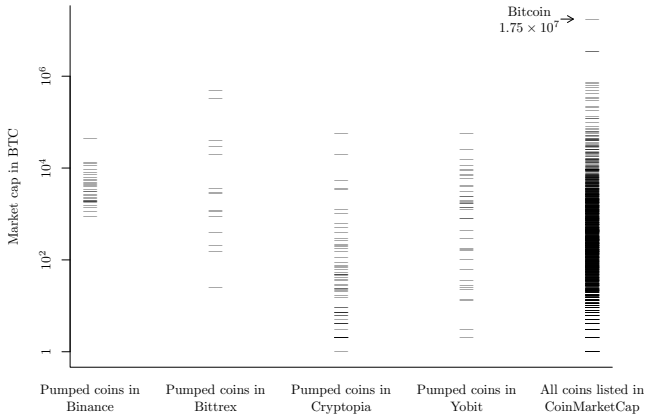


Figure 12: Distribution of coin market caps. Market cap information was extracted from CoinMarketCap on November 5, 2018.

Pump-and-dump organizers' preference for small-cap coins resembles equity market manipulators' taste for microcap stocks [3, 24], and can be explained by the empirical finding of Hamrick et al. [18] and Li et al. [23]: the smaller the market cap of the pumped coin, the more successful the pump would be.

Price movement: Figure 13 depicts time series of hourly log returns of pumped coins between 48 hours before and 3 hours after a pump. We detect anomalous return signals before pump-and-dump admins' announcement of the pumped coin. The signals appear most jammed one hour prior to the pump, and less so before that. This is to a certain degree in accord with Kamps et al. [20] who find that a shorter, 12-hour rolling estimation window is more suitable for anomaly detection in the crypto-market than a longer, 24-hour one.

The return signal before the pump is the strongest with Cryptopia, where in numerous pumps, coin prices were elevated to such an extent that the hourly return before the pump even exceeds the hourly return during the pump. This can be explained by the assumption that pump organizers might utilize their insider information to purchase the to-be-pumped coin before the coin announcement, causing the coin price elevation and usual return volatility before the pump. The analysis above provides grounds for predicting the pumped coin before coin announcement using coin features and market movement.

5 Predicting Pump-and-Dump Target Coins

5.1 Feature Selection

Based on the preliminary analysis in the last section, we believe pump-and-dump organizers have specific criteria for coin selection and they generally purchase the to-be-pumped coin before naming it to the investors. Thus, it should be possible to use coin features and market movements prior to a

Feature	Description	Notation
Market cap	Market cap information extracted from CoinMarketCap at 08:42 GMT, November 5, 2018 when no pump-and-dump activity in Telegram channels was observed *	*caps*
Returns before pump	x-hour log return of the coin within the time window from $x+1$ hours to 1 hour before the pump	$return[x]h$ †
Volumes in coin before pump	Total amount of the coin traded within the time window from $x+1$ hours to 1 hour before the pump	$volumefrom[x]h$ †
Volumes in BTC before pump	Total trading volume of the coin measured in BTC within the time window from $x+1$ hours to 1 hour before the pump	$volumeto[x]h$ †
Return volatilities before pump	Volatility in the hourly log return of the coin within the time window from $y+1$ hours to 1 hour before the pump	$returnvola[y]h$ ‡
Volume volatilities in coin before pump	The volatility in the hourly trading volume in coin within the time window from $y+1$ hours to 1 hour before the pump	$volumefromvola[y]h$ ‡
Volume volatilities in BTC before pump	The volatility in the hourly trading volume in BTC within the time window from $y+1$ hours to 1 hour before the pump	$volumetovola[y]h$ ‡
Last price before pump	Open price of the coin one hour before the coin announcement	*lastprice*
Time since existence	The time difference between the time when the first block of the is mined and the pump time	*age*
Pumped times before	Number of times the coin been pumped in Cryptopia before	*pumpedtimes*
Coin rating	Coin rating displayed on Cryptopia, 0 being the worst, 5 being the best. The rating considers the following criteria wallet on {Windows, Linux, Mac, mobile, web, paper}, premine ratio, website and block explorer	*rating*
Withdrawal fee	Amount of coin deducted when withdrawing the coin from Cryptopia	*WithdrawFee*
Minimum withdrawal	Minimum amount of coin that can be withdrawn from Cryptopia	*MinWithdraw*
Maximum withdrawal	Daily limit on the amount of coin that can be withdrawn from Cryptopia	*MaxWithdraw*
Minimum base trade	Minimum base trade size of the coin	*MinBaseTrade*

Table 2: Features included in the prediction model. *The feature is designed to represent a coin's market cap in a normal setting, i.e. absent market manipulation. While it might be useful to also collect coins' historical market cap before each pump-and-dump, we have not found a public source that provides this type of data. †$x \in \{1,3,12,24,36,48,60,72\}$. ‡$y \in \{3,12,24,36,48,60,72\}$.

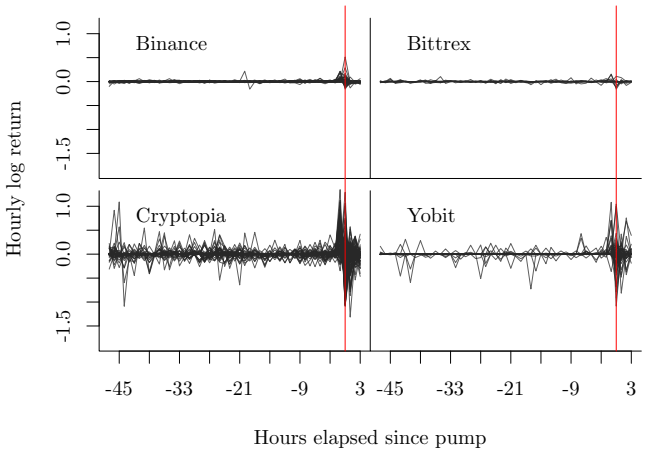


Figure 13: Time series of coin returns before and after pump. In each subplot, the hourly log return of each pumped coin before and shortly after the pump is superimposed. The vertical red line represents the pump hour during which the coin was announced.

coin announcement to predict which coin might be pumped.

In the following exercise, we focus on predicting coins pumped in one specific exchange for the ease of data harmonization. We choose Cryptopia due to sufficient data collected for modelling. Although the exchange ceased to operate on May 15, 2019, our exercise demonstrates a proof of concept for strategic crypto-trading that can be adapted for any exchange.

For each coin before a pump event, we predict whether it will be pumped (`TRUE`) or not (`FALSE`). The formula for the prediction model is:

$$Pumped = M(feature_1, feature_2, \dots)$$

where the dependent variable *Pumped* is a binary variable that equals 1 (`TRUE`) when the coin is selected for the pump, and 0 (`FALSE`) otherwise. Table 2 lists the features considered in the prediction model.

Previous analyses indicate unusual market movements prior to the pump-and-dump might signal organizers' pre-pump behavior, which could consequently give away the coin selection information. Therefore, we place great emphasis on features associated with market movements, such as price, returns and volatilities covering various lengths of time. Those features, 46 in total, account for 85% of all the features considered.

5.2 Model Application

Sample specification: We consider all the coins listed on Cryptopia at each pump-and-dump event. On average, we have 296 coin candidates at each pump, out of which one is the actual pumped coin. The number of coins considered varies for each event due to constant listing/delisting activities on the part of exchanges. The full sample contains 53,208 pump-coin observations, among which 180 are pumped cases,[21] accounting for 0.3% of the entire sample population. Apparently, the sample is heavily skewed towards the unpumped class and needs to be handled with care at modelling.

For robustness tests, we split the whole sample into three chronologically consecutive datasets: training sample, validation sample and and test sample:

Pumped?	Training	Validation	Test	Total	
TRUE	60	60	60	180	(0.3%)
FALSE	17,078	17,995	18,135	53,028	(99.7%)
Total	17,138	18,055	18,195	53,208	(100.0%)

The training sample covers the period of June 19, 2018 to September 5, 2018 and consists of 17,138 data points (32.2% of full sample); the validation sample covers September 5, 2018 to October 29, 2018 and consists of 18,055 data points (33.9% of full sample); the test sample covers October 29, 2018 to January 11, 2019 and consists of 18,195 data points (34.2% of full sample).

Model selection: We test both classification and logit regression models for the prediction exercise. Specifically, for the classification model, we choose random forest (RF) with stratified sampling; for the logit regression model, we apply generalized linear model (GLM). Both RF and GLM are widely adopted in machine learning and each has its own quirks.

RF is advantageous in handling large quantities of variables and overcoming overfitting issues. In addition, RF is resilient to correlations, interactions or non-linearity of the features, and one can be agnostic about the features. On the flip side, RF relies upon a voting mechanism based on a large number of bootstrapped decision trees, which can be time-consuming, and thus challenging to execute. In addition, RF provides information on feature importance, which is less intuitive to interpret than coefficients in GLM.

GLM is a highly interpretable model [28] that can uncover the correlation between features and the dependent variable. It is also highly efficient in terms of processing time, which is a prominent advantage when coping with large datasets. However, the model is prone to overfitting when fed with too many features, which potentially results in poor out-of-sample performance.

[21] Due to missing data on several delisted coins, this number deviates from the total number of 211 pump events in Cryptopia, as presented in Figure 7.

Hyperparameter specification: Due to the heavily imbalanced nature of our sample, we stratify the dataset when using RF [9], such that the model always includes TRUE cases when bootstrapping the sample to build a decision tree. Specifically, we try the following three RF variations:

Model	Sample size per tree			Number of trees
	TRUE	FALSE	Total	
RF1	60	20,000	20,060	5,000
RF2	60	5,000	5,060	10,000
RF3	60	1,000	1,060	20,000

We fix the number TRUEs at 60 for each RF variation, so that the model may use the majority of TRUEs to learn their pattern when building each tree. Model RF1 stays loyal to our sample's original TRUE/FALSE ratio, with 0.3% of TRUEs contained in each tree-sample. RF2 and RF3 raise the TRUE/FALSE ratio to 1.2% and 6%, respectively. Note that while the sample size per tree decreases from RF1 to RF2 to RF3, we are mindful to increase the number of trees accordingly to ensure that whichever model we use, every input case is predicted a sufficient number of times. We use the R package randomForest to model our data with RF1, RF2 and RF3.

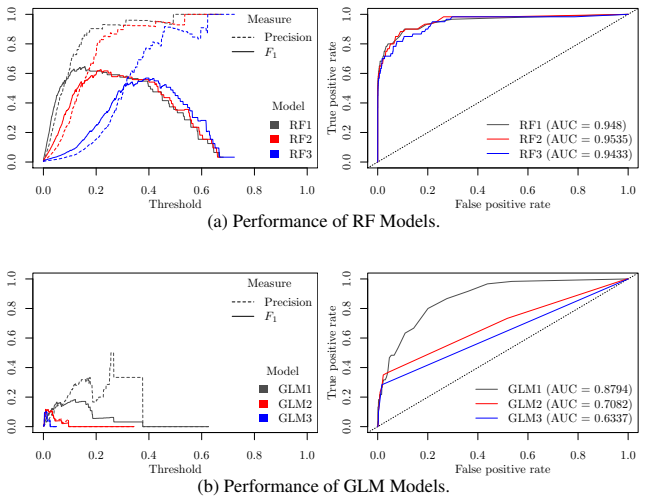
With conventional binomial GLM, problems can arise not only when the dependent variable has a skewed distribution, but also when features are skewed. With heavy-tailed coin price distribution and market cap distribution, conventional binomial GLM can be insufficient to handle our sample. Therefore, we apply LASSO (least absolute shrinkage and selection operator) regularization to the GLM models. After preliminary testing, we choose to focus on three representative LASSO-GLM models with various shrinkage parameter values (λ):

Model	Shrinkage parameter (λ)
GLM1	10^{-8}
GLM2	10^{-3}
GLM3	5×10^{-3}

Higher values of λ causes elimination of more variables. We use the R package glmnet to model our data with GLM1, GLM2, and GLM3.

Variable assessment: By applying the specified models on the training sample, we are able to assess the features' relevance to coin prediction. Figure 17 presents features' importance based on mean decrease in Gini coefficient with RF models. We find that:

- Coin market cap *caps* and last hour return before the pump *return1h* appear to be the two most important features in predicting pumped coin using RF models.
- Features describing market movements shortly before the pump, e.g. *return1h*, *volumeto1h* and *volumefrom1h*, appear to be more important than features describing longer-term movements.



(a) Performance of RF Models.

(b) Performance of GLM Models.

Figure 14: Model performance on the training sample measured by Precision, F_1 (left) and ROC AUC (right) at different threshold levels.

- Among all the features related to market movements, return features are generally more important than volume or volatility features.
- Exchange-specific features including *MinBaseTrade*, *MinWithdraw*, *MaxWithdraw*, and *WithdrawFee* are least important.

Figure 18 presents the estimated coefficients of variables with GLM models, from which we obtain several findings in line with what is indicated by RF models above. Specifically, we notice that:

- When only one variable is included, *return1h* appears to have the highest explanatory power on coins' pump likelihood;
- The positive coefficients of return features imply that the higher the return a coin shows before the pump, the more likely the coin is to be pumped;
- The positive coefficient of *pumpedtimes* implies that pumped coins are more likely to get pumped again.

The variable assessment performed by RF and GLM is coherent in that both find features representing market movement shortly before the pump to be more important than longer-term features. This echoes our exploratory analysis illustrated in Figure 13 and aligns with Kamps et al. [20]. The finding suggests the spontaneity of admins' coin selection, and the importance for strategic traders to obtain real-time market data.

5.3 Assessing Prediction Accuracy

Both the random forest model and GML predict whether a given coin will be pumped as a likelihood ranging between 0 and 1. We apply thresholding to get a binary `TRUE`/`FALSE` answer.

Figure 14 depicts the in-sample fitting of model candidates with the training sample as the threshold value changes. The fitting measurements include precision, the F_1 measure and area under ROC (Receiver operating characteristic) curve. Figure 14(a) describes the performance of RF models and Figure 14(b) GLM models.

Precision represents the number of true positive divided by number of predicted positive, and the precision line ends when the denominator equals zero, i.e. when no `TRUE` prediction is produced. Figure 14 shows that, among the three RF models, the threshold value at which the line ends is the lowest with RF1, and highest with RF3. This indicates that absent balanced bootstrapping, an RF model tends to systematically underestimate pump likelihood, leading to zero predicted `TRUE` cases even when the threshold value is small.

Compared to RF models, none of the GLM models is able to produce high precision.

In terms of F_1 measure, RF models again appear superior to GLM models. Among the three RF models, the RF1 performs best at a low threshold range (< 0.2), while RF3 performs best at a high threshold range (> 0.4). RF2 resides in between.

The RF models' superiority to GLM models is further demonstrated by the ROC (Receiver operating characteristic) curve in Figure 14. Among the three RF models, no discernible difference can be found in terms of ROC AUC: all exhibit high performance with AUC > 0.94. The GLM models, in contrast, render an AUC between 0.63 and 0.88.

Due to their obvious inferiority, we eliminate GLM models from further analysis. Figure 15 illustrates the out-of-sample performance of RF models. The model performance with the validation sample resembles that of the training sample, remaining strong with regard to all three indicators (precision, F_1 and AUC). This suggests that the classification model trained and calibrated on one period of data can accurately predict a later period.

Both Figure 14(a) and Figure 15 suggest that balancing the sample with various `TRUE`/`FALSE` ratios only changes the *absolute* value of the pump likelihood output, but not the *relative* one. This means the three RF models can perform similarly in terms of Precision and F_1 measure, when the appropriate threshold value is chosen in correspondence with the model (specifically, $\text{Treshold}_{\text{RF1}} < \text{Treshold}_{\text{RF2}} < \text{Treshold}_{\text{RF3}}$).

5.4 Testing an Investment Strategy

To explore the model's practical utility, we devise a simple investment strategy. At each pump, we check which coin's predicted pump likelihood surpasses a predetermined threshold, and we purchase all those coins before the actual coin announcement (if no coin's vote exceeds the threshold, we will not pre-purchase any coin). Note that if we had the ability to short or use margin trading on the exchanges we use,

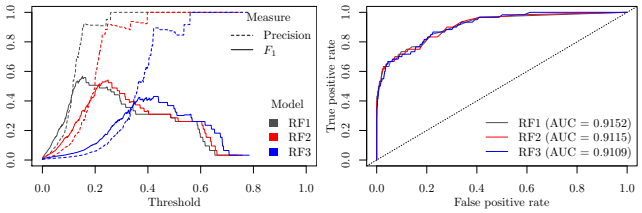
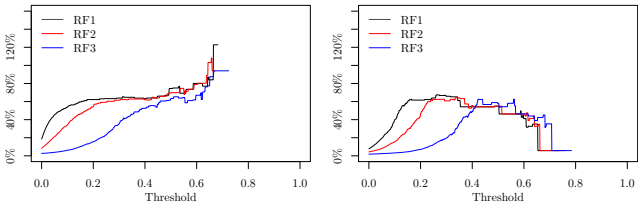


Figure 15: Performance of RF models on the validation sample measured by Precision, F_1 (left) and ROC AUC (right) at different threshold levels.



(a) Training sample. (b) Validation sample.

Figure 16: Investment return using different models at different threshold levels.

potentially more options would open up for us.

Strategy: Specifically, for each coin that we pre-purchase, we buy the coin at the open price one hour before the coin announcement with the amount of BTC equivalent to k times the vote where k is a constant. That is to say, with all the coins we purchase, the investment, measured in BTC, on each coin is proportionate to its vote supplied by the random forest model. This is logical because a higher vote implies a higher likelihood of being pumped, and thus worth a higher investment.

We further assume that among all the coins we purchased, those coins that do not get pumped (false positive, "false alarms") will generate a return of zero, i.e. their price will remain at the same level as the purchase price; those coins that get pumped (true positive, "hits") will be sold at an elevated price during the pump. To be conservative, we assume that with each purchased coin that gets pumped we only obtain half of the pump gain, expressed as:
$\text{pump gain} = \frac{\text{high price} - \text{open price}}{\text{open price}}$.

Returns: Figure 16 presents the relationship between the aggregate return and the threshold choice.

Figure 16(a) illustrates the performance of the trading strategy with the training sample. The figure shows that, in general, the *higher* the threshold, which means we buy coins with higher pump likelihoods and disregard others, the *higher* the return.

Figure 16(b) illustrates the performance of the trading strategy with the validation sample. As the threshold increases, the return first increases and then decreases. This is because the coins with the highest predicted pump likelihood in the validation sample happen to have very low pump gain. When the threshold is high, only those coins with high likelihood but low gain are included in the investment portfolio, resulting in a low overall return.

		Predicted		
		TRUE	FALSE	Total
Actual	TRUE	9	51	60
	FALSE	0	18,135	18,135
	Total	9	18,186	18,195

Table 3: Confusion matrix of RF1 with threshold value 0.3 applied to test sample.

As already mentioned at the end of Section 5.3, every model has its own optimal threshold value. In terms of the magnitude of the profit, with the right combination of threshold and model, investors would theoretically enjoy a return of 140% with the training sample cases (RF1 with threshold of 0.7), and a return of 80% with the validation sample cases (RF1 with threshold of 0.3).

One should be mindful that if the threshold is set *too high* (e.g., greater than 0.8), then the investor might end up not buying any coins, and consequently gaining no profit. In addition, although high threshold comes with high precision, it also leads to a low number of coins being purchased, increasing the risk associated with an undiversified investment portfolio, as demonstrated in Figure 16(b).

5.5 Final Test

Based on the training and validation results of specified models, we need to select one model and an accompanying threshold value to apply to the test sample. Our ultimate goal to maximize the trading profit using the selected model in combination with the proposed trading strategy on a set of out-of-sample data. Therefore, we base our decision primarily on Figure 16(b). We apply RF1 and a threshold of 0.3 — the combination that delivers the highest return in Figure 16(b) — on our test sample.

To determine the investment amount in BTC for our trading strategy, we need to examine the market depth. This is particularly important for exchanges with low trading volume such as Cryptopia and Yobit. When trading in those exchanges, it has to be ensured that during the pump-and-dump, the market would provide sufficient depth for us to liquidate the coins purchased prior to the pump. For example, if the total trading volume in one event is 0.4 BTC, it would make no sense to spend 0.8 BTC on the coin.

To this end, we calculate the average trading volume per pump-and-dump at Cryptopia. We only consider "uptick" transactions, i.e. where the buyer is the aggressor. This yields a ballpark estimation of the market depth on the buy side. We use this number, 0.37 BTC, as the baseline investment quantity. This baseline amount, discounted by the predicted pump likelihood, would be the investment value in BTC.

Coin	Date	Pumped?	weight wt	BTC invested $q = \overline{Q} \times wt$	Pump gain pg	Assumed gain $ag = pg/2$	BTC gained $q \times ag$
BVB	Nov 14	TRUE	0.30	0.11	283%	142%	0.16
CON	Nov 16	TRUE	0.44	0.16	33%	17%	0.03
FLAX	Nov 10	TRUE	0.58	0.21	135%	67%	0.14
MAGN	Nov 13	TRUE	0.37	0.14	70%	35%	0.05
MAGN	Dec 16	TRUE	0.39	0.14	85%	43%	0.06
OSC	Nov 13	TRUE	0.65	0.24	297%	148%	0.36
OSC	Nov 25	TRUE	0.52	0.19	100%	50%	0.10
SOON	Nov 01	TRUE	0.58	0.21	10%	5%	0.01
UMO	Nov 15	TRUE	0.55	0.20	60%	30%	0.06
				1.61			**0.96**

Table 4: Purchased coins based on pump likelihood predicted by RF1. Only coins with predicted pump likelihood of greater than 0.3 are purchased. Investment weight equals pump likelihood. $\overline{Q} = 0.37$, the average of total transaction volume in a pump-and-dump event in Cryptopia. Only transaction volume where the buyer is the aggressor is considered.

Table 3 displays the confusion matrix of the model prediction with the test sample. The model suggests us to purchase 9 coins in total, all of which are ultimately pumped. Table 4 lists those 9 coins, their respective investment weight and assumed profit. The return on the investment amounts to 60% (2.61/4.38) over the test sample period of two and a half months. Note that the effect of transaction fees (0.2% on Cryptopia) on the investment profitability is negligible. The result of the final test is very similar to that with both the training sample and the validation sample when the same combination of model (RF1) and threshold (0.3) is applied (Figure 16), confirming the model's robustness.

5.6 Caveats and Improvement Potential

Data: Upon availability, order book data, tick-by-tick data before a pump and traders' account information can also be included as features.

Modelling method: Random forest with unsupervised anomaly detection has the potential to improve the model performance. In addition, other classification (e.g. k-NN) and regression (e.g. ridge) models are worth considering.

Additional considerations: Regarding investment weights, one may consider coin price increase potential (based on e.g. historical returns) in combination with coin pump likelihood. One must beware that in liquid exchanges, the trading strategy only applies to *small* retail investment, since big purchase orders prior to a pump can move the market, such that pump organizers may cancel the pump or switch the coin last-minute. Also worth factoring in is the market risk (e.g. security risk, legal risk) associated with the nascent crypto-market.

6 Related Work

Over the past year, a handful of studies researching cryptocurrency pump-and-dump activities have been conducted, notably Kamps et al. [20] , Li et al. [23] and Hamrick et al. [18]. Our work differs from the aforementioned studies in terms of motivation, methodology, data, and contribution. We aim for *prospective* prediction as opposed to *retrospective* investigation of pump-and-dump activities. We use a homogeneous set of data that only includes clearly announced pump-and-dump events on Telegram.[22] Regarding the sample period, our data cover a recent time span of June 17, 2018 to February 26, 2019 (Table 5).

Our paper is also closely linked to literature on market manipulation in non-cryptocurrency contexts. Lin [24] explains potential damage of various manipulation methods including pump-and-dump, front running, cornering and mass misinformation, and argues for swift regulatory action against those threats. Austin [3] calls for authorities' demonstration of their ability to effectively deter market manipulation such as pump-and-dump in exchanges for small-capped companies, in order to recover investors' confidence in trading in those markets, which would consequently foster economic growth.

Our paper is further related to research on crypto trading. Gandal et al. [17] demonstrate that the unprecedented spike in the USD-BTC exchange rate in late 2013 was possibly caused by price manipulation. Makarov et al. [25] probe arbitrage opportunities in crypto markets. Aune et al. [2] highlight potential manipulation in the blockchain market resulting from the exposure of the footprint of a transaction after its broadcast and before its validation in a blockchain, and proposes a cryptographic approach for solving the information leakage problems in distributed ledgers.

Our paper is also akin to existing literature on cryptocurrencies' market movements. The majority of related literature still orients its focus on Bitcoin. Many scholars use GARCH models to fit the time series of Bitcoin price. Among them, Dyhrberg et al. [13] explore the financial asset capabilities of Bitcoin and suggests categorizing Bitcoin as something between gold and US Dollar on a spectrum from pure medium of exchange to pure store of value; Bouoiyour et al. [7] argue that Bitcoin is still immature and remains reactive to negative rather than positive news at the time of their writing; 2 years later, Conrad et al. [10] present the opposite finding that negative press does not explain the volatility of Bitcoin; Dyhrberg [14] demonstrates that bitcoin can be used to hedge against stocks; Katsiampa [21] emphasizes modelling accuracy and recommends the AR-CGARCH model for price retro-fitting. Bariviera et al. [4] compute the Hurst exponent by means of the Detrended Fluctuation Analysis method and conclude that the market liquidity does not affect the level of long-range dependence. Corbet et al. [11] demonstrate that Bitcoin shows characteristics of an speculative asset rather than a currency also with the presence of futures trading in Bitcoin.

Among the few research studies that also look into the financial characteristics of other cryptocurrencies, Fry et al. [16]

[22] As suggested earlier, all the coin announcements we found on Discord overlap with our Telegram data

	Kamps et al. [20]	Hamrick et al. [18]	Li et al. [23]	This paper
Motivation	Locating suspicious transactions patterns through automated anomaly detection	Identifying success factors for historical pumps	Examining how pump-and-dumps are correlated with cryptocurrency price	Predicting the coin to be pumped with input of Telegram signals
Methodology	Breakout indicators & reinforcers	Ordinary least squares (OLS)	OLS, difference in difference	RF, GLM
Data	Market data of cryptocurrencies from April 2018 to May 2018 on Binance, Bittrex, Kraken, Kucoin and Lbank	Explicit (with coin announcement) and suspected (no coin announcement) pump-and-dumps from January 2018 to July 2018	Pump-and-dump events from May 2017 to August 2018 on Binance, Bittrex, and Yobit, with a focus on Bittrex	Pump-and-dump events from June 2018 to February 2019 on Binance, Bittrex, Cryptopia and Yobit, with a focus on Cryptopia
Main finding / contribution	The authors develop a defining criteria set for detecting suspicious activity like pump-and-dumps.	Pumping obscure, small-market-cap coins is more likely to be successful.	Pump-and-dumps are detrimental to the liquidity and price of cryptocurrencies.	Pump-and-dumps schemes can be found and foiled by machine learning.

Table 5: Comparison of studies on cryptocurrency pump-and-dump.

examine bubbles in the Ripple and Bicoin markets; Baur et al. [6] investigate asymmetric volatility effects of large cryptocurrencies and discover that in the crypto market positive shocks increase the volatility more than negative ones. Jahani et al. [19] assess whether and when the discussions of cryptocurrencies are truth-seeking or hype-based, and discover a negative correlation between the quality of discussion and price volatility of the coin.

7 Conclusions

This paper presents a detailed study of pump-and-dump schemes in the cryptocurrency space. We start by presenting the anatomy of a typical attack and then investigate a variety of aspects of real attacks on crypto-coins over the last eight months on four crypo-exchanges. The study demonstrates the persisting nature of pump-and-dump activities in the crypto-market that are the driving force behind tens of millions of dollars of phony trading volumes each month. The study reveals that pump-and-dump organizers can easily use their insider information to profit from a pump-and-dump event at the sacrifice of fellow pumpers.

Through market investigation, we further discover that market movements prior to a pump-and-dump event frequently contain information on witch coin will be pumped. Using LASSO regularized GML and balanced random forests, we build various models that are predicated on the time and venue (exchange) of a pump-and-dump broadcast in a Telegram group. Multiple models display high performance across all subsamples, implying that pumped coins can be predicted based on market information. We further propose a simple but effective trading strategy that can be used in combination with the prediction models. Out-of-sample tests show that a return of as high as 60% over two and half months can be consistently exploited even under conservative assumptions.

In sum, we wish to raise the awareness of pump-and-dump schemes permeating the crypto-market through our study. We show that with fairly rudimentary machine learning models, one can accurately predict pump-and-dump target coins in the crypto-market. As such, we hope our research could, on one hand, lead to fewer people falling victim to market manipulation and more people trading strategically, and on the other hand, urge the adoption of new technology for regulators to detect market abuse and criminal behavior. If such advice would be heeded, admins' schemes would crumble, which would in turn lead to a healthier trading environment, accelerating the market towards a fairer and more efficient equilibrium.

References

[1] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A Survey of Attacks on Ethereum Smart Contracts. *Proceedings of the 6th International Conference on Principles of Security and Trust*, 10204:164–186, 2017.

[2] Rune Tevasvold Aune, Adam Krellenstein, Maureen O'hara, and Ouziel Slama. Leakage in Distributed Ledgers Trading and Information Footprints on a Blockchain. *Journal of Trading*, 12(3):5–13, 2017.

[3] Janet Austin. How Do I Sell My Crowdfunded Shares? Developing Exchanges and Markets to Trade Securities Issued by Start-Ups and Small Companies. *Harvard Business Law Review*, 8:21–35, 2018.

[4] Aurelio F. Bariviera, María José Basgall, Waldo Hasperué, and Marcelo Naiouf. Some stylized facts of the Bitcoin market. *Physica A: Statistical Mechanics and its Applications*, 484:82–90, 2017.

[5] Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, and Roberto Saia. Dissecting Ponzi schemes on Ethereum: identification, analysis, and impact. 2017.

[6] Dirk G. Baur and Thomas Dimpfl. Asymmetric volatility in cryptocurrencies. *Economics Letters*, 173:148–151, 2018.

[7] Jamal Bouoiyour and Refk Selmi. Bitcoin: A beginning of a new phase? *Economics Bulletin*, 36(3):1430–1440, 2016.

[8] CFTC. Customer Advisory: Beware Virtual Currency Pump-and-Dump Schemes. 2018.

[9] Chao Chen and Andy Liaw. Using Random Forest to Learn Imbalanced Data. Technical report, 2004.

[10] Christian Conrad, Anessa Custovic, and Eric Ghysels. Long- and Short-Term Cryptocurrency Volatility Components: A GARCH-MIDAS Analysis. *Journal of Risk and Financial Management*, 11(2):23, 5 2018.

[11] Shaen Corbet, Brian Lucey, Maurice Peat, and Samuel Vigne. Bitcoin Futures – What use are they? *Economics Letters*, 172:23–27, 2018.

[12] Crypto Insider. CFTC offers $100,000+ bounty for crypto pump and dump whistleblowers.

[13] Anne Haubo Dyhrberg. Bitcoin, gold and the dollar - A GARCH volatility analysis. *Finance Research Letters*, 16:85–92, 2015.

[14] Anne Haubo Dyhrberg. Hedging capabilities of bitcoin. Is it the virtual gold? *Finance Research Letters*, 16:139–144, 2016.

[15] Federal Reserve. Federal Trade Commission Act Section 5: Unfair or Deceptive Acts or Practices, 2016.

[16] John Fry and Eng-Tuck Cheah. Negative bubbles and shocks in cryptocurrency markets. *International Review of Financial Analysis*, 47:343–352, 2016.

[17] Neil Gandal, JT Hamrick, Tyler Moore, and Tali Oberman. Price manipulation in the Bitcoin ecosystem. *Journal of Monetary Economics*, 95(4):86–96, 2018.

[18] JT Hamrick, Farhang Rouhi, Arghya Mukherjee, Amir Feder, Neil Gandal, Tyler Moore, and Marie Vasek. The Economics of Cryptocurrency Pump and Dump Schemes. In *Workshop on the Economics of Information Security*, 2019.

[19] Eaman Jahani, Peter M. Krafft, Yoshihiko Suhara, Esteban Moro, and Alex Pentland. ScamCoins, S*** Posters, and the Search for the Next Bitcoin™: Collective Sensemaking in Cryptocurrency Discussions. *Proceedings of the ACM on Human-Computer Interaction*, 2:79, 2018.

[20] Josh Kamps and Bennett Kleinberg. To the moon: defining and detecting cryptocurrency pump-and-dumps. *Crime Science*, 7(1):18, 2018.

[21] Paraskevi Katsiampa. Volatility estimation for Bitcoin: A comparison of GARCH models. *Economics Letters*, 158:3–6, 2017.

[22] Jiasun Li and William Mann. Initial Coin Offering and Platform Building. 2018.

[23] Tao Li, Donghwa Shin, and Baolian Wang. Cryptocurrency Pump-and-Dump Schemes. 2019.

[24] Tom C.W. Lin. The New Market Manipulation. *Emory Law Journal*, 66:1253–1314, 2016.

[25] Igor Makarov and Antoinette Schoar. Trading and Arbitrage in Cryptocurrency Markets. 2018.

[26] SEC. Ponzi schemes Using virtual Currencies Ponzi Schemes Generally. *Investor Alert*, (153), 2017.

[27] SEC. SEC Files Subpoena Enforcement Against Investment Company Trust and Trustee for Failure to Produce Documents. 2018.

[28] Lin Song, Peter Langfelder, and Steve Horvath. Random generalized linear model: a highly accurate and interpretable ensemble predictor. *BMC bioinformatics*, 14:5, 2013.

[29] Oscar Williams-Grut. 'Market manipulation 101': 'Wolf of Wall Street'-style 'pump and dump' scams plague cryptocurrency markets. *Business Insider*, 2017.

Appendix

	RF1	RF2	RF3
caps	8.52	8.53	7.67
return1h	4.60	6.65	7.30
return3h	2.88	3.83	4.62
return12h	2.89	3.22	3.88
return24h	2.53	2.59	2.63
return36h	2.45	2.84	3.68
return48h	3.84	3.95	4.17
return60h	2.65	2.71	3.10
return72h	2.55	2.95	3.60
volumefrom1h	2.22	2.45	2.84
volumefrom3h	1.53	1.34	1.21
volumefrom12h	1.58	1.41	1.14
volumefrom24h	1.70	1.60	1.38
volumefrom36h	1.85	1.68	1.37
volumefrom48h	1.84	1.69	1.41
volumefrom60h	1.93	1.81	1.53
volumefrom72h	1.95	1.87	1.66
volumeto1h	2.64	3.27	3.19
volumeto3h	1.85	1.87	1.60
volumeto12h	1.86	1.70	1.45
volumeto24h	2.23	2.07	1.79
volumeto36h	2.42	2.18	1.87
volumeto48h	2.31	2.19	1.86
volumeto60h	2.40	2.30	2.01
volumeto72h	2.81	2.51	2.16
returnvola3h	2.02	2.30	3.07
returnvola12h	2.08	1.94	1.87
returnvola24h	2.17	1.96	1.73
returnvola36h	2.22	1.99	1.78
returnvola48h	2.44	2.10	1.69
returnvola60h	2.39	2.17	1.80
returnvola72h	2.30	2.09	1.67
volumefromvola3h	1.39	1.34	1.31
volumefromvola12h	1.57	1.42	1.16
volumefromvola24h	1.65	1.51	1.25
volumefromvola36h	1.75	1.55	1.21
volumefromvola48h	1.81	1.56	1.22
volumefromvola60h	1.79	1.56	1.25
volumefromvola72h	1.81	1.66	1.33
volumetovola3h	1.86	2.06	1.96
volumetovola12h	1.77	1.74	1.52
volumetovola24h	2.10	1.94	1.70
volumetovola36h	2.16	1.94	1.65
volumetovola48h	2.12	1.96	1.64
volumetovola60h	2.15	1.99	1.67
volumetovola72h	2.26	2.04	1.70
lastprice	2.14	2.02	1.66
age	2.20	1.88	1.69
pumpedtimes	1.31	1.65	2.52
rating	1.77	1.64	1.37
WithdrawFee	0.73	0.71	0.63
MinWithdraw	1.02	1.03	0.98
MaxWithdraw	0.43	0.36	0.28
MinBaseTrade	0.00	0.00	0.00

Figure 17: Features' importance indicated by mean decrease in Gini coefficient. Higher importance is marked by darker cell color.

	GLM1	GLM2	GLM3
caps	0.00	-	-
return1h	2.76	4.75	5.02
return3h	-0.04	-	-
return12h	1.08	-	-
return24h	-4.81	-	-
return36h	1.41	0.11	-
return48h	3.64	2.33	-
return60h	0.07	-	-
return72h	1.21	-	-
volumefrom1h	0.00	-	-
volumefrom3h	-0.00	-	-
volumefrom12h	-	-	-
volumefrom24h	-	-	-
volumefrom36h	-	-	-
volumefrom48h	0.00	-	-
volumefrom60h	-	-	-
volumefrom72h	-	-	-
volumeto1h	1.61	-	-
volumeto3h	5.99	-	-
volumeto12h	-	-	-
volumeto24h	-	-	-
volumeto36h	-	-	-
volumeto48h	-	-	-
volumeto60h	-2.88	-	-
volumeto72h	-0.49	-	-
returnvola3h	3.94	-	-
returnvola12h	4.41	-	-
returnvola24h	-9.39	-	-
returnvola36h	10.40	-	-
returnvola48h	9.10	-	-
returnvola60h	-12.57	-	-
returnvola72h	-3.93	-	-
volumefromvola3h	-0.00	-	-
volumefromvola12h	0.00	-	-
volumefromvola24h	-	-	-
volumefromvola36h	-	-	-
volumefromvola48h	-	-	-
volumefromvola60h	-	-	-
volumefromvola72h	-0.00	-	-
volumetovola3h	-7.46	-	-
volumetovola12h	1.32	-	-
volumetovola24h	-9.96	-	-
volumetovola36h	-2.13	-	-
volumetovola48h	18.83	-	-
volumetovola60h	-	-	-
volumetovola72h	8.65	-	-
lastprice	-91.74	-	-
age	0.00	-	-
pumpedtimes	0.69	0.66	-
rating	-0.16	-	-
WithdrawFee	-0.00	-	-
MinWithdraw	-0.00	-	-
MaxWithdraw	0.00	-	-
MinBaseTrade	-	-	-
(Intercept)	-5.43	-6.15	-5.95

Figure 18: Variable coefficients (unstandardized) using GLM. Coefficients of variables not selected by the model are shown as "-".

Inadvertently Making Cyber Criminals Rich: A Comprehensive Study of Cryptojacking Campaigns at Internet Scale

Hugo L.J. Bijmans
Delft University of Technology

Tim M. Booij
Delft University of Technology

Christian Doerr
Delft University of Technology

Abstract

Since the release of a browser-based cryptominer by Coinhive in 2017, the easy use of these miners has skyrocketed illicit cryptomining in 2017 and continued in 2018. This method of monetizing websites attracted website owners, as well as criminals seeking new ways to earn a profit. In this paper, we perform two large studies into the world of cryptojacking, focused on organized cryptomining and the spread of cryptojacking on the Internet. We have identified 204 cryptojacking campaigns, an order of magnitude more than previous work, which indicates that these campaigns are heavily underestimated by previous studies. We discovered that criminals have chosen third-party software – such as WordPress – as their new method for spreading cryptojacking infections efficiently. With a novel method of using NetFlow data we estimated the popularity of mining applications, which showed that while Coinhive has a larger installation base, CoinImp WebSocket proxies were digesting significantly more traffic in the second half of 2018. After crawling a random sample of 49M domains, ~20% of the Internet, we conclude that cryptojacking is present on 0.011% of all domains and that adult content is the most prevalent category of websites affected.

1 Introduction

Unlike traditional currencies, such as the Euro or Dollar, cryptocurrencies are digital assets created as a medium of exchange based on cryptography and a blockchain, which are used to secure both the creation and transactions of units. In 2009, Satoshi Nakamoto released the Bitcoin [33], the first ever decentralized cryptocurrency, which made it possible to transfer monetary value to another person by creating a transaction and committing this to the blockchain, a list of blocks secured by cryptographic operations maintained by a peer-to-peer network of miners. These miners secure the blockchain by constantly collecting transaction data from the network and validating it by solving cryptographic challenges based on the previous block, the transaction and the receiver of the transaction. After validation, the confirmed transaction is inserted into the blockchain again in the form of a validated block. As a reward, the miner gets a (part of a) cryptocurrency. This network guarantees that only the rightful owner of a Bitcoin wallet can make transactions and prevents malicious actors from inserting false information into the blockchain.

Solving these cryptographic challenges as a miner has however become so difficult that Bitcoin cannot efficiently be mined anymore on regular PCs. Over the past years, over 4,000 other cryptocurrencies have been created, so-called alt-coins. One of them is Monero (XMR), launched in 2014 and nowadays the most popular cryptocurrency in browser-based mining [34]. In contrast to Bitcoin, Monero uses a private blockchain, meaning that while anybody can use it to make transactions, nobody is allowed to view them [47]. It also builds upon a different proof-of-work algorithm to validate its transactions, called CryptoNight, a fork of CryptoNote [43]. This algorithm is designed to be memory-hard and therefore requires a large set of bytes in memory to perform frequent read and write operations on. Simple consumer-grade CPUs have exactly that memory available at their processor caches, making this kind of mining the most efficient on regular consumer-grade hardware. To speed up the mining process, mining jobs can be distributed among individual miners in a mining pool. In such a pool, miners work together to mine new blocks and share the rewards. Work is distributed among miners in the pool based on the difficulty of the cryptographic challenge. As a consequence, powerful machines solve the more difficult puzzles, while low-end machines receive the easier ones. Rewards are shared according to the same principle. Mining pools closely monitor the submissions from their miners and state that they will block any wallet address after receiving evidence that a wallet is used for malware or botnet activities [28].

The introduction of alt-coins that by design can be effectively mined on regular PCs also made them an attractive target for cybercriminals. Both the private blockchain and the ASIC-resistant mining algorithm of Monero quickly made Monero one of the preferred choices. In addition to being

included in malware [37], there also exist implementations to perform *drive-by mining* or *cryptojacking*, where cryptocurrency is mined in the user's web browser while visiting a web site. While originally developed as an alternative mechanism to donate to the upkeep of a website in presence of now ubiquitous ad-blockers, many methods exist to maliciously apply browser-based mining: for example, criminals hack vulnerable websites to install mining scripts [3] or create malicious advertisements with cryptojacking code that are displayed on benign websites [30], but actors have also compromised routers [35] or setup malicious Wi-Fi networks [38] to inject cryptominers into their users' traffic.

Previous studies have performed surveys on the use of cryptominers across the most commonly visited websites and have identified groups of criminals installing cryptominers on a large number of domains for their own profit [22, 39]. It makes sense for a cyber criminal to lure as many users as possible into such mining, which could be accomplished not only by deploying the cryptojacking code into popular websites, but hacking a large number of websites or injecting a resource such as a common library that is used by a large number of unsuspecting websites. These individual installations are working together in a coordinated campaign, thus significantly increasing the profits of the criminal, but at the same time also indicating an elevated level of knowledge and sophistication of the adversary. The presence and extent of such coordination is however largely unknown.

In this paper, we address this gap and systematically investigate the coordination and collaboration of cryptojackers on the Internet and make the following four contributions:

- We are the first to systematically analyze the relationships between websites that perform cryptomining and the actors behind them. By this campaign analysis, we find the existence of massive installations. In fact, we have identified 3 times as much cryptojacking activity as [39] and the five largest campaigns we detected exceed the *total* size of cryptomining reported in [22].
- We show that the bulk of organized mining activity is the result of compromised (parts of) third-party software and that comparatively little organized activity is the result of hacked websites or an explicit choice to mine by the website owner.
- Through a survey of 1,136 top level domains and by comparing the installation base with actual mining traffic on the Internet using NetFlow data, we find that the most prominently installed miner is actually not the one that generates the most mining activity in practice. We also see that applications and attack vectors come and go, and that different TLD zones exhibit clear differences in mining application popularity.
- Estimating cryptojacking by solely crawling the Alexa Top 1M results is an overestimation of the size, as we see that cryptojacking activity is almost 6 times higher in that subset compared to the rest of the Internet.

To enable follow up research, we make our data and software publicly available at `https://www.cyber-threat-intelligence.com/cryptojacking-campaigns`.

2 Background

WebAssembly & asm.js To enable faster execution of code inside the browser, Mozilla developed *asm.js*, a technique for translating high-level languages, such as C and C++, into JavaScript to be used by the browser [29]. Multiple validation methods enable the JavaScript engine to compile this code ahead-of-time and improve execution speed. This technique made it possible to execute code faster inside the browser after its release in 2013.

WebAssembly (*Wasm*) is a more recently released scripting language developed by the World Wide Web consortium in 2017 and is able to compile high-level languages like C, C++ and Rust inside the browser to be used in web applications [50]. It runs in a sandbox within the browser and it aims to execute as fast as native machine code. Wasm is complementary to JavaScript, as it is being controlled by JavaScript code after its compilation.

The difference between asm.js and Wasm is the fact that the latter is compiled only once and is started directly at native speed, whereas code in asm.js is compiled and optimized at run time, therefore decreasing execution speed. Both techniques are supported by all four major browsers (Chrome, Firefox, Edge and Safari) and have drastically improved the execution speed of applications inside the browser, which made them very attractive for browser-based mining.

WebSockets & Stratum WebSockets is a HTML5 protocol providing two-way communication between the client and a server over a single TCP connection [52]. The protocol enables easy real-time data transfer without refreshing (a part of) the web page. Communication is done over the same TCP ports as the web browser, making it robust to strict firewall rules or other blocking.

Developers are free to define the format of messages sent over WebSocket connections. However, there is a protocol specifically designed for cryptomining communications: the Stratum Mining Protocol, a line-based protocol with messages encoded in plain-text JSON-RPC format [46]. Servers communicate with their clients using Stratum to authorize new miners in the pool, distribute jobs based on difficulty and retrieve found hashes from the miners. An example of a WebSocket connection using the Stratum protocol is given in Table 1.

Browser-based mining Triggered by the rise of CPU-mineable cryptocurrencies (such as Monero) and the rapid

WebSocket traffic frames
⇑ {"type":"**auth**", "params":{"site_key":"<site_key_of_website>", "type":"anonymous","user":null,"goal":0, "version":3000,"coin":"xmr"}}
⇓ {"type":"**authed**", "params":{"token":"<random_36_characters>", "hashes":0}}
⇓ {"type":"**job**", "params":{"blob":"<random_152_characters>", "job_id":"<random_28_characters>", "target":"ffffff01", "id":"<random_36_characters>", "algo":"cn","variant":"4","height":1808537}}
⇑ {"type":"**submit**", "params":{"job_id":"<random_28_characters>", "nonce":"377c32b8", "result":"<found_64_characters_hash>"}}
⇓ {"type":"**hash_accepted**", "params":{"hashes":128}}
⇓ {"type":"**job**", "params":{"blob":"<random_152_characters>", "job_id":"<random_28_characters>", "target":"ffffff01", "id":"<random_36_characters>", "algo":"cn","variant":"4","height":1808537}}

Table 1: Example of a WebSocket connection using the Stratum Mining Protocol to communicate with a mining pool

development of useful web standards (e.g.WebAssembly and the Stratum protocol), browser-based cryptomining gained an enormous momentum in the autumn of 2017. Coinhive, a German company, created an easy to use browser-based mining application as an alternative to advertisements [9, 23]. They provide a JavaScript library, an API and a WebSocket proxy infrastructure to developers to easily integrate a browser-based miner into their website and let their visitors mine for Monero. 70% of the mined Monero is transferred to the owner of the account, the remaining 30% is kept by Coinhive [10]. Soon after Coinhive released their miner application, similar ones appeared, such as Cryptoloot [11] and Coin-Have [6]. Nowadays, miner applications come and go, with various capabilities and usage fees, but Coinhive still has a prominent place in the cryptojacking landscape.

Overview of a cryptojacking attack Although different mining applications exist, all browser-based miners show great similarities. As depicted in Figure 1, the user visits the cryptomining website (1) and receives a valid HTTP response (2). The cryptomining website requests a JavaScript file (3), which controls the mining operation. This script first explores the host system, searches for the number of CPU threads available, downloads the WebAssembly mining script

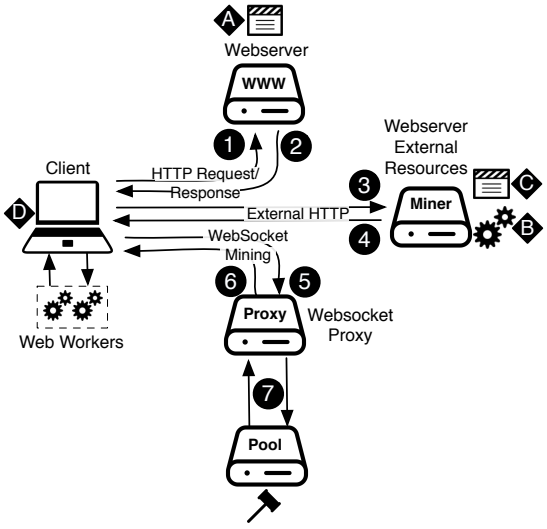


Figure 1: Browser-based cryptomining attack

for the actual mining operation (4) and distributes it over a number of WebWorkers (a JavaScript instance running in the background, without affecting the page performance). It also sets up a WebSocket connection with the mining pool through a proxy (5). The script authenticates itself to the mining pool server (in Stratum format) and, if successful, receives the first job to work on (6). The WebWorkers start working on that job and found hashes are submitted to the mining pool by the controller script (7).

Campaign analysis Campaign analysis is the field of research focused on discovering clusters of malicious online entities. The term originates from analysis of large volumes of SPAM or phishing emails, but can also be used in other areas, such as browser-based cryptomining. In this particular case, campaign analysis is focused on finding clusters of the same cryptominers on different domains. Since those miners always include a form of identification to which funds need to be transferred, clustering cryptojacking websites can be done relatively easily. Most mining applications define a *siteKey*, a unique (random) string used to identify the user to which earnings have to be transferred, which can be found in either the source code or the WebSocket traffic. A similar *siteKey* guarantees that the same account is rewarded for the mining that takes place. Identifying campaigns can also be done by searching for similar WebSocket proxy servers, if the website is not using a popular one, but instead hosting its own server. We have used these, and other techniques to discover campaigns as discussed in Section 6.1. We have chosen to define a cluster of websites as a campaign once they share identical features more than 5 times. E.g. a cluster of 6 websites with the same *siteKey* or private WebSocket proxy server is considered a campaign.

3 Attack vectors

Mining cryptocurrencies with the computing power of website visitors is not illegal, as long as users are asked permission to mine. When a user cannot consent to the mining activities their computer is involved in, it is called *cryptojacking*. Although browser-based cryptomining is a recent phenomenon, jurisdiction on cryptomining without consent already exists. In 2015, a US court settled a case with a developer of Bitcoin-mining software, in which the Attorney General stated that no website should tap into a person's computer processing power and that the user has to be informed about the cryptomining activities which take place on the visited website [18]. However, this is often not the case. In this section, we summarize the attack surface for cryptojacking on the Internet. All attack vectors are marked in Figure 1 by their corresponding characters.

Website owner (A) The owner of a website can add a cryptomining script to his web page without informing its users. This can be done as a replacement for advertisements, which was the case for The Pirate Bay, one of the most popular torrent websites [48]. Only a few days after the Coinhive service was launched, they added a miner to their website which started mining without user consent, as a replacement for the intrusive advertisements they would normally show. Nowadays, the website shows a disclaimer on the bottom of the homepage, notifying their visitors that their CPU will be used for cryptomining. Another major source of website owner initiated cryptojacking is parked domains [13].

Compromised websites (A) A cryptomining script can also be present on a web page without knowledge of the website owner. When a website gets hacked, an attacker is able to inject cryptomining scripts. Now, the attacker receives the rewards for the visitors mining on that website. There are numerous examples of this kind of attack. There have been cryptojacking scripts found on web pages of the Indian government [3], CBS Showtime [26] and many others.

Third-party software (B) Gaining unsolicited access to large number of domains is a time-consuming operation. As a consequence, attackers have tried different tactics to infect multiple websites at once by infecting third-party software. In the last year, we have seen attacks in which cryptojacking code is injected into popular third-party software, such as JQuery or Google Tag Manager [5]. Drupal, a widely used open-source CMS, was the victim of a large attack involving more than 100,000 websites [32] and WordPress, a similar CMS, suffered from a weather plugin [53] secretly injecting a cryptojacking script into the website it was installed on.

Malicious advertisements (C) Advertisement-supported websites let their advertisements be sold by advertisement networks, such as Google. The downside of this system is that attackers can attach cryptomining scripts to advertisements and distribute them through an advertisement network over a large number of websites. In January 2018, Youtube was a victim of this kind of attack, in which cryptomining scripts were injected in the ads shown on the website [30].

Man in the middle (D) The most effective method of gaining large groups of miners for an attacker is by being the man-in-the-middle. In August 2018, 200,000 MikroTik routers were infected by malware, which inserted a Coinhive script into every website the user visits [35]. The bug was patched within a day, but many MikroTik routers are not, leaving them still vulnerable. In our research, we are not able to detect these attacks, since they are not originating from a website.

4 Related Work

Academic research on browser-based cryptomining has only started in 2017 and is, due to the recent developments of the used web standards, very topically. The first explorations into this research field have been performed by Eskandari et al. [13]. In their analysis, the authors queried two large source code datasets for strings known to be part of cryptomining scripts (such as `coinhive.min.js` or `load.jsecoin.com`) and found a large number of domains. This method is only able to detect known mining applications, not the obfuscated or new ones. While calculating the profitability, the authors stumbled upon a Coinhive campaign which ran a miner on over 11,000 parked websites. This study kicked-off a number of subsequent investigations, which were all aimed at detecting browser-based cryptomining. Rauchberger et al. [39] created their *MiningHunter*, a crawler able to detect mining scripts even when their malicious activities are obfuscated. The detection method relied on analyzing executed JavaScript code and WebSocket traffic frames. After a successful crawl of the Alexa Top 1M in the beginning of December 2017, they were able to detect 3,178 websites running a cryptominer. 1,210 unique keys were retrieved and one large campaign involving 1,116 websites infected by a malicious advertisement network was identified. At the same time Parra Rodriguez et al. [40] worked on *RAPID*, a resource and API-based detection method, which is able to detect browser-based cryptomining and is resistant to JavaScript obfuscation. Their classification was able to classify mining samples with a precision of 96%. Eventually 656 actively mining websites were found in the Alexa Top 330,550. A similar classification study was performed by Carlin et al. [2], in which they demonstrated that dynamic opcode tracing is extremely effective at detecting cryptomining behavior. Liu et al. [24] proposed a novel approach for detecting browser-based mining applications by

creating *BMDetector*, a detection system based on a modified Chrome kernel. Using this modified kernel, the authors were able to perform JavaScript code block analysis on the compiled JavaScript code, which allowed them to detect heavily obfuscated miner applications as well. Hong et al. [19] built *CMTracker*, a behavior-based detector with two runtime profilers for tracking browser-based cryptomining. The first profiler monitors incoming JavaScript files for known fingerprints, the second profiler observes the call stack and searches for periodic executions. Their approach was able to detect 868 actively mining websites among the Alexa Top 100K in April 2018. More than half of the found keys were used only once and they noticed that domains hosting mining scripts were migrating faster than the mining pool domains. The authors also mentioned evasion techniques, such as code obfuscation and payload hiding inside third-party libraries. Periodic execution in mining scripts was also noticed by Wang et al. [49], who created *SEISMIC*, a monitoring service to interrupt browser-based mining scripts based on this finding.

A different view on the subject was given by Papadopoulos et al. [36], who tried to answer the question whether browser-based cryptomining could be a suitable alternative to advertisements. After crawling a dataset of 200K websites running advertisements or cryptominers, they concluded that advertisements are still more than 5 times more profitable than cryptominers. This will only change once a visitor stays on the same website for more than 5.3 minutes or when Monero becomes more valuable [36]. A broader view of the browser-based cryptomining ecosystem is given by Saad et al. [42], who researched both cryptomining code and user impact. Besides various JavaScript static code analysis clustering methods and battery drainage studies when cryptomining, they did not perform any crawling of the web. This is in great contrast to the work of Rüth et al. [41], who digged deep into browser-based cryptomining by conducting two large web crawls. A first crawl using `zgrab`, which downloaded the first 256 kB of 137M *.com*, *.net*, and *.org* domains, as well as from the Alexa Top 1M websites. Consequently, the resulting HTML file was checked against the NoCoin [14] block list. A second crawl was performed on a subset of 10M websites, with a customized Chrome browser, instructed to dump WebAssembly modules for further inspection. They conclude their work by stating that 0.08% of the probed websites is actively mining [41].

Another large web crawl study is conducted by Konoth et al. [22] as a study for the creation of *MineSweeper*. Again, the Alexa Top 1M (including three internal pages) was crawled, with a crawler extracting information from all loaded JavaScript and HTML files, WebSocket traffic, and requests made while visiting the website. A total of 1,735 websites was found to be actively mining, the majority of them using Coinhive. 20 mining campaigns were discovered in their analysis, of which the largest involved 139 websites. Based on these findings, a novel detection technique was developed, which focused on the aspects all mining scripts have in common: high CPU cache usage and WebAssembly. They developed *MineSweeper*, able to successfully identify mining scripts based on the CPU's L1 and L3 cache usage and cryptomining characteristics in WebAssembly, thus hardening it against miner obfuscation.

As shown by this summary of related work, most attention of academic investigation has been on detecting these browser-based cryptominers. Multiple studies have shown to be able to detect them with high precision [19, 22, 24, 39–41]. Academic research is less focused on finding campaigns of cryptomining websites, while the online research community (such as Badpackets [31] or Krebs on Security [23]) is particularly interested in finding those relations. The first explorations into this area have been taken by [22], [13] and [39], but campaigns have not been systematically explored in their research. This paper aims to resolve this gap, by focusing on identifying campaigns, methods used in these campaigns and their evolution. We are also interested in the spread of cryptojacking on the Web, but as previous work is mostly crawling (subsets of) the Alexa Top 1M, we will analyze a broader set of websites online. In this paper we will not try to create a new detection method, but we build upon the work of [22] to perform our crawls.

5 Methodology

In a measurement study like this, suitable datasets and methods are essential for conducting proper research. In this section we first discuss the datasets used or created, followed by a summary of our crawler implementation.

5.1 Dataset creation

In our first crawl, we focus on finding campaigns of cryptojacking websites. Previous work of [19, 22, 39, 41] mainly investigated the popular parts of the Internet by crawling the Alexa Top 1M, or subsets of it. But, as pointed out by Scheitle et al., the Alexa Top 1M is not the only list measuring the popular Internet and the method Alexa uses to create this list raises questions whether it is the most reliable list to use for research on cryptojacking [44]. To overcome this issue, we have decided to use the union of three top lists on the Internet; the Alexa Top 1M [1], the Cisco Umbrella 1M [4] and the Majestic 1M [25], all using different measurement strategies, to include the popular part of the Internet in our dataset. These last two also include subdomains and domains not serving a web page. Therefore, we have only added the domains to the list of URLs to be crawled and omitted the subdomains from the latter two. Since we are interested in finding as many cryptojacking domains as possible for our campaign analysis, we have decided to extend our list even further with a list of websites gathered from querying PublicWWW – a source code search engine – with the keywords listed in Appendix A.

Table 2: Dataset creation for the campaign focused crawl

List	No. of websites	Date (2018)
Alexa Top 1M	1,000,000	Dec 24
Cisco Umbrella 1M	233,145	Dec 24
Majestic 1M	897,767	Dec 24
Custom PublicWWW set	87,051	Nov 23 – Dec 24
Total	**1,896,503**	

The union of these sets formed the dataset to be crawled and consisted out of 1,896,503 websites (unique effective TLDs + 1), as listed in Table 2. To estimate the prevalence of cryptojacking on the Internet in general, we will not use a top list as the Alexa Top 1M, because it is not a random sample of the Internet. We therefor also download a random sample of ~20% of the websites in 1,136 TLDs. We discuss this crawl in more detail in Section 7.

Operator NetFlows While the aforementioned datasets provide insights into the landscape of cryptomining installations at a given moment, these data sources do not reveal much about the actual usage of such services. In order to bridge this gap, we analyzed NetFlow traces from the network of a Tier 1 operator from September 2017 until December 2018, which were collected at a 1:8192 sampling ratio. For our analysis, we obtained NetFlow records for all traffic from and to the various WebSocket proxy servers belonging the mining services. Although NetFlows do not reveal the actual contents of a connection, the used ports and packet sizes can indicate connection types. The identity of the source connecting to the WebSocket proxy is however irrelevant, and was anonymized to a pseudo-random value by the operator using the CryptoPan algorithm [54].

5.2 Crawler implementation

As mentioned in Section 4, this research builds upon the work of Konoth et al. [22]. Therefore, we have used their crawler implementation as a starting point for our crawler. The following paragraphs will highlight the major changes and additions made to their work for our research.

Addition of new miner applications The publicly available *Minesweeper* crawler supports 22 different mining applications. Based on previous work and online research, we have added another 9 miner applications to the crawler, in order to also identify the newest miner applications. The added applications and their keywords are listed in Appendix B. For some of the already supported miner applications we have extended the fingerprints and improved the regular expressions to find *siteKeys*.

Active mining detection We have instructed the crawler to never explicitly consent to any mining operation. Therefore, we define that website to be actively mining without consent when: a mining code signature is found, together with a *siteKey*, more than two WebWorkers and a WebSocket connection, or, when the Stratum protocol communication or login credentials for a mining pool are found in WebSocket traffic. If one of these conditions holds, we mark the domain as actively cryptojacking.

WebSocket stack trace The miner application communicates with the mining pool using WebSocket connections. WebSocket traffic was already logged in the crawler, but the initiator of the WebSocket connection was not. By inspecting the stack trace of the WebSocket initiation, we can determine which script was responsible for opening the WebSocket connection and therefore the mining initiator. Using this method, we can easily distinguish between miners started from the main HTML page or the ones hidden inside other resources. Moreover, similar stack traces are a powerful indicator for campaign analysis, since it shows what component started the mining application. We have used this method successfully in our campaign analysis to identify attack vectors. Miners hidden inside third-party software such as WordPress are easily noticed in the stack trace, as we will show in Section 6.1.

Changed logic and exhaustive key finding Our crawler visits every website twice. First, by using a custom Chrome build, with the `-dump-wasm-module` flag enabled to dump any WebAssembly on the page. If present, these Wasm modules are analyzed for cryptojacking code by the *MineSweeper* application. Second, by using another Chrome build, which visits the website and saves every file it encounters. Instead of visiting 3 internal pages (as Konoth et al. did), we instructed the crawler to visit just one internal page. Besides that, we have implemented a more exhaustive *siteKey* search. The crawler first searches for fingerprints of known miner applications and afterwards for the *siteKey* in the following order: WebSocket traffic, the HTML page and finally in all other HTML and JavaScript resources. A minor addition has been made to automatically decode a base64 encoded *siteKey* of the Mineralt miner [27]. This addition allowed us to retrieve more *siteKeys*, which improves the campaign analysis afterwards.

5.3 Infrastructure

We deployed the crawler in Docker containers on 60 servers within the university network, each running 8 Docker instances in parallel. The crawl started on the December 24, 2018 and completed on January 9, 2019. In total, 1,769,183 websites have been successfully visited in this initial crawl. Afterwards, we have performed a second crawl using the same infrastructure, which we discuss in Section 7.

Table 3: Summary of the results of the first crawl

Crawling period	24/12/2018 – 9/1/2019
# websites crawled	1,769,183 (93%)
# potential cryptojacking websites	21,022
# active cryptojacking websites	10,100
# active miner applications	22
# websites with unknown miners	323
# cryptojacking campaigns identified	204
# websites in largest campaign	987
# websites in Alexa Top 1M	648 (0.065%)
# websites in Cisco Umbrella 1M	109 (0.047%)
# websites in Majestic 1M	506 (0.056%)

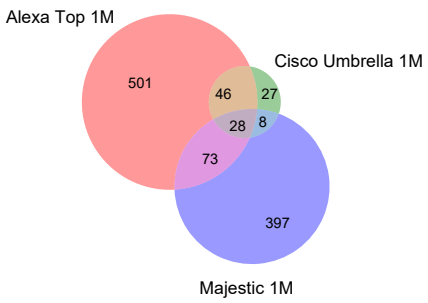


Figure 2: Venn-diagram showing the distribution of identified cryptojacking domains over the used top lists

6 Current state of cryptojacking campaigns

We have identified 21,022 websites with traces of cryptomining activities of which 10,100 websites are actively mining without the visitor's explicit consent. Only 648 of these websites are listed in the Alexa Top 1M. 22 different miner applications have been identified among the crawled websites, most of them running at least the Coinhive miner application (71%). Also, 509 websites are deploying multiple miners. For 323 websites, the used miner application could not be detected, which indicates heavily obfuscated or unknown miner applications. The results are summarized in Table 3.

Among the identified websites, 204 campaigns have been detected, of which the largest one covers 987 websites. This number of campaigns is a magnitude larger compared to previous work [22, 39]. We have identified the use of third-party software, such as Drupal and WordPress, to be the driving factor behind the largest cryptojacking campaigns.

Mining with consent There are two mining applications focused on mining solely with visitor consent. JSEcoin, a mining service presenting itself as *"The future blockchain & ecosystem for ecommerce and digital advertising"*, allows website owners to let their users mine JSE tokens, after explicit opt-in consent [20]. Another consent-focused mining application is AuthedMine, the opt-in version of Coinhive, introduced after adblockers started blocking Coinhive [8]. In our crawl, we have identified 2,477 websites using the JSEcoin miner and 227 websites using AuthedMine. None of the websites using AuthedMine opened a WebSocket connection, which indicates that no mining activity took place. 143 websites using JSEcoin did however open a WebSocket connection, but never actually started mining. By analyzing the WebSocket traffic, we observed that in most cases the WebSocket connection initiation was followed by two probes sent back and forth, waiting for the user to opt-in. Since these mining applications did not started mining without consent of the visitor, we have omitted them from our results.

Identified domains in top lists Of the 10,100 domains identified as actively cryptojacking, only 925 were found in one of the three top lists. The Alexa Top 1M contains the most cryptojacking domains (648), meaning that 0.065% of the websites in the Alexa Top 1M are cryptojacking, slightly less than previous work [22, 41]. For both other lists this number is lower. The addition of the Cisco Umbrella 1M resulted in only 27 additional findings, whereas the addition of the Majestic 1M led to the discovery of 397 new cryptojacking domains. In Figure 2, a Venn diagram depicts these differences in subsets. Only a small number of websites is shared among the Alexa Top 1M and the Majestic 1M. Also note that 9,175 (86%) of the identified websites are not listed in any of these top lists. This finding stresses the necessity of looking further than top lists while performing campaign analysis and to study the current state of cryptojacking on the Internet.

Categorization of websites We have discovered various sorts of cryptojacking websites on the Internet. By complementing the list of identified domains with website categorization data of Webshrinker [51], we categorized each cryptojacking website. We confirm previous work by identifying adult content (such as pornography) as the most prevailing category within our dataset, with over 2,000 websites in this category. Illegal content, a category known being home to abusive web resources, contains a lower number of cryptojacking websites compared to what we expected.

Installation base Coinhive is still the most popular cryptomining application installed on the identified cryptojacking websites (75%), followed by Cryptoloot (5.3%) and CoinImp (3.2%). But, there are noticeable differences between the complete crawl and the subset of domains in the Alexa Top 1M. Coinhive's share is halved, whereas CoinImp and Cryptoloot installations are doubled in size. Nerohut and Webminerpool miners are relatively more present in the Alexa Top 1M subset, while Mineralt has a similar share in that subset. The bottom

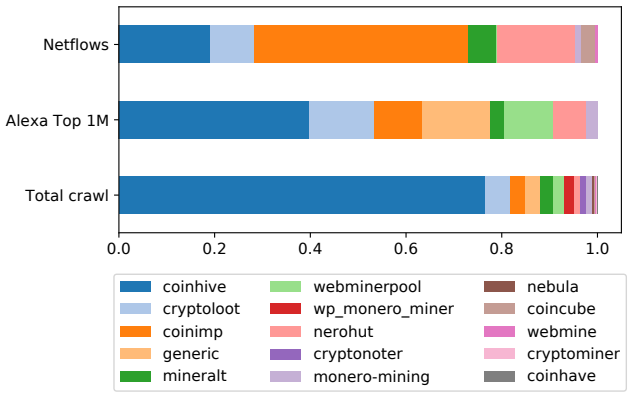


Figure 3: Distribution of cryptomining applications based on the total crawl, the Alexa Top 1M and NetFlows analysis.

two stacked bars in Figure 3 show the distribution of miners according to our analysis.

We have also discovered services which combine multiple cryptomining applications. The most popular mining combination is the set of Coinhive, Cryptoloot and Cryptonoter, which are bundled in the implementation of the WordPress Monero Miner plugin [21]. A combination of a Nerohut miner with a Cryptoloot or Webminerpool miner is also regularly encountered. Usually, only one miner starts (due to another script deciding which one to use), but we also encountered domains on which multiple miners were started concurrently.

Actual mining activity The distribution of mining applications installed on domains gives an insight into their popularity by actors pursuing cryptomining, but not into their actual usage. The amount of actual mining that takes place can however be estimated by tracing the connections website visitors make to the mining application's WebSocket proxy, as explained in Figure 1. We obtained a trace of connections transported by a Tier 1 network operator in 1:8192 sampling for a period of 14 months, and followed the WebSocket proxy server IPs from these mining applications to estimate the traffic to these servers. This gives an insight into how much traffic these WebSocket proxies digest, and is therefore a more reliable source for popularity measures. The upper stacked bar chart in Figure 3 shows the distribution of NetFlows to the WebSocket proxy servers of known mining applications for the month of December. The results show a drastic difference between installation base and mining traffic: while Coinhive is found on most websites, CoinImp proxy servers handle more than twice as much traffic than the dominant application. WebSocket traffic to servers of Cryptoloot is similar in size compared to its installation base.

Table 4: Mining pools the identified domains are mining in

Mining pool	Occurrence
`supportxmr.com`	93
`xmrpool.eu`	15
`greenpool.site`	13
`minexmr.com`	6
`xmr.omine.org`	4
`moneroocean.stream`	2
`seollar.me`	1
`xmr.nanopool.org`	1

Mining pool participation Most mining applications do not disclose the actual mining pool they are mining for in WebSocket traffic. However, on 135 identified domains, the WebSocket traffic did reveal that, as listed in Table 4. Most of these websites are participating in the `supportxmr.com` mining pool, which is commonly orchestrated by a Webminerpool or Nerohut mining script. Other pools are less commonly used or were not revealed in WebSocket traffic.

Throttling of applications Most cryptomining applications allow for a throttle value to be set, which limits the percentage of the CPU the miner can use. It is not necessary to set a throttle value, in this case the miner uses 100% of the available processing power. We have discovered that when a throttle value is set, this is often set to 0.3, meaning that 70% of the processing power can be used by the miner. Setting a throttle to use 70% of the resources seems to be balancing between gaining enough profit and not disturbing the browsing experience too much. In the identified campaigns, the throttle value is mostly set to the same value on all domains. An exception is listed in Table 5, in which a campaign involving 180 websites uses two different throttle values.

Attack vectors encountered We were able to retrieve the *siteKey* of actively cryptomining websites in 92% of the cases. Most of the gathered *siteKeys* are only used once (78%) and only a small portion (5%) is used on more than 5 different websites. However, the *siteKeys* in this last category are found on 4,663 different websites (46% of the total). The high number of *siteKeys* used only once suggests a large amount of website owner initiated cryptojacking, since every domain uses its own key. The fact that almost half of the websites is part of a campaign involving at least 5 websites also indicates different attack vectors. We have manually analyzed the used *siteKeys* in the latter category, and we can conclude that, besides website owner initiated cryptojacking, the use of third party software is a prevailing attack vector. Third-party applications like WordPress, Drupal or Magento are often abused to spread cryptojacking injections. These applications play a major part in campaign analysis, as discussed in Section 6.1.

Hiding techniques With the rise of cryptomining blocking applications such as NoCoin [17] or Minerblock [16], mining scripts are more often hidden to prevent detection. We have encountered a number of hiding techniques in our crawl and distinguish the following levels of obfuscation:

1. *No obfuscation.* The script is loaded in clear text, key and other options are visible to the user.

   ```
   var miner = new CoinHive.Anonymous('key');
   miner.start();
   ```

2. *Limiting CPU usage.* Script is loaded in clear text, key and other options are visible to the user, but CPU usage is throttled, so detection by the user is less likely.
3. *Renamed variables.* The script is loaded in clear text, but (some) variable names have been changed. These variable names are either replaced by random strings, or by completely different words, such as on `http://www.2001.com.ve/`:

   ```
   startHarryPotter("boddington", "2001");
   ```

4. *Renamed mining script.* The loaded script is still in clear text, but hosted on the web server itself instead of fetched from a mining service. The file name is changed to prevent blacklist blocking, frequently to general names, such as `jquery.js` or `stat.js`.
5. *Hidden inside other scripts*. The miner is appended or inserted into another script. The benign script still functions as normal, but also starts up the mining process.
6. *Obfuscated code.* The loaded scripts are masked by a code obfuscator and contain packed or CharCode code. All application-specific strings are encoded, stored in an array and variable names are replaced by random strings.

   ```
   var _0x5d02=["\x75\x73\x65\x20\x73\x74", ..]
   ```

7. *Obfuscated code and WebSocket traffic*. The loaded script is obfuscated by a code obfuscator and WebSocket traffic is sent encrypted to the proxy server.
8. *Obfuscated and hidden*. Scripts are hidden inside other files and/or via multiple redirects. Every script is randomly named and obfuscated, and so is the WebSocket traffic. WebAssembly is not retrieved from the server, but included inside the script.

In our crawl, most website owner initiated cryptojacking is not obfuscated, often not even throttling CPU usage. Attacks using third-party software are usually hiding cryptomining code inside other scripts and apply some obfuscation. We have encountered multiple WordPress themes and Drupal plugins with such a hidden miner. Only 391 websites with encrypted WebSocket have been identified, whereas most websites are using plain text Stratum communication. The highest level of obfuscation is rarely encountered.

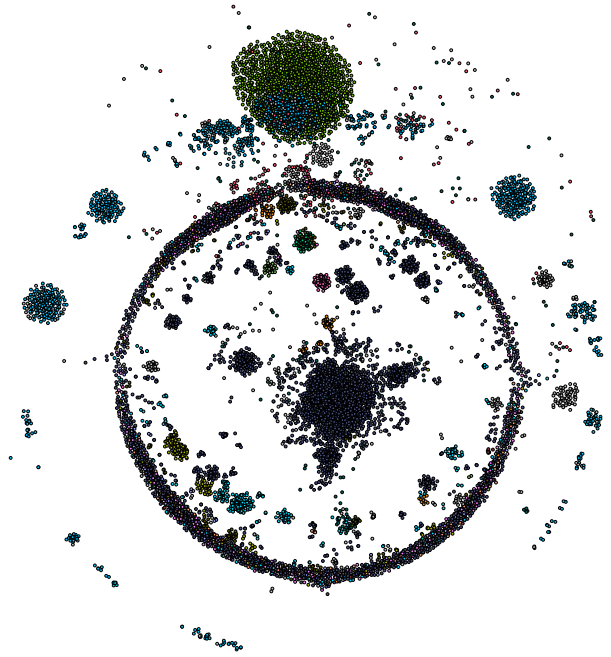


Figure 4: Relationships between the identified cryptojacking domains depicted in a force-directed graph

6.1 Cryptojacking campaigns

We have identified 204 cryptojacking campaigns, covering 5,733 websites, meaning that 57% of all cryptojacking websites encountered are part of a campaign. We define a cluster of more than 5 websites to be a campaign, as stated in Section 2. Figure 4 shows all the identified cryptojacking domains in a force-directed graph, where domains with similar features attract each other, colored according to the used application. Clear clusters can be distinguished, such as a Monero-Mining campaign shown in pink and a large Mineralt campaign shown in green right above it. Coinhive, the application used the most, is shown in dark blue with multiple large clusters all over the graph. The circle represents the cryptojacking domains not part of a campaign. In the following paragraphs, we highlight our findings based on different possibilities for identifying campaigns as introduced in Section 2.

Found on shared siteKey We were able to successfully retrieve the *siteKey* of 92% of the actively cryptojacking domains, which enabled us to cluster domains sharing the same *siteKey*. A shared *siteKey* guarantees that the rewards for mining will be transferred to the same account. We have identified 192 cryptojacking campaigns based on the same *siteKey* being installed on more than 5 different websites. As shown in Table 5, the largest campaign covers 987 websites, all using WordPress. A variety of plugins and themes include a malicious file named `jquory.js`, which is responsible for starting

a Coinhive miner. A similar attack vector is observed in a campaign involving 317 Drupal websites. This campaign is part of the Drupalgeddon 2 and 3 attacks, which took advantage of major remote code execution vulnerabilities in Drupal to inject their malicious scripts [45]. The only large campaign using the Mineralt miner, also focused on WordPress, has base64 encoded its *siteKey* inside the `script` tags. This makes them seem different, but match once decoded, since only the throttle value is changed. Not just vulnerabilities in CMS systems are used to spread cryptojacking code, also Magento, an e-commerce system, is involved in a Coinhive mining campaign targeting 175 websites in our crawl. The largest campaign using the compromised websites attack vector involved 376 Chinese websites, which share a miner script injected on the bottom of the page. A provider of The Pirate Bay proxies orchestrates the largest website owner initiated campaign on our list, with 70 proxy domains using the same Cryptoloot miner. These findings indicate that the most successful and largest cryptojacking campaigns are created by abusing third-party software.

Found on shared WebSocket proxy server Most cryptojacking campaigns are using the infrastructure of popular applications, such as Coinhive, to connect to a mining pool. Thus, clustering domains on these WebSocket proxy servers will not create meaningful clusters. However, when we discard these popular proxy servers, we are able to identify another 12 campaigns, which have not already been identified by shared *siteKeys*. Those are listed in Table 6. A Coincube miner campaign involving 27 websites uses `coin-services.info` as a WebSocket proxy server on a variety of ports. This campaign hosts its miner scripts on code repositories such as GitHub and BitBucket, where a number of accounts is created to host the miner files, which are all named `main.js`. On one of the GitHub accounts, even a picture of stacked Ukrainian money can be found [15]. 28 very similar websites, all offering illegal video streams, were found to be using a WebSocket proxy server on `wss://ws**.1q2w3.life/proxy` with, after manual inspection, `seriesf.lv` as the accompanied *siteKey*. This proxy server was also discovered by [22] on 5 websites in their crawl. They estimated that this campaign made a profit of $2,012.90 per month, which is likely to be a lot more, since we have found almost 6 times as many domains involved in this campaign. We have discovered that websites using a private WebSocket proxy are more likely to hide their activities by using higher levels of obfuscation.

Additionally, we have discovered 14 WebSocket proxy servers with very similar addresses on 75 domains (e.g. `nflying.bid`, `flightzy.bid` and `flightsy.bid`). These servers are contacted by the most obfuscated miner encountered in this crawl. The miner code is hidden inside a randomly named file, the miner code is heavily obfuscated and the WebSocket traffic is sometimes encrypted. Our efforts to reverse engineer the obfuscated miner code are so far unsuccessful. Therefore, we can not cluster them as being a campaign based on the shared proxy servers, but we have added the signature to our crawler as a separate mining application for the next crawls.

Found on shared initiator file In our crawling process, the stack trace of an initialized WebSocket connection is saved for every website. While examining these stack traces, some file names emerged and lead to the identification of another 4 cryptojacking campaigns. The oddly named file `gninimorenomv2.js`, responsible for opening WebSocket connections on 24 websites seemed to be part of a malicious advertisement campaign, which injects cryptojacking scripts into served advertisements. As shown in Table 6, this file opens a connection to `wss://heist.thefashiontip.com:8182/` to earn the profits from the displayed mining advertisements. Another campaign was identified by grouping the websites in which `adsmine.js` was responsible for opening a WebSocket connection. These websites turned out to be 17 very similar pornography websites, which indicates that this campaign is website owner initiated. The newly discovered mining application, as described in the previous section, served obfuscated mining scripts to its miners. Although obfuscated, inspection of the random file names revealed clusters of websites injected with the same randomly named miner, which lead to the discovery of another 3 campaigns, all targeting solely WordPress websites.

Found on shared mining pool login Most miner applications submit their solved hashes to a WebSocket proxy server, which combines the hashes of multiple miners before forwarding it to the actual mining pool. However, we have discovered 238 websites directly submitting their hashes to a mining pool. These websites are using only six unique cryptocurrency wallet addresses. The shared wallet addresses guarantee that profits made by cryptojacking are transferred to the exact same wallet. These findings did not lead to the discovery of any new campaigns, but did confirm previous findings. E.g., proxy `wss://delagrossemerde.com:8181/` (used by 15 sites) is solely receiving traffic from domains using the same wallet.
The different methods used in this section enabled us to find 204 cryptojacking campaigns. We can conclude that the largest campaigns are using third-party services like WordPress, Drupal or Magento as their method of spreading. Only one campaign using advertisements with injected cryptojacking scripts has been identified, this in contrast to previous work by [22, 39], who reported malicious advertisements as a significant attack vector. Compromised websites or website owner initiated campaigns are generally smaller in size. The obfuscation level used in most campaigns is rather low, heavily obfuscated code is encountered rarely and in more than half of the identified campaigns a miner added in plain text.

Table 5: Identified campaigns based on a shared *siteKey* (HT = hiding technique encountered)

SiteKey	#	Type	Attack vector	HT
I2OG8vG[..]coQL & hn6hNEm[..]w1hE	987	Coinhive	Third-party software (WordPress)	5
I8rYivhV3ph1iNrKfUjvdqNGfc7iXOEw	376	Coinhive	Compromised websites	2
oHaQn8u[..]EvOS, XoWXAWvi[..]JfGx, no2z8X4[..]w2yK	317	Coinhive	Third-party software (Drupal)	2
TnKJQivLdI92CHM5VDumySeVWinv2yfL	213	Coinhive	Third-party software (WordPress)	1
GcxML3FZ;60;1 & GcxML3FZ;-70;1	180	Mineralt	Third-party software (WordPress)	6
ZjAbjZv[..]9FiZ, PQbIwg9H[..]gfVW	175	Coinhive	Third-party software (Magento & WordPress)	4
w9WpfXZJ9POkztDmNpey3zA1eq3I3Y2p	103	Coinhive	Compromised websites	2
j7Bn4I56Mj7xPR2JrUNQ9Bjt6CeHS3X1	79	Coinhive	Third-party software (WordPress)	2
cb8605f33e66d9d[..]6af74f86e6882899a8	70	Cryptoloot	Website owner initiated (The Pirate Bay)	2
49dVbbCFDuhg9nX[..]K2fkq5Nd55mLNnB4WK	70	Coinhive	Compromised websites	1

Table 6: Identified campaigns based on shared WebSocket proxy servers (HT = hiding technique encountered)

WebSocket proxy server	#	Type	Attack vector	HT
wss://ws**.1q2w3.life/proxy	28	Nebula	Website owner initiated	6
wss://coin-services.info:****/proxy	27	Coincube	Compromised websites	6
wss://heist.thefashiontip.com:8182/	24	Webminerpool	Malicious advertisements	5
wss://delagrossemerde.com:8181//	15	Webminerpool	Website owner initiated	8
wss://wss.rand.com.ru:8843/	13	Coinhive	Third-party software (WordPress)	8
ws://185.165.169.108:8181/	8	Webminerpool	Website owner initiated	2
ws://68.183.47.98:8181/	7	Webminerpool	Website owner initiated	2
wss://gtg02.bestsecurepractice.com/proxy2/	6	Unknown	Third-party software (WordPress)	3

6.2 A in-depth campaign search

The sizes of the campaigns identified in Section 6.1 depend on the dataset we crawled, so they could have been incomplete. To find more websites belonging to the identified campaigns, we have taken the indicators of compromise for a large number of campaigns and queried PublicWWW for domains matching these IoCs. This resulted in a dataset of 7,892 websites. Combined with the 21,022 potentially cryptojacking websites from the initial crawl, a total of 25,121 URLs was crawled on February 12, 2019, more than a month after the initial crawl. We successfully obtained 24,187 (96%) of them.

Most of the campaigns remained of similar size in this crawl, except for a campaign involving three keys, ef937f99557277ff62a6fc0e5b3da90ea9550ebcdfac, 06d93b846706f4dca9996baa15d4d207e82d1e86676c and dd27d0676efdecb12703623d6864bbe9f4e7b3f69f2e. This advanced campaign is targeting domains using Bitrix24, a CRM platform used by a variety of organizations. The most remarkable website it has been found on is the website of the Ministry of Education of Belarus (https://edu.gov.by/). The malicious code is hidden as the core loader of Bitrix24 and uses both Nerohut and Cryptoloot to mine with. It has a built-in anti-detection method, since it stops mining once a developer tools window is opened. In our initial crawl, we have identified only 68 domains belonging to this campaign, which turned out to be 855 in our in-depth search, making this campaign the second largest campaign we have identified so far. Another campaign, involving key vPfPDHk89TxmH1arysiJDrutpYGntofP, is displaying fake loading screens on 86 websites, whereas only 47 of these have been identified in our initial crawl.

All other campaigns remained similar or slightly smaller in size. Except for the two aforementioned campaigns, we conclude that our initial crawl likely identified the correct size of campaigns, given the database of PublicWWW. Their database contains source code snapshots of over 544M websites, which should provide a proper approximation.

6.3 Evolution of cryptojacking

To study the evolution of cryptojacking on the Internet, data is needed from different moments in time. Fortunately, Konoth et al. [22] shared their crawling results and Hong et al. [19] shared their list of identified cryptojacking domains, which made it possible for us to crawl these exact same sets of URLs and to analyze whether these domains were still mining. Additionally, we have followed the domains identified in our crawls over a period of 3 months, and analyzed WebSocket proxy traffic over time using operator NetFlows.

Comparison with previous crawls Konoth et al. [22] crawled from March 12 until 19, 2018 and identified 1,735 potential cryptojacking domains. We crawled their list on January 21, 2019 and obtained 1,725 of them. 85% of the

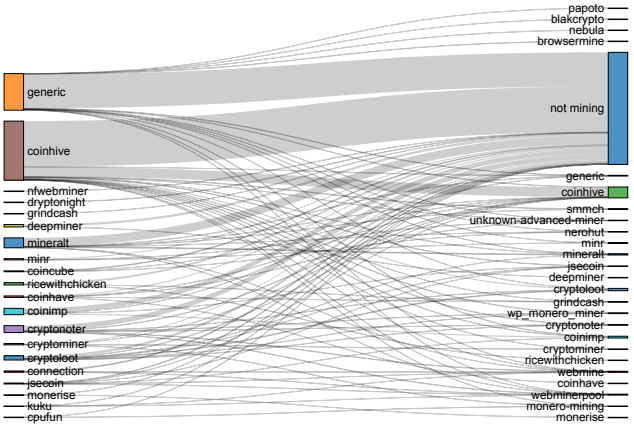


Figure 5: Usage evolution between March 2018 and January 2019 in the list of identified domains by [22]

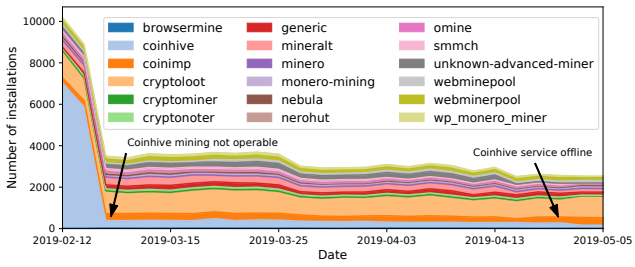


Figure 6: Evolution of the cryptojacking domains per type

websites are not cryptomining anymore, and only 10% is still using the same application. On 136 websites (7%), the same key was found in both crawls. As Figure 5 shows, a large number of websites using a Coinhive miner removed the miner application. Some continued using Coinhive, but also a small shift into less popular mining applications can be observed. Websites already using these miners tend to stick to their choice and are still using the same miner almost a year later. We have also seen a number of mining applications become extinct, such as Deepminer and NF Webminer. Hong et al. [19] also published the list of identified cryptojacking domains from their crawl in February 2018. A year later, on February 12, 2019, we have crawled this list of 2,770 domains. We obtained 2,435 (88%) of them and only 340 (14%) domains are still actively cryptojacking. Both crawls show that a large number of websites stopped cryptojacking themselves or removed the miner infection. After one year, approximately 85% of the domains are not actively cryptojacking anymore. We have also observed a small portion of domains switching to less popular applications. The low number of 7% of websites that are still mining with the same *siteKey* indicates the fast changes in the cryptojacking threat landscape.

Evolution of identified domains We have followed all previously identified cryptojacking domains for a period of 3 months (until May 5, 2019) and crawled them initially occasionally, but afterwards every other day. Within this time period, Coinhive announced to end its mining application, due to decreased Monero prices and hash rate [7]. The announcement was made on February 26, 2019 and stated that mining would not be operating anymore after March 8, 2019, and that the service would be discontinued by the end of April 2019. This lead to a drastic change in the cryptojacking landscape, as Coinhive's dominance in actively mining installations collapsed when their mining service was set non-operationally. Mining applications were however not massively replaced, which confirms our finding that a large portion of browser-based cryptomining is not initiated by the website owner. Only when the Coinhive mining service was actually discontinued and errors were shown while requesting the offline Coinhive mining resources, we observe a small increase in Cryptoloot and CoinImp installations.

WebSocket proxy traffic over time As discussed in Section 2, most miner applications use a WebSocket proxy server to forward traffic from their miners to the mining pool. Using NetFlow data mentioned earlier, we have analyzed traffic towards popular WebSocket proxies from September 2017 till December 2018, which gives an insight into the evolution of cryptomining applications usage, as shown in Figure 7. We have taken the set of WebSocket proxy IPs the miners connect to as a basis, which we extended by using passive DNS data to discover other WebSocket proxy server IPs used by these applications, but hosted on different servers, not encountered during our crawls. The same passive DNS data was used to verify whether these IP addresses were solely used as WebSocket proxy servers. To prevent other traffic to these servers from being in our dataset, we have both set the maximum packet size to 550 kB and verified that only WebSocket traffic was counted towards these servers. For most proxies, this is traffic towards port 80 or 443, and for a few servers using specific ports, this could be different. An example is the WebSocket proxy server of the WP-monero-miner which uses port 8020.

The blue line from September 2017 on shows how the web-mining ecosystem is monopolized by innovator Coinhive at the start, where after copycats like Cryptoloot and Webmine start to emerge in October. We see that CoinImp essentially starts to eclipse all other miner applications from mid April 2018 onwards in terms of mining traffic to the proxies, which is unexpected given the distribution of installations on websites and previous studies. Some mining proxies only have transient success: a remarkable example is the WP-monero-miner, released shortly after Coinhive in 2017. The applica-

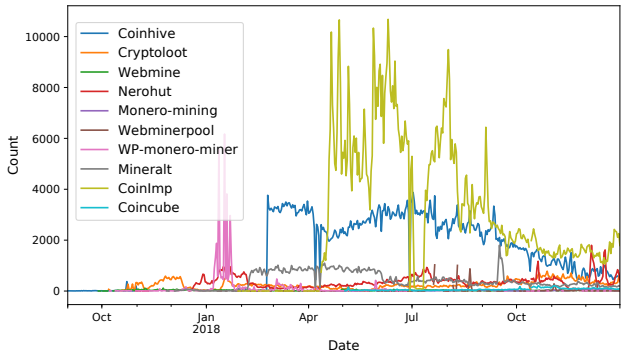


Figure 7: Number of NetFlows involving WebSocket proxy servers for popular miners between Sep 2017 and Dec 2018

tion hosts its own mining pool and digested a lot of traffic in January 2018, only to almost disappear again weeks later. Coinhive, the application used by most websites, is a constant factor in the miner landscape with over 4,000 NetFlows a day in mid 2018 (given our 1:8192 sampling, thus 32M connections per day), but not as large as one would expect from its installation base. Additionally, a clear declining trend can be observed in the NetFlow counts to all mining services after the summer of 2018. The last months of NetFlow data show a diverse set of mining applications actively used.

7 An Internet scale study on cryptojacking

In order to estimate the prevalence of browser-based cryptojacking on the Internet and to indicate any differences between Top Level Domains (TLDs), we have performed another crawl, in which we have crawled ~20% of the websites belonging to each of the 1,136 existing TLDs. We obtained a daily zone transfer for all generic top level domains (gTLDs) – such as *.top*, *.loan* – from the Internet Corporation for Assigned Names and Numbers (ICANN), as well as a feed of registered country code top-level domains (ccTLDs) – such as *.uk*, *.jp*, or *.ru* – from a security intelligence provider. From these lists, we randomly picked a sample of ~20% of the size of each TLD [12]. Based on the results of the previous crawl, we have added another 5 mining applications to the crawler implementation, as listed in Appendix C. From January 11 until April 3, 2019, we crawled the random sample including 48.9M domains. This yielded a total of 125 TB of network traffic.

7.1 General findings

After crawling a random sample of 48.9M websites in a large number of different top level domains, we are able to draw conclusions about the prevalence of browser-based cryptojacking on the Internet. We estimate that 0.011% of all domains are actively cryptomining without their visitors' explicit consent,

Table 7: Distribution of cryptomining applications installations in the Internet scale crawl (sum of percentages is >100%, because of websites using multiple applications)

Type	# of websites	Percentage
Coinhive	2,531	48.767%
Unknown	689	13.276%
CoinImp	513	9.884%
Cryptoloot	504	9.711%
Mineralt	276	5.318%
Nerohut	247	4.760%
Webminerpool	233	4.489%
Unknown-advanced-miner	92	1.773%
SMMCH	80	1.541%
Browsermine	73	1.407%
Webminepool	62	1.195%
WP-Monero-Miner	60	1.156%
Omine	56	1.079%
Monero-mining	55	1.060%
Cryptonoter	50	0.963%
Cryptominer	26	0.501%
Minero	24	0.462%
Nebula	23	0.443%
Webmine	19	0.366%
Coincube	19	0.366%
Project-poi	4	0.077%
Adless	1	0.019%

meaning that one in every 9,090 websites is cryptojacking. Comparing this number to the statistics of the top lists used in our initial crawl, we conclude that cryptojacking activity is mainly focused on the popular parts of the Internet. In the Alexa Top 1M, 0.065% of the websites was actively cryptojacking, in this random sample only 0.011% of the websites, which is almost 6 times lower. This can be explained by the lucrativeness of cryptojacking, in which a higher popularity means more visitors, yielding more potential miners and thus higher potential profits. Additionally, it shows that researching the prevalence of cryptojacking by crawling the Alexa Top 1M overestimates the problem size. However, the distribution of used applications in our random sample is fairly similar to the distribution in the Alexa Top 1M. The distribution of mining applications in this crawl is listed in Table 7.

The categories of domains identified in this crawl are very similar to the initial crawl. As depicted in Figure 8, *Adult content* remains the most prevailing category, while other large categories are *Technology* and *Under Construction*, the category involving parked, expired or yet-to-be developed domains. Based on these two very different crawls we can conclude that cryptojacking is indeed more prevailing on domains hosting adult content.

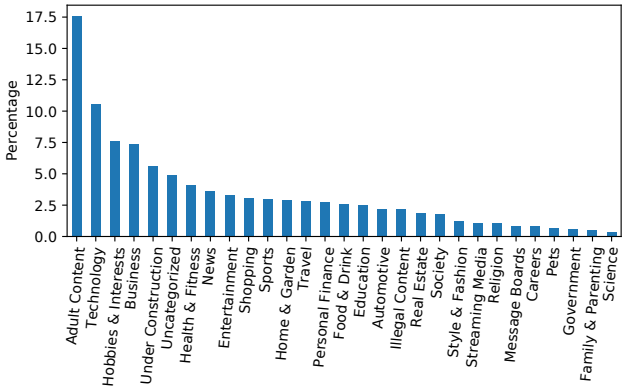


Figure 8: Categories of mining domains in the second crawl

7.2 Cryptojacking on different TLDs

We have crawled domains of roughly ~20% of 1,136 different TLDs in order to analyze the prevalence of cryptojacking. As Table 8 shows, cryptojacking activity varies enormously within different TLD zones. The four largest TLDs, *.com*, *.de*, *.net* and *.org* have a similar percentage of cryptojacking websites, but we have discovered almost 6 times as much cryptojacking activity in the Russian TLD. Also, domains in the Brazilian and Spanish zones are more susceptible to cryptojacking, having respectively 4 and 3 times more cryptojacking activity than average. On the contrary, the *.top*, *.us* and *.loan* zones host only a few cryptojacking websites.

Our website category analysis showed that adult content is the most prevailing category for cryptojacking activities. This triggered our attention for the *.xxx* domain, which is specially created for adult content, which we therefore crawled completely instead of ~20%. Surprisingly, the *.xxx* domain contains only one website actively cryptomining.

When comparing used mining applications on the different TLDs, large differences can be distinguished, as shown in Figure 9. Coinhive is the most popular miner in most zones, whereas Cryptoloot is preferred in the Russian zone, and French and Czech websites contain more Nerohut miners. The Russian zone is also the only TLD where browsermine is used regularly. The high number of generic miner applications in the Dutch and Belgian zone is remarkable. A large number of these domains in the *.nl* and *.be* zone are part of a campaign using expired domain names of a Dutch registrar (*Totaaldomein B.V.*) to host porn and unknown cryptominers.

Our results show a different popularity of used mining applications compared to previous work of [41]. They detected Coinhive on 85% to 90% of the *.com*, *.net* and *.org* TLDs, whereas we determine that this market share is significantly lower (~50%). This result proves that a simple solution like the NoCoin block list is unable to detect all miners and analyses with such techniques result in different outcomes.

Table 8: Results of the TLD crawl. Listed are the top 10 largest domains, followed by remarkable TLDs

TLD	Size	Crawled	Cryptojacking
.com	149,937,597	27,555,546 (18.4%)	2,353 (0.009%)
.net	15,008,406	2,741,550 (18.3%)	238 (0.009%)
.de	15,089,860	2,244,139 (14.9%)	254 (0.011%)
.org	11,330,764	2,021,630 (17.8%)	145 (0.007%)
.info	6,524,248	1,309,323 (20.6%)	77 (0.005%)
.ru	5,480,467	998,422 (20.0%)	593 (0.059%)
.nl	5,360,173	880,122 (16.4%)	191 (0.022%)
.top	4,024,497	788,748 (19.6%)	19 (0.002%)
.br	3,813,745	383,910 (10.1%)	185 (0.048%)
.fr	3,449,775	567,887 (16.5%)	133 (0.023%)
.pl	2,621,515	523,497 (20.0%)	81 (0.015%)
.us	2,409,802	472,323 (19.6%)	2 (0.000%)
.loan	2,228,165	445,749 (20.0%)	0 (0.000%)
.es	2,010,710	327,810 (16.3%)	110 (0.036%)
.online	1,105,999	219,447 (19.8%)	67 (0.031%)
.pro	295,201	58,999 (14.2%)	32 (0.054%)
.space	268,846	53,363 (20.0%)	19 (0.036%)
.website	276,063	54,704 (19.8%)	21 (0.038%)
.xxx	93,101	91,877 (98.7%)	1 (0.001%)
Total		**48,948,669**	**5,190 (0.011)%**

8 Discussion

Crawling the Internet inevitably comes with its shortcomings. Limitations in the crawler implementation, network used and analysis can produce both false positives and negatives. The latter category can occur for example when extreme obfuscation is used, as we have seen in Section 6. However, we believe that due to our double crawling strategy, based on both WebAssembly and code signatures, this could not have happened very often. Finally, the use of worldwide NetFlow traffic from a Tier 1 network operator allowed us to analyze the popularity of cryptojacking services in a revolutionary way, although BGP policies, and a specific PoP and IXP footprint could lead to a bias of certain autonomous systems just as some discrepancies might arise due to 1:8192 random sampling. Additionally, since the NetFlows do not reveal the actual contents of the connection, we can never be sure about the contents. However, during our crawls we could confirm the mining applications to contact the WebSocket proxy servers in question, and passive DNS lookups did not show any other domains pointed to that IP. Furthermore, the NetFlows both revealed no traffic to other ports than those seen from our crawlers and packet sizes resembling those observed in our crawls, thus the methodology should provide valid results.

Future work The additional angle provided by the NetFlow data allowed us to study the evolution of cryptojacking over a longer period of time, something which has not been done be-

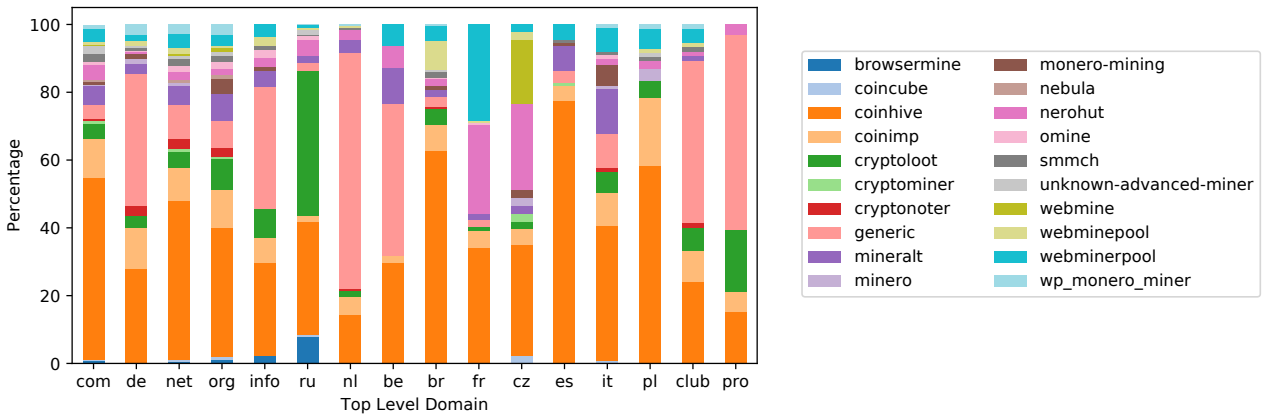


Figure 9: The distribution of used mining applications in various TLDs

fore. Regular crawls of the Internet, especially of the already identified cryptojacking domains gives more insight in this practice, as will it increase the innovation of defense mechanisms. The most influential defense against cryptojacking will nonetheless be frequent patching, as most cryptominers are installed exploiting known vulnerabilities. CMS providers, such as Drupal or WordPress, have shown agility in patching vulnerabilities, but the responsibility of installing these patches remains with the website owner. Finally, as we have seen a decline in the price of Monero (-85% in 2018), we believe that cryptojacking infections on individual websites will decrease, but that cyber criminals will search for other possibilities to exploit cryptojacking at an even larger scale. As we have mentioned in Section 3, the most effective method of collecting large groups of miners is by launching a MITM attack. Investigating the prevalence of this attack vector for cryptomining is something we preserve for future work.

9 Conclusions

In this paper, we have studied the prevalence of cryptojacking as well as of cryptojacking campaigns on the Internet. We have performed multiple large crawls, each with a different focus. In our first crawl, we have analyzed the 1.7M most popular domains to identify organized campaigns. We found 204 campaigns, from which we conclude that the size of cryptojacking campaigns is heavily underestimated by current academic research. Additionally, using solely the Alexa Top 1M shows significantly different results in terms of the size of organized activity and infection rate, which we found to be almost 6 times lower in a random sample compared to the Alexa Top 1M, hence overestimating the problem. Third-party software is often used by attackers to spread cryptojacking scripts over a large number of domains. The share of domains serving advertisements injected with cryptojacking scripts is lower compared to previous work, most likely because of stricter monitoring by advertisement networks. We have seen that obfuscation of cryptojacking scripts is definitely present, but only occasionally used. Comparing our results with data from previous studies (in both February and March 2018) shows that after a year, only 15% of the websites is still actively mining. This, and our novel way of estimating miner application popularity by analyzing NetFlows, led to the conclusion that the cryptojacking landscape is constantly changing and involves a variety of actors.

A second, Internet-scale crawl involving ~20% of 1,136 TLDs (48.9M websites), which represents a truly random sample of the Internet, allows us to conclude that cryptojacking is present on 0.011% of all domains. Not unexpectedly, this percentage increases in the more popular parts of the Internet, because cryptojacking on popular domains is much more lucrative. Both of our crawls have shown that cryptojacking mostly takes place on websites hosting adult content, although the *.xxx* TLD is home to only one cryptojacking website. Based on the applications used within the time span of our analysis, we can conclude that Coinhive was the largest mining application in terms of installation base, but that CoinImp's WebSocket proxy servers were digesting much more traffic in 2018. Looking at the different TLDs, we conclude that Russian, Brazilian and Spanish zones are home to a disproportionate number of cryptojacking domains.

With the discontinuation of Coinhive in March 2019, the landscape of cryptojacking has changed enormously, but based on our results, we are only expecting a further decline in individual cryptojacking activities given that the Monero value keeps diminishing. However, this only stresses the importance of organized cryptojacking campaigns, as cyber criminals will find new ways to spread their cryptojacking infections to still be profitable. Here, campaign analysis will be an important asset: as adversaries are unlikely to develop a unique approach for each infected website, the reuse of resources and methods will provide an effective angle to detect and mitigate these activities.

References

[1] ALEXA. Top 1M sites. http://s3.amazonaws.com/alexa-static/top-1m.csv.zip (December 2018).

[2] CARLIN, D., O'KANE, P., SEZER, S., AND BURGESS, J. Detecting cryptomining using dynamic analysis. In *16th Annual Conference on Privacy, Security and Trust, PST 2018, Belfast, Northern Ireland, Uk, August 28-30, 2018* (2018), pp. 1–6.

[3] CHRISTOPHER, N. Hackers mined a fortune from indian websites, Sep 2018. https://economictimes.indiatimes.com/small-biz/startups/newsbuzz/hackers-mined-a-fortune-from-indian-websites/articleshow/65836088.cms (December 2018).

[4] CISCO. Cisco Umbrella 1 Million. http://s3-us-west-1.amazonaws.com/umbrella-static/top-1m.csv.zip (December 2018).

[5] CLABURN, T. Crypto-jackers enlist google tag manager to smuggle alt-coin miners, Jan 2018. https://www.theregister.co.uk/2017/11/22/cryptojackers_google_tag_manager_coin_hive/ (December 2018).

[6] COIN-HAVE. Coinhave – monero javascript mining. https://coin-have.com/ (December 2018).

[7] COINHIVE. Blog: Discontinuation of coinhive. https://coinhive.com/blog/en/discontinuation-of-coinhive (April 2019).

[8] COINHIVE. Coinhive blog: Authedmine – non-adblocked. https://coinhive.com/blog/en/authedmine (April 2019).

[9] COINHIVE. First week status report, Sep 2017. https://coinhive.com/blog/en/status-report (December 2018).

[10] COINHIVE. Coinhive - monero mining club, Jan 2018. https://coinhive.com/ (December 2018).

[11] CRYPTOLOOT.COM. Cryptoloot - earn more from your traffic. https://crypto-loot.com/ (December 2018).

[12] DOMAINTOOLS.COM. Domain Count Statistics for TLDs. http://research.domaintools.com/statistics/tld-counts/ (January 2019).

[13] ESKANDARI, S., LEOUTSARAKOS, A., MURSCH, T., AND CLARK, J. A first look at browser-based cryptojacking. *2018 IEEE European Symposium on Security and Privacy Workshops, EuroS&P Workshops 2018, London, United Kingdom, April 23-27, 2018* (2018), 58–66.

[14] GITHUB.COM. hoshsadiq/adblock-nocoin-list. https://github.com/hoshsadiq/adblock-nocoin-list (December 2018).

[15] GITHUB.COM. leonidackov901/leonidackov901.github.io. https://github.com/leonidackov901/leonidackov901.github.io (January 2019).

[16] GOOGLE.COM. minerblock. https://chrome.google.com/webstore/detail/minerblock/emikbbbebcdfohonlaifafnoanocnebl?hl=en (January 2019).

[17] GOOGLE.COM. No coin - block miners on the web! https://chrome.google.com/webstore/detail/no-coin-block-miners-on-t/gojamcfopckidlocpkbelmpjcgmbgjcl (January 2019).

[18] HOFFMAN, J. J., LEE, S. C., AND JACOBSON, J. S. New jersey division of consumer affairs obtains settlement with developer of bitcoin-mining software found to have accessed new jersey computers without users' knowledge or consent, May 2015. https://nj.gov/oag/newsreleases15/pr20150526b.html (December 2018).

[19] HONG, G., YANG, Z., YANG, S., ZHANG, L., NAN, Y., ZHANG, Z., YANG, M., ZHANG, Y., QIAN, Z., AND DUAN, H. How you get shot in the back: A systematical study about cryptojacking in the real world. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018* (2018), pp. 1701–1713.

[20] JSECOIN. Jsecoin: Digital currency - designed for the web. https://jsecoin.com/ (April 2019).

[21] KEIL, D. Wp monero miner - home. https://www.wp-monero-miner.com/ (December 2018).

[22] KONOTH, R. K., VINETI, E., MOONSAMY, V., LINDORFER, M., KRUEGEL, C., BOS, H., AND VIGNA, G. Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018* (2018), pp. 1714–1730.

[23] KREBS, B. Krebs on security - who and what is coinhive. https://krebsonsecurity.com/2018/03/who-and-what-is-coinhive/ (December 2018).

[24] LIU, J., ZHAO, Z., CUI, X., WANG, Z., AND LIU, Q. A novel approach for detecting browser-based silent miner. *Third IEEE International Conference on Data Science in Cyberspace, DSC 2018, Guangzhou, China, June 18-21, 2018* (2018), 490–497.

[25] MAJESTIC. Majestic Million CSV now free for all, daily. http://downloads.majestic.com/majestic_million.csv (December 2018).

[26] MCCARTHY, K. Cbs's showtime caught mining crypto-coins in viewers' web browsers, Jan 2018. https://www.theregister.co.uk/2017/09/25/showtime_hit_with_coinmining_script/ (December 2018).

[27] MINERALT. Developer api documentation and reference. https://support.mineralt.io/support/solutions/articles/36000047274-js-miner-usage-and-api-reference (December 2018).

[28] MONERO OCEAN. Monero ocean – faq. https://moneroocean.stream/#/help/faq (May 2019).

[29] MOZILLA FOUNDATION. asm.js - working draft — 18 august 2014. http://asmjs.org/spec/latest/ (November 2018).

[30] MURPHY, M. Youtube shuts down hidden cryptojacking adverts, Jan 2018. https://www.telegraph.co.uk/technology/2018/01/29/youtube-shuts-hidden-crypto-jacking-adverts/ (November 2018).

[31] MURSCH, T. Cryptojacking malware coinhive found on 30,000 websites, Feb 2018. https://badpackets.net/cryptojacking-malware-coinhive-found-on-30000-websites/ (December 2018).

[32] MURSCH, T. Over 100,000 drupal websites vulnerable to drupalgeddon 2 (cve-2018-7600), Jun 2018. https://badpackets.net/over-100000-drupal-websites-vulnerable-to-drupalgeddon-2-cve-2018-7600/ (January 2019).

[33] NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system, 2009.

[34] OGONO, U. Monero cryptojacking: Monero cryptocurrency mining malware disrupts government site, Sep 2018. https://smartereum.com/35507/monero-cryptojacking-monero-cryptocurrency-mining-malware-disrupts-government-site-monero-news-today/ (December 2018).

[35] OSBORNE, C. Mikrotik routers enslaved in massive coinhive cryptojacking campaign, Aug 2018. https://www.zdnet.com/article/mikrotik-routers-enslaved-in-massive-coinhive-cryptojacking-campaign/ (December 2018).

[36] PAPADOPOULOS, P., ILIA, P., AND MARKATOS, E. P. Truth in web mining: Measuring the profitability and cost of cryptominers as a web monetization model. *CoRR abs/1806.01994* (2018).

[37] PASTRANA, S., AND SUAREZ-TANGIL, G. A first look at the crypto-mining malware ecosystem: A decade of unrestricted wealth. *CoRR abs/1901.00846* (2019).

[38] PEARSON, J. Starbucks Wi-Fi Hijacked People's Laptops to Mine Cryptocurrency. https://motherboard.vice.com/en_us/article/gyd5xq/starbucks-wi-fi-hijacked-peoples-laptops-to-mine-cryptocurrency-coinhive (February 2019).

[39] RAUCHBERGER, J., SCHRITTWIESER, S., DAM, T., LUH, R., BUHOV, D., PÖTZELSBERGER, G., AND KIM, H. The other side of the coin: A framework for detecting and analyzing web-based cryptocurrency mining campaigns. In *Proceedings of the 13th International Conference on Availability, Reliability and Security, ARES 2018, Hamburg, Germany, August 27-30, 2018* (2018), pp. 18:1–18:10.

[40] RODRIGUEZ, J. D. P., AND POSEGGA, J. RAPID: resource and api-based detection against in-browser miners. *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018* (2018), 313–326.

[41] RÜTH, J., ZIMMERMANN, T., WOLSING, K., AND HOHLFELD, O. Digging into browser-based crypto mining. In *Proceedings of the Internet Measurement Conference 2018, IMC 2018, Boston, MA, USA, October 31 - November 02, 2018* (2018), pp. 70–76.

[42] SAAD, M., KHORMALI, A., AND MOHAISEN, A. End-to-end analysis of in-browser cryptojacking. *CoRR abs/1809.02152* (2018).

[43] SABERHAGEN, N. V. Cryptonote v 2.0, Oct 2013. https://cryptonote.org/whitepaper.pdf.

[44] SCHEITLE, Q., HOHLFELD, O., GAMBA, J., JELTEN, J., ZIMMERMANN, T., STROWES, S. D., AND VALLINA-RODRIGUEZ, N. A long way to the top: Significance, structure, and stability of internet top lists. *CoRR abs/1805.11506* (2018).

[45] SEGURA, J. A look into drupalgeddon's client-side attacks, Jun 2018. https://blog.malwarebytes.com/threat-analysis/2018/05/look-drupalgeddon-client-side-attacks/ (January 2019).

[46] SLUSHPOOL. Stratum mining protocol. https://slushpool.com/help/topic/stratum-protocol/ (November 2018).

[47] THE MONERO PROJECT. Monero: What is monero (xmr)? https://www.getmonero.org/get-started/what-is-monero/ (December 2018).

[48] THE PIRATE BAY. The pirate bay - miner, Sep 2017. https://thepiratebay.org/blog/242 (December 2018).

[49] WANG, W., FERRELL, B., XU, X., HAMLEN, K. W., AND HAO, S. SEISMIC: secure in-lined script monitors for interrupting cryptojacks. In *Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part II* (2018), pp. 122–142.

[50] WEBASSEMBLY.ORG. Webassembly. https://webassembly.org/ (November 2018).

[51] WEBSHRINKER. Webshrinker apis. https://www.webshrinker.com/apis/ (January 2019).

[52] WEBSOCKET.ORG. Html5 websocket - a quantum leap in scalability for the web. http://www.websocket.org/aboutwebsocket.html (November 2018).

[53] WORDFENCE.COM. Wordpress plugin banned for crypto mining, Nov 2017. https://www.wordfence.com/blog/2017/11/wordpress-plugin-banned-crypto-mining/ (January 2019).

[54] XU, J., FAN, J., AMMAR, M., AND MOON, S. B. On the design and performance of prefix-preserving ip traffic trace anonymization. In *ACM SIGCOMM Workshop on Internet Measurement* (2001).

A Search queries for PublicWWW

Table 9: All search queries for the PublicWWW database

Miner	Search term(s)
Coinhive	coinhive.min.js, CoinHive.Anonymous(
JSECoin	load.jsecoin.com
Webmine	webmine.cz
Cryptoloot	/crypta.js, /crlt.js, crlt.anonymous, CryptoLoot.Anonymous
CoinImp	CoinImp.Anonymous, www.hashing.win, hostingcloud.racing
Cryptonoter	minercry.pt/processor.js, cryptonoter
NFWebminer	nfwebminer.com/lib/, NFMiner(
Deepminer	deepMiner
Monerise	monerise_builder, monerise_payment_address(
Coinhave	minescripts.info
Nebula	CoinNebula.Instance
Mineralt	play.gramombird.com/app.js
Munero	munero.me
Minr	cdn.jquery-uim.download, cnt.statistic.date, ad.g-content.bid
Webminerpool	webmr.js
WPMoneroMiner	wp-monero-miner.js
Nerohut	nhm.min.js, nerohut.com/srv
Adless	adless.js
Monero-mining	Perfektstart(
Miscellaneous	function echostat(){var, function printju, pocketgolf.host/start.php async, startMining(, jquory.js

B Added miner applications and their keywords for the campaign crawl

Table 10: The added miner applications and their keywords

Miner	Keywords
Nebula	CoinNebula.Instance
WP Monero miner	wp_js_options | wp-monero-miner
Nerohut	nhm.min.js | NHpwd | nhsrv.cf/srv/serve.php?key=
Webminerpool	webmr.js | startMining(
Minero	minero.cc
Adless	adless.js | adless.io
Monero-mining	PerfektStart | perfekt.js
ProjectPoi	ProjectPoi\b | projectpoi.min.js
Papoto	papoto

C Added miner applications in the Internet scale crawl

Table 11: The added miner applications and their keywords in the latest version of the crawler

Miner	Keywords
SMMCH	simple-monero-miner-coin-hive smmch-public | smmch-mine.js
Webminepool	webminepool.com/lib/base.js
Unknown miner	proofly.date | flightsy.date | gettate.trade alflying.date | flightzy.date | joytate.date zymerget.faith | nflying.win | flightzy.bid flightsy.win | zymerget.bid | nflying.bid baseballnow.press | flightsy.bid
Omine	omine.org
Browsermine	browsermine.com.cc | bmcm.pw | bmnr.pw lm-sdfhfad.ml | new BMCM | asdvhsrtsb.ml

D Human Subjects and Ethical Considerations

For the analysis of cryptojacking usage in the wild, this paper uses NetFlow statistics from a Tier 1 network operator. This data access was cleared by the institutional review board. The research team did not obtain direct access to the NetFlow data containing source and destination IP addresses as personally identifiable information, but instead provided a list of IP addresses of cryptomining proxies and mining pools to the data owner, based on which the corresponding flow records were provided with the connection's source IP protected by a salted hash.

Rendered Private: Making GLSL Execution Uniform to Prevent WebGL-based Browser Fingerprinting

Shujiang Wu, Song Li, Yinzhi Cao, and Ningfei Wang[†*]
Johns Hopkins University, [†]Lehigh University
{swu68, lsong18, yinzhi.cao}@jhu.edu, wangningfei7@gmail.com

Abstract

Browser fingerprinting, a substitute of cookies-based tracking, extracts a list of client-side features and combines them as a unique identifier for the target browser. Among all these features, one that has the highest entropy and the ability for an even sneakier purpose, i.e., cross-browser fingerprinting, is the rendering of WebGL tasks, which produce different results across different installations of the same browser on different computers, thus being considered as fingerprintable.

Such WebGL-based fingerprinting is hard to defend against, because the client browser executes a program written in OpenGL Shading Language (GLSL). To date, it remains unclear, in either the industry or the research community, about how and why the rendering of GLSL programs could lead to result discrepancies. Therefore, all the existing defenses, such as these adopted by Tor Browser, can only disable WebGL, i.e., a sacrifice of functionality over privacy, to prevent WebGL-based fingerprinting.

In this paper, we propose a novel system, called UNIGL, to rewrite GLSL programs and make uniform WebGL rendering procedure with the support of existing WebGL functionalities. Particularly, we, being the first in the community, point out that such rendering discrepancies in state-of-the-art WebGL-based fingerprinting are caused by floating-point operations. After realizing the cause, we design UNIGL so that it redefines all the floating-point operations, either explicitly written in GLSL programs or implicitly invoked by WebGL, to mitigate the fingerprinting factors.

We implemented a prototype of UNIGL as an open-source browser add-on (https://www.github.com/unigl/). We also created a demo website (http://test.unigl.org/), i.e., a modified version of an existing fingerprinting website, which directly integrates our add-on at the server-side to demonstrate the effectiveness of UNIGL. Our evaluation using crowdsourcing workers shows that UNIGL can prevent state-of-the-art WebGL-based fingerprinting with reasonable FPSes.

*The last author, Ningfei Wang, contributed to the paper when he was a master student financially supported and mentored by Dr. Yinzhi Cao.

1 Introduction

Browser fingerprinting [12, 13, 20, 23, 34, 45, 63], a substitute of traditional cookie-based approaches, is recently widely adopted by many real-world websites to track users' browsing behaviors potentially without their knowledge, leading to a violation of user privacy. In particular, a website performing browser fingerprinting collects a vector of browser-specific information called browser fingerprint, such as user agent, a list of browser plugins, and installed browser fonts, to uniquely identify the target browser.

Among all the possible fingerprintable vectors, the rendering behavior of WebGL, i.e., a Web-level standard that follows OpenGL ES 2.0 to introduce complex graphics functionalities to the browser, is an important factor that contributes the most, in terms of entropy, to the overall distinguishability of browser fingerprints [19]. Specifically, WebGL-based fingerprinting is first discovered by Mowery et al. [41], and then further explored by Cao et al. [19], who not only show that WebGL-based fingerprinting has the highest entropy among all fingerprinting factors, but also demonstrate the ability of WebGL-based fingerprinting for an even sneakier purpose, i.e., cross-browser fingerprinting, compared to traditional fingerprinting vectors like user agents.

In order to prevent WebGL from being used as a vector of browser fingerprinting, Tor Browser, the pioneer private browser for the Web makes WebGL click-to-play, i.e., disabling it by default, so that a website cannot use it for the tracking purpose. However, there exists a tradeoff between privacy and functionality: Tor Browser sacrifices an important functionality—i.e., all the computer graphics features brought by WebGL, which are particularly useful for modern web applications like games [10] and virtual reality [50]—for privacy. Specifically, according to a 2016 study [55], about 10% of Top 10K Alexa websites, including famous ones visited by billions of users such as Google Map and Earth [2], adopt WebGL to augment user experience—and the number keeps increasing as the WebGL community grows. Therefore, the research question that we want to ask in the paper is whether

a browser can allow Web applications to use WebGL and its abundant functionalities without violating users' privacy.

Before answering this question, we first take a look at how existing works prevent browser fingerprinting that does not use WebGL. There are two categories of approaches in defending against browser fingerprinting in general (e.g., these based on fonts, plugins, and user agent), which are randomization and uniformity. The former, adopted by PriVaricator [44] and some browser add-ons [1, 9], adds noise to the fingerprinting results so that an adversary cannot obtain an accurate fingerprint each time. However, according to prior work [18, 49], such randomization-based defense can be defeated if the adversary fingerprints the browser multiple times and averages the results. In addition, according to a recent work [59], inconsistencies in browser fingerprints may cause further privacy violations. Because of these concerns, Tor Browser also explicitly prefers the latter, i.e., uniformity, over randomization in its design document [49].

Therefore, our detailed research question becomes how to make uniform WebGL rendering results and prevent WebGL-based browser fingerprinting. The answer to this question is unknown in the community as indicated in Tor Browser's practice of disabling WebGL. The reason is that unlike other forms of fingerprinting (e.g., user agent and fonts), which rely on the outputs of a browser API, WebGL-based fingerprinting runs a program, i.e., a rendering task, in OpenGL Shading Language (GLSL). One possible solution, i.e., an idea floated in the design document [49] of Tor Browser without any implementation, is to adopt software rendering and make uniform WebGL rendering. However, Cao et al. [19] show that even if software rendering is enabled, WebGL rendering results are still fingerprintable.

Now, to answer the specific question of making WebGL uniform in the paper, we need to understand why a single WebGL rendering task differs much from one browser to another. From a high level, the reason is that computer graphic tasks pursue visual rather than computational uniformity. One single WebGL task on different browsers is rendered by a different combination of a variety of computer graphics rendering layers, such as browsers, graphics libraries (e.g., DirectX and OpenGL) including conversion interfaces (e.g., Almost Native Graphics Layer Engine, i.e., ANGLE), rendering mechanisms, device drivers and graphics cards. Therefore, different implementations and even versions of these various layers will lead to a computationally different rendering result. This high-level answer also partially explains the reason that software rendering cannot prevent fingerprinting: Software rendering, belonging to rendering mechanisms, is just one of the many layers that could lead to the rendering discrepancies, and it may also have different versions and implementations.

While this problem appears hard to solve unless we make uniform all the graphics layers, the root reason, after our intensive manual study and experiment, can be summarized as one surprisingly concise and abstract sentence—i.e., the results of floating-point operations on different machines are different inside and across various graphics layers, leading to rendering differences. This one-sentence, intuitive reason can be further broken down into many sub-reasons when the WebGL rendering performs various operations in different graphics layers. Let us illustrate two examples.

First, we consider the color value, i.e., RGB, in WebGL, which semantically ranges from 0 to 255 but is represented as a floating-point from 0 to 1. Therefore, a conversion is required when WebGL renders a 0–1 color value on the screen to be a 0–255 RGB value—and most importantly the conversion, i.e., a floating-point operation, will lead to rendering difference. Say one WebGL implementation, i.e., a combination of different graphics layer, multiplies the color value with 255 and applies $floor$ to convert it to the RGB value, and the other applies *round*. Then, a color value of <0.5, 0.5, 0.5> will be rendered as <127, 127, 127> in the former, but as <128, 128, 128> in the latter. This float-to-int conversion issue can be generalized in many other representation, such as alpha value, texture size, and canvas size—and also other conversion algorithms beyond $floor$ and *round*, such as linear interpolation in texture mapping.

Second, let us consider another common graphics operation involving several float multiplications and then a subtraction, i.e., we need to decide whether a given point, very close to one triangle edge, is inside the given triangle. Say, there are two WebGL implementations, one adopting 10-bit float numbers and the other a higher precision, i.e., 16-bit. The multiplication results on these two implementations differ slightly, because the former has fewer decimals than the latter. Because the given point is very close to the triangle, such slight difference will propagate to the float subtraction, leading to a positive in the former implementation but a negative the latter. Therefore, the point will be judged as either inside or outside the triangle in these two implementations, causing a rendering difference.

That said, the *key* insight of the paper is that we need to make uniform all the floating-point operations across various computer graphics layers. Specifically, we adopt two integers, one as the numerator and the other as the denominator, to simulate floating-point operations in GLSL programs so that the underlying layers, regardless of their implementation or approximation for floating-point operations, always produce the same results. When we need to feed simulated values into WebGL, we convert the value to a float based on its semantics. In the aforementioned color value example, we can use 127/255 to represent 127 and 128/255 for 128, leading to no confusions under different implementations.

While the idea is intuitively simple, the major challenge is that WebGL rendering process involves implicit floating-point operations. In order to understand this challenge, let us briefly describe the three-stage WebGL rendering process—i.e., (i) vertex rendering, (ii) rasterization and interpolation, and (iii) fragment rendering—and corresponding floating-point operations in each stage. The first stage, controlled by a

GLSL program, i.e., vertex shader, generates vertices information using graphics operations, e.g., transformation and rotation, and also associates attribute values with each vertex. These aforementioned graphics operations are all related to floating-point operations. Then, the second stage, an implicit one implemented by WebGL and not controlled by any GLSL program, generates fragments, called rasterization, and then interpolates values based on fragments using floating-point operations. Lastly, the third stage, controlled by another GLSL program, called fragment shader, colors each fragment, which also involves floating-point operations, such as texture mapping.

In this paper, we propose UNIGL, a novel system that rewrites GLSL programs and redefines all the floating-point operations in the aforementioned three stages of WebGL rendering. Specifically, UNIGL hooks JavaScript APIs—which accept GLSL programs and the corresponding parameters such as vertex and index arrays—and then rewrites both vertex and fragment shaders via three phases mapping to the three stages of WebGL rendering. First, UNIGL converts the vertex shader to a JavaScript program and executes it. During the execution, floating-point operations in vertex shader, e.g., matrix multiplication, are executed as JavaScript, thus kept with uniformity. Second, UNIGL takes the execution results of JavaScript vertex shader and feeds them into a customized rasterization and interpolation engine written as a fragment shader. In this phase, UNIGL preserves uniformity for floating-point operation implemented natively in WebGL's rasterization and interpolation module via integer simulation. Lastly, UNIGL rewrites the original fragment shader and redefines floating-point operations, such as texture mapping, in the fragment shader via integer simulation.

In designing UNIGL, we realize the following additional challenges from the viewpoint of system building.

- Backward Compatibility. We want UNIGL to be backward compatible with existing commercial Web Browsers. Specifically, we deploy UNIGL as a browser add-on, easily installable, to protect Web users' privacy.
- Performance. We want to keep the high-performance benefits brought by WebGL, especially when it runs on GPU. Therefore, we design UNIGL so that the rendering bottleneck, i.e., rasterization, interpolation and coloring, runs as a GLSL program on fragment shader possibly via GPU (depending on whether the underlying rendering mechanism is software or hardware rendering). Note that the vertex shader has to run as a JavaScript program because otherwise we do not have access to intermediate results between two shaders without modifying the browser. Because of this, we adopt multiple optimization techniques, such as caching, typed array, and code-data separation, to speed up UNIGL's vertex shader.
- Variable Number Limits. Since we choose backward compatibility, we are constrained by the limit that is enforced by the current version of WebGL. Particularly, all current implementations of WebGL have enforced a limit [26] for "uniform" variables, sometimes 256 or 1024 depending on the graphics card and operating system. Therefore, in designing UNIGL, we have to divide one rendering task iteratively until each small one can fit into and run as a fragment shader.

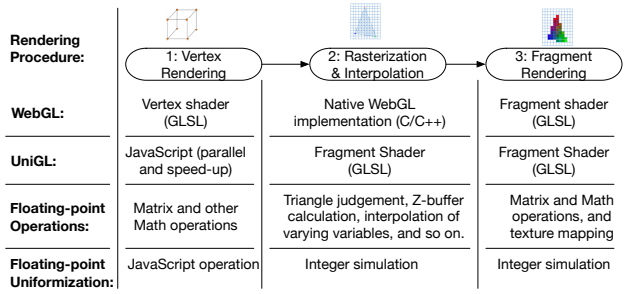


Figure 1: A High-level Overview of Rendering Procedure (i.e., the execution of GLSL programs) in both WebGL and UNIGL as well as a Description of Corresponding Floating-point Operations

We implemented a prototype of UNIGL as an open-source Google Chrome add-on, which is available at the repositories of this GitHub user (https://www.github.com/unigl/). We also created a demo website, i.e., http://test.unigl.org/, which works on modern web browsers including Chrome, Firefox, and Safari. Specifically, the demo is a modified version of Cao et al.'s fingerprinting website [19], which directly integrates our add-on at the server-side, to show that UNIGL can prevent WebGL-based browser fingerprinting.

We make the following contributions in the paper:

- We are the first to point out that WebGL-based browser fingerprinting is caused by floating-point operations.
- We design UNIGL to rewrite GLSL programs and redefine all, i.e., explicit and implicit, floating-point operations embedded deeply inside WebGL.
- We show that our prototype of UNIGL can defend against WebGL-based browser fingerprinting with reasonable FPS.

2 Overview

We give an overview of WebGL and UNIGL rendering procedure, and then present a running example.

2.1 WebGL's Rendering and Floating-point Operations

WebGL's rendering procedure can be roughly divided into three stages as shown in Figure 1. First, the vertex shader in WebGL performs operations, e.g., rotation via a matrix multiplication, related to the vertices of a computer graphics model. Specifically, the shader accepts two types of variables, i.e., attributes and uniforms, binds attributes, such as texture coordinates, to vertices, and then outputs transformed vertices, i.e., *gl_Position*, and varyings. Many operations in the vertex shader, such as matrix multiplication and Math functions like *sqrt*, have floating-point values involved.

Second, the outputs of the vertex shader are fed into the rasterization and varying interpolation module implemented

natively by WebGL. The module maps a computer graphics model to each pixel on the canvas and interpolates each varying variable based on the attribute values on each vertex. Let us illustrate three major floating-point related operations in this module. (i) The module decides whether a given pixel is inside a triangle. (ii) The module calculates the z-buffer of each point, i.e., determining whether a point is in front of or behind another on the canvas. (iii) The module interpolates a varying variable based on attributes.

Lastly, the outputs of the rasterization and varying interpolation module are fed into the fragment shader in WebGL, which paints all pixels on the canvas by assigning a value to *gl_FragColor*. All the floating-point related operations in the vertex shader, such as matrix operations, also exist in the fragment shader. Additionally, color lookup operations, such as texture mapping, need to fetch a color from a texture, which involve floating-point operations as well.

2.1.1 An Explanation of Floating-point Operation and Rendering Discrepancies

We now explain that floating-point operations will cause rendering discrepancies. Figure 2 shows a classic computer graphics model, i.e., a 3D monkey head, covered with a random texture. We render this WebGL task in Google Chrome browser on two different machines, one iMac and the other Dell with Windows system—the rendering results, though being visually the same (Left and Middle of Figure 2), are quite different if we compare them pixel by pixel (Right of Figure 2).

This model is complex with many floating-point operations and thus hard to explain the discrepancies. We now decompose the complex model into several small experiments with only one or a few types of floating-point operations for explanation.

- A Varying Experiment. We setup a thin 3×100 rectangle and then specify a varying variable spanning along the long edge from 0 to 100. The variable starts from a color <0, 0, 0> at position 0 and ends to a color <0, 0, 255> at position 100. That is, we are rendering a simple spectrum on a canvas with only one type of floating-point operation, i.e., the interpolation of a varying variable between 0 and 100.
  When we perform this varying experiment on different machines and compare the rendering results, we find that the result differences are several single-pixel lines orthogonal to the long edge. This explains that floating-point operations for interpolating varying variable will lead to rendering discrepancies.
- A Triangle Experiment. We setup a triangle on a 255×255 canvas: two vertices at <127, 0> and <0, 128>, and the third movable along the line from <255, 0> to <255, 255>. We then color the triangle with only a single color, such as black. By doing so, we create a rendering task with another type of floating-point operation, i.e., determining whether a pixel is inside a triangle.
  When we perform this triangle experiment on different machines and move the point along the edge, we find that the result difference is a single pixel close to the triangle edge for a special third vertex position. This explains that floating-point operations for triangle judgement will lead to rendering discrepancies.
- A Texture Experiment. We setup a triangle and then map a random texture onto the triangle. That is, we create a rendering task with a texture mapping operation, which has floating-point operations in color interpolation.
  When we perform this texture experiment on different machines, we find that the result differences are several pixels within the triangle. This explains that floating-point operations for texture mapping will lead to rendering discrepancies.

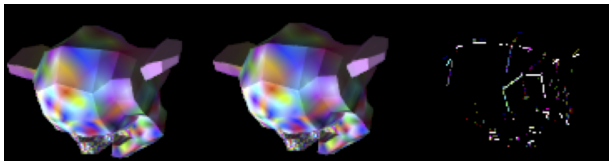


Figure 2: An Illustration of Rendering Task Difference for Browser Fingerprinting Purpose (Left: A classic monkey head model rendered in Google Chrome on an iMac, Middle: the same model rendered in Google Chrome on a Windows machine, Right: The pixel difference between these two rendering results.)

2.2 UNIGL's Rendering and Floating-point Operations

Now let us go over the three-stage rendering procedure (Figure 1) again in UNIGL and show how to make uniform floating-point operations in aforementioned steps. First, UNIGL moves the vertex shader from GLSL to JavaScript, i.e., all the floating-point operations are executed on JavaScript interpreter and thus handled by CPU without any discrepancies. Note that we need to adopt parallel workers and many other speed-up techniques to run the JavaScript version of vertex shader fast.

Second, UNIGL implements a customized rasterization and varying interpolation module via GLSL in which all the floating-point values are represented via integer simulation, and therefore, all the aforementioned floating-point operations in this module are made uniform. It is worth noting that theoretically we can also implement the module via JavaScript for uniformization, but the performance is unacceptable.

Lastly, UNIGL executes the fragment shader in GLSL, but rewrites all the floating-point operations via integer simulation. Additionally, UNIGL also implements a customized texture mapping algorithm using integers so that the color lookup operation in texture mapping is also made uniform.

2.3 A Running Example

After a high-level overview of UNIGL rendering, we now present a detailed running example to illustrate our target problem and how UNIGL solves the problem via uniformity.

JavaScript:

```
1  vers=[...]; // vertices information
2  inds=[...]; // indices information
3  ...
4  var versBufferObject = gl.createBuffer();
5  gl.bindBuffer(gl.ARRAY_BUFFER, versBufferObject);
6  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vers),
       gl.STATIC_DRAW);
7  var indexBufferObject = gl.createBuffer();
8  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBufferObject
       );
9  gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(
       inds), gl.STATIC_DRAW);
10 ...
11 gl.bindBuffer(gl.ARRAY_BUFFER, versBufferObject);
12 var posAttr=gl.getAttribLocation(program, 'aVersPos');
13 ...
14 var uMVLoc=gl.getUniformLocation(program, "uMVMatrix");
15 gl.unifomMatrix4fv(uMVLoc, [...]);
16 ...
17 // bind other attributes and uniforms
18 gl.drawElements(gl.TRIANGLES, length, gl.UNSIGNED_SHORT
       ,0);
```

Vertex Shader:

```
1  attribute vec3 aVersPos, aVersNormal;
2  attribute vec2 aTextureCoord;
3  uniform mat4 uMVMatrix, uPMatrix;
4  uniform mat3 uNMatrix;
5  uniform vec3 uAmbColor,uLtDir,uDirColor;
6  varying vec2 vTextureCoord;
7  varying vec3 vLightWeighting;
8  void main(void) {
9   gl_Position=uPMatrix * uMVMatrix * vec4(aVersPos.xyz,
        1.0);
10  vTextureCoord=aTextureCoord;
11  vLightWeighting=uAmbColor + uDirColor * max(dot(
        uNMatrix * aVersNormal, uLtDir), 0.0);
12 }
```

Fragment Shader:

```
1  varying vec2 vTextureCoord;
2  varying vec3 vLightWeighting;
3  uniform sampler2D uSampler;
4  void main(void) {
5   vec4 texColor=texture2D(uSampler, vTextureCoord);
6   gl_FragColor=vec4(texColor.rgb * vLightWeighting,
        texColor.a);
7  }
```

JavaScript:

```
1  vers = [...]; // vertices information
2  inds = [...]; // indices information
3  ... // hook JS APIs to obtain arguments
4  var UniGL_aVersPos=UniGLAttr('aVersPos');
5  var UniGL_uMVMatrix=UniGLUniform('uMVMatrix');
6  ...
7  for(UniGL_I=0;UniGL_I<AttrLen;UniGL_I++) {
8  // Transformed JS vertex shader
9   aVersPos=UniGL_aVersPos[UniGL_I];
10  uMVMatrix=UniGL_uMVMatrix;
11  ...
12  UniGL_Position=UniGLMultiply(UniGLMultiply(uPMatrix,
        uMVMatrix), UniGLVec4(UniGLExtract(aVersPos,
        [1,1,1]), 1.0));
13  vTextureCoord=aTextureCoord;
14  vLightWeighting=UniGLAdd(uAmbColor, UniGLMultiply(
        uDirColor, UniGLMax(UniGLDot(UniGLMultiply(uNMatrix,
         aVersNormal), uLtDir), 0.0)));
15 // end of transformation
16  UniGL_vTextureCoord.push(vTextureCoord);
17  UniGL_vLightWeighting.push(vLightWeighting);
18 }
19 UniGL_CallFragmentShader(UniGL_Position,
       UniGL_vTextureCoord, UniGL_vLightWeighting);
```

Fragment Shader:

```
1  #define N AttributeNumber
2  uniform ivec3 UniGL_Position[N];
3  uniform ivec2 UniGL_vTextureCoord[N];
4  uniform ivec3 UniGL_vLightWeighting[N];
5  uniform sampler2D uSampler;
6  void main(void) {
7   for (UniGL_I=0;UniGL_I<N;UniGL_I+=3) {
8    if (UniGL_InTriangleZBuffer(gl_FragCoord, UniGL_I)) {
9     vTextureCoord = UniGL_Interpolate(gl_FragCoord,
       UniGL_I, UniGL_vTextureCoord[UniGL_I],
       UniGL_vTextureCoord[UniGL_I+1], UniGL_vTextureCoord[
       UniGL_I+2]);
10    vLightWeighting = ...;
11    // Transformed fragment shader
12    ivec4 texColor=UniGL_texture2D(uSampler,
       vTextureCoord);
13    gl_FragColor=UniGL_i2f(ivec4(UniGL_Multiply(texColor
       .rgb, vLightWeighting), texColor.a));
14    // end of transformation
15   }
16  }
17 }
```

Figure 3: A Running Example (Left: The original code, Right: The code rewritten by UNIGL—JavaScript WebGL APIs are hooked, the vertex shader is rewritten as JavaScript code, and the fragment shader is still as fragment shader with floating-point operations redefined.)

We now show the source code of this rendering task of Figure 2 in Figure 3 (Left). The source code contains three parts: JavaScript, Vertex Shader and Fragment Shader. First, the JavaScript code prepares data, such as attributes, uniforms, and texture, for both vertex and fragment shaders. Lines 4–13 (Left, JavaScript) show an example of passing attributes to the vertex shader. Next, Lines 14–16 (Left, JavaScript) show another example of passing uniforms to the vertex shader. Second, the vertex shader code accepts attribute and uniform values from the JavaScript and then performs operations, i.e., Lines 9–11 (Left, Vertex Shader), on each attribute in a parallel manner. The outputs of the vertex shader to the fragment shader are a special variable, *gl_position*, which indicates the transformed vertices, and multiple varyings. Third, the fragment shader accepts outputs, i.e., vertices and varying, from the vertex shader, performs rasterization and interpolation, and then runs the code (i.e., Line 5–6, Left, Fragment Shader).

Now, let us describe the floating-point related operations that cause the rendering result difference. First, the vertices, i.e., *gl_position*, and the varyings, i.e., *vTextureCoord* and *vLightWeighting*, are passed and interpolated between the vertex and the fragment shader. Such interpolation and accompanied rasterization involve floating-point operations and may cause difference. Second, WebGL functions, such as *dot* at Line 11 (Left, Vertex Shader) and *texture2D* at Line 5 (Left, Fragment Shader), include floating-point operations and may cause difference. Lastly, floating-point operations, such as Lines 9–10 (Left, Vertex Shader) and Line 6 (Left, Fragment Shader), may cause difference.

Next, we use Figure 3 (Right) to illustrate how UNIGL rewrites the original code and prevents such differences caused by floating-point operations. First, UNIGL will hook all the JavaScript APIs, such as *bindBuffer* and *bufferData*, to obtain vertices and indices information and associate them with attributes and uniforms in the vertex shader. Then, UNIGL rewrites the original vertex shader by replacing

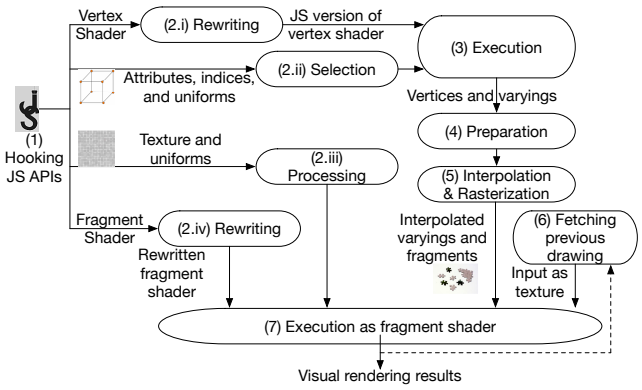


Figure 4: Overall Architecture of UNIGL (Steps 1–4 are executed as JavaScript, and Steps 5–7 as a fragment shader in a GLSL program).

floating-point operations like *dot* with JavaScript functions like *UniGLDot*, and executes it as JavaScript at Line 9–15 (Right, JavaScript). Second, UNIGL collects the execution results from JavaScript-based vertex shader (Lines 16–17, Right, JavaScript) and then feeds them into a rewritten fragment shader as uniforms (Line 2–4, Right, Fragment Shader). Third, UNIGL rewrites the original fragment shader and execute it with customized rasterization and interpolation. The rasterization process at Lines 7–8 (Right, Fragment Shader) goes through all the triangles for each pixel on the canvas and determines whether a given pixel lies inside a triangle. If yes, UNIGL calculates the z-buffer of the given pixel for the triangle, decides whether the triangle is in the front and needs to be rendered, and also calculates pixel values based on alpha. Then, the interpolation process (Lines 9–10, Right, Fragment Shader) interpolates varyings based on vertices and attributes passed from JavaScript as uniforms. Lastly, UNIGL executes a rewritten fragment shader at Lines 12–13.

3 Design

We present the design of UNIGL in this section.

3.1 System Architecture

In this subsection, we show and describe the overall architecture of UNIGL. The main function of UNIGL is encapsulated as JavaScript files executed directly in the website runtime context. The add-on part is a thin layer used to inject UNIGL into the runtime context of the target website with WebGL tasks. That is, the purpose of the add-on is to ensure the injected UNIGL JavaScript code is executed before any website code. Once the injected scripts are running, the add-on code is disposable. We make this design choice so that UNIGL can be easily transferred between browsers.

Once the main part of UNIGL is injected into the target website, UNIGL performs seven detailed steps to render a WebGL task on a canvas as shown in Figure 4. First, UNIGL needs to perform some preparation tasks outside the three stages in Figure 1. In Step (1), UNIGL hooks WebGL related JavaScript APIs to obtain four types of information: (i) vertex shader (via *shaderSource* API), (ii) inputs to vertex shader, such as attributes, indices and uniforms (via APIs, such as *bufferData* and *bindBuffer*), (iii) fragment shader (via *shaderSource* API), and (iv) inputs to fragment shader, such as texture and uniforms (via APIs, such as *texImage2D* and *texParameteri*). Then, in Step (2), UNIGL processes all the information obtained in Step (1). Particularly, UNIGL rewrites both vertex and fragment shaders in Step (2.i) and (2.iv), and extracts and prepares inputs in Step (2.ii) and (2.iii). For example, UNIGL reads indices and attributes, associates each attribute buffer with corresponding attribute variable in the vertex shader, and then prepares data based on the drawing mode (e.g., *drawElements* vs. *drawArray*, and *gl.POINTS* vs. *gl.LINES* vs. *gl.TRIANGLES*). Similarly, UNIGL also extracts texture information, such as image width and height, and texture mapping algorithm (e.g., *GL_LINEAR* vs. *GL_NEAREST*).

After that, in Step (3), UNIGL executes the rewritten vertex shader and generates outputs, i.e., vertices and varyings, which also belongs to Stage 1 in Figure 1. Next, in Step (4), UNIGL prepares inputs to the fragment shader by processing the outputs in Step (3), i.e., Stage 2 in Figure 1. Specifically, UNIGL performs backface culling on the triangles and iteratively divides the visible triangles by half so that each rendering task only contains uniforms within the limit enforced by WebGL. Then, in Step (5), UNIGL loops through all the pixels on canvas and determines whether each pixel falls inside the triangles or on the lines depending on the drawing mode. If yes, UNIGL interpolates all the varyings based on the given pixel and vertices. If no, UNIGL fetches previous drawing results from a special texture and uses the pixel color in Step (6). Lastly, UNIGL executes the rewritten fragment shader to calculates the pixel color in Step (7), i.e., performing Stage 3 in Figure 1.

3.2 Floating-point Operation Simulation

In this subsection, we present how to simulate floating-point operations using integers, i.e., the integer simulation method in Figure 1 for fragment shader. Such method is used in both Stage 2: Rasterization&Interpolation and Stage 3: Fragment Rendering, i.e., Step (5) and (7) in Figure 4.

3.2.1 Floating-point Representation and Operation

UNIGL adopts two integers, i.e., one numerator (p) and the other denominator (q), to represent an arbitrary floating-point value in the fragment shader. In this paper, we also refer the denominator as a *base*, because UNIGL can easily perform operations, such as addition and subtraction, on two values with the same base. Note that such representation also aligns well with the physical meaning of WebGL floating-point values. Take coordinates for example. The original vertices or texture coordinates are specified as an integer in terms of the canvas or texture size, which can serve as the numerator, and then the canvas or texture size can serve as the base. For

another example, 255 will be the base for all the color values, because all RGB colors are within the range of 255.

One important operation for UNIGL's floating-point numbers is to change base for a given number. Such operation will be used when UNIGL converts simulated floating-point values to real ones represented by WebGL, such as the specification of texture coordinates at Line 12 and color values at Line 13 of Figure 3 (Right, Fragment Shader). The choice of a base depends on the underlying physical meaning of the WebGL functions, e.g., UNIGL adopts 255 as the base for *gl_FragColor*.

We now explain why the base representation can make uniform rendering results across browsers. Specifically, although a color value is represented as a float value internally in WebGL, WebGL has to convert it back to an integer when rendering the color on canvas. That is, if WebGL accepts a value p in between $1/255$ and $2/255$, it may render p as 1 or 2 depending on the underlying conversion algorithm. At contrast, if WebGL accepts a value as either $1/255$ or $2/255$, the ambiguity disappears and the results are uniform across browsers. In sum, UNIGL adopts different bases according to the physical meaning when UNIGL passes the floating-point value back to WebGL.

Next, we describe how to change base for a given number especially when it does not have the required base. Intuitively, we can multiply the value with the new base and divide the product with the old base. However, such intuitive approach does not work, because the division of one integer over another involves floating-point operations in some WebGL implementations. That is, the division result differs from one browser to another. Therefore, after obtaining the quotient, UNIGL needs to search within a range (i.e., ± 1) of the quotient for the real quotient. Other than the base change operation, UNIGL also supports basic arithmetic operations, which follow fraction operations. Due to simplicity, we skip details here. One thing worth noting is that UNIGL needs to avoid result overflows—if so, UNIGL needs to increase the base to accommodate a larger value.

In the next two subsections, we show how to use such base representation of floating-point values and replace existing ones in two important types of WebGL functions.

3.2.2 Floating-point Operations in Rasterization and Interpolation

Rasterization and interpolation are automatically performed in between the vertex and fragment shaders. Specifically, for all pixels on the canvas and all triangles, rasterization needs to decide whether the given pixel is inside the triangle and, if so, calculate the z-buffer. Note that both procedures involve floating-point values and operation, which need to adopt the base representation during calculation. Then, the interpolation calculates weights for three vertices at a given triangle and then outputs the interpolated varying, i.e., a weighted sum of the attribute values at three vertices. All the weights and calculations are in the aforementioned base representation.

Another thing here, being different from normal rasterization and interpolation, is that UNIGL divides the entire, rectangle canvas into two triangles with four vertices as shown in Figure 5a instead of using the original vertices. The usage of such vertices is necessary to design a GLSL-version of rasterization, because the fragment shader will only expose a pixel to the GLSL program when the pixel is inside one triangle. Therefore, if we adopt the original vertices as the inputs to the fragment shader, some pixels, especially when they are on the edge, may be considered as inside a triangle by one browser but outside by another. The division shown in Figure 5a will consider all pixels on the canvas as being within either of these two triangles on any browser. This will give UNIGL the capability to go through all the pixelsand decide whether a pixel, such as (x,y) in Figure 5a, is inside triangles consisted of the original vertices, i.e., $(x_1,y_1)...(x_4,y_4)$.

3.2.3 Floating-point Operations in Fragment Shader

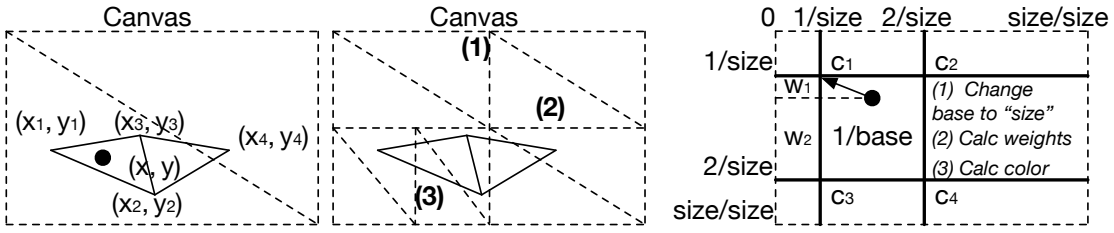
The fragment shader uses many float-point related functions, such as "texture2D", "normalize" and "sqrt". In this subsection, we use texture mapping, i.e., "texture2D", as an example to show the procedure of adopting integer simulation and replacing floating-point operations.

Texture mapping, in its normal definition, is a method of applying a two-dimensional surface upon a three-dimensional graphics model. There are many variations of texture mapping algorithms, such as linear interpolation (i.e., GL_LINEAR) and nearest neighbor (GL_NEAREST). Sometimes, mipmaps are also generated to process the texture before mapping. In this paper, we use linear interpolation as a proof of concept algorithm to show how to redefine texture mapping in UNIGL.

One of the major tasks in redefining texture mapping is to pass texture data from JavaScript to the fragment shader. The naïve method is to utilize "uniforms" just as vertices. However, because there exists a limit for the number of "uniform" variables and texture cannot be divided in multiple draws, we have to rely on existing texture information stored in WebGL.

Here is how UNIGL redefines a linear interpolation algorithm for texture mapping. UNIGL stores texture information using the default WebGL method with a nearest neighbor algorithm—therefore, WebGL will just directly fetch color values from the texture instead of performing any computation. When UNIGL has a texture coordinate, say, for example, $1/base$ in Figure 5c, UNIGL will first change the base to the size of the texture. Note that we use a square-shape texture as an example and a rectangle-shape will be similar.

Then, UNIGL fetches the colors, i.e., $c_1...c_4$, of four texture points in Figure 5c, which locate around the target texture coordinate. Because UNIGL uses the texture size as the new base, the ambiguity among browsers will disappear. Next, UNIGL calculates two weights, i.e., w_1 and w_2, based on the distance between the target texture coordinate and four texture



(a) Rasterization *(b)* Canvas Division *(c)* Texture Mapping

Figure 5: Explanation of Different UNIGL Operations. In subfigure (a), during rasterization, UNIGL needs to determine the triangle that each point belongs to. Specifically, Point (x, y) is inside the triangle consisting of vertices (x_1, y_1), (x_2, y_2), and (x_3, y_3), but outside the other triangle consisting of vertices (x_2, y_2), (x_3, y_3), and (x_4, y_4). Therefore, UNIGL will interpolate the values of all the varyings at (x, y) based on (x_1, y_1), (x_2, y_2), and (x_3, y_3). In subfigure (b), UNIGL first divides the entire canvas into two parts, i.e, left and right, and then top and bottom. After that, because the number of vertices in some divided parts is small enough to be handled by WebGL, UNIGL will not further divide such parts, like top right and top left. If the number of vertices in a part is still larger than the minimum number of allowed uniform variables, UNIGL will further divide the canvas, like cut (3). Subfigure (c) adopts a square texture as an example to show how the linear interpolation algorithm of texture mapping works in UNIGL. UNIGL calculates an interpolated color value for a given point based on four color values around this point.

Table 1: WebGL Data and Corresponding JavaScript APIs that Accept such Data

Intercepted WebGL Data	JavaScript APIs
GLSL Program	createProgram, attachShader
Shader	attachShader, getShaderSource
Buffer	bufferData
Attribute	vertexAttribPointer
Attribute Location	getAttribLocation
Uniform	uniform*** (e.g., uniform1f)
Uniform Location	getUniformLocation

fetched points—these two weights can be in any base. Lastly, UNIGL calculates the color for the target texture coordinate using these two weights, i.e., $color = w_2w_2c_1 + w_1w_2c_2 + w_2w_1c_3 + w_1w_1c_4$.

3.3 Rendering Preparation

In this subsection, we introduce how to prepare inputs to both rewritten vertex and fragment shaders, i.e., Steps (1), (4), and (6). These steps do not have direct involvement with floating-point values, but are essential in preparing the vertex and fragment shaders in UNIGL.

JavaScript Hooking and Data Extraction. UNIGL hooks WebGL-related JavaScript APIs in Step (1) of Figure 4. Specifically, UNIGL utilizes the dynamic feature of JavaScript to redefine such JavaScript APIs, intercepts all arguments, i.e., parameters to WebGL, processes the arguments, and stores them in an internal data structure of UNIGL. Table 1 shows a list of WebGL data and corresponding JavaScript APIs. Such data can be roughly divided into two major categories: programs and inputs. "GLSL program" and "shader data" in Table 1 are intercepted by UNIGL for rewriting purpose. All others, such as "attribute" and "uniform", are inputs to the program—UNIGL intercepts them and then feeds them to the rewritten programs. Both "attribute" and "uniform" refer to the data, e.g., colors and vertices, and "attribute location" and "uniform location" are the corresponding variables defined in the shaders.

Backface Culling. UNIGL adopts backface culling [11] in Step (4) of Figure 4 to determine whether a polygon, such as a triangle, of a graphical object is visible. Specifically, UNIGL calculates the normal vector of all the triangles and filters these triangles of which the normal vector does not face the camera.

Rendering Task Division. Rendering task division, part of Step (4), is designed specifically in UNIGL to overcome the limit of the uniform variable numbers in fragment shader. Figure 5b shows an illustration of such division. UNIGL first divides the canvas vertically by half, e.g., division (1) in Figure 5b, and then horizontally, e.g., division (2). Such vertical or horizontal division will be performed alternatively in each iteration on a small region until the number of vertices in that region is smaller than the limit enforced by WebGL implemented in a specific browser. Note that if a triangle is partially inside a region, e.g., the right corner region in Figure 5b, UNIGL will count all three vertices towards the limit. The reason is that all three vertices are required to determine whether a given pixel lies inside a triangle.

Reading Previous Draw Results. In this part, we describe how UNIGL handles multiple draws in Step (6). For example, a WebGL program may first draw a triangle by calling one GLSL program and then a rectangle by calling another. In such multiple draws, the rendering results are treated as a background in the latest draw. Because UNIGL goes through all the pixels in the fragment shader each time, UNIGL needs to read previous draw results. Specifically, UNIGL relies on the readPixels API to obtain the canvas contents, constructs a texture based on the contents and then passes the texture to the fragment shader. Then, in the fragment shader, UNIGL first reads the color value from the texture, and assigns the color to *gl_FragColor*. Later on, if the current drawing renders a

```
1 attribute vec2 vertPosition;
2 void main() {
3  gl_Position =  vec4(vertPosition, 0.0, 1.0);
4  gl_PointSize = 1.0;
5 }
```

Figure 6: Dummy Vertex Shader (The vertex shader performs a self-mapping with a z-value as 0.0 and a w-value as 1.0. The point size is also set to be 1.0.)

color on this pixel, the assigned color will be overwritten; otherwise, the assigned color is treated as the background.

3.4 Rewriting and Rendering

In this subsection, we describe UNIGL's rewriting and rendering process, i.e., Step (2). In both Steps (2.i) and (2.iv), UNIGL preprocesses the GLSL program, i.e., replacing preprocessor directives with a hash symbol at the beginning, and then parses the processed program into Abstract Syntax Tree (AST). We then discuss how to rewrite vertex and fragment shaders separately.

- Vertex Shader. UNIGL traverses through the AST, redefines corresponding node, e.g., replacing the operator plus with a function UniGL_Plus, and then converts the AST back to either JavaScript or GLSL code. All the type information is kept the same because UNIGL has redefined all the types in JavaScript to be the same as in GLSL. Note that one additional step is that UNIGL needs to divide the *w* value of *gl_Position* from the *x*, *y* and *z* values so that UNIGL can switch the rendering results from an orthographic view to a perspective view. This step was performed implicitly in the original WebGL, and UNIGL needs to do so as well.
- Fragment Shader. UNIGL traverses through the AST and redefines the following three types of nodes: (i) operators, (ii) constant float number, and (iii) type information related to floating point values. First, UNIGL redefines all the existing operations, such as multiply, with the corresponding GLSL function, such as UniGL_Multiply. Second, UNIGL converts all the constant floating point values, e.g., 0.32 as an alpha value, to our base representation, e.g., 32 as the value and 100 as the base for all alphas. Lastly, UNIGL need to convert all the types related to floating point values, such as float and vec3, to the corresponding integer type, such as int and ivec3.

Note that WebGL requires that all GLSL programs have both vertex and fragment shaders. Therefore, UNIGL executes a dummy vertex shader as shown in Figure 6. The dummy vertex shader needs to set gl_PointSize, because the default value also differs on different OSes, e.g., Mac vs. Windows.

3.5 Execution of JavaScript Vertex Shader and Corresponding Floating-point Operations

In this subsection, we describe the execution of JavaScript vertex shader in Step (3). All the floating-point operations in the vertex shader are thus executed on CPUs. We adopt five runtime optimizations to speed up the execution as shown below.

- Multiple Web Workers. Once one frame comes in, UNIGL puts the vertex rendering tasks of the frame into a queue. Then, multiple workers keep fetching the tasks from the queue, and run them in parallel.
- Caching. UNIGL adopts a caching mechanism for matrix operations, e.g., *UniGLAdd* and *UniGLMultiply* at Lines 12 and 14 of Figure 3 (Right, JavaScript), in Step (3). That is, UNIGL will cache the calculation results of a matrix operation, e.g., the multiplication of two matrices. If the rewritten vertex shader asks for the results of an operation upon the same matrices, UNIGL will directly return the result directly instead of calculating it again.
- Typed, fixed-size Array. UNIGL adopts typed arrays, such as Float32Array, instead of the normal JavaScript array to store data. The reason is that typed array are stored in a contiguous memory region can fast accessed by the browser. In addition, UNIGL needs to specify the length to avoid array resizing. The reason is that array resizing may involve additional memory allocation and data copy. Note that we also need to avoid using some heavy JavaScript array operations, such as Array.map().
- Code and Data Separation. UNIGL separates the code and the data for the rewritten shader. That is, the code will be prepared in the hooked *useProgram* API, which does not have an influence on the runtime performance, and JIT-ed for performance speed-up. Note that UNIGL triggers the JIT engine by executing the code once with initial data. Then, further data will be prepared in the draw stage.
- WebAssembly. UNIGL executes some heavyweight operations, such as matrix multiplication, using native code like WebAssembly.

4 Implementation

We implemented a prototype of the core function of UNIGL with around 8,500 lines of JavaScript code, around 650 lines of GLSL code, and around 3,000 lines of code for auxiliary components such as WebAssembly and add-on. The rewriting component of UNIGL is modified from one open-source GitHub repository (namely, https://github.com/stackgl/glsl-transpiler). Other than this repository, we also use several other libraries, such as glMatrix.

UNIGL is open-source, available at repositories under this GitHub user (https://www.github.com/unigl/). We also provide a demo at this url (http://test.unigl.org/), a modified version of Cao et al. [19]'s fingerprinting websites, to demonstrate that UNIGL can prevent state-of-the-art WebGL-based browser fingerprinting. All rendering results are the same in this new, UNIGL rewritten version.

5 Evaluation

We evaluate UNIGL prototype on three metrics: anti-fingerprinting capability, performance, compatibility, and CPU energy consumption.

5.1 Anti-fingerprinting Capability

We adopt a state-of-the-art WebGL-based browser fingerprinting work [19] as our benchmark to evaluate the anti-fingerprinting capability and performance of UNIGL. Specifically, the benchmark contains 17 different WebGL rendering tasks including plain WebGL tasks and these relying on three.js, a WebGL library, to explore various WebGL features, such as varyings, light and texture. The first column of Table 2 shows the names of all the rendering tasks and the second column a rendering result example on a dell desktop installed with Windows 10.

We evaluate UNIGL by asking Amazon Mechanical Turks to visit our website—including a demo site of UNIGL together with the original fingerprinting site from Cao et al.—using Firefox, Chrome and Safari. In total, we have collected 656 fingerprints from these three types of browsers. Among all the 656 fingerprints, UNIGL only renders one unique fingerprint for each rendering task across different browsers. This unique fingerprint, being visually the same to the original rendering result, is shown in the "Example" column under UNIGL of Table 2. As a comparison, we also show the number of unique fingerprints of Cao et al. in the "# Unique Results" column under "Original" of Table 2—this confirms Cao et al.'s findings that WebGL is a high-entropy vector for browser fingerprinting. We also list some more statistics in Appendix A.

In addition to the Amazon Mechanical Turk experiment, we also perform a local experiment that enumerates a large varieties of factors across the graphics layers. Specifically, we test UNIGL with the following different settings: OS (Windows 7, 8, 10, iOS 10.14.1, and Ubuntu 18.04), graphics card (Nvidia Geforce GTX 1070, Intel Iris plus Graphics 640, AMD Radeon R9 M390, and AMD Radeon HD 6770m), drivers (Nvidia, Intel, and AMD drivers), screen resolution (all these provided by the OS, such as 2560x1440 and 1920x1080), and DPI scaling (100%, 125%, 150%, 175%, 200%, and 225%). UNIGL only produces one unique result, which is the same as the Amazon Mechanical Turk experiment.

5.2 Performance

We test the performance of UNIGL by using both micro- and macro-benchmarks. All the experiments, except for the crowdsourced results in Table 2, are performed on an iMac – the machine has an Intel Core i5, 3.2 Hz, 4-core CPU, 24 GB memory, and an AMD Radeon R9 M390 GPU with 2048 MB VRAM.

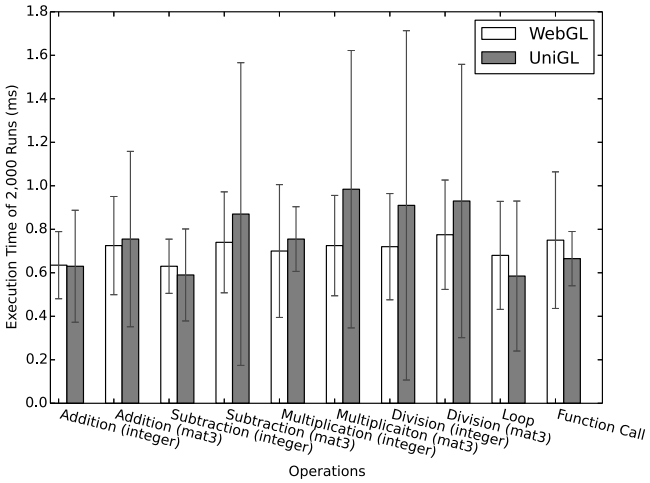


Figure 7: Micro-benchmark of Vertex Shader.

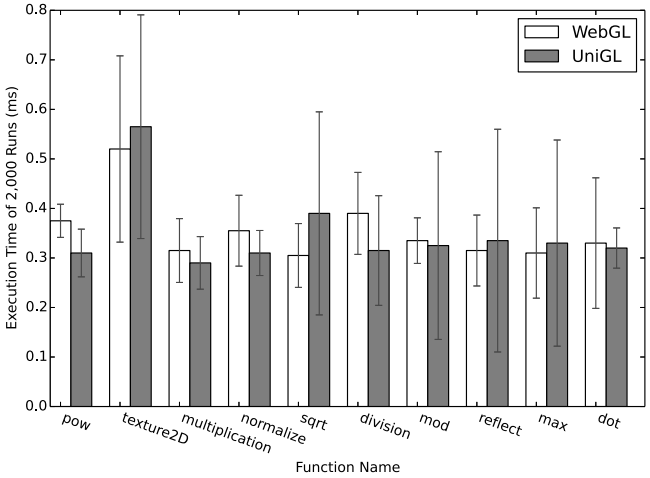


Figure 8: Micro-benchmark of Fragment Shader (all the operations in WebGL are using floating-point values; these in UNIGL adopts integer simulation).

5.2.1 Micro-benchmark

In the micro-benchmark, we test several atomic WebGL operations and compare the original version with the one defined in UNIGL. Specifically, we run each operation in either fragment or vertex shader for 2,000 times and calculate the interval between two draws. Each experiment is performed 20 times to obtain a standard deviation. Note that we adopt a simple model, i.e., a cube, in the micro-benchmark experiment. When we are testing one shader, the other shader will contain a one-line, dummy statement, i.e., the assignment of either *gl_Position* or *gl_FragColor*.

Figure 7 shows the micro-benchmark performance of the vertex shader. The vertex shader of UNIGL outperforms the one written in GLSL in some aspects, such as these operations that have integers involved. The reason is that CPU is well designed for integer operation when compared with GPU. On the contrary, the original shader written in GLSL is better at matrix operations, because such operations can be performed in parallel using shading languages.

Table 2: Macro-benchmark WebGL Tasks [19] and Corresponding Rendering Results with UNIGL ("# Vertices" means the number of vertices in the model, which are two per line segment in a 2D model and three per triangle in a 3D model. "Example" is one rendering example collected from users. In "# Unique Results" columns, X/Y means the number of unique fingerprints collected from all the users out of the total number of fingerprints. "FPS" means frames per second—which is around 60 Hz due to the screen refresh rate.)

WebGL Task	# Vertices	Original					UNIGL		
		Example	# Unique Results			FPS	Example	# Unique Results	FPS
			Chrome	Firefox	Safari				
Curve and Line	262		74/496	23/108	18/52	60.32±0.38		1/656	61.77±0.54
Curve and Line (AA)	262		83/496	32/108	21/52	60.78±0.54		1/656	61.83±0.97
Cube	36		4/496	2/108	5/52	60.67±0.49		1/656	62.50±0.80
Cube (AA)	36		55/496	20/108	23/52	60.46±0.15		1/656	62.39±1.36
Cube (Camera)	36		56/496	18/108	7/52	60.02±0.22		1/656	61.75±1.32
Monkey head (Texture)	2,904		5/496	18/108	2/52	60.14±1.17		1/656	61.88±1.61
Monkey head (Light)	2,904		36/496	11/108	15/52	59.95±0.60		1/656	61.07±1.02
Two models (Light)	2,988		44/496	18/108	14/52	60.02±1.19		1/656	61.90±0.97
Two models (Complex light)	2,988		52/496	6/108	20/52	60.18±0.40		1/656	60.02±1.13
Two models (Texture)	2,988		79/496	31/108	21/52	60.31±0.54		1/656	62.33±1.22
Two models (Transparency)	2,988		87/496	25/108	22/52	59.97±1.13		1/656	60.13±1.76
Two models (Tex&Light)	2,988		38/496	14/108	11/52	60.04±0.31		1/656	59.74±0.75
Thousands of rings (three.js)	5,376		53/496	25/108	15/52	60.52±0.53		1/656	57.47±1.87
Clipping plane (three.js)	36		44/496	14/108	19/52	59.98±0.44		1/656	59.67±1.29
Bubble (three.js)	974		49/496	17/108	22/52	60.20±1.52		1/656	60.07±1.43
Compressed Texture (three.js)	98		72/496	21/108	19/52	60.04±0.56		1/656	59.59±0.73
Shadow (three.js)	156		53/496	17/108	19/52	59.84±0.35		1/656	60.12±1.02
Combined fingerprint			123/496	41/108	26/52			1/656	

Figure 8 shows the micro-benchmark performance of the fragment shader. Similar to the vertex shader, while UNIGL is slower than the original WebGL in some cases, such as "texture2D" as UNIGL redefines the function, it is worth noting that UNIGL is faster than the original WebGL in many other cases, such as "pow" and "multiplication". The reason is that an integer operation is indeed sometimes cheaper than a floating-point one. For example, it takes less time to multiply two integers than two floating-point values. Note that we are referring to integer operations that exist in the original vertex shader during this discussion. Floating-point values are still represented as floats in the vertex shader of UNIGL.

5.2.2 Macro-benchmark

In this subsection, we use the WebGL tasks provided by Cao et al. [19] as our macro-benchmark to measure the FPS of these rendered by UNIGL. The column "FPS" under UNIGL of Table 2 shows the FPS of each rendering task and we also show the FPS without UNIGL, i.e., these rendered directly by WebGL in the same table. The performance of UNIGL can satisfy the required screen refresh rate, i.e., 60 Hz. The FPS of UNIGL for all the tasks are similar to the original one rendered by WebGL alone.

There are two things worth noting. First, the FPS of UNIGL is even sometimes a little bit higher than the one of WebGL. The reason is that when the model is simple, our highly optimized vertex shader with the help of WebAssembly is faster than the original one. Second, the FPSes of both UNIGL and WebGL are a little bit higher than 60 Hz in some tasks, because modern browsers reduce the precision of *performance.now* to prevent timing attacks [7], which may

Table 3: Overhead Breakdown for the "Two models (Complex light)" Task

Procedure	Overhead
Data Preparation	0.08±0.07ms
Backface Culling	2.16±0.06ms
Vertex Shader	2.26±0.18ms
Rendering Task Division	5.34±0.73ms
Fragment Shader	6.51±0.85ms
Total	16.35±0.74ms

Table 4: Vertex Shader Optimization (all numbers are averaged from 10 experiments and rounded to ms)

Unoptimized Vertex Shader	110 ms
Result Caching of Matrix Operation	-24 ms
Typed, Fixed-size Array for Data	-22 ms
Code and Data Separation	-39 ms
Parallelization	-19 ms
WebAssembly	-4 ms
Optimized Vertex Shader	2 ms

lead to a small measurement error. Such measurement errors are consistent across UNIGL and WebGL.

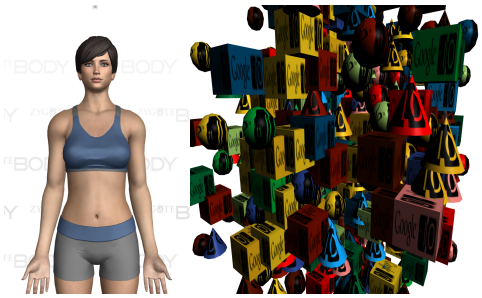
We further look at one specific task, i.e., "Two models (Complex light)", and analyze the overhead brought by UNIGL. Table 3 shows the overhead breakdown by different procedures of UNIGL. The rendering task division and fragment shader are the most time-consuming procedures, i.e., each taking one third of the entire overhead. Both data preparation and backface culling are lightweight, taking up a small portion of the overhead.

We then look at how our optimization reduces overhead of UNIGL, especially the vertex shader, using the same task. Specifically, we evaluate five optimizations and their impact on the performance in Table 4. The unoptimized vertex shader in JavaScript takes 110ms and each optimization reduces the overhead to some degree. Code and data separation is the most effective one, i.e., about 40ms reduction, because JIT engine will execute code natively rather than on an interpreter. Then, both caching and typed, fixed-size array speed up the shader by reducing around 20ms, and parallelization also reduces the overhead by around 19ms. Lastly, if we apply WebAssembly optimization, the overhead can also be reduced by 4 ms.

5.3 Compatibility

In this section, we evaluate the compatibility of UNIGL with existing WebGL applications. Specifically, in addition to the WebGL tasks from Cao et al. [19], we run UNIGL using two other real-world WebGL applications shown below:

- Zygote Body. Zygote Body (https://www.zygotebody.com/), formerly known as Google Body, is created by Zygote Media Group to renders a manipulable 3D model of human body from outside, such as skins, muscle tissues and hairs, to inside, such as blood vessels and skeletons.



(a) Zygote Body *(b)* Google I/O Application

Figure 9: Two Screenshots of Zygote (Google) Body and Google I/O Application Rendered by UNIGL

- Google I/O 2011 Applications. Google has presented WebGL applications (https://webglsamples.org/google-io/2011/index.html) at its I/O event in 2011 to show the new technique and performance.

Our evaluation result shows that UNIGL is compatible with both applications. First, we run UNIGL with Zygote Body—the human body is rendered correctly with no visual difference. We can also manipulate it by looking at different layers, such as skeleton and muscle. Second, we run UNIGL with Google I/O applications—all the objects are shown and displayed correctly with the right texture, moving on the canvas the same as ones with WebGL directly. Two screenshots of both applications are also shown in Figure 9: Figure 9a shows the front page of Zygote Body, a default rendering of a human, and Figure 9b a screenshot of one Google I/O application after it runs for two seconds.

5.4 CPU Energy Consumption

In this section, we evaluate the CPU package power of our macro-benchmark. Specifically, we use CPUID's HWMonitor [3], a program that monitors PC systems' main health sensors, to calculate the CPU package power consumption for each macro-benchmark task. Note that our experiment is performed on a Dell Desktop because HWMonitor is only available on PC systems.

Figure 10 shows our evaluation results, i.e., CPU package power consumption when each macro-benchmark task is rendered by WebGL (hardware rendering), WebGL (software rendering), and UNIGL. The power consumption for WebGL (hardware rendering) is the smallest for all the tasks because hardware rendering relies on GPU to perform computation. UNIGL is the second, because UNIGL relies on CPU for rendering vertex shader, but GPU for fragment shader. WebGL (software rendering) is the highest as all the rendering tasks are performed on CPU.

It is also worth noting that CPU package powers for "thousands of rings" and "clipping plane" are the highest compared with other tasks. The reason is that the vertex shader for both tasks are computationally heavy. Take "thousands of rings" for example. The position and shapes for all the rings are calculated in the vertex shader.

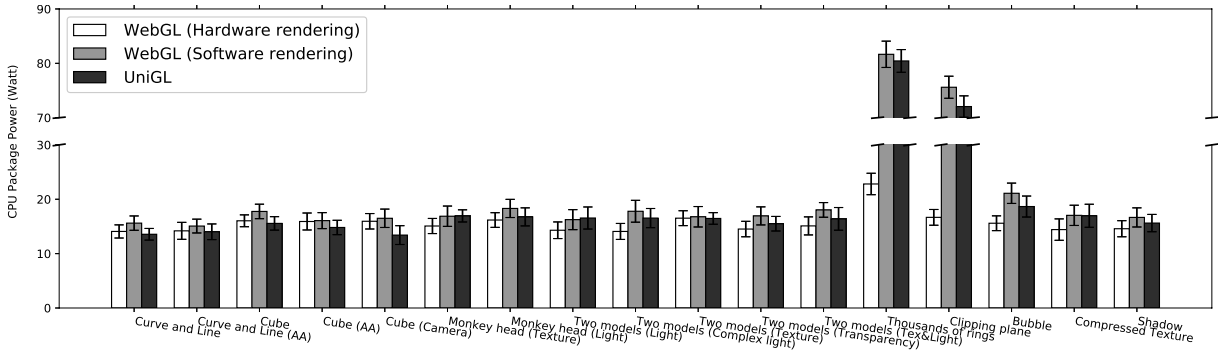


Figure 10: CPU Package Power Comparison among WebGL (hardware rendering), WebGL (software rendering), and UNIGL. (Note that all other WebGLs mentioned in the paper except for this figure and corresponding texts refer to the default hardware rendering.)

6 Discussion

We discuss several issues regarding UNIGL in this section.

- Timing side-channel attacks. Timing information, such as the rendering speed of WebGL tasks, may also be used for fingerprinting. For example, Naghibijouybari et al. [43] shows that timing side channels exist in GPU, e.g., WebGL rendering tasks. Similarly, the execution time of floating-point operations can also be used as a timing side channels [29, 51]. Many existing works, such as Deterministic Browser [18], JavaScript Zero [54], and CTFP [14], are proposed to defend against such timing channels, and therefore we would consider such timing-based fingerprinting out of scope of the paper.
- Self-modifying code (i.e., an strong adversary aware of UNIGL). There are three techniques used to prevent self-modifying code that is aware of the existence of UNIGL from tampering the UNIGL code and logics. First, UNIGL adopts anonymous closure to encapsulate all the core code of UNIGL from access by any potentially malicious website JavaScript. Specifically, anonymous closure makes sure that all the private variables and original WebGL functions, such as drawElements, are securely protected. Second, UNIGL obtains all the system object, such as "undefined", to avoid tampering from an adversary. Lastly, the add-on code injects the main function of UNIGL as the first script to execute before any other website JavaScript. It is worth noting that we are aware that there exists an active Chrome bug [4] at the time when we write the paper, which is about "document_start" hook on child frames. We believe that this bug should be fixed to follow the specification of Chrome extension.
- WebGL Vulnerability. WebGL may expose some low-level vulnerabilities in device drivers to web applications [62]. As discussed in Milkomeda [62], WebGL has already imported security checks to prevent an attacker exploiting such vulnerabilities.
- Fingerprinting via WebGL meta-information. We realize that not only the rendering behaviors of WebGL tasks but also the meta-information of WebGL engine and implementation can be used for fingerprinting. For example, as shown by this website (https://browserleaks.com/webgl), different WebGL meta-information, such as vendor, renderer, and shader parameters, can all be used as part of browser fingerprinting.

  We would like to point out that such fingerprinting relying on WebGL meta-information is relatively easy to prevent as shown by Tor Browser's in uniformization of the reported values of such meta-information. Specifically, Tor Browser changes and makes uniform the return values of WebGL meta-information functions, such as getParameter(), getSupportedExtensions(), and getExtension(). Therefore, we encourage one to rely on Tor Browser for prevention of such fingerprinting.
- Floating-point Value Precision. We now discuss the precision of floating-point values used in UNIGL. Our integer simulation of floating-point values can meet the needs of graphics tasks, because WebGL does not need a high-precision definition of floating-point variables. Specifically, the semantic meaning of many WebGL variables only requires a relatively low-precision value. Take color values for example: Each color value only ranges from 0 to 255 and thus a high-precision, 16-bit floating-point value will not represent more colors. In fact, many WebGL implementation only support a mediump (10 bits) or lowp (8 bits) for float variables.
- Future WebGL-based fingerprinting. State-of-the-art WebGL fingerprinting is based on differences in floating-point operations. We empirically verify this via carefully-designed experiments following WebGL specifications in Section 2.1.1 and our evaluation of UNIGL against Cao et al.'s WebGL fingerprinting [19]. Future WebGL fingerprinting techniques and those that do not rely on WebGL are out of scope of the paper.
- Ethic concerns. We discussed with our Institutional Review Board (IRB) about the ethics of the proposed research and experiment, because we require a human to run our experiment on their machines. The conclusion is that the proposed research does not require IRB approval. The reason is that although browser fingerprinting may be used to collect private information, the fingerprint itself, just like

cookies, is just an identifier, which does not contain any private information. Our experiment only collects fingerprints but not any private information associated with the fingerprint.

7 Related Work

We discuss related work in this section.

Browser Fingerprinting. Browser fingerprinting is a second-generation web tracking that goes beyond cookies or super cookies [30–33, 35, 53] to utilize inherent features inside web browsers. For example, there are many works [12, 13, 20, 23, 45, 63] performing measurement works on browser fingerprinting. Browser fingerprinting can rely on many features. Laperdrix et al. [34], i.e., AmIUnique, is a comprehensive study on 17 features of browser fingerprinting. Then, Vastel et al. studied the dynamics of fingerprints [59], i.e., how browser fingerprints change over time, and the privacy implication of browser fingerprint inconsistencies [60]. Researchers also study specific features of browser fingerprinting, such as fonts [23], AudioContext [20], and JavaScript engine [40, 42]. The defense target of the paper is WebGL-based fingerprinting, which was first proposed by Mowery et al. [41] and then thoroughly examined by Cao et al. [19].

Floating-point Timing Channel. A floating-point timing channel [29, 51] of web browser refers to that the duration of a floating-point operation can be used to break same-origin policy. Other than floating-point timing channel, many other timing channels [16, 22, 24, 25, 28, 39, 46, 57, 58, 64, 65] have also been studied. As a comparison, the side channel studied in the paper is caused by the different results of floating-point operations, such as conversion from low resolution to high resolution, but not the different duration of floating-point operations. Therefore, we consider that floating-point timing channels are out of scope of the paper, and one should refer to existing works [18] for solutions.

Defense against Fingerprinting. To the best of knowledge, none of existing works can defend against WebGL-based browser fingerprinting while still preserving its functionality. The reason is that we believe we are the first to point out floating-point operations are the root cause for WebGL-based browser fingerprinting. UNIGL is also the first system to find out and then redefines such floating-point operations that cause rendering discrepancies.

In the related work, Tor Browser [48], as discussed, is the pioneer work in defending against browser fingerprinting, but it disables WebGL for privacy. PriVaricator [44] adds noises to browser fingerprinting, which can be defeated if the adversary runs the fingerprinting multiple times. Similarly, Multilogin Browser [5] also creates a virtual browser profile with a random fingerprint. TrackingFree [47] only defends against the first-generation web tracking, i.e., these based on cookies or super cookies, but not browser fingerprinting.

Rewriting Technique. In the past, both academia [21, 27, 38] and industry [6, 8] have adopted rewriting techniques in different scenarios. In academia, WebShield [38] and BrowserShield [52] rewrite webpages in a proxy to enable web defense techniques. Erlingsson et al. [21] enforce cyber security policies by rewriting binaries. In industry, ShapeSecurity [8], a commercial company, provide products to rewrite websites and prevent bots and malware. Google's PageSpeed Module [6] also rewrites webpages to improve their performance.

As a comparison, there are unique challenges in rewriting GLSL languages in UNIGL, because it contains two shaders and many internal variables, such as varyings and uniforms. Specifically, UNIGL not only redefines floating-point operations, but also implements rasterization and interpolation in fragment shader so that corresponding floating-point operations in these two procedures can be made uniform.

Determinism. Determinism is a technique used to defend against side-channel attacks. For example, StopWatch [36,37], Deterministic Browser [18], and DeterLand [61] adopt determinism to defend against timing channels. Burias et al. [17] design a deterministic information-flow control system to defend against cache attacks and then Stefan et al. [56] prove that such cache attacks are still possible given a reference clock. Aviram et al. [15] use provider-enforced deterministic execution to prevent timing channels within a shared cloud domain. As a comparison, UNIGL makes the execution results the same but not the execution time across different browsers.

8 Conclusion

In this paper, we propose UNIGL, a novel system that rewrites GLSL programs and renders them uniformly across different browsers, thus preventing WebGL-based browser fingerprinting. UNIGL redefines all the floating-point operations, either explicitly written in the shader, or implicitly invoked by the WebGL system.

We implemented an open-source prototype of UNIGL as a browser add-on. Our evaluation shows that UNIGL can defend against state-of-the-art WebGL-based fingerprinting, i.e., there exists only one rendering result when Amazon Mechanical Turks visit our demo website from different browsers on different machines. Our evaluation also shows that the performance of UNIGL can satisfy the needs for the screen refresh rate, i.e., the FPSes of graphics tasks rendered by UNIGL are around 60 Hz.

In the future, we believe that browser vendors should integrate UNIGL natively into browsers. If they choose to do so, they can directly use integer simulation for all the components including vertex shader, rasterization & interpolation engine, and fragment shader, because the outputs from vertex shader, though unavailable in the JavaScript-level, are accessible and can be made uniform directly in native browser. Additionally, a native implementation does not need the rendering task division step in UNIGL because there will be no "uniform" variable limit in the low-level.

Acknowledgment

We would like to thank our shepherd, Ben Stock, and anonymous reviewers for their helpful comments and feedback. This work was supported in part by National Science Foundation (NSF) grant CNS-18-12870 and an Amazon Research Award. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF or Amazon.

References

[1] Canvas defender. https://multiloginapp.com/canvasdefender-browser-extension/.

[2] Google maps meets webgl. https://www.youtube.com/watch?v=X3EO_zehMkM.

[3] Hwmonitor—voltages, temperatures and fans speed monitoring. https://www.cpuid.com/softwares/hwmonitor.html.

[4] Issue 793217: “document_start" hook on child frames should fire before control is returned to the parent frame. https://bugs.chromium.org/p/chromium/issues/detail?id=793217.

[5] Multilogin. https://multilogin.com/.

[6] Pagespeed module: open-source server modules that optimize your site automatically. https://developers.google.com/speed/pagespeed/module/.

[7] Reduce resolution of performance.now to prevent timing attacks. https://bugs.chromium.org/p/chromium/issues/detail?id=506723.

[8] Shape security. https://www.shapesecurity.com/.

[9] Trackoff privacy software. https://www.trackoff.com/en.

[10] Webgl games. https://www.crazygames.com/t/webgl.

[11] [wikipedia] back-face culling. https://en.wikipedia.org/wiki/Back-face_culling.

[12] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juarez, Arvind Narayanan, and Claudia Diaz. The web never forgets: Persistent tracking mechanisms in the wild. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’14, pages 674–689, New York, NY, USA, 2014. ACM.

[13] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. FPDetective: Dusting the web for fingerprinters. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’13, pages 1129–1140, 2013.

[14] Marc Andrysco, Andres Nötzli, Fraser Brown, Ranjit Jhala, and Deian Stefan. Towards verified, constant-time floating point operations. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’18, pages 1369–1382, New York, NY, USA, 2018. ACM.

[15] Amittai Aviram, Sen Hu, Bryan Ford, and Ramakrishna Gummadi. Determinating timing channels in compute clouds. In *Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop*, CCSW ’10, pages 103–108, New York, NY, USA, 2010. ACM.

[16] Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *Proceedings of the 16th International Conference on World Wide Web*, WWW ’07, pages 621–628, New York, NY, USA, 2007. ACM.

[17] Pablo Buiras, Amit Levy, Deian Stefan, Alejandro Russo, and David Mazieres. A library for removing cache-based attacks in concurrent information flow systems. In *International Symposium on Trustworthy Global Computing*, pages 199–216. Springer, 2013.

[18] Yinzhi Cao, Zhanhao Chen, Song Li, and Shujiang Wu. Deterministic browser. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security*, CCS ’17, 2017.

[19] Yinzhi Cao, Song Li, and Erik Wijmans. (cross-)browser fingerprinting via os and hardware level features. In *Annual Network and Distributed System Security Symposium*, NDSS, 2017.

[20] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS ’16, 2016.

[21] Ulfar Erlingsson and Fred B Schneider. Irm enforcement of java stack inspection. In *IEEE S&P*, 2000.

[22] Edward W. Felten and Michael A. Schneider. Timing attacks on web privacy. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, CCS ’00, pages 25–32, New York, NY, USA, 2000. ACM.

[23] David Fifield and Serge Egelman. Fingerprinting web users through font metrics. In *Financial Cryptography and Data Security*, pages 107–124. Springer, 2015.

[24] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. Aslr on the line: Practical cache attacks on the mmu. In *Annual Network and Distributed System Security Symposium*, NDSS, 2017.

[25] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP ’13, pages 191–205, Washington, DC, USA, 2013. IEEE Computer Society.

[26] Darius Kazemi. Counting uniforms in webgl. https://bocoup.com/blog/counting-uniforms-in-webgl.

[27] Emre Kiciman and Benjamin Livshits. Ajaxscope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. In *SIGOPS*, 2007.

[28] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO ’96, pages 104–113, London, UK, UK, 1996. Springer-Verlag.

[29] David Kohlbrenner and Hovav Shacham. On the effectiveness of mitigations against floating-point timing channels. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, pages 69–81, 2017.

[30] Balachander Krishnamurthy, Konstantin Naryshkin, and Craig Wills. Privacy leakage vs. protection measures: the growing disconnect. In *Web 2.0 Security and Privacy Workshop*, 2011.

[31] Balachander Krishnamurthy and Craig Wills. Privacy diffusion on the web: a longitudinal perspective. In *Proceedings of the 18th international conference on World wide web*, pages 541–550. ACM, 2009.

[32] Balachander Krishnamurthy and Craig E Wills. Generating a privacy footprint on the internet. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 65–70. ACM, 2006.

[33] Balachander Krishnamurthy and Craig E Wills. Characterizing privacy in online social networks. In *Proceedings of the first workshop on Online social networks*, pages 37–42. ACM, 2008.

[34] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *37th IEEE Symposium on Security and Privacy (S&P 2016)*, 2016.

[35] Adam Lerner, Anna Kornfeld Simpson, Tadayoshi Kohno, and Franziska Roesner. Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016. In *25th USENIX Security Symposium (USENIX Security 16)*, Austin, TX, 2016.

[36] Peng Li, Debin Gao, and Michael K. Reiter. Mitigating access-driven timing channels in clouds using stopwatch. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Budapest, Hungary, June 24-27, 2013*, pages 1–12, 2013.

[37] Peng Li, Debin Gao, and Michael K. Reiter. Stopwatch: A cloud architecture for timing channel mitigation. *ACM Trans. Inf. Syst. Secur.*, 17(2):8:1–8:28, November 2014.

[38] Zhichun Li, Yi Tang, Yinzhi Cao, Vaibhav Rastogi, Yan Chen, Bin Liu, and Clint Sbisa. Webshield: Enabling various web defense techniques without client side modifications. In *NDSS*, 2011.

[39] Yali Liu, Dipak Ghosal, Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Stefan Katzenbeisser. Hide and seek in time - robust covert timing channels. In Michael Backes and Peng Ning, editors, *ESORICS*, volume 5789 of *Lecture Notes in Computer Science*, pages 120–135. Springer, 2009.

[40] Keaton Mowery, Dillon Bogenreif, Scott Yilek, and Hovav Shacham. Fingerprinting information in javascript implementations. In *WEB 2.0 SECURITY & PRIVACY (W2SP)*, 2011.

[41] Keaton Mowery and Hovav Shacham. Pixel perfect: Fingerprinting canvas in html5. In *W2SP*, 2012.

[42] Martin Mulazzani, Philipp Reschl, Markus Huber, Manuel Leithner, Sebastian Schrittwieser, Edgar Weippl, and FC Wien. Fast and reliable browser identification with javascript engine fingerprinting. In *WEB 2.0 SECURITY & PRIVACY (W2SP)*, 2013.

[43] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered insecure: Gpu side channel attacks are practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 2139–2153, New York, NY, USA, 2018. ACM.

[44] Nick Nikiforakis, Wouter Joosen, and Benjamin Livshits. Privaricator: Deceiving fingerprinters with little white lies. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, pages 820–830, New York, NY, USA, 2015. ACM.

[45] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *IEEE Symposium on Security and Privacy*, 2013.

[46] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1406–1418, New York, NY, USA, 2015. ACM.

[47] Xiang Pan, Yinzhi Cao, and Yan Chen. I do not know what you visited last summer - protecting users from third-party web tracking with trackingfree browser. In *NDSS*, 2015.

[48] M Perry, E Clark, and S Murdoch. The design and implementation of the tor browser [draft][online], united states, 2015.

[49] Mike Perry, Erinn Clark, Steven Murdoch, and Georg Koppen. The design and implementation of the tor browser. https://www.torproject.org/projects/torbrowser/design/.

[50] Jason Peterson. How to start building your own webgl-based vr app. https://medium.com/adventures-in-consumer-technology/how-to-start-building-your-own-webgl-based-vr-app-cdaf47b8132a.

[51] Ashay Rane, Calvin Lin, and Mohit Tiwari. Secure, precise, and fast floating-point operations on x86 processors. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 71–86, 2016.

[52] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. Browsershield: vulnerability-driven filtering of dynamic html. In *OSDI: USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[53] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. Detecting and defending against third-party tracking on the web. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 12–12, Berkeley, CA, USA, 2012. USENIX Association.

[54] Michael Schwarz, Moritz Lipp, and Daniel Gruss. Javascript zero: Real javascript and zero side-channel attacks. In *NDSS*, 2018.

[55] Peter Snyder, Lara Ansari, Cynthia Taylor, and Chris Kanich. Browser feature usage on the modern web. In *Proceedings of the 2016 Internet Measurement Conference*, IMC '16, pages 97–110, New York, NY, USA, 2016. ACM.

[56] Deian Stefan, Pablo Buiras, Edward Z Yang, Amit Levy, David Terei, Alejandro Russo, and David Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In *European Symposium on Research in Computer Security*, pages 718–735. Springer, 2013.

[57] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1382–1393, New York, NY, USA, 2015. ACM.

[58] Tom Van Goethem, Mathy Vanhoef, Frank Piessens, and Wouter Joosen. Request and conquer: Exposing cross-origin resource size. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security, 2016.

[59] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. Fp-stalker: Tracking browser fingerprint evolutions.

[60] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. FP-Scanner: The Privacy Implications of Browser Fingerprint Inconsistencies. In *Proceedings of the 27th USENIX Security Symposium*, Baltimore, United States, August 2018.

[61] Weiyi Wu and Bryan Ford. Deterministically deterring timing attacks in deterland. In *Conference on Timely Results in Operating Systems (TRIOS)*, 2015.

[62] Zhihao Yao, Saeed Mirzamohammadi, Ardalan Amiri Sani, and Mathias Payer. Milkomeda: Safeguarding the mobile gpu interface using webgl security checks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1455–1469, New York, NY, USA, 2018. ACM.

[63] Ting-Fang Yen, Yinglian Xie, Fang Yu, Roger Peng Yu, and Martın Abadi. Host fingerprinting and tracking on the web: Privacy and security implications. In *Proceedings of NDSS*, 2012.

[64] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 313–328, Washington, DC, USA, 2011. IEEE Computer Society.

[65] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 305–316, New York, NY, USA, 2012. ACM.

Appendix

A Statistics of Collected Fingerprints

In the appendix, we show some statistics about collected fingerprints using the WebGL tasks provided by the original Cao et al.'s website. Figure 11 shows the anonymous set for the collected data, which is broken down into three different browsers. The size of anonymous set is relatively small—if we limit it to be three, we include 77% of all the collected fingerprints. The largest anonymous set is just with about 10 fingerprints. Among all the browsers, Safari is the most fingerprintable as compared with others: It is probably also because the number of Safari users is relatively small.

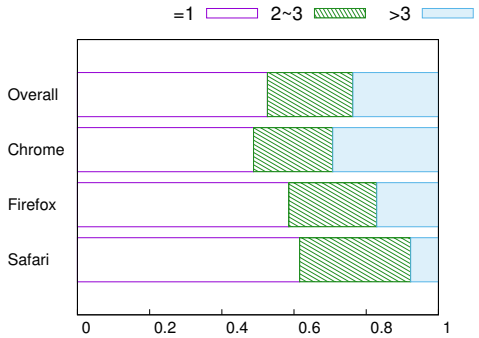


Figure 11: Anonymous Set for Collected Fingerprints

Site Isolation: Process Separation for Web Sites within the Browser

Charles Reis
Google
creis@google.com

Alexander Moshchuk
Google
alexmos@google.com

Nasko Oskov
Google
nasko@google.com

Abstract

Current production web browsers are multi-process but place different web sites in the same renderer process, which is not sufficient to mitigate threats present on the web today. With the prevalence of private user data stored on web sites, the risk posed by compromised renderer processes, and the advent of transient execution attacks like Spectre and Meltdown that can leak data via microarchitectural state, it is no longer safe to render documents from different web sites in the same process. In this paper, we describe our successful deployment of the *Site Isolation* architecture to all desktop users of Google Chrome as a mitigation for process-wide attacks. Site Isolation locks each renderer process to documents from a single site and filters certain cross-site data from each process. We overcame performance and compatibility challenges to adapt a production browser to this new architecture. We find that this architecture offers the best path to protection against compromised renderer processes and same-process transient execution attacks, despite current limitations. Our performance results indicate it is practical to deploy this level of isolation while sufficiently preserving compatibility with existing web content. Finally, we discuss future directions and how the current limitations of Site Isolation might be addressed.

1 Introduction

Ten years ago, web browsers went through a major architecture shift to adapt to changes in their workload. Web content had become much more active and complex, and monolithic browser implementations were not effective against the security threats of the time. Many browsers shifted to a multi-process architecture that renders untrusted web content within one or more low-privilege sandboxed processes, mitigating attacks that aimed to install malware by exploiting a rendering engine vulnerability [43, 51, 70, 76].

Given recent changes in the security landscape, that multi-process architecture no longer provides sufficient safety for visiting untrustworthy web content, because it does not provide similar mitigation for attacks between different web sites. Browsers load documents from multiple sites within the same renderer process, so many new types of attacks target rendering engines to access cross-site data [5, 10, 11, 33, 53]. This is increasingly common now that the most exploitable targets of older browsers are disappearing from the web (e.g., Java Applets [64], Flash [1], NPAPI plugins [55]).

As others have argued, it is clear that we need stronger isolation between security principals in the browser [23, 33, 53, 62, 63, 68], just as operating systems offer stronger isolation between their own principals. We achieve this in a production setting using *Site Isolation* in Google Chrome, introducing OS process boundaries between web site principals.

While Site Isolation was originally envisioned to mitigate exploits of bugs in the renderer process, the recent discovery of *transient execution attacks* [8] like Spectre [34] and Meltdown [36] raised its urgency. These attacks challenge a fundamental assumption made by prior web browser architectures: that software-based isolation can keep sensitive data protected within an operating system process, despite running untrustworthy code within that process. Transient execution attacks have been demonstrated to work from JavaScript code [25, 34, 37], violating the web security model *without requiring any bugs in the browser*. We show that our long-term investment in Site Isolation also provides a necessary mitigation for these unforeseen attacks, though it is not sufficient: complementary OS and hardware mitigations for such attacks are also required to prevent leaks of information from other processes or the OS kernel.

To deploy Site Isolation to users, we needed to overcome numerous performance and compatibility challenges not addressed by prior research prototypes [23, 62, 63, 68]. Locking each sandboxed renderer process to a single site greatly increases the number of processes; we present process consolidation optimizations that keep memory overhead low while preserving responsiveness. We reduce overhead and latency by consolidating painting and input surfaces of contiguous same-site frames, along with parallelizing process creation with network requests and carefully managing a spare process. Supporting the entirety of the web presented additional compatibility challenges. Full support for *out-of-process iframes* requires proxy objects and replicated state in frame trees, as well as updates to a vast number of browser features. Finally, a privileged process must filter sensitive cross-site data without breaking existing cross-site JavaScript files and other subresources. We show that such filtering requires a new type of confirmation sniffing and can protect not just HTML but also JSON and XML, beyond prior discussions of content filtering [23, 63, 68].

With these changes, the privileged browser process can keep most cross-site sensitive data out of a malicious document's renderer process, making it inconsequential for a web attacker to access and exfiltrate data from its address space. While there are a set of limitations with its current implementation, we argue that Site Isolation offers the best path to mitigating the threats posed by compromised renderer processes and transient execution attacks.

In this paper, Section 2 introduces a new browser threat model covering *renderer exploit attackers* and *memory disclosure attackers*, and it discusses the current limitations of Site Isolation's protection. Section 3 presents the challenges we overcame in fundamentally re-architecting a production browser to adopt Site Isolation, beyond prior research browsers. Section 4 describes our implementation, consisting of almost 450k lines of code, along with critical optimizations that made it feasible to deploy to all desktop and laptop users of Chrome. Section 5 evaluates its effectiveness against compromised renderers as well as Spectre and Meltdown attacks. We also evaluate its practicality, finding that it incurs a total memory overhead of 9-13% in practice and increases page load latency by less than 2.25%, while sufficiently preserving compatibility with actual web content. Given the severity of the new threats, Google Chrome has enabled Site Isolation by default. Section 6 looks at the implications for the web's future and potential ways to address Site Isolation's current limitations. We compare to related work in Section 7 and conclude in Section 8.

Overall, we answer several new research questions:

- Which parts of a web browser's security model can be aligned with OS-level isolation mechanisms, while preserving compatibility with the web?
- What optimizations are needed to make process-level isolation of web sites feasible to deploy, and what is the resulting performance overhead for real users?
- How well does process-level isolation of web sites upgrade existing security practices to protect against compromised renderer processes?
- How effectively does process-level isolation of web sites mitigate Spectre and Meltdown attacks, and where are additional mitigations needed?

2 Threat Model

We assume that a *web attacker* can lure a user into visiting a web site under the attacker's control. Multi-process browsers have traditionally focused on stopping web attackers from compromising a user's computer, by rendering untrusted web content in sandboxed renderer processes, coordinated by a higher-privilege browser process [51]. However, current browsers allow attackers to load victim sites into the same renderer process using iframes or popups, so the browser must trust security checks in the renderer process to keep sites isolated from each other.

In this paper, we move to a stronger threat model emphasizing two different *types* of web attackers that each aim to steal data across web site boundaries. First, we consider a *renderer exploit attacker* who can discover and exploit vulnerabilities to bypass security checks or even achieve arbitrary code execution in the renderer process. This attacker can disclose any data in the renderer process's address space, as well as lie to the privileged browser process. For example, they might forge an IPC message to retrieve sensitive data associated with another web site (e.g., cookies, stored passwords). These attacks imply that the privileged browser process must validate access to all sensitive resources without trusting the renderer process. Prior work has shown that such attacks can be achieved by exploiting bugs in the browser's implementation of the Same-Origin Policy (SOP) [54] (known as universal cross-site scripting bugs, or UXSS), with memory corruption, or with techniques such as data-only attacks [5, 10, 11, 33, 53, 63, 68].

Second, we consider a *memory disclosure attacker* who cannot run arbitrary code or lie to the browser process, but who can disclose arbitrary data within a renderer process's address space, even when the SOP would disallow it. This can be achieved using transient execution attacks [8] like Spectre [34] and Meltdown [36]. Researchers have shown specifically that JavaScript code can manipulate microarchitectural state to leak data from within the renderer process [25, 34, 37].[1] While less powerful than renderer exploit attackers, memory disclosure attackers are not dependent on any bugs in web browser code. Indeed, some transient execution attacks rely on properties of the hardware that are unlikely to change, because speculation and other transient microarchitectural behaviors offer significant performance benefits. Because browser vendors cannot simply fix bugs to mitigate cases of these attacks, memory disclosure attackers pose a more persistent threat to the web security model. It is thus important to reason about their capabilities separately and mitigate these attacks architecturally.

2.1 Scope

We are concerned with isolating sensitive web site data from execution contexts for other web sites within the browser. Execution contexts include both *documents* (in any frame) and *workers*, each of which is associated with a *site* principal [52] and runs in a renderer process. We aim to protect many types of content and state from the attackers described above, including the HTML contents of documents, JSON or XML data files they retrieve, state they keep within the browser (e.g., cookies, storage, saved passwords), and permissions they have been granted (e.g., geolocation, camera).

Site Isolation is also able to strengthen some existing security practices for web application code, such as upgrading clickjacking [30] protections to be robust against com-

[1] In some cases, transient execution attacks may access information across process or user/kernel boundaries. This is outside our threat model.

promised renderers, as discussed in Section 5.1. Not all web security defenses are in scope, such as mitigations for XSS [46].

2.2 Limitations

For both types of attackers we consider, Site Isolation aims to protect as much site data as possible, while preserving compatibility. Because we isolate *sites* (i.e., scheme plus registry-controlled domain name [52]) rather than *origins* (i.e., scheme-host-port tuples [54]) per Section 3.1, cross-origin attacks within a site are not mitigated. We hope to allow some origins to opt into origin-level isolation, as discussed in Section 6.3.

Cross-site subresources (e.g., JavaScript, CSS, images, media) are not protected, since the web allows documents to include them within an execution context. JavaScript and CSS files were already somewhat exposed to web attackers (e.g., via XSSI attacks that could infer their contents [26]); the new threat model re-emphasizes not to store secrets in such files. In contrast, cross-site images and media were sufficiently opaque to documents before, suggesting a need to better protect at least some such files in the future.

The content filtering we describe in Section 3.5 is also a best-effort approach to protect HTML, XML, and JSON files, applying only when it can confirm the responses match the reported content type. This confirmation is necessary to preserve compatibility (e.g., with JavaScript files mislabeled as HTML). Across all content types, we expect this filtering will protect most sensitive data today, but there are opportunities to greatly improve this protection with headers or web platform changes [21, 71, 73], as discussed in Section 6.1.

Finally, we rely on protection domains provided by the operating system. In particular, we assume that the OS's process isolation boundary can be trusted and consider cross-process and kernel attacks out of scope for this paper, though we discuss them further in Sections 5.2 and 6.2.

3 Site Isolation Browser Architecture

The *Site Isolation* browser architecture treats each web site as a separate security principal requiring a dedicated renderer process. Prior production browsers used rendering engines that predated the security threats in Section 2 and were architecturally incompatible with putting cross-site iframes in a different process. Prior research browsers proposed similar isolation but did not preserve enough compatibility to handle the full web. In this section, we present the challenges we overcame to make the Site Isolation architecture compatible with the web in its entirety.

3.1 Site Principals

Most prior multi-process browsers, including Chrome, Edge, Safari, and Firefox, did not assign site-specific security principals to web renderer processes, and hence they did not enforce isolation boundaries between different sites at the process level. We advance this model in Chrome by partitioning web content into finer-grained principals that correspond to web *sites*. We adopt the *site* definition from [52] rather than *origins* as proposed in research browsers [23, 62, 63, 68]. For example, an origin `https://bar.foo.example.com:8000` corresponds to a site `https://example.com`. This preserves compatibility with up to 13.4% of page loads that change their origin at runtime by assigning to `document.domain` [12]. Site principals ensure that a document's security principal remains constant after `document.domain` modifications.

For each navigation in any frame, the browser process computes the site from the document's URL, determining its security principal. This is straightforward for HTTP(S) URLs, though some web platform features require special treatment, as we discuss in Appendix A (e.g., `about:blank` can inherit its origin and site).

3.2 Dedicated Processes

Site Isolation requires that renderer processes can be dedicated to documents, workers, and sensitive data from only a single site principal. In this paper, we consider only the case where *all* web renderer processes are locked to a single site. It would also be possible for the browser to isolate only some sites and leave other sites in shared renderer processes. In such a model, it is still important to limit a dedicated renderer process to documents and data from its own site, but it is also necessary to prevent a shared process from retrieving data from one of the isolated sites. When isolating all sites, requests for site data can be evaluated solely on the process's site principal and not also a list of which sites are isolated.

The browser's own components and features must be also partitioned in a way that does not leak cross-site data. For example, the network stack cannot run within the renderer process, to protect `HttpOnly` cookies and so that filtering decisions on cross-site data can be made before the bytes from the network enter the renderer process. Similarly, browser features must not proactively leak sensitive data (e.g., the user's stored credit card numbers with autofill) to untrustworthy renderer processes, at least until the user indicates such data should be provided to a site [49]. These additional constraints on browser architecture may increase the amount of logic and state in more privileged processes. This does not necessarily increase the attack surface of the trusted browser process if these components (e.g., network stack) can move to separate sandboxed processes, as in prior microkernel-like browser architectures [23, 62].

3.3 Cross-Process Navigations

When a document in a frame navigates from site A to site B, the browser process must replace the renderer process for site A with one for site B. This requires maintaining state in the browser process, such as session history for the tab, related window references such as openers or parent frames,

and tab-level session storage [74]. Due to web-visible events such as `beforeunload` and `unload` and the fact that a navigation request might complete without creating a new document (e.g., a download or an HTTP "204 No Content" response), the browser process must coordinate with both old and new renderer processes to switch at the appropriate moment: after `beforeunload`, after the network response has proven to be a new document, and at the point that the new process has started rendering the new page. Note that cross-site server redirects may even require selecting a different renderer process before the switch occurs.

Session history is particularly challenging. Each stop in the back/forward history can contain information about multiple cross-site documents in various frames in the page, and it can include sensitive data for each document, such as the contents of partially-filled forms. To meet the security goals of Site Isolation, this site-specific session history state can only be sent to renderer processes locked to the corresponding site. Thus, the browser process must coordinate session history loads at a frame granularity, tracking which data to send to each process as cross-site frames are encountered in the page being loaded.

3.4 Out-of-process iframes

The largest and most disruptive change for Site Isolation is the requirement to load cross-site iframes in a different renderer process than their embedding page. Most widely-used browser rendering engines were designed and built before browsers became multi-process. The shift to multi-process browsers typically required some changes to these existing engines in order to support multiple instances of them. However, many core assumptions remained intact, such as the ability to traverse all frames in a page for tasks like painting, messaging, and various browser features (e.g., find-in-page). Supporting *out-of-process iframes* is a far more intrusive change that requires revisiting such assumptions across the entire browser. Meanwhile, prior research prototypes that proposed this separation [23, 63, 68] did not address many of the challenges in practice, such as how to ensure the iframe's document knows its position in the frame tree. This section describes the challenges we overcame to make out-of-process iframes functional and compatible with the web platform.

Frame Tree. To support out-of-process iframes, multi-process browser architectures must change their general abstraction level from *page* (containing a tree of frames) to *document* (in a single frame). The browser process must track which document, and thus principal, is present in each frame of a page, so that it can create an appropriate renderer process and restrict its access accordingly. The cross-process navigations described in Section 3.3 must be supported at each level of the frame tree to allow iframes to navigate between sites.

Each process must also keep a local representation of documents that are currently rendered in a different process,

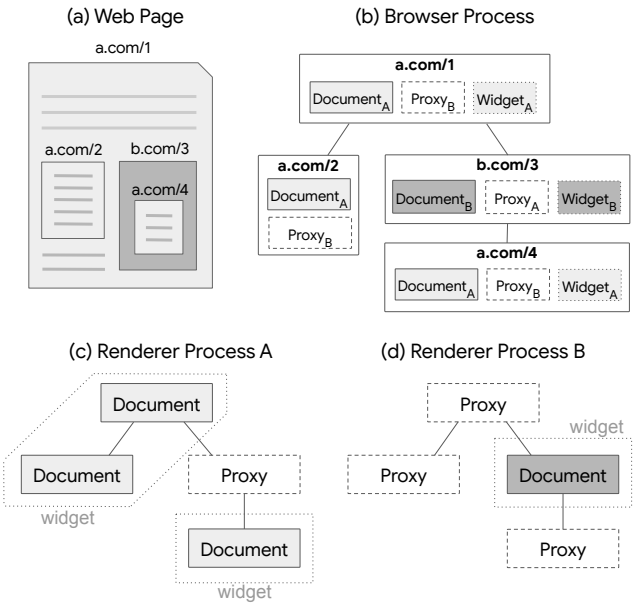


Figure 1: **An example of out-of-process iframes.** To render the web page shown in (a), the browser process (b) coordinates two renderer processes, shown in (c) and (d).

which we call *proxies*. Proxies offer cross-process support for the small set of cross-origin APIs that are permitted by the web platform, as described in [52]. These APIs may be accessed on a frame's `window` object and are used for traversing the frame hierarchy, messaging, focusing or navigating other frames, and closing previously opened windows. Traversing the frame hierarchy must be done synchronously within the process using proxies, but interactions between documents can be handled asynchronously by routing messages. Note that all same-site frames within a frame tree (or other reachable pages) must share a process, allowing them to synchronously script each other.

An example of a page including out-of-process iframes is shown in Figure 1 (a), containing three documents from `a.com` and one from `b.com`, and thus requiring two separate renderer processes. Figure 1 (b) shows the browser process's frame tree, with representations of each document annotated by which site's process they belong to, along with a *set* of proxy objects for each frame (one for each process which might reference the frame). Figure 1 (c-d) shows the corresponding frame trees within the two renderer processes, with proxy objects for any documents rendered in a different process. Note that the actual document and proxy objects live in renderer processes; the corresponding browser-side objects are stubs that track state and route IPC messages between the browser and renderer processes.

For example, suppose the document in `a.com/2` invokes `window.parent.frames["b"].postMessage("msg", "b.com")`. Renderer Process A can traverse its local frame tree to find the parent frame and then its child frame named "`b`", which is a proxy. The renderer process will send the message to the corresponding Proxy$_A$ object for `b.com/3` in

the browser process. The browser process passes it to the current Document$_B$ object in this frame, which sends the message to the corresponding Document object in Renderer Process B. Similar message routing can support other cross-origin APIs, such as focus, navigation, or closing windows.

State Replication. The renderer process may need synchronous access to some types of state about a frame in another process, such as the frame's current name (to find a frame by name, as in the example above) or iframe sandbox flags. As this state changes, the browser process broadcasts it to all proxies for a frame across affected processes. Note that this state should never include sensitive site-specific data (e.g., full URLs, which may have sensitive URL parameters), only what is necessary for the web platform implementation.

Painting and Input. To preserve the Site Isolation security model, the rendered appearance of each document cannot leak to other cross-site renderer processes. Otherwise, an attacker may be able to scrape sensitive information from the visible appearance of frames in other processes. Instead, each renderer process is responsible for the layout and paint operations within each of its frames. These must be sent to a separate process for compositing at the granularity of *surfaces*, to form the combined appearance of the page. The compositing process must support many types of transforms that are possible via CSS, without leaking surface data to a cross-site renderer process.

Often, many frames on a page come from the same site, and separate surfaces for each frame may be unnecessary. To reduce compositing overhead, we use a *widget* abstraction to combine contiguous same-site frames into the same surface. Figure 1 shows how `a.com/1` and `a.com/2` can be rendered in the same widget and surface without requiring compositing. `b.com/3` requires its own widget in Renderer Process B. Since `a.com/4` is not contiguous with the other two `a.com` frames and its layout may depend on properties assigned to it by `b.com/3` (e.g., CSS filters), it has a separate widget within Renderer Process A, and its surface must be composited within `b.com/3`'s surface.

Widgets are also used for input event routing, such as mouse clicks and touch interactions. In most cases, the compositing metadata makes it possible for the browser process to perform sufficient hit testing to route input events to the correct renderer process. In some cases, though, web platform features such as CSS transforms or CSS `pointer-events` and `opacity` properties may make this difficult. Currently, the browser process uses *slow path hit testing* over out-of-process iframes, i.e., asking a parent frame's process to hit-test a specific point to determine which frame should receive the event, without revealing any further details about the event itself. This is only used for mouse and touch events; keyboard events are reliably delivered to the renderer process that currently has focus.

Note that images and media from other sites can be included in a document. The Site Isolation architecture does not try to exclude these from the renderer process, for multiple reasons. First, moving cross-origin image handling out of the renderer process and preventing renderers from reading these surfaces would require a great deal of complexity in practice. Second, this would substantially increase the number of surfaces needed for compositing. This decision is consistent with other research browsers [23, 62, 63], including Gazelle's implementation [68]. Thus, we leave cross-site images and media in the renderer process and rely on servers to prevent unwanted inclusion, as discussed in Section 6.1.

Affected Features. In a broad sense, almost all browser features that interact with the frame tree must be updated to support out-of-process iframes. These features could traditionally assume that all frames of a page were in one process, so a feature like find-in-page could traverse each frame in the tree in the renderer process, looking for a string match. With out-of-process iframes, the browser process must coordinate the find-in-page feature, collecting partial results from each frame across multiple renderer processes. Additionally, the feature must be careful to avoid leaking information to renderer processes (e.g., whether there was a match in a cross-site sibling frame), and it must be robust to renderer processes that crash or become unresponsive.

These updates are required for many features that combine data across frames or that perform tasks that span multiple frames: supporting screen readers for accessibility, compositing PDFs for printing, traversing elements across frame boundaries for focus tracking, representations of the full page in developer tools, and many others.[2]

3.5 Cross-Origin Read Blocking

Loading each site's documents in dedicated renderer processes is not sufficient to protect site data: there are many legitimate ways for web documents to request cross-site URLs within their own execution context, such as JavaScript libraries, CSS files, images, and media. However, it is important not to give a renderer process access to cross-site URLs containing sensitive data, such as HTML documents or JSON files. Otherwise, a document could access cross-site data by requesting such a URL from a <`script`>, <`style`>, or <`img`> tag. The response may nominally fail within the requested context (e.g., an HTML file would produce syntax errors in a <`script`> tag), but the data would be present in the renderer process, where a compromised renderer or a transient execution attack could leak it.

Unfortunately, it is non-trivial to perfectly distinguish which cross-site URLs must be allowed into a renderer process and which must be blocked. It is possible to categorize content types into those needed for subresources and those that are not (as in Gazelle [68]), but content types of re-

[2] A list of these features is included in Appendix B.

sponses are often inaccurate in practice. For example, many actual JavaScript libraries have content types of `text/html` rather than `application/javascript` in practice. Changing the browser to block these libraries from cross-site documents would break compatibility with many existing sites.

It may be desirable to require sites to correct their content types or proactively label any resources that need protection (e.g., with a new `Cross-Origin-Resource-Policy` header [21]), but such approaches would leave many existing resources unprotected until developers update their sites.

Until such shifts in web site behavior occur, browsers with Site Isolation can use a best effort approach to protect as many sensitive resources as possible, while preserving compatibility with existing cross-site subresources. We introduce and standardize an approach called *Cross-Origin Read Blocking (CORB)* [17, 20], which prevents a renderer process from receiving a cross-site response when it has a *confirmed* content type likely to contain sensitive information. CORB focuses on content types that, when used properly, cannot be used in a subresource context. Subresource contexts include scripts, CSS, media, fetches, and other ways to include or retrieve data within a document, but exclude iframes and plugins (which can be loaded in separate processes). CORB filters the following content types:

- HTML, which is used for creating new documents with data that should be inaccessible to other sites.
- JSON, which is used for conveying data to a document.
- XML, which is also often used for conveying data to a document. An exception is made for SVG, which is an XML data type permitted within <`img`> tags.

Since many responses have incorrect content types, CORB requires additional confirmation before blocking the response from the renderer process. In other contexts, web browsers perform MIME-type sniffing when a content type is missing, looking at a prefix of the response to guess its type [4]. OP2 and IBOS use such sniffing to confirm a response is HTML [23, 63], but this will block many legitimate JavaScript files, such as those that begin with HTML comments (i.e., "`<!--`"). In contrast, CORB relies on a new type of *confirmation sniffing* that looks at a prefix of the response to confirm that it matches the claimed content type and not a subresource [17]. For example, a response labeled as `text/html` starting with "`<!doctype`" would be blocked, but one starting with JavaScript code would not. (CORB attempts to scan past HTML comments when sniffing.) This is a default-allow policy that attempts to protect resources where possible but prioritizes compatibility with existing sites. For example, CORB allows responses through when they are polyglots which could be either HTML or JavaScript, such as:

```
<!--/*--><html><body><script type="text/javascript"><!--//*/
var x = "This is both valid HTML and valid JavaScript.";
//--></script></body></html>
```

CORB skips confirmation sniffing in the presence of the existing `X-Content-Type-Options: nosniff` response header, which disables the browser's existing MIME sniffing logic. When this header is present, responses with incorrect content types are already not allowed within subresource contexts, making it safe for CORB to block them. Thus, we recommend that web developers use this header for CORB-eligible URLs that contain sensitive data, to ensure protection without relying on confirmation sniffing.

If a cross-site response with one of the above confirmed content types arrives, and if it is not allowed via CORS headers [18], then CORB's logic in the network component prevents the response data from reaching the renderer process.

3.6 Enforcements

The above architecture changes are sufficient to mitigate *memory disclosure attackers* as described in Section 2. For example, transient execution attacks might leak data from any cross-site documents present in the same process, but such attacks cannot send forged messages to the browser process to gain access to additional data. However, a *renderer exploit attacker* that compromises the renderer process or otherwise exploits a logic bug may indeed lie to the browser process, claiming to be a different site to access its data.

The browser process must be robust to such attacks by tracking which renderer processes are locked to which sites, and thus restricting which data the process may access. Requests for site data, actions that require permissions, access to saved passwords, and attempts to fetch data can all be restricted based on the site lock of the renderer process. In normal execution, a renderer process has its own checks to avoid making requests for such data, so illegal requests can be interpreted by the browser process as a sign that the renderer process is compromised or malfunctioning and can thus be terminated before additional harm is caused. The browser process can record such events in the system log, to facilitate audits and forensics within enterprises.

These enforcements may take various forms. If the renderer process sends a message labeled with an origin, the browser process must enforce that the origin is part of the process's site. Alternatively, communication channels can be scoped to a site, such that a renderer process has no means to express a request for data from another site.

The CORB filtering policy in Section 3.5 also requires enforcements against compromised renderers, so that a renderer exploit attacker cannot forge a request's initiator to bypass CORB. One challenge is that extensions had been allowed to request data from extension-specified sites using scripts injected into web documents. Because these requests come from a potentially compromised renderer process, CORB cannot distinguish them from an attacker's requests. This weakens CORB by allowing responses from any site that an active extension can access, which in many cases is all sites. To avoid having extensions weaken the security

of Site Isolation, we are changing the extension system to require these requests to be issued by an extension process instead of by extension scripts in a web renderer process, and we are helping extension developers migrate to the new approach [9].

4 Implementation

With the Chrome team, we implemented the Site Isolation architecture in Chrome's C++ codebase. This was a significant 5-year effort that spanned approximately 4,000 commits from around 350 contributors (with the top 20 contributors responsible for 72% of the commits), changing or adding approximately 450,000 lines of code in 9,000 files.

We needed to re-architect a widely deployed browser without adversely affecting users, both during development and when deploying the new architecture. This section describes the steps we took to minimize the impact on performance and functionality, while Section 5 evaluates that impact in practice.

4.1 Optimizations

Fundamentally, Site Isolation requires the browser to use a larger number of OS processes. For example, a web page with four cross-site iframes, all on different sites, will require five renderer processes versus one in the old architecture. The overhead of additional processes presents a feasibility risk, due to extra memory cost and process creation latency during navigation. To address these challenges, we have implemented several optimizations that help make Site Isolation practical.

4.1.1 Process Consolidation

Our security model dictates that a renderer process may never contain documents hosted at different sites, but a process may still be shared across separate instances of documents from the same site. Fortunately, many users keep several tabs open, which presents an opportunity for process sharing across those tabs.

To reduce the process count, we have implemented a process consolidation policy that looks for an existing *same-site* process when creating an out-of-process iframe. For example, when a document embeds an `example.com` iframe and another browser tab already contains another `example.com` frame (either an iframe or a main frame), we consolidate them in the same process. This policy is a trade-off that avoids process overhead by reducing performance isolation and failure containment: a slow frame could slow down or crash other same-site frames in the process. We found that this trade-off is worthwhile for iframes, which tend to require fewer resources than main frames.

The same policy could also be applied to main frames, but doing this unconditionally is not desirable: when resource-heavy documents from a site are loaded in several tabs, using a single process for all of them leads to bloated processes that perform poorly. Instead, we use process consolidation for same-site main frames only after crossing a *soft process limit* that approximates memory pressure. When the number of processes is below this limit, main frames in independent tabs don't share processes; when above the limit, all new frames start reusing same-site processes when possible. Our threshold is calculated based on performance characteristics of a given machine. Note that Site Isolation cannot support a hard process limit, because the number of sites present in the browser may always exceed it.

4.1.2 Avoiding Non-essential Isolation

Some web content is assigned to an opaque origin [29] without crossing a site boundary, such as iframes with `data:` URLs or sandboxed *same-site* iframes. These could utilize separate processes, but we choose to keep these cases in-process as an optimization, focusing our attention on true cross-site content.

Other design decisions that help reduce process count include isolating at a site granularity rather than origin, keeping cross-site images in-process, and allowing extensions to share processes with each other. Section 6.3 discusses improving isolation in these cases in the future.

4.1.3 Reducing the Cost of Process Swaps

Section 3.3 implies that many more navigations must create a new process. We mask some of this latency by (1) starting the process in parallel with the network request, and (2) running the old document's `unload` handler in the background after the new document is created in the new process.

However, in some cases (e.g., back/forward navigations) documents may load very quickly from the cache. These cases can be significantly slowed by adding process creation latency. To address this, we maintain a warmed-up *spare* renderer process, which may be used immediately by a new navigation to any site. When a spare process is locked to a site and used, a new one is created in the background, similar to process pre-creation optimizations in OP2 [23]. To control memory overhead, we avoid spare processes on low memory devices, when the system experiences memory pressure, or when the browser goes over the soft process limit.

4.2 Deployment

Shipping Site Isolation in a production browser is challenging. It is a highly disruptive architecture change affecting significant portions of the browser, so enabling it all at once would pose a high risk of functional regressions. Hence, we deployed incrementally along two axes: isolation targets and users. Before launching full Site Isolation, we shipped two milestones to enable process isolation for selective targets:

1. **Extensions.** As the first use of out-of-process iframes from Section 3.4, we isolated *web* iframes embedded inside *extension* pages, and vice versa [50]. This provided a meaningful security improvement, keeping ma-

licious web content out of higher-privileged extension processes. It also affected only about 1% of all page loads, reducing the risk of widespread functional regressions.

2. **Selective isolation.** We created an enterprise policy allowing administrators to optionally isolate a set of manually selected high-value web sites [6].

Deploying these preliminary isolation modes provided a valuable source of bug reports and performance data (e.g., at least 24 early issues reported from enterprise policy users). These modes also show how some form of isolation may be deployed in environments where full Site Isolation may still be prohibitively expensive, such as on mobile devices.

We also deployed each of these milestones incrementally to users. All feature work was developed behind an opt-in flag, and we recruited early adopters who provided bug reports. For each milestone (including full Site Isolation), we also took advantage of Chrome's A/B testing mechanism [13], initially deploying to only a certain percentage of users to monitor performance and stability data.

5 Evaluation

To evaluate the effectiveness and practicality of deploying Site Isolation, we answer the following questions: **(1)** How well does Site Isolation upgrade existing security practices to mitigate renderer exploit attacks? **(2)** How effectively does Site Isolation mitigate transient execution attacks, compared to other web browser mitigation strategies? **(3)** What is the performance impact of Site Isolation in practice? **(4)** How well does Site Isolation preserve compatibility with existing web content? Our findings have allowed us to successfully deploy Site Isolation to all desktop and laptop users of Google Chrome.

5.1 Mitigating Renderer Vulnerabilities

We have added numerous enforcements to Chrome (version 76) to prevent a compromised renderer from accessing cross-site data.[3] This section evaluates these enforcements from the perspective of web developers. Specifically, we ask which existing web security practices have been transparently upgraded to defend against renderer exploit attackers, who have complete control over the renderer process.

New Protections. The following web developer practices were vulnerable to renderer exploit attackers before Site Isolation but are now robust.

- **Authentication.** `HttpOnly` cookies are not delivered to renderer processes, and `document.cookie` is restricted based on a process's site. Similarly, the password manager only reveals passwords based on a process's site.
- **Cross-origin messaging.** Both `postMessage` and `BroadcastChannel` messages are only delivered to processes if their sites match the target origin, ensuring that confidential data in the message does not leak to other compromised renderers. Source origins are also verified so that incoming messages are trustworthy.
- **Anti-clickjacking.** `X-Frame-Options` is enforced in the browser process, and CSP `frame-ancestors` is enforced in the embedded frame's renderer process. In both cases, a compromised renderer process cannot bypass these policies to embed a cross-site document.
- **Keeping data confidential.** Many sites use HTML, XML, and JSON to transfer sensitive data. This data is now protected from cross-site renderer processes if it is filtered by CORB (e.g., has a `nosniff` header or can be sniffed), per Section 3.5.
- **Storage and permissions.** Data stored on the client (e.g., in `localStorage`) and permissions granted to a site (e.g., microphone access) are not available to processes for other sites.

Potential Protections. The Site Isolation architecture should be capable of upgrading the following practices to mitigate compromised renderers as well, but our current implementation does not yet fully cover them.

- **Anti-CSRF.** CSRF [3] tokens remain protected from other renderers if they are only present in responses protected by CORB. `Origin` headers and `SameSite` cookies can also be used for CSRF defenses, but our enforcement implementation is still in progress.
- **Embedding untrusted documents.** The behavioral restrictions of `iframe sandbox` (e.g., creating new windows or dialogs, navigating other frames) and `Feature-Policy` are currently enforced in the renderer process, allowing compromised renderers to bypass them. If sandboxed iframes are given separate processes, many of these restrictions could happen in the browser process.

Renderer Vulnerability Analysis. We also analyzed security bugs reported for Chrome for 2014-2018 (extending the analysis by Moroz et al [41]) and found 94 UXSS-like bugs that allow an attacker to bypass the SOP and access contents of cross-origin documents. Site Isolation mitigates such bugs by construction, subject to the limitations discussed in Section 2.2. Similar analyses in prior studies have also shown that isolating web principals in different processes prevents a significant number of cross-origin bypasses [19, 63, 68].

In the six months after Site Isolation was deployed in mid-2018, Chrome has received only 2 SOP bypass bug reports, also mitigated by Site Isolation (compared to 9 reports in the prior six months). The team continues to welcome and fix such reports, since they still have value on mobile devices where Site Isolation is not yet deployed. We also believe that going forward, attention will shift to other classes of bugs seen during this post-launch period, including:

[3] A list of these enforcements is included in Appendix C.

- **Bypassing Site Isolation.** These bugs exploit flaws in the process assignment or other browser process logic to force cross-site documents to share a process, or to bypass the enforcement logic. For example, we fixed a reported bug where incorrect handling of `blob` URLs created in opaque origins allowed an attacker to share a victim site's renderer process.
- **Targeting non-isolated data.** For example, 14 bugs allowed an attacker to steal cross-site images or media, which are not isolated in our architecture, e.g., by exploiting memory corruption bugs or via timing attacks.
- **Cross-process attacks.** For example, 5 bugs are side channel attacks that rely on timing events that work even across processes, such as a frame's `onload` event, to reveal information about the frame.

In general, we find that Site Isolation significantly improves robustness to renderer exploit attackers, protecting users' web accounts and lowering the severity of renderer vulnerabilities.

5.2 Mitigating Transient Execution Attacks

Transient execution attacks represent *memory disclosure attackers* from Section 2, where lying to the browser process is not possible. Thus, Site Isolation mitigations here depend on process isolation and CORB, but not the enforcements in Section 3.6. This section compares the various web browser mitigation strategies for such attacks, evaluating their effectiveness against known variants.

Strategy Comparison. Web browser vendors have pursued three types of strategies to mitigate transient execution attacks on the web, with varying strengths and weaknesses.

First, most browsers attempted to *reduce the availability of precise timers* that could be used for attacks [14, 39, 48, 67]. This focuses on the most commonly understood exploitation approach for Spectre and Meltdown attacks: a Flush+Reload cache timing attack [75]. This strategy assumes the timing attack will be difficult to perform without precise timers. Most major browsers reduced the granularity of APIs like `performance.now` to 20 microseconds or even 1 millisecond, introduced jitter to timer results, and even removed implicit sources of precise time, such as `SharedArrayBuffers` [59]. This strategy applies whether the attack targets data inside the process or outside of it, but it has a number of weaknesses that limit its effectiveness:

- It is likely incomplete: there are a wide variety of ways to build a precise timer [35, 58], making it difficult to enumerate and adjust all sources of time in the platform.
- It is possible to amplify the cache timing result to the point of being effective even with coarse-grained timers [25, 37, 58].
- Coarsening timers hurts web developers who have a legitimate need for precise time to build powerful web applications. Disabling `SharedArrayBuffers` was a particularly unfortunate consequence of this strategy, since it disrupted web applications that relied on them (e.g., AutoCAD).
- Cache timing attacks are only one of several ways to leak information from transient execution, so this approach may be insufficient for preventing data leaks [8].

As a result, we do not view coarsening timers or disabling `SharedArrayBuffers` as an effective strategy for mitigating transient execution attacks.

Second, browser vendors pursued *modifications to the JavaScript compiler and runtime* to prevent JavaScript code from accessing victim data speculatively [37, 48, 65]. This involved array index masking and pointer poisoning to limit out of bounds access, `lfence` instructions as barriers to speculation, and similar approaches. The motivation for this strategy is to disrupt all "speculation gadgets" to avoid leaking data within and across process boundaries. Unfortunately, there are an increasingly large number of variants of transient execution attacks [8], and it is difficult for a compiler to prevent all the ways an attack might be expressed [37]. This is especially true for variants like Spectre-STL (also known as Variant 4), where store-to-load forwarding can be used to leak data [28], or Meltdown-RW which targets in-process data accessed after a CPU exception [8]. Additionally, some of these mitigations have large performance overheads on certain workloads (up to 15%) [37, 65], which risk slowing down legitimate applications. The difficulty to maintain a complete defense combined with the performance cost led Chrome's JavaScript team to conclude that this approach was ultimately impractical [37, 49].

Site Isolation offers a third strategy. Rather than targeting the cache timing attack or disrupting speculation, Site Isolation assumes that transient execution attacks may be possible within a given OS process and instead attempts to *move data worth stealing outside of the attacker's address space*, much like kernel defenses against Meltdown-US [15, 24].

Variant Mitigation. Canella et al [8] present a systematic evaluation of transient execution attacks and defenses, which we use to evaluate Site Isolation. Spectre attacks rely on branch mispredictions or data dependencies, while Meltdown attacks rely on transient execution after a CPU exception [8]. Table 1 shows how both types of attacks are able to target data inside or outside the attacker's process, and thus both Spectre and Meltdown are relevant to consider when mitigating memory disclosure attacks.

Site Isolation mitigates same-address-space attacks by avoiding putting vulnerable data in the same renderer process as a malicious principal. This targets the most practical variants of transient execution attacks, for which an attacker has a large degree of control over the behavior of the process (relative to attacks that target another process). Site Isolation does not depend on the absence of precise timers for

	Inside Process			Outside Process		
Attack	**Site Isolation**	**Timers**	**Compiler**	**Site Isolation**	**Timers**	**Compiler**
Spectre-PHT	●	◐	●	○	◐	●
Spectre-BTB	●	◐	●	○	◐	●
Spectre-RSB	●	◐	◐	○	◐	◐
Spectre-STL	●	◐	○	-	-	-
Meltdown-US	-	-	-	○	◐	○
Meltdown-P	-	-	-	○	◐	○
Meltdown-GP	-	-	-	○	◐	○
Meltdown-NM	-	-	-	○	◐	○
Meltdown-RW*	●	◐	○	-	-	-
Meltdown-PK*	●	◐	○	-	-	-
Meltdown-BR*	●	◐	○	-	-	-

Table 1: **Web browser mitigations for Spectre and Meltdown attacks, for targets inside and outside the attacker's process.** Symbols show if an attack is mitigated (●), partially mitigated (◐), not mitigated (○), or not applicable (-). Site Isolation mitigates all applicable same-process attacks, and it depends on other mitigations for cross-process attacks.
** Only affects browsers that use these hardware features.*

mitigating same-process attacks, and it can mitigate attacks like Spectre-STL that are difficult or costly for compilers to prevent [37]. For Meltdown attacks that target same-process data (e.g., Meltdown-RW, which can transiently overwrite read-only data), Site Isolation applies as well. It is less clear whether Meltdown-PK and Meltdown-BR [8] are relevant in the context of the browser, but Site Isolation would mitigate them if browsers used protection keys [38] or hardware-based array bounds checks, respectively.

Site Isolation does not attempt to mitigate attacks targeting data in other processes or the kernel, such as the "Outside Process" variants in Table 1 and Microarchitectural Data Sampling (MDS) attacks [40, 57, 66]. Site Isolation can and must be combined with hardware and OS mitigations for such attacks to prevent web attackers from leaking data across process boundaries or from the kernel. For example, PTI is a widely used mitigation for Meltdown-US, eliminating kernel memory from the address space of each user process [15, 24]. Similarly, microcode updates and avoiding sibling Hyper-Threads for untrustworthy code may be useful for mitigating MDS attacks [40, 57, 66].

Ultimately, cross-process and user/kernel boundaries must fundamentally be preserved by the OS and hardware and cannot be left to applications to enforce. Within a process, however, the OS and hardware have much less visibility into where isolation is needed. Thus, applications that run code from untrustworthy principals (e.g., browsers) must align their architectures with OS-enforced abstractions to isolate these principals. As a result, we have chosen Site Isolation as the most effective mitigation strategy for Chrome. When it is enabled, Chrome re-enables `SharedArrayBuffer` and other precise timers and removes JavaScript compiler mitigations, to restore powerful functionality to the web and regain lost performance.

5.3 Performance

Enabling Site Isolation can affect the browser's performance, so we evaluate its effect on memory overhead, latency, and CPU usage in the wild and in microbenchmarks. We find that the new architecture has low enough overhead to be practical to deploy.

5.3.1 Observed Workload

We first focus on measuring performance in the field, because this more accurately reflects real user workloads (e.g., many tabs, long-tail sites) than microbenchmarks do. The data in this section was collected using pseudonymous metric reporting over a two-week period starting October 1, 2018, from desktop and laptop users of Chrome (version 69) on Windows who have this reporting enabled. We compare results from equal-sized test and control groups within the general user population. (These metrics are enabled by default, but users can opt out during installation or later in settings. Our experimental design and data collection were reviewed under Google's processes.)

Process Count. With Site Isolation, the browser process must create more renderer processes to keep sites isolated from each other: at least as many as unique sites open at a time. Using periodic samples, we found that users had 6.0 unique sites open across the entire browser at the 50th percentile of the distribution, and 41.9 unique sites at the 99th percentile. This only provides a lower bound for the number of renderer processes; each instance of a site might live in a separate process. If this were the case, our metrics give an upper bound estimate of 79.7 processes at the 99th percentile. However, thanks to the process sharing heuristics described in Section 4.1.1, far fewer processes were used in practice, as shown in Figure 2. At the 50th percentile, the number of processes increased 43.5% from 4.4 without Site Isolation to 6.2 with Site Isolation. At the 99th percentile, the process count increased 50.6% from 35.0 to 52.7 processes. This indicates that many more processes are needed for Site Isolation, but also that the process consolidation heuristics greatly reduce the count at the upper percentiles.

Memory Overhead. On its own, the 50% increase in renderer process count is significant, but this does not necessarily translate to an equivalent increase in memory overhead or performance slowdowns. Site Isolation is effectively dividing an existing workload across more processes, so each renderer process is correspondingly smaller and shorter lived. In reported metrics, we found that private memory use per renderer process decreased 51.5% (87.2 MB to 42.3 MB) at

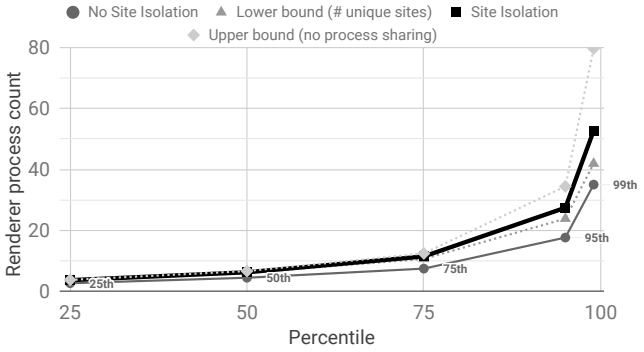


Figure 2: **Renderer process count.** This graph shows the number of renderer processes before and after Site Isolation, as well as an estimated lower and upper bound on process count, controlled by the amount of process sharing for instances of the same site. Site Isolation finds a middle ground between no process sharing and having at most one process per site.

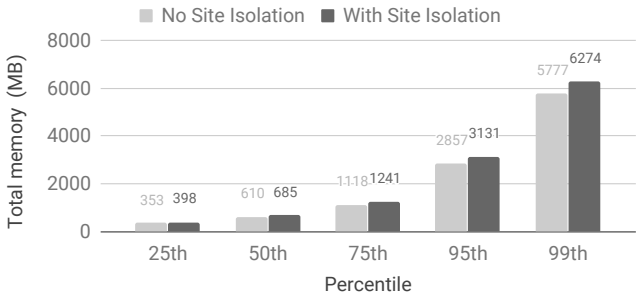


Figure 3: **Total browser memory usage across all processes.** Overall, Site Isolation has a 9-13% overhead.

the 50th percentile and 28.6% (from 714.2 MB to 509.7 MB) at the 99th percentile. Renderer process lifetime decreased 4.3% at the 50th percentile and 55.5% at the 99th percentile.

This leaves an open question about the overhead of each process relative to the workload of the process, which determines the total memory use. Figure 3 compares the total private memory use across all processes (including browser process, renderer processes, and other types of utility processes) with and without Site Isolation. In practice, we see that total memory use increased only 12.6% at the 25th percentile, and only 8.6% at the 99th percentile. This is significantly lower than the 50% increase in process count might suggest, indicating that the large number of extra processes has a relatively small impact on the total memory use of the browser. We confirmed that this is not due to a change in workload size: there were no statistically significant differences in page load count, and we saw at most a 1.5% decrease in the number of open tabs (at the 99th percentile).

Due to the severity of transient execution attacks and the drawbacks of other mitigation strategies in Section 5.2, the Chrome team was willing to accept 9-13% memory overhead for the security benefits of enabling Site Isolation.

Latency. Site Isolation also impacts latency in multiple ways, from the time it takes to load a page to the responsiveness of input events. On one hand, more navigations need to create new processes, which can incur latency due to process startup time. There may also be greater contention for IPC messages and input event routing, leading to some delays. On the other hand, there is a significant amount of new parallelism possible now that the workload for a given page can be split across multiple independent threads of execution. We use observed metrics from the field to study the combined impact of these changes in practice.

Site Isolation significantly increased the percentage of navigations that cross a process boundary, from 5.73% to 56.0%. However, we mask some of the latency of process creation in Chrome by starting the renderer process in parallel with making the network request. Combined with the increased parallelism of loading cross-site iframes in different processes, we see very little change to a key metric for page load time: the time from navigation start to the first paint of page content (e.g., text, images, etc) [22]. Across all navigations, we observe this to increase at most 2.25% at the 25th percentile (457 ms to 467 ms) and 1.58% (14.6 s to 14.8 s) at the 99th percentile. This metric also benefits from the spare process optimization described in Section 4.1.3, which avoids the process startup latency on many navigations. Without the spare process, this "First Contentful Paint" time increases 5.1% at the 25th percentile and 2.4% at the 99th percentile.

If we look closer at various types of navigations, the most significantly affected category is back/forward navigations, which frequently load pages from the cache without waiting for the network. This eliminates most of the benefit of parallelizing process startup with the network request. Here, we see time to First Contentful Paint increase 28.3% (177 ms to 227 ms) at the 25th percentile and 6.8% (4637 ms to 4952 ms) at the 99th percentile. Again, this is better than without using a spare process, in which case we see increases of 40.7% and 12.5% at these percentiles, respectively.

We also looked at the latency impact on input events. The current implementation uses slow path hit testing for mouse and touch events over out-of-process iframes, which results in small increases to input event latency. For key presses, there are no statistically significant differences at the 50th or 99th percentiles, and only a 1.0% latency increase at the 75th percentile (43.6 ms to 44.0 ms). For mouse scroll update events, latency increased 1.3% (21.8 ms to 22.1 ms) at the 50th percentile and 8.6% (228.8 ms to 248.6 ms) at the 99th percentile. For touch scroll update events, latency increased 2.6% (18.4 ms to 18.9 ms) and 10.7% (134.0 ms to 148.3 ms) at these percentiles. We expect to improve these by updating hit testing to avoid the slow path in most cases.

CPU Usage. Finally, we study the impact of Site Isolation on CPU usage. Average CPU usage in the browser process increased 8.2% (32.0% to 34.6%) at the 99th percentile,

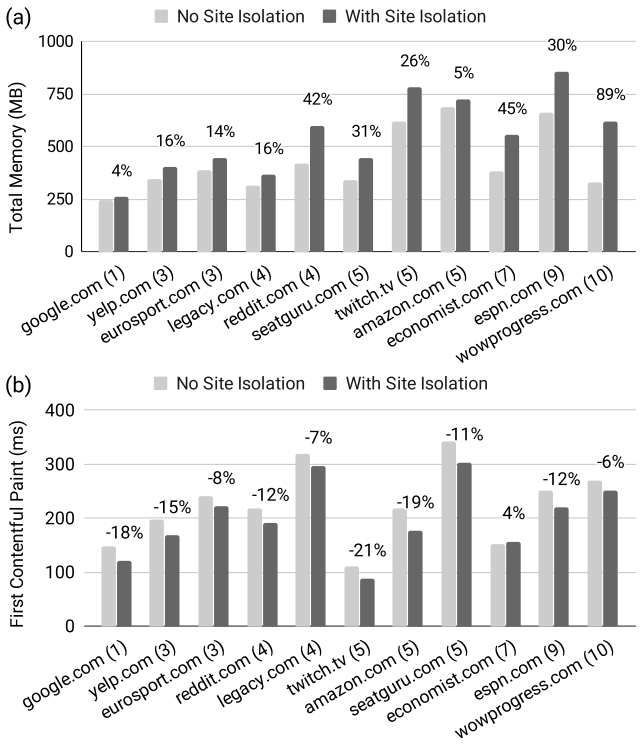


Figure 4: **(a) Total browser memory usage and (b) Time to First Contentful Paint for individual sites.** Parentheses denote the number of renderer processes required to load each site with Site Isolation. Without Site Isolation, each site requires one renderer process.

due to additional IPC messages and coordination across processes. While there were more renderer processes, each renderer's average CPU usage dropped 33.5% (47.7% to 31.8%) at the 99th percentile, since the workload was distributed across more processes.

Overall, we found that enabling Site Isolation had a much smaller performance impact than expected due to the properties of the workload. Given the importance of mitigating the attacks in the threat model described in Section 2, the Chrome team has chosen to keep Site Isolation enabled for all users on desktop and laptop devices.

5.3.2 Microbenchmarks

We also report microbenchmark results showing the overhead of Site Isolation on individual web pages when loaded in a single tab, with nothing else running in the browser. This setup does not benefit from process consolidation across multiple tabs as discussed in Section 4.1.1, and hence it is not representative of the real-world workloads used in the previous section. However, these measurements establish a baseline and provide a reproducible reference point for future research.

To study a mix of the most popular (likely highly optimized) and slightly less popular sites, we selected the top site as well as the 50th-ranked site in Alexa categories for news, sports, games, shopping, and home, as well as `google.com` as the top overall URL.[4] This set provides pages with a range of cross-site iframe counts, showing how the browser scales with more processes per page.

Next, we started Chrome version 69.0.3497.100 with a clean profile, and we loaded each site in a single tab, both with and without Site Isolation. We report the median of five trials for each data point to reduce variability, and we replayed recorded network data for all runs using WprGo [69]. Our experiments were performed on a Windows 10 desktop with an Intel Core i7-8700K 3.7 GHz 6-core CPU and 16 GB RAM. Our data collection script is available online [45].

Figure 4 (a) shows the total browser memory use for each site, sorted by the number of renderer processes (shown in parentheses) that each site utilizes when loaded with Site Isolation. As expected, the relative memory overhead generally increases with the number of processes, peaking at 89% for `wowprogress.com` with 10 processes. Sites that use more memory tend to have smaller relative overhead, as their memory usage outweighs the cost of extra processes. For example, a heavier `amazon.com` site has a 5% overhead compared to `seatguru.com`'s 31%, even though both require five processes. `google.com` does not have any cross-site iframes and requires no extra processes, but it shows a 4% increase in memory use due to the spare process that we maintain with Site Isolation, as explained in Section 4.1.3.

The overhead seen in these results is significantly higher than the 9-13% overhead we reported from real-world user workloads in the previous section. This underscores the limitations of microbenchmarks: users tend to have multiple tabs (four at 50th percentile) and a variety of open URLs. In practice, this helps reduce memory overhead via process consolidation, while iframe-heavy sites like `wowprogress.com` may represent only a small part of users' browsing sessions.

Figure 4 (b) shows time to First Contentful Paint [22] for each site, to gauge impact on page load time. Most paint times improve with Site Isolation because the spare process helps mask process startup costs, which play a larger role than network latency due to the benchmark's use of recorded network traffic. The speedups are not correlated with process counts; Site Isolation offloads some of the work from the main frame into iframe renderers, which may make the main frame more responsive regardless of process count.

5.4 Compatibility

Site Isolation strives to avoid web-visible changes. For example, we found that CORB blocks less than 1% of responses, most of which are not observable; if it only relied on content type and not confirmation sniffing, it would block 20% of responses [17]. Also, since cross-origin frame interactions had been mostly asynchronous prior to our work, making these interactions cross-process is largely transpar-

[4] If a site's main content required logging in, we picked the next highest-ranked site.

ent to web pages. During deployment, we closely monitored bug reports for several months to judge the impact on actual users and content. We have received around 20 implementation bugs, most of which are now fixed. We did uncover some behavior changes, described below. Overall, however, none of the bug reports warranted turning Site Isolation off, indicating that our design does not result in major compatibility problems when deployed widely.

Asynchronous Full-page Layout. With Site Isolation, full-page layout is no longer synchronous, since the frames of a page may be spread across multiple processes. For example, if a page changes the size of a frame and then sends it a `postMessage`, the receiving frame may not yet know its new size when receiving the message. We found that this disrupted behavior for some pages, but since the HTML spec does not guarantee this behavior and relatively few sites were affected, we chose not to preserve the old ordering. Instead, we provided guidance for web developers to fix the few affected pages [7] and are pursuing specification changes to explicitly note that full-page layout is asynchronous [27].

Partial Failures. Site Isolation can expose new failure modes to web pages, because out-of-process iframes may crash or become unresponsive independently from their embedder, after having been loaded. Although this may lead to unexpected behavior in the page, it happens rarely enough to avoid being a problem in practice, and for users, losing an iframe is usually preferable to losing the entire page.

Detecting Site Isolation. A web page should not know if it is rendered with or without Site Isolation, and we have avoided introducing APIs for doing so: a browser's process model is an implementation detail that developers should not depend on. We did encounter and fix some bugs that allowed detection of Site Isolation, such as differing JavaScript exception behavior for in-process and out-of-process frames. Fundamentally, though, it is possible to detect Site Isolation via timing attacks. For example, a cross-process `postMessage` will take longer than a same-process `postMessage`, due to an extra IPC hop through the browser process; a web page could perform a timing analysis to detect whether a frame is in a different process. However, such timing differences are unlikely to affect compatibility, and we have not received any such reports.

6 Future Directions

Site Isolation protects a great deal of site data against renderer exploit attackers and memory disclosure attackers, but there is a strong incentive to address the limitations outlined in Section 2.2.

It is worth noting that web browsers are not alone in facing a new security landscape. Other software systems that isolate untrustworthy code may require architecture changes to avoid leaking data via microarchitectural state. For example, SQL queries in databases might pose similar risks [47]. Applications that download and render untrustworthy content from the web, such as document editors, should likewise leverage OS abstractions to isolate their own principals [42].

6.1 Protecting More Data

CORB currently only protects HTML, XML, and JSON responses, and only when the browser can confirm them using sniffing or headers. There are several options for protecting additional content, from using headers to protect particular responses, to expanding CORB to cover more types, to changing how browsers request subresources.

First, web developers can explicitly protect sensitive resources without relying on CORB, using a `Cross-Origin-Resource-Policy` response header [21] or refusing to serve cross-site requests based on the `Sec-Fetch-Site` request header [71].

Second, the Chrome team is working to isolate cross-site PDFs and other types [2, 60]. Developer outreach may also cut down on mislabeled subresources, eliminating the need for CORB confirmation sniffing.

Third, recent proposals call for browsers to make cross-origin subresource requests without credentials by default [73]. This would prevent almost all sensitive cross-site data from entering a renderer process, apart from cases of ambient authority (e.g., intranet URLs which require no credentials).

These options may close the gaps to ensure essentially all sensitive web data is protected by Site Isolation.

6.2 Additional Layers of Mitigation

Because Site Isolation uses OS process boundaries as an isolation mechanism, it is straightforward to combine it with additional OS-level mitigations for attacks. This may include other sandboxing mechanisms (e.g., treating different sites as different user accounts) or mitigations for additional types of transient execution attacks. For example, microcode updates and OS mitigations (e.g., PTI or disabling Hyper-Threading) may be needed for cross-process or user/kernel attacks [15, 24, 40, 57, 66]. These are complementary to the mitigations Site Isolation offers for same-process attacks, where the OS and hardware have less visibility.

6.3 Practical Next Steps

Mobile Devices. This paper has described deploying Site Isolation to users on desktop and laptop devices, but the new web attackers are important to consider for mobile phone browsers as well. Site Isolation faces greater challenges on mobile devices due to fewer device resources (e.g., memory, CPU cores) and a different workload: there are fewer renderer processes in the working set due to proactive discarding by the mobile OS, and thus fewer opportunities for process sharing. We are investigating options for deploying similar mitigations on mobile browsers, such as isolating a subset of sites that need the protection the most.

Isolation in Other Browsers. There are opportunities for other browsers to provide a limited form of process isolation without the significant implementation requirements of out-of-process iframes. For example, sites might adopt headers like `Cross-Origin-Opener-Policy` to opt into a mode that can place a top-level document in a new process by disrupting some cross-window scripting [44].

Origin Isolation. Within browsers with Site Isolation, further isolation may be practical by selectively moving from a site granularity to a finer origin granularity. Too many web sites rely on modifying `document.domain` to deploy origin isolation by default, but browsers may allow sites to opt out of this feature and thus become eligible for origin isolation [72]. Making this optional may reduce the impact on the process count. Similarly, we plan to evaluate the overhead impact of isolating opaque origins, especially to improve security for sandboxed same-site iframes.

Performance. Finally, there are performance opportunities to explore to reduce overhead and take advantage of the new architecture. More aggressive renderer discarding may be possible with less cross-site sharing of renderer processes. Isolating cross-origin iframes from some web applications may also provide performance benefits by parallelizing the workload, moving slower frames to a different process than the primary user interface to keep the latter more responsive.

7 Related Work

Prior to this work, all major production browsers, including IE/Edge [76], Chrome [52], Safari [70], and Firefox [43], had multi-process architectures that rendered untrustworthy web content in sandboxed renderer processes, but they did not enforce process isolation between web security principals, and they lacked architectural support for rendering embedded content such as iframes out-of-process. Site Isolation makes Chrome the first widely-adopted browser to add such support. Other research demonstrated a need for an architecture like Site Isolation by showing how existing browsers are vulnerable to cross-site data leaks, local file system access via sync from cloud services, and transient execution attacks [25, 33, 53].

Several research browsers have proposed isolating web principals in different OS processes, including Gazelle [68], OP and its successor OP2 [23, 62], and IBOS [63]. Compared to these proposals, Site Isolation is the first to support the web platform in its entirety, with practical performance and compatibility. First, these proposals all define principals as origins, but this cannot support pages that change `document.domain` [12]. Other research browsers isolate web applications with principals that are similarly incompatible: Tahoma [16] uses custom manifests, while SubOS [31, 32] uses full URLs that include path in addition to origin. To preserve compatibility, we adopt the *site* principal proposed in [52]; this also helps reduce process count compared to origins. Second, we describe new optimizations that make Site Isolation practical, and we evaluate our architecture on a real workload of Chrome users. This shows that Site Isolation introduces almost no additional page load latency and only 9-13% memory overhead, lower than expected from microbenchmark evaluations. Third, we comprehensively evaluate the implications of new transient execution attacks [8] for browser security. Fourth, we show that protecting cross-origin network responses requires new forms of confirmation sniffing to preserve compatibility; content types and even traditional MIME sniffing are insufficient. Finally, while Gazelle, OP2, and IBOS have out-of-process iframes, our work overcomes many challenges to support these in a production browser, such as supporting the full set of cross-process JavaScript interactions, challenges with painting and input event routing, and updating affected features (e.g., find-in-page, printing).

The OP and OP2 browsers [23, 62] also use OS processes to isolate other browser components, including the network stack, storage, and display. Such additional process separation is orthogonal to Site Isolation and offers complementary benefits, such as making the browser more modular, reducing the size of the browser process, and keeping crashes in one component isolated from the rest of the browser.

Dong et al [19] argued that practical browser designs will require a trade-off between finer-grained isolation and performance. Our experience echoes this finding, and we indeed make trade-offs to reduce memory overhead, such as isolating sites rather than origins. Dong et al's evaluation relied on sequentially browsing top Alexa sites; we additionally collect measurements from browsing workloads in the wild, providing a more realistic performance evaluation. For example, this factors in process sharing across multiple tabs, which significantly reduces overhead in practice.

Other researchers propose disabling risky JavaScript features unless user-defined policies indicate they are safe for a desired site [56, 61]. These approaches aim to disrupt a wide variety of attacks (including microarchitectural), but they impose barriers to adoption of powerful web features, and they rely on users or third parties to know when features are safe to enable. Site Isolation's scope is more limited by compatibility, but it does not require actions from users or disabling powerful features.

8 Conclusion

The web browser threat model has changed significantly. Web sites face greater threats of data leaks within the browser due to compromised renderer processes and transient execution attacks. Site Isolation offers the best path to mitigating these attacks in the browser, protecting a significant amount of site data today with future opportunities to expand the coverage. We have shown that Site Isolation is practical to deploy in a production desktop web browser, incurring a 9-13% total memory overhead on real-world work-

loads. We recommend that web developers and browser vendors continue down this path, protecting additional sensitive resources, adding more mitigations, and pursuing similar isolation in environments like mobile browsers.

9 Acknowledgements

We would like to thank Łukasz Anforowicz, Jann Horn, Ken Buchanan, Chris Palmer, Adrienne Porter Felt, Franziska Roesner, Tadayoshi Kohno, Antoine Labour, Artur Janc, our shepherd Adam Doupé, and the anonymous reviewers for their input on this paper. We also thank the many Chrome team members who made this work possible.

References

[1] Adobe. Flash & The Future of Interactive Content. https://theblog.adobe.com/adobe-flash-update/, 2017.

[2] L. Anforowicz. More CORB-protected MIME types - adding protected types one-by-one. https://github.com/whatwg/fetch/issues/860, Jan. 2019.

[3] A. Barth, C. Jackson, and J. C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In *CCS*, 2008.

[4] A. Barth, D. Song, and J. Caballero. Secure Content Sniffing for Web Browsers, or How to Stop Papers from Reviewing Themselves. In *IEEE Symposium on Security and Privacy*, 2009.

[5] A. Barth, J. Weinberger, and D. Song. Cross-origin JavaScript Capability Leaks: Detection, Exploitation, and Defense. In *USENIX Security*, 2009.

[6] M. Blumberg. Security enhancements and more for enterprise Chrome browser customers. https://www.blog.google/products/chrome-enterprise/security-enhancements-and-more-enterprise-chrome-browser-customers/, Dec. 2017.

[7] M. Bynens. Site Isolation for web developers. https://developers.google.com/web/updates/2018/07/site-isolation, July 2018.

[8] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security*, 2019.

[9] Changes to Cross-Origin Requests in Chrome Extension Content Scripts. https://www.chromium.org/Home/chromium-security/extension-content-script-fetches, Jan. 2019.

[10] S. Chen, H. Chen, and M. Caballero. Residue Objects: A Challenge to Web Browser Security. In *EuroSys*, 2010.

[11] S. Chen, D. Ross, and Y.-M. Wang. An Analysis of Browser Domain-Isolation Bugs and A Light-Weight Transparent Defense Mechanism. In *CCS*, 2007.

[12] Chrome Platform Status: DocumentSetDomain. https://www.chromestatus.com/metrics/feature/popularity#DocumentSetDomain, Dec. 2018.

[13] Chromium Blog: Changes to the Field Trials infrastructure. https://blog.chromium.org/2012/05/changes-to-field-trials-infrastructure.html, May 2012.

[14] Chromium Security: Mitigating Side-Channel Attacks. https://www.chromium.org/Home/chromium-security/ssca, Jan. 2018.

[15] J. Corbet. The current state of kernel page-table isolation. https://lwn.net/Articles/741878/, Dec. 2017.

[16] R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy. A Safety-Oriented Platform for Web Applications. In *IEEE Symposium on Security and Privacy*, 2006.

[17] Cross-Origin Read Blocking (CORB). https://chromium.googlesource.com/chromium/src/+/master/services/network/cross_origin_read_blocking_explainer.md, Mar. 2018.

[18] Cross-Origin Resource Sharing (CORS). https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS, 2019.

[19] X. Dong, H. Hu, P. Saxena, and Z. Liang. A Quantitative Evaluation of Privilege Separation in Web Browser Designs. In *ESORICS*, 2013.

[20] Fetch Standard: CORB. https://fetch.spec.whatwg.org/#corb, May 2018.

[21] Fetch Standard: Cross-Origin-Resource-Policy header. https://fetch.spec.whatwg.org/#cross-origin-resource-policy-header, Jan. 2019.

[22] First Contentful Paint. https://developers.google.com/web/tools/lighthouse/audits/first-contentful-paint, 2019.

[23] C. Grier, S. Tang, and S. T. King. Designing and Implementing the OP and OP2 Web Browsers. *TWEB*, 5:11, May 2011.

[24] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard. KASLR is Dead: Long Live KASLR. In *ESSoS*, 2017.

[25] N. Hadad and J. Afek. Overcoming (some) Spectre browser mitigations. https://alephsecurity.com/2018/06/26/spectre-browser-query-cache/, June 2018.

[26] V. Hailperin. Cross-Site Script Inclusion. https://www.scip.ch/en/?labs.20160414, Apr. 2016.

[27] C. Harrelson. Adjust event loop processing model to allow asynchronous layout of frames. https://github.com/whatwg/html/issues/3727, May 2018.

[28] J. Horn. Speculative Execution, Variant 4: Speculative Store Bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018.

[29] HTML Living Standard: opaque origin. https://html.spec.whatwg.org/multipage/origin.html#concept-origin-opaque, Jan. 2019.

[30] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schecter, and C. Jackson. Clickjacking: Attacks and Defenses. In *USENIX Security*, 2012.

[31] S. Ioannidis and S. M. Bellovin. Building a secure web browser. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, 2001.

[32] S. Ioannidis, S. M. Bellovin, and J. M. Smith. Sub-operating systems: a new approach to application security. In *Proceed-*

ings of the 10th SIGOPS European workshop, 2002.

[33] Y. Jia, Z. L. Chua, H. Hu, S. Chen, P. Saxena, and Z. Liang. "The Web/Local" Boundary Is Fuzzy: A Security Study of Chrome's Process-based Sandboxing. In *CCS*, 2016.

[34] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy*, 2019.

[35] M. Lipp, D. Gruss, M. Schwarz, D. Bidner, C. Maurice, and S. Mangard. Practical Keystroke Timing Attacks in Sandboxed JavaScript. In *ESORICS*, 2017.

[36] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security*, 2018.

[37] R. McIlroy, J. Sevcík, T. Tebbi, B. L. Titzer, and T. Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *CoRR*, abs/1902.05178, 2019.

[38] Memory Protection Keys for Userspace. `https://www.kernel.org/doc/Documentation/x86/protection-keys.txt`, Jan. 2019.

[39] Microsoft Edge Team. Mitigating speculative execution side-channel attacks in Microsoft Edge and Internet Explorer. `https://blogs.windows.com/msedgedev/2018/01/03/speculative-execution-mitigations-microsoft-edge-internet-explorer/`, Jan. 2018.

[40] M. Minkin, D. Moghimi, M. Lipp, M. Schwarz, J. V. Bulck, D. Genkin, D. Gruss, B. Sunar, F. Piessens, and Y. Yarom. Fallout: Reading Kernel Writes From User Space. `https://mdsattacks.com`, 2019.

[41] M. Moroz and S. Glazunov. Analysis of UXSS exploits and mitigations in Chromium. Technical report, Google, 2019. `https://ai.google/research/pubs/pub48028`.

[42] A. Moshchuk, H. J. Wang, and Y. Liu. Content-based Isolation: Rethinking Isolation Policy Design on Client Systems. In *CCS*, 2013.

[43] N. Nguyen. The Best Firefox Ever. `https://blog.mozilla.org/blog/2017/06/13/faster-better-firefox/`, 2017.

[44] R. Niwa. Restricting cross-origin WindowProxy access (Cross-Origin-Opener-Policy). `https://github.com/whatwg/html/issues/3740`, June 2018.

[45] N. Oskov. Site Isolation Benchmark Script. `https://github.com/naskooskov/site-isolation-benchmark`, May 2019.

[46] OWASP. XSS (Cross Site Scripting) Prevention Cheat Sheet. `https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.md`, Feb. 2019.

[47] C. Palmer. Isolating Application-Defined Principals. `https://noncombatant.org/application-principals/`, July 2018.

[48] F. Pizlo. What Spectre and Meltdown Mean For WebKit. `https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/`, Jan. 2018.

[49] Post-Spectre Threat Model Re-Think. `https://chromium.googlesource.com/chromium/src/+/master/docs/security/side-channel-threat-model.md`, May 2018.

[50] C. Reis. Improving extension security with out-of-process iframes. `https://blog.chromium.org/2017/05/improving-extension-security-with-out.html`, May 2017.

[51] C. Reis, A. Barth, and C. Pizano. Browser Security: Lessons from Google Chrome. *Commun. ACM*, 52(8):45–49, Aug. 2009.

[52] C. Reis and S. D. Gribble. Isolating Web Programs in Modern Browser Architectures. In *EuroSys*, 2009.

[53] R. Rogowski, M. Morton, F. Li, F. Monrose, K. Z. Snow, and M. Polychronakis. Revisiting Browser Security in the Modern Era: New Data-Only Attacks and Defenses. In *IEEE European Symposium on Security and Privacy*, 2017.

[54] J. Ruderman. The Same Origin Policy. `https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy`, 2019.

[55] J. Schuh. The Final Countdown for NPAPI. `https://blog.chromium.org/2014/11/the-final-countdown-for-npapi.html`, 2014.

[56] M. Schwarz, M. Lipp, and D. Gruss. JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks. In *NDSS*, 2018.

[57] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. `https://zombieloadattack.com`, 2019.

[58] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *Financial Cryptography and Data Security*, Jan 2017.

[59] SharedArrayBuffer. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer`, 2019.

[60] Site Isolate PDFium. `https://crbug.com/809614`, Jan. 2019.

[61] P. Snyder, C. Taylor, and C. Kanich. Most Websites Don't Need to Vibrate: A Cost-Benefit Approach to Improving Browser Security. In *CCS*, 2017.

[62] S. Tang, S. T. King, and C. Grier. Secure Web Browsing with the OP Web Browser. In *IEEE Symposium on Security and Privacy*, 2008.

[63] S. Tang, H. Mai, and S. T. King. Trust and Protection in the Illinois Browser Operating System. In *OSDI*, 2010.

[64] D. Topic. Moving to a Plugin-Free Web. `https://blogs.oracle.com/java-platform-group/moving-to-a-plugin-free-web`, Jan. 2016.

[65] Untrusted code mitigations. `https://v8.dev/docs/untrusted-code-mitigations`, Jan. 2018.

[66] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. RIDL: Rogue In-flight Data Load. In *IEEE Symposium on Security and Privacy*, 2019.

[67] L. Wagner. Mitigations landing for new class of timing attack. https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/, Jan. 2018.

[68] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The Multi-Principal OS Construction of the Gazelle Web Browser. In *USENIX Security*, 2009.

[69] Web Page Replay. https://github.com/catapult-project/catapult/blob/master/web_page_replay_go/README.md, Sept. 2017.

[70] WebKit2. https://trac.webkit.org/wiki/WebKit2, July 2011.

[71] M. West. Fetch Metadata Request Headers. https://mikewest.github.io/sec-metadata, 2018.

[72] M. West. Proposal: Control over 'document.domain'. https://github.com/w3c/webappsec-feature-policy/issues/241, Nov. 2018.

[73] M. West. Incrementally Better Cookies. https://mikewest.github.io/cookie-incrementalism/draft-west-cookie-incrementalism.html, May 2019.

[74] Window.sessionStorage. https://developer.mozilla.org/en-US/docs/Web/API/Window/sessionStorage, 2019.

[75] Y. Yarom and K. Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*, 2014.

[76] A. Zeigler. IE8 and Loosely-Coupled IE (LCIE). https://blogs.msdn.microsoft.com/ie/2008/03/11/ie8-and-loosely-coupled-ie-lcie/, 2008.

A Determining Site Principals

This appendix provides additional details on how we define principals used in Site Isolation. Figure 5 compares principal definitions in monolithic browsers, multi-process browsers that isolate coarser-grained groups of principals, Site Isolation, and Origin Isolation. Origin Isolation, where principals are defined as origins, offers stronger security guarantees at the cost of breaking `document.domain` compatibility and performance challenges due to a larger number of principals.

As noted in Section 3.1, computing site URL for most HTTP(S) URLs is straightforward, but some web platform features require special treatment. For example, frames may be navigated to `about:blank`, a special URL which must inherit the security origin, and hence the site, from the frame initiating the navigation. The web also supports *nested URLs* such as `blob:` URLs. These URLs embed an origin; e.g., `blob:http://example.com/UUID` addresses an in-memory blob of data controlled by the `http://example.com` origin. In these cases, we extract the inner origin from the URL and then convert it to a site.

A document may also embed a frame and specify its HTML content *inline* rather than from the network, either using the `srcdoc` attribute (e.g., `<iframe srcdoc="<html>content</html>">`) or a `data:` URL (e.g., `data:text/html,<html>content</html>`). `Srcdoc` frames inherit their creator's origin and must stay in the principal of their embedding document. In contrast, `data:` URLs load in an opaque origin [29], which cannot be accessed from any other origin. Browsers may choose to load each `data:` URL in its own separate principal and process, but our current implementation uses the creator's principal (which typically controls the content) to reduce the number of processes required. Similarly, our current implementation keeps same-site iframes with the `sandbox` attribute, which typically load in an opaque origin, in the principal of their URL's site. In practice, sites often use sandboxed iframes for untrustworthy content that they wish to isolate from the rest of the site; we discuss opportunities for finer-grained isolation within a site in Section 6.3.

Non-web Principals. Many browsers can load documents that do not originate from the web, including content from local files, extensions, browser UI pages, and error pages. These forms of content utilize the web platform for rendering, so the browser must define principals for them. Each local URL (e.g., `file:///homes/foo/a.html`) is typically treated as its own origin by the browser, so each path could use a separate principal and process. Our current implementation treats all local files as part of the same *file* principal to reduce the process count, since they ultimately belong to a local user. We may revise this to isolate each file in the future, since this group of local files may contain less trustworthy pages saved from the web.

We assign content from extensions to a separate shared principal, and we isolate all browser UI pages, such as settings or download manager, from one another. These pages require vastly different permissions and privileges, and a compromise of one page (e.g., a buggy extension) should not be able to take advantage of permissions granted to a more powerful page (e.g., a download management page that can download and open files). We do allow extensions to share processes with each other to reduce the process count; thus, Figure 5 (c) shows extensions in a shared principal. However, extensions never share processes with other types of pages.

B Features Updated to Support Out-of-process iframes

This appendix lists a subset of Chrome features that needed to be updated to support out-of-process iframes, beyond those discussed in Section 3.4.

- Accessibility (e.g., screen readers).
- Developer tools.
- Drag and drop.
- Extensions (e.g., injecting scripts into frames of a page).
- Find-in-page.

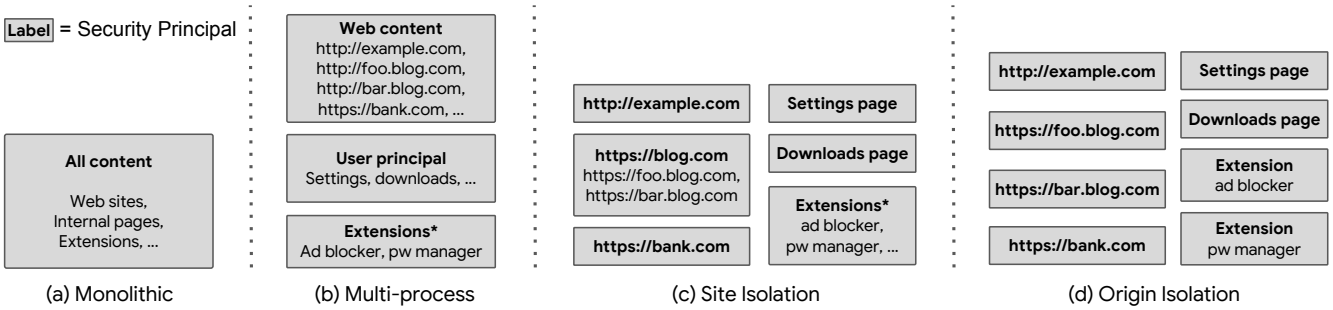


Figure 5: **Evolution of security principals in browser architectures.** Compared to prior browser architectures, Site Isolation defines finer-grained principals that correspond to sites. Origin Isolation (d) further refines sites to origins and is the most desirable principal model in the long term, but backward compatibility and performance challenges currently limit its practicality.
** In pre-Site-Isolation browsers (b), extensions were isolated in higher-privileged processes, but with a caveat: extensions could embed web URL iframes which would stay in the extension's process. With Site Isolation (c), process sharing across the web/extension boundary is no longer possible, though extensions may still share a process with one another.*

- Focus (e.g., tracking focused page and frame, focus traversal when pressing Tab).
- Form autofill.
- Fullscreen.
- IME (Input Method Editor).
- Input gestures.
- JavaScript dialogs.
- Mixed content handling.
- Multiple monitor and device scale factor support.
- Password manager.
- Pointer Lock API.
- Printing.
- Task manager.
- Resource optimizations (e.g., deprioritizing offscreen content).
- Malware and phishing detection.
- Save page to disk.
- Screen Orientation API.
- Scroll bubbling.
- Session restore.
- Spellcheck.
- Tooltips.
- Unresponsive renderer detector and dialog.
- User gesture tracking.
- View source.
- Visibility APIs.
- Webdriver automation.
- Zoom.

C Compromised Renderer Enforcements

This appendix lists the current places that privileged browser components in Chrome (version 76) limit the behavior of a renderer process based on its associated site, to mitigate compromised renderers.

- Cookie reads and writes (`document.cookie`, `HttpOnly` cookies).
- Cross-Origin Read Blocking implementation [20].
- `Cross-Origin-Resource-Policy` blocking [21].
- Frame embedding (`X-Frame-Options`).
- JavaScript code cache.
- Messaging (`postMessage`, `BroadcastChannel`).
- Password manager, Credential Management API.
- Storage (`localStorage`, `sessionStorage`, `indexedDB`, `blob` storage, Cache API, WebSQL).
- Preventing web page access to `file://` URLs.
- Web permissions (e.g., geolocation, camera).

We expect the following enforcements to be possible as well, with additional implementation effort.

- Address bar origin.
- Custom HTTP headers requiring CORS.
- Feature Policy.
- iframe sandbox behaviors.
- `Origin` Header and CORS implementation.
- SameSite cookies.
- `Sec-Fetch-Site` [71].
- User gestures.

Everyone is Different: Client-side Diversification for Defending Against Extension Fingerprinting

Erik Trickel[⋆]*, Oleksii Starov*[†]*, Alexandros Kapravelos*[‡]*, Nick Nikiforakis*[†]*, and Adam Doupé*[⋆]

[⋆]*Arizona State University*
{etrickel, doupe}@*asu.edu*

[†]*Stony Brook University*
{ostarov, nick}@*cs.stonybrook.edu*

[‡]*North Carolina State University*
akaprav@*ncsu.edu*

Abstract

Browser fingerprinting refers to the extraction of attributes from a user's browser which can be combined into a near-unique fingerprint. These fingerprints can be used to re-identify users without requiring the use of cookies or other stateful identifiers. Browser extensions enhance the client-side browser experience; however, prior work has shown that their website modifications are fingerprintable and can be used to infer sensitive information about users.

In this paper we present CloakX, the first client-side anti-fingerprinting countermeasure that works without requiring browser modification or requiring extension developers to modify their code. CloakX uses client-side diversification to prevent extension detection using anchorprints (fingerprints comprised of artifacts directly accessible to any webpage) and to reduce the accuracy of extension detection using structureprints (fingerprints built from an extension's behavior). Despite the complexity of browser extensions, CloakX automatically incorporates client-side diversification into the extensions and maintains equivalent functionality through the use of static and dynamic program analysis. We evaluate the efficacy of CloakX on 18,937 extensions using large-scale automated analysis and in-depth manual testing. We conducted experiments to test the functionality equivalence, the detectability, and the performance of CloakX-enabled extensions. Beyond extension detection, we demonstrate that client-side modification of extensions is a viable method for the late-stage customization of browser extensions.

1 Introduction

As the web expands and continues being the platform of choice for delivering applications to users, the browser becomes a core component of a user's interactions with the web. Modern browsers advertise a wide range of features, from cloud-syncing and notifications to password management and peer-to-peer video and audio communications. An important feature of modern browsers is their ability to be extended by users, as they see fit, by installing *browser extensions*. Namely, Google Chrome and Mozilla Firefox, the browsers with the largest market share, offer dedicated browser extension stores that house tens of thousands of extensions. In turn, these extensions advertise a wide range of additional features, such as enabling the browser to store passwords with online password managers, blocking ads, and saving articles for later reading.

From a security perspective, the ability to load third-party code into the browser comes at a cost, even though extensions rely on web technologies such as HTML, JavaScript, and CSS. Browsers afford extensions significantly more privileges than they do to a webpage. For example, the same origin policy restricts webpages from accessing content, such as a cookie, that does not originate from the same domain. For a webpage to bypass this restriction, it must implement cross-origin resource sharing, whereas extensions may not only access resources of any domain but may also alter the content. Historically, malicious extensions abuse these privileges to perform advertising fraud and to steal private and financial user data [22, 28, 44, 47].

Next to security issues, using browser extensions can also lead to the loss of privacy. Given that users choose the extensions to install, it is possible to make inferences about a user's thoughts and beliefs *based solely on the extensions she keeps.* For example, the detection of a coupon-finding extension [1] reveals information about the user's income-level. Additionally, an extension that hides articles about certain political figures [20, 21] reveals the user's political leanings. Lastly, the use of browser extensions may provide a means for websites to persistently identify a user over the course of distinct browser sessions.

Although browser vendors do not offer any programmatic methods for a webpage's JavaScript to detect the extensions currently installed in a user's browser, researchers recently discovered side-channel techniques for fingerprinting many extensions. Sjösten et al. were the first to demonstrate a new method for detecting browser extensions that exploited the public nature of *web-accessible resources* (WARs) [38]. A

WAR is any resource (e.g., JavaScript or image) within an extension that the extension identifies as externally accessible. As a result, a webpage can determine whether a visitor uses an extension by requesting one of the exposed WARs. Sjösten et al. showed that more than 50% of the top 1,000 browser extensions use WARs, which any webpage might use to detect extensions. Later, Starov and Nikiforakis demonstrated another technique for fingerprinting extensions that uses an extension's modifications to the document-object-model (DOM) to detect their presence [42]. The authors developed XHOUND, a system that automatically discovers the DOM side-effects of extensions. Through their experiments, they showed that more than 10% of the top 50K extensions were fingerprintable.

One approach to reducing fingerprintable extensions is through education and developer training. However, historically, developers — and web developers in particular — ignore even well-known security concerns. Even after nearly 20 years, the most common website vulnerabilities are still SQL injection vulnerabilities [43]. Therefore, it is unlikely that asking extension developers to make their extensions less fingerprintable will have the desired effect on the ecosystem.

To empower users to protect their own privacy, in this paper we propose CloakX, a client-side countermeasure against extension detection using fingerprints. Instead of trying to remove the fingerprintable attributes of extensions, our approach is to automatically alter, randomize, and add to these attributes without requiring web browser modifications or any involvement from the extension's developer. Through these modifications, CloakX diversifies the extension's anchorprints, which are fingerprints consisting of items that can be accessed directly from a webpage, and structureprints, which are fingerprints that embody the structural changes an extension makes to a webpage (for more details refer to Section 2.2). On the surface, client-side diversification of the fingerprintable attributes seems straightforward; however, the dynamic nature of JavaScript and the complexity of the browser extension's architecture necessitated a complex approach that relies on both static and dynamic program analysis.

CloakX uses static and dynamic analysis techniques to automatically diversify the extension's fingerprint without modifying the browser, without requiring any changes by the extension's author, and without altering the extension's functionality. To diversify the extension's anchorprint, CloakX automatically renames WARs, IDs, and class names and corrects any references to them in the extension's code, which severs the link between the published extension and the currently installed version. In addition to static changes, the diversification is also performed by our dynamic DOM proxy (Droxy), which intercepts DOM modifications from the extension's code and makes the changes on-the-fly. To diversify the extension's structureprint, Droxy also injects random tags, attributes, and custom attributes into each webpage, which obfuscates the extension's structureprint. As a result, an extension cloaked by CloakX is undetectable by a webpage using anchorprints and is obfuscated from a webpage using structureprints; however, from the user's point of view, the extension operates the same.

In summary, we make the following contributions:

- We present the design of a novel system that automatically identifies and randomizes browser extension fingerprints to defend against existing extension fingerprinting techniques without requiring any browser changes or any involvement from the extension's developer.
- We describe the implementation of our design into a prototype, CloakX, that uses a combination of: (1) static rewriting of extension JavaScript code and (2) a dynamic DOM proxy, Droxy, that intercepts and rewrites extension requests on-the-fly.
- We use a combination of high-fidelity testing (extensive manual testing) and low-fidelity testing (broad automated testing) on the extensions rewritten by CloakX to quantify the breakage caused by our system, demonstrating that client-side modification of extensions introduces minimal defects.
- We also evaluate the detectability of cloaked extensions and show that some cloaked extensions are undetectable while others are more difficult to detect.

2 Background

In this section, we provide insights into the complexity of modern browser extension frameworks that must be taken into account when designing a client-side countermeasure against extension fingerprinting. We start by describing the architecture of browser extensions, focusing on the details that pertain to their fingerprintability. Next, we discuss fingerprinting and detecting extensions using anchorprints (fingerprints that are comprised of items directly accessible from a tracking webpage's JavaScript) and structureprints (fingerprints built from the extension's behavior). Last, we finish this section by presenting the threat model that CloakX can defend against.

2.1 Browser Extensions Explained

While modern web browsers provide an ever-increasing range of functionality to users and webpages, an off-the-shelf browser cannot possibly provide a sufficiently large set of features to satisfy every user's browsing needs. To improve the user's browsing experience, browsers enable users to enhance their functionality through extensions. Users add extensions to their browsers to change the browser's look, to add helpful toolbars, to block ads, and to enhance popular webpages [5].

Although extensions utilize web technologies such as HTML, CSS, and JavaScript, they also have access to powerful extension-only APIs that enable them to, among others, access and modify cross-origin content and a browser's

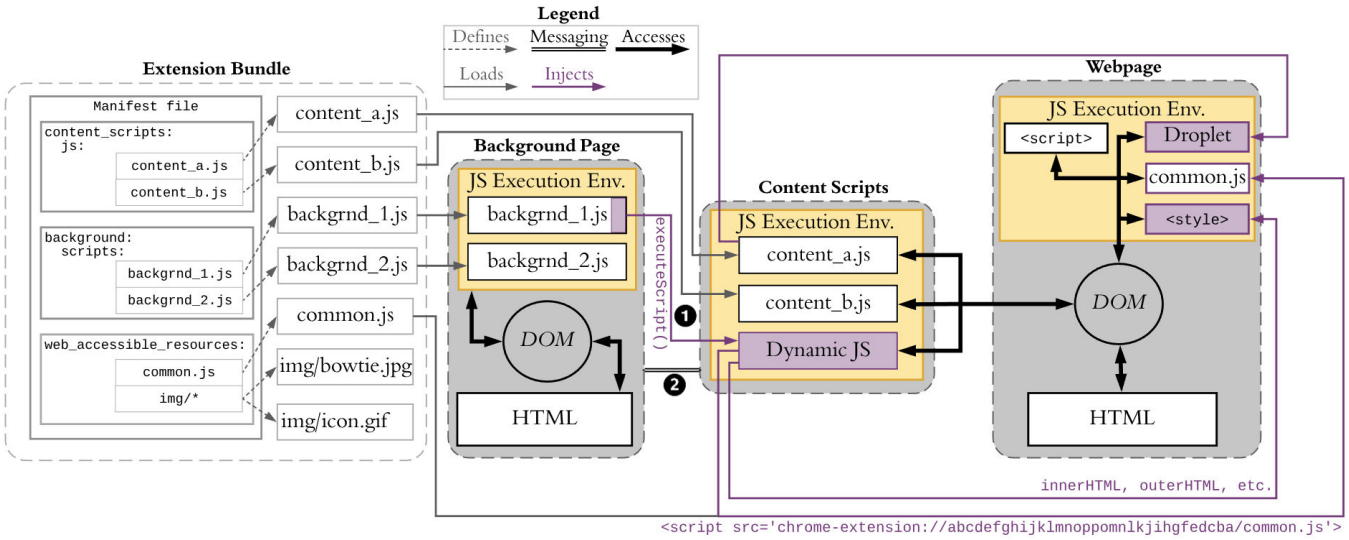


Figure 1: Extension architecture. A high-level overview of Chrome's extension architecture with the static content of the extension on the left side and the multiple execution environments on the right. Background pages can 1) inject content scripts dynamically using the executeScript() *method in Chrome's extension API and 2) send and receive messages from the content scripts.*

client-side storage. However, before an extension can access broader privileges or interact with a webpage, it must request this access from the browser. As Figure 1 depicts, the modern extension architecture implements a layered security approach within the browser that creates multiple execution environments with varying levels of persistence and privileges for each extension and webpage.

The left-hand side of Figure 1 depicts the static parts of an extension, including items such as the manifest, JavaScript, HTML, and image files. For the browser to parse and install an extension, it must have a *manifest* file which defines the extension's properties. Similar to the manifest shown in Figure 1, extensions commonly rely on three properties, which describe background pages, content scripts, and web accessible resources [5].

When the *background* property is included in the extension's manifest, the browser automatically constructs a hidden background page for the extension. The background page contains HTML, a DOM, and a separate JavaScript execution environment (labeled as "Background Page" in Figure 1). The JavaScript executed in the background page often contains the main logic of the extension, maintains long-term state, and operates independently from the life-cycle of the webpages [28].

Content scripts bridge the gap between the background page and the current webpage. An extension uses content scripts to modify the current webpage and communicate with the background page. These content scripts are either statically declared by an extension in the manifest file or programmatically injected into the current webpage. For example, on the left-hand side of Figure 1, the manifest declares the two content scripts content_a.js and content_b.js. To programmatically inject a content script, an extension must call executeScript() from a background page (see ❶ in Figure 1).

To modify a webpage, a content script uses the webpage's DOM [10]. DOM APIs provide a systematic way for interacting with a webpage. In this paper, we call the content script's interaction with the DOM APIs *DOM requests*.

Notice in Figure 1 that the background page, content scripts, and webpage each run their own JavaScript execution environment. The separate execution environments prevent the JavaScript variables and functions from directly interacting. Google Chrome's documentation states that content scripts "live in an isolated world, allowing a content script to make changes to its JavaScript environment without conflicting with the page or additional *content scripts*" [3] (emphasis added). This statement, however, is misleading because we experimentally discovered that content scripts loaded from the same extension share variables and can call functions from other content scripts. Thus, an extension's content scripts share a single execution environment; however, they do not share an environment with the background page, webpage, or other extensions (depicted in Figure 1).

Using DOM requests, a content script has significant control over the rendered webpage. Content scripts can inject HTML into the webpage (using DOM element properties such as innerHTML or DOM methods such as appendChild()). We call this injected HTML *droplets* (the extension drops them onto the webpage). Among other elements, droplets may contain <script> tags where the extension includes either inline or remote JavaScript. By injecting JavaScript, the content script purposefully bypasses the isolation between the content scripts and the webpage's execution environments.

The Chrome Extension API provides privileged functionality available only to extensions. Chrome grants back-

ground scripts broad access to the API's capabilities. However, Chrome grants content scripts limited access to the API while making the API inaccessible to webpages. For example, only an extension's background page can access network resources, view platform information, and communicate with native applications. However, both content scripts and background scripts may use the API to initiate and listen for communications from one another via the appropriate Chrome APIs (as shown by the double lines towards the bottom of Figure 1). Background scripts cannot directly interact with a webpage, however they can *indirectly* send messages to it via the extension API using the method chrome.runtime.sendMessage() [2]. Part of the reason for this layered security model, including the separate execution environments, is to isolate the components and prevent webpages from unauthorized access to the extension API's more sensitive functions.

Another important property in the manifest is the *web-accessible-resources* property [7]. Prior to January 2014, Chrome permitted external access to all of an extension's resources, i.e., a webpage could reference resources belonging to installed extensions. In more modern versions of Google Chrome, an extension must explicitly whitelist a resource before a webpage may retrieve it [8]. An extension whitelists its resources by adding them to the *web-accessible-resources* property in the manifest. Once added, a resource becomes accessible to any webpage or any installed extension.

To access a web accessible resource (WAR) from the context of a web page, a webpage developer uses a URL of the format: chrome-extension://[extId]/[path-to-resource]. The extId in the URL is a unique identifier generated by the Google Web Store upon publication of an extension which does not change when extensions are updated.

2.2 Extension Fingerprinting and Detection

In 2017, Sjösten et al. demonstrated that, with WAR fingerprinting, any extension using WARs is trivially detectable by a webpage [38] by creating a database of which WARs are utilized by each extension available in the Google Store. Given that an extension's ID is globally unique and permanent, a tracker can detect an extension by requesting any one of its previously identified WARs. If the request is successful, then the corresponding extension is installed on the user's browser. Next to its simplicity and the 16,479 (28%) of extensions that utilize WARs (and are thus fingerprintable), WAR fingerprinting works in the browser's private mode.

Orthogonally to WAR fingerprinting, Starov et al.'s Extension Hound (XHOUND) [42] creates a DOM fingerprint based on the extension's DOM modifications. XHOUND uses dynamic analysis to exercise extensions and detect changes introduced to the DOM through the extension's operation. By loading a set of webpages with and without a given extension, XHOUND can compare the two resulting DOMs and isolate the DOM changes that were performed by the given extension. These changes can straightforwardly be converted into fingerprints which trackers can use to detect the presence of any DOM-modifying extension.

When using WAR and DOM fingerprints for detection of extensions, we reclassify all such fingerprints into *anchorprints* and *structureprints* to describe the method and accuracy of the detection techniques. Anchorprints rely on an *anchor* between the webpage's JavaScript and the extension. An anchor is a unique identifier formed to facilitate access and communication between webpages and extensions. An anchor provides a way to directly access elements and resources available to the webpage. Some examples of anchors include WARs, IDs, class names, and custom attributes. For example, the Chrome extension Grammarly adds a unique class to the root <html> element on each webpage. Thus, if a webpage uses document.getElementsByClassName() and receives the <html> element, it is likely the user has Grammarly installed.

An anchorprint is comprised of all the WARs, IDs, class names, and custom attributes made available by an extension. With the items in an anchorprint, a webpage need only to query the DOM or send an XMLHttpRequest to detect an extension. WARs are the most powerful of the anchorprint elements because, due to the unique extension identifier, an anchorprint with even one WAR is always 100% accurate. Although IDs, class names, and custom attributes might be 100% accurate, they often have a much lower per element accuracy than WARs because webpages and extensions alike often use some of the same names. Despite this limitation, the accuracy of the anchorprint improves dramatically with each additional element included in it.

Structureprints are less precise (in terms of fingerprinting) but are formed based on the *structure* of the changes the extension makes to the underlying webpage. Structureprints effectively create a DOM fingerprint that uses the extension's unique and intended behavior to identify the extension. The idea of a structureprint is that it can be used to detect a specific extension because the extension always behaves in a predictable manner and alters a webpage consistently, thus creating a *structure* that is unique among extensions. For instance, consider a popular Google Calendar extension that is the *only* extension with a structureprint that contains the tags a and img with the following attribute names href, location, target, blank, width, height, src, alt and style Surprisingly, we found during our experiments that a tracking webpage can reliably detect 28.93% (1,511) of extensions using only the tagName of the DOM elements added or deleted from a webpage by an extension. Adding attribute names, attribute values, and the text of the DOM elements to the structureprint increases the number of detectable extensions to 73.65% (3,847).

An important *subset* of structureprints that target an extension's behavior are called *behaviorprints*. For example,

Grammarly creates a green button inside a `text area`. With manual analysis, it is possible to identify whether the green button has been added to the webpage without relying on the IDs or class names injected by Grammarly. Another example of using behaviorprints are in the detection of ad-blocking extensions, such as Detect AdBlock [4]. However, no recent research has shown how to create a behaviorprint in an automated way at scale. As a result, current behaviorprints are limited to targeted attacks against specific extensions or narrowly constrained categories of extensions (e.g., ad-blocking extensions).

Beyond the obvious implementation differences between anchorprints and structureprints, the fingerprint classes differ in their accuracy and their destructibility. For most anchorprints, matching the WAR, ID, class name, and custom attributes of a published extension often provides a (unique) one-to-one match. However, for structureprints, finding a match is often less certain because many extensions have similar behavior, which results in the same structureprint. Another key difference between anchorprints and structureprints is the permanence of their link between the published extension and the user's installed version. For anchorprints using WARs, IDs, and class names, CloakX completely renames the values. By renaming the values, CloakX completely destroys the link between the published extension and the user's installed version. Without that link, it is impossible for a tracking webpage to use the anchorprint to identify the installed extension because the anchorprint no longer matches the published extension. Whereas with structureprints, the destruction of the link between the published extension and the user's installed version is difficult. This difficulty occurs because of the requirement that a cloaked extension retain the same behavior (i.e., user experience). By maintaining the same behavior, the structureprint of a cloaked extension is only being obfuscated, which means that with enough effort a tracker can eventually deobfuscate the cloaked structureprint and, thus, detect the cloaked extension.

2.3 Threat Model

In our threat model, attackers use a database of fingerprints to detect the extensions installed by a visitor to the site. However, we limit the attackers to the information and privileges afforded to the webpage's JavaScript execution environment. In essence, we assume that there are no zero-day vulnerabilities that would allow webpages to bypass the layered-security architecture depicted in Figure 1. Therefore, the attackers cannot access the content of an extension installed on a visitor's device.

In this paper, we explore two different types of attackers. The automated attacker uses automated extension detection techniques. Specifically, we limit the automated attacker to anchorprints and structureprints. To detect an extension, the automated attacker must find either an exact or fuzzy match to an entry in their fingerprint database. The targeted attacker is permitted to manually generate targeted structureprints using portions of the structureprint (i.e., behaviorprints) for extension detection. While we focus on defending against the automated attacker because automated large-scale detection is a feasible attack, we also include the targeted attacker to explore how CloakX can defend against the targeted attacks.

3 CloakX

The core idea behind CloakX is to diversify each extension's fingerprint from the client-side while maintaining equivalent functionality *without making any changes to the browser and without requiring the developers to alter their extensions*. Client-side diversification of the anchorprints (fingerprints comprised of items directly accessible from a tracking webpage's JavaScript) and structureprints (fingerprints built from the extension's behavior) reduces the extension's detectability by breaking a webpage's ability to link together a published extension and the one installed on the user's machine. CloakX defeats detection using an anchorprint by randomizing the names of the WARs, IDs, and classes. However, CloakX does not completely defeat anchorprint detection using custom attributes. CloakX's approaches combat custom attribute-based detection by randomly injecting more unique custom attributes into each webpage. CloakX reduces the efficacy of structureprints by introducing random attributes and tags into the webpage. Although CloakX does not completely prevent detection using custom attributes or structureprints, it is a step beyond current solutions and CloakX achieves these protections without any changes to the browser and without requiring the intervention of extension developers.

Figure 2 shows the overall process of CloakX, a multiphase tool that leverages static- and dynamic-analysis techniques to achieve extension diversification while maintaining functional equivalence. In the first phase, CloakX analyzes the extension for the DOM fingerprints and CloakX identifies the droplets that must be statically analyzed. In the second phase, CloakX renames each WAR within the extension to a unique random value, finds all the references to the original name, and replaces them with their randomized counterpart. In the third phase, CloakX adds a dynamic proxy (Droxy) to the extension's content and background scripts. Droxy dynamically intercepts DOM and WAR requests and substitutes the original ID, class names, and WAR names with their random counterparts. In the last phase, CloakX statically analyzes and rewrites the DOM IDs and class names inside droplets that cannot be dynamically intercepted by Droxy.

3.1 XHOUND Analysis

CloakX uses XHOUND (we obtained a copy of the XHOUND prototype by contacting the paper's authors [42]) to generate a DOM fingerprint for the extension and to identify the droplets

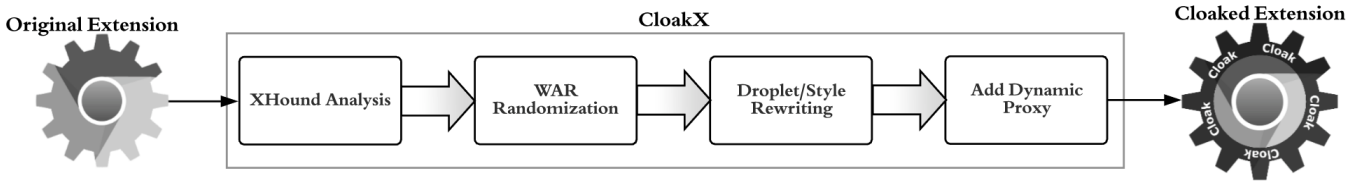


Figure 2: Overview of the CloakX process.

injected into the webpage. Each DOM fingerprint consists of four types of artifacts: (1) adding a new DOM element, (2) deleting a DOM element, (3) setting or altering an element's attribute, and (4) changing text on the page.

Of the four types, DOM additions are the most common type of detectable artifacts according to XHOUND [42]. This is because DOM additions are generic operations that often rely on loose coupling with a webpage for them to be triggered. Whereas most DOM modifications or deletions require a tighter coupling between the extension and the webpage, which limits their applicability to the problems often solved by developers. For instance, consider a password manager extension that injects a stylized element into every password form field (so that the user can invoke the password manager interface). The extension adds the element to the webpage and gives it a unique ID and a custom class name. It requires the ID to communicate with the element once it's placed on the webpage. Using the added ID and class name (i.e., the extension's anchorprint), a webpage can detect the extension by checking for the presence of either the unique ID or class name on the webpage.

Next, CloakX uses XHOUND to identify any droplets the extension injects into the webpage's execution environment so that CloakX can preprocess the droplets to identify the ID and class names within them. As discussed in Section 2.1, droplets (purple-colored boxes in Figure 1) are JavaScript strings that an extension injects directly into a rendered webpage. Droplets can include any text literal such as HTML, JavaScript, or base64-encoded images; however, the preprocessing is only performed on droplets containing inline JavaScript and those <script> elements that reference WARs.

Finally, during this phase, CloakX creates a map from the original ID and class names used to fingerprint the extension to the new randomized values.

3.2 Diversification of Web-Accessible Resources (WARs)

The principle behind the diversification of Web-Accessible Resources (WARs) is straightforward: if each installation of an extension has different filenames for the same WARs, then a tracker can no longer create a global database of WARs and, therefore, can no longer detect the presence or absence of any given extension based on its WAR anchorprint.

In the first stage of the WAR diversification process, CloakX identifies all the resources declared as WARs in the manifest file of each extension. Although many extensions explicitly list the resources they wish to make accessible, it is also possible to use a * wildcard [34]. With wildcards, an entire folder, its contents, and all its subfolders can be designated as web-accessible — this includes using a single *, which designates every file in the extension as web-accessible. Even though making every file in the extension web-accessible is likely an implementation error, we discovered 419 extensions that made all of their resources web-accessible, out of 59K analyzed extensions. In the second stage, CloakX computes the shortest unique file path to facilitate the search-and-replace in the final stage. Specifically, CloakX reduces the full path of each WAR to the minimum length necessary to uniquely identify the resource (compared to all the other resources in the extension). This operation reduces the number of resource references missed (i.e., false negatives) associated with dynamic string concatenation (often a directory path).

In the final stage of the WAR diversification process, CloakX uses the shortest unique path to find every use of the WAR within the extension's files and to replace that with the appropriate random value, maintaining the correctness of WAR references for each extension.

In addition to the static alterations described above, CloakX relies on Droxy, discussed in the Section 3.3, to dynamically translates any WAR requests missed by the static replacement method.

3.3 Droxy

The next step in the CloakX process adds Droxy to the extension. Droxy is a content script that injects random attributes and tags into the DOM to further obfuscate the extension's DOM fingerprint while also translating any uncloaked WAR requests and the IDs and class names used in DOM requests into their cloaked versions. CloakX patches Droxy into the extension and configures Droxy to execute before any of the extension's content scripts.

Droxy adds random attributes and tags to the DOM to reduce the accuracy of detection using structureprints. As each webpage is loaded, Droxy adds a random number of randomly generated tags to the DOM to make extension detection less accurate. To further frustrate detection using structureprint matching, Droxy adds random attributes to the DOM elements added by each extension.

Droxy also uses cross injection of custom attributes to frustrate anchorprint detection. For trackers using custom attributes to detect extensions, cross injection allows the user to impersonate other extensions, which increases a tracker's false positives when using anchorprint detection. This is done by adding custom attributes that are randomly selected from a list of the 244 unique custom attributes used by other extensions with a DOM fingerprint.

Droxy also dynamically catches any WAR requests made using the resource's original filename, which serves as a backup for the static replacement method described in Section 3.2. Droxy achieves this by watching for changes to the DOM using a MutationObserver() that checks for uncloaked WAR requests inside the DOM elements altered by the extension. In addition, Droxy overrides the XMLHttpRequest.open() method and adds functionality to translate any WAR requests for the original filename to the new, randomized filename.

Droxy translates the ID and class names used to create a DOM anchorprint. As the first content script to load, Droxy overrides DOM accessor and mutator methods before the extension uses them to interact with the DOM, which effectively wraps all DOM requests in a translation layer (blue area in Figure 3). Each of the overridden methods are augmented to intercept and translate ID and class names used to create the DOM fingerprint. Droxy determines which ID and class names to translate by checking the ID and class names against the cloaking map, created in Section 3.1. The cloaking map contains name–value pairs where each XHOUND-discovered ID and class name is paired with a randomized version. If it finds a match in the cloaking map, it translates the original value on-the-fly into the randomized version. By intercepting and translating the fingerprintable ID and class names to randomized values, Droxy alters the extension's DOM fingerprint from the perspective of a tracker's execution context breaking the link between the user's installed extension and the publicly available version.

To prevent the use of anchorprint detection, Droxy translates IDs and class names into random values according to the map created in Section 3.1. For ID and class name translation, Droxy also tracks DOM queries and DOM mutations. Droxy intercepts and inspects the extension's queries that use IDs, element names, class names, and query selectors, which include the methods getElementById(), getElementsByName(), getElementsByTagName(), getElementsByClassName(), querySelector(), and querySelectorAll(). To handle more complex query selectors, Droxy parses the selectors using the open-source Sizzle engine to accurately identify the ID and class names [9].

For DOM mutations performed via JavaScript, Droxy intercepts all the ways in which an ID or class name can be introduced to the DOM. This dynamic interception of ID and class names is done by overriding setAttribute() and getAttribute() methods and redefining id and className properties to use the overridden setAttribute() and getAttribute(). In addition, Droxy overrides the classList property. Because classList is an object, Droxy overrides the add(), contains(), and remove() methods of the classList. As a result, Droxy translates the extension's use of IDs and class names whether it is done when a DOM element is created or modified.

For DOM mutations performed via the injection of raw HTML, Droxy uses static and dynamic analysis to make the translation of ID and class names straight-forward and precise. Droxy overrides the methods used to inject raw HTML, such as the innerHTML property and insertAdjacentHTML() method. Droxy uses the browser to parse the HTML by creating a mock container and adding the HTML to it without attaching the mocked container to the DOM. Droxy queries the mock container to identify and transform the ID and class names into their randomized versions. Droxy then exports the string representation from the mock container's DOM and then calls the original method to apply the modified string to the webpage's DOM.

In addition to DOM queries and mutations, Droxy intercepts styles and translates on-the-fly. An extension can include styles via text content inside <style> or CSSStyleSheet's methods such as addRule() or insertRule(). Once intercepted, Droxy uses CSS parsing to locate the IDs and class names. If found, Droxy replaces the ID or class name with its randomized counterpart.

Droxy replaces an extension's droplets with the statically rewritten version (content_a.js and Dynamic JS in Figure 3). As a part of the droplet rewriting process described in Section 3.4, Droxy receives a hash value of the original droplet and modified version of the code for each droplet used by the extension. Droxy then matches the current droplet's hash to the ones provided and replaces it with its cloaked counterpart. Droxy performs the matching and replacement by customizing the properties textContent, innerText, and HTMLScriptElement's and the methods append() and appendChild() This process is depicted by the dashed arrow near ❶ in Figure 3. Droxy relies on the preprocessed JavaScript because rewriting the code on-the-fly in the browser efficiently is currently infeasible.

3.4 Static Droplet Rewriting

As discussed previously and shown in Figure 1, Droxy cannot intercept a droplet with dynamically inserted JavaScript because when the inserted code is executed in the webpage's JavaScript execution environment. Unfortunately, Droxy is also unable to cloak the droplet before inserting it because the heavy-weight static analysis necessary would significantly degrade the extension's performance. Therefore, CloakX statically analyzes the droplets offline, identifies where the extension adds the fingerprintable ID and class names to the

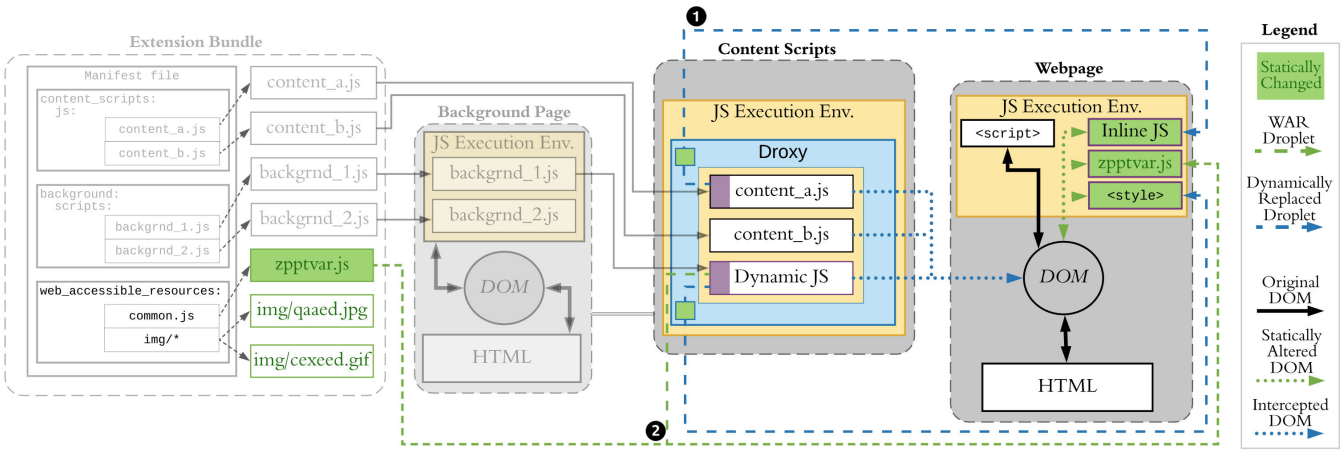


Figure 3: Diversified CloakX rewritten extension. CloakX hides fingerprints by rewriting the droplets, content styles, and renaming of web-accessible resources (WARs) and through Droxy's on-the-fly substitution. As a result, a tracking webpage cannot access the original identifiers; however, the internal logic of the extension still can because Droxy translates those requests.

DOM, rewrites the JavaScript code, and Droxy dynamically substitutes the original code with its rewritten counterpart.

Extensions commonly use *generic* values for IDs and class names, which often overlap with JavaScript keywords or JavaScript code constructs that refer to the class names and IDs dynamically. In addition, the expressiveness of JavaScript means that the ID and class names usage are context-sensitive. For example, if the fingerprintable class name is content, CloakX should only replace the instance of #content and ignore element.content, content.maximizer, and content-shaper, as each have a different semantic meaning. Developers often construct ID and class names dynamically in the code, which necessitates a more sophisticated form of static analysis. For example, an extension might attempt to access an element with the ID content by using getElementById("con" + "tent"), which would be missed by a regular expression searching for the full word.

CloakX statically rewrites droplets offline (i.e., before an extension is installed) using static analysis to identify the appropriate locations in the JavaScript. CloakX limits its rewrites to the ID and class names that occur in the JavaScript and are added to the webpage via the DOM. By identifying and only altering these DOM altering instances, CloakX limits the possibility of breaking the extension with the alterations. In essence, the static rewriting requires a tool that performs taint analysis where it labels DOM interactions as sinks and then analyzes the backward slices of the control flow graph (CFG) until it finds the fingerprintable IDs and class names as sources.

We decided to use TAJS — a state-of-the-art and feature rich JavaScript analyzer — as the program analysis core of the CloakX static rewriting. We chose TAJS because it (1) performs type analysis on JavaScript, (2) supports most of the ECMAScript 5 standard and DOM functionality, (3) is under active development, (4) is open source [6], and (5) is the product of recent research [13, 14, 15, 23, 24, 25, 27].

TAJS performs dataflow analysis by using techniques that examine the flow of data along program execution paths. As TAJS iterates over the CFG, it creates a semilattice of program states that are unique for each basic block in the CFG [26]. For each variable represented in the lattice at a given basic block, TAJS assigns a set of possible values. The dataflow analysis completes when the values inside the lattice reach a fixed point and no longer change with each iteration. Using these values, it is possible to follow data both forwards and backwards through the CFG [26].

3.4.1 TAJS for Extensions

We enhanced TAJS to support static rewriting of the droplets by adding support for Chrome extensions, adding DOM taint analysis, and maximizing its exploration of the CFG. In addition, we plan to make our changes to TAJS publicly available because there are currently no other program analysis tools for browser extensions.

We added extension support to TAJS by creating stubs for Chrome's extension API and implementing support for necessary methods such as sendMessage(), getURL(), executeScript(), onMessage.addListener() .

We implemented taint analysis within TAJS that tracks data through an application until it reaches a sink, where a sink is a location of interest within the CFG [16]. For the purposes of this analysis, TAJS tracks string literals matching the fingerprintable IDs and class names through the CFG until they are used to interact with the DOM. As a part of the taint tracking, we added functionality that maintains an *audit trail* of the changes to each variable while traversing the CFG so that upon reaching a sink CloakX can trace the values of interest to their origins.

We increased TAJS's code coverage by adding edges to the end of the CFG that force a call to every named and anonymous function defined within the code. For the purposes

of extension rewriting, it is necessary that TAJS analyzes all the JavaScript within a droplet because some functions appear unreachable without complete semantic understanding of Chrome's extension execution environment. However, the dynamic aspects used by TAJS itself to strike a balance between soundness and precision came at the cost of code coverage [15]. For example, TAJS does not analyze functions unless they are called by the JavaScript and the call is also reachable from the beginning of the CFG. Because extension rewriting requires TAJS to analyze all of the JavaScript within a droplet, we added edges to the end of the CFG that simulates a call to every named and anonymous function in the droplet. The potential downside to adding the edges is the decreased precision of our analysis (i.e., we are adding behavior to the application that does not exist at run-time), however for the purposes of identifying DOM fingerprints the trade-off is acceptable.

3.4.2 Static Analysis Results

Automated analysis of real-world JavaScript code is a difficult problem and despite all the advances made by TAJS, it, as well as similar tools, cannot analyze some JavaScript programs. As a JavaScript program increases in complexity and size, it becomes increasingly less likely TAJS will complete the analysis due to the explosion of dataflows (i.e., the classic state space explosion problem). As acknowledged by the authors, TAJS initially targeted hand-written JavaScript applications of a "few thousand lines of code" [26]. Plus, the addition of the fake edges dramatically increased the complexity of the CFG and the number of states, which decreased the code TAJS could successfully analyze to about 1,000 lines of code.

Fortunately, CloakX only needs TAJS analysis for the 197 extensions using droplets, which is only 3.2% of the extensions identifiable by XHOUND, because Droxy handles the rest of the extensions. Out of those 197 extensions, TAJS analyzed 212 total scripts of which 94 were JavaScript files that were designated as a WAR (and thus accessed via a src attribute, see Figure 3) and 118 were inline JavaScript. TAJS successfully completed analysis of 134 scripts (63.2%) finding 19,380 basic blocks and analyzing 18,497 (95.44%). However, TAJS was unable to analyze 78 (36.8%) of the inline JavaScript and WARs because the analysis for 34 scripts timed out, 6 scripts failed with an analysis exception, 6 scripts failed due to syntax errors in the JavaScript, and 32 scripts failed when TAJS crashed.

After manually analyzing the results we found the following reasons for why TAJS failed.

Exceeded timeout threshold. Most of the JavaScript code that caused TAJS to timeout were large JavaScript files that varied in size from 75 kilobytes to over a megabyte. In other cases, TAJS failed to finish analyzing smaller JavaScript code because of a bug in the forced path exploration code.

Analysis exceptions. TAJS failed to complete the analysis because it was missing support for the ECMAScript standard.

Syntax errors. TAJS was unable to analyze scripts with error in the JavaScript syntax.

Crashed. Some of the scripts triggered a bug in TAJS, causing it to crash with null pointer, stack overflow, or other miscellaneous exceptions.

3.5 Cloaked Extension

Once CloakX completes its modifications to the extension, the extension is cloaked and it appears to a webpage using anchorprint or structureprint detection techniques as though the user no longer has that particular extension installed. Architecturally, the resulting extension is similar to Figure 3 with Droxy surrounding the content scripts and translating the extension's DOM requests and droplet injections. To a webpage, the results look similar to the HTML source shown in Figure 4.

The permanence of the cloaked anchorprint and structureprint depends on whether the extension is subject to static rewriting. For cloaked extensions that rely on purely dynamic mutations, the structureprint changes each time the cloaked extension is loaded. Droxy alters the structureprint by injecting new randomly generated noise into the DOM and re-randomizing the cloaked ID and class names. However, CloakX must statically alter extensions with WARs or droplets. As a result, the cloaked fingerprint of extensions requiring static rewriting remains the same until a new version of the extension is reprocessed by CloakX. Although guessing the name of a cloaked WAR is unlikely because CloakX generates a random alphanumeric value that is at least ten characters in length for each WAR; even if an adversary guesses the name of a WAR, the detectability would cease when a new version of the extension was released.

3.6 Deployment

Although we describe CloakX as a client-side mechanism (as this is where the fingerprint rewriting is done), to reduce end-user friction, we envision CloakX as the final step in an extension's release and update process, all of which can be performed by the extension store and would require no intervention by the users. Prior to releasing the extension to users, the store sends the extension to CloakX for preprocessing. During CloakX's preprocessing, CloakX installs Droxy and generates a cloaking-template for the extension. The cloaking-template contains a configuration file that identifies the static variable replacements necessary for WARs, IDs, and class names. When a user requests a preprocessed extension, CloakX uses the cloaking-template to quickly generate and implement random WAR, IDs, and class names for the current user.

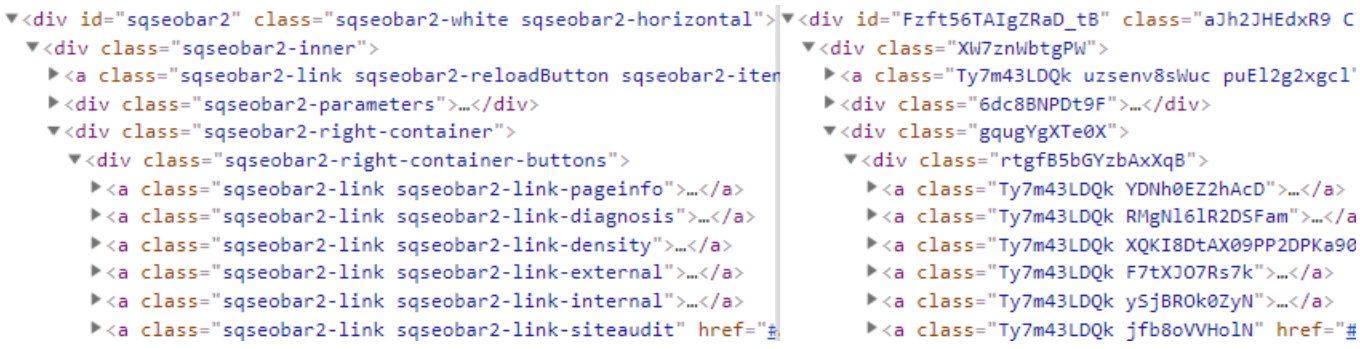


Figure 4: Original code of SEOquake extension (left) and SEOQuake extension when patched by CloakX (right).

4 Evaluation

Altering extensions without modifying the browser or relying on extension developers to make changes is a complex process, and while CloakX is a prototype and does not cover every possible scenario, we wanted to evaluate its current effectiveness. Thus, in this section we evaluate the efficacy of CloakX by (1) testing the breakage introduced by its use (2) the detectability of the cloaked extensions and (3) the performance of the cloaked extensions.

In November 2017, we extracted 59,255 extensions from the Chrome Store. Of those, we identified 13,693 extensions with only WAR fingerprints; however, 67 of the extensions had errors that prevented them from loading. Next, we identified 2,537 extensions having only DOM fingerprints, but Chrome could not load nine of the extensions. The last set of 2,786 extensions had both WAR and DOM fingerprints, one of which would not load in Chrome.

4.1 Functionality Experiments

Testing the functionality of a large set of applications is subject to two problems. First, the tests must explore all the relevant execution paths in the application. Second, the tests should test the entire set of applications. Furthermore, any testing approach will leave code unexplored and applications uncovered, and thus the results form an estimation of functionality breakage. In this work, we perform two different experiments to address both of these challenges: a low-fidelity and a high-fidelity experiment.

The low-fidelity experiment tested the entire population and the high-fidelity experiment randomly sampled from the population. The low-fidelity experiment automatically exercised the original and cloaked extensions and compared the error messages generated by each. The low-fidelity experiment provides a lower bound on the breakage across the entire population. The high-fidelity experiment involved manually — and extensively — exercising the extension, which provided deeper coverage of the extension's functionality. Due to the time-consuming nature of each high-fidelity run, we used a random sample of the extensions from each population.

4.1.1 Low-fidelity Functionality Experiments

To measure functionality breakage introduced by CloakX broadly across all extensions, we performed automated experiments that measured the change in errors from the original extension to the cloaked extension. To execute the experiment, we created a headless browser session using Selenium's ChromeDriver with full logging enabled, which includes errors from the extension's content scripts. Next, we visited a triggering web page, which is similar to the webpage used by XHOUND to activate the extension's functionality. In addition, for those extensions with DOM fingerprints identified by XHOUND, the triggering webpage also included dynamically generated triggers. After the page loaded, the browser waited 30 seconds for any delayed actions to execute. Other than the static and dynamic triggers, the automated experiments do not simulate additional user actions, which might be necessary to execute all the extension's functionality. These steps comprise a *run*, which is completed once for the original extension and once for the cloaked extension.

After both runs finish, we compared the severe JavaScript error messages between the two runs. If the cloaked extension generated the same errors, then the extension passed. Otherwise, if the cloaked extension generated any new or different errors, then the extension failed. Because the automated tests exercise limited functionality and only compare errors, this experiment represents the best case scenario (i.e., the lower bound) on the errors introduced by CloakX. However, the automation allowed us to run the experiment across the entire population.

Table 1 shows the results for WAR and DOM cloaking separately. Note that at the time we ran the experiments, which took place several months after collecting the extensions, some of the original versions stopped working because of Chrome browser updates, obsolete back-end servers, etc. As a result, we only tested working extensions and, therefore, the results only contain errors introduced by CloakX.

In the low-fidelity experiments, CloakX retained equivalent functionality for 99.02% (13,493) of the WAR fingerprintable extensions, 98.69% (2,493) of DOM fingerprintable extensions, and 97.92% (2,727) of WAR and DOM fingerprintable

Table 1: Automated Test Results

Extension set	Total	Tested	Passed	Results	
				Pass	*Fail*
WAR Fingerprintable	13,693	13,626	13,493	99.02%	.98%
DOM Fingerprintable	2,537	2,526	2,493	98.69%	1.31%
WAR & DOM Fingerprintable	2,786	2,785	2,727	97.92%	2.08%
Totals	**19,016**	**18,937**	**18,713**	**98.82%**	1.18%

Table 2: Manual Test Results

Extension set	Random	Top 25	Overall
	Pass/Fail	*Pass/Fail*	*Pass/Fail*
WAR Fingerprintable	25 / 0	25 / 0	50 / 0
DOM Fingerprintable	24 / 1	24 / 1	48 / 2
WAR & DOM Fingerprintable	24 / 1	24 / 2	47 / 3

extensions. For the WAR fingerprintable extensions, we found that the most frequent cause of the failures was the loading of WARs from remote websites. For the DOM fingerprintable extensions, most of the new error messages generated by the cloaked extensions were severe JavaScript errors caused by (1) extensions loading remote content or (2) missing functionality in Droxy. For the WAR and DOM fingerprintable extensions, we found the same errors as seen in the WAR and DOM only tests. To verify the WAR and DOM cloaking did not interfere with one another, we also ran this group using only one of the modifications at a time. The total number of errors was the same for the joint run as it was for the two additional runs with the single modifications, which indicates the modifications did not interfere with one another.

4.1.2 High-fidelity Functionality Experiments

The high-fidelity experiments consisted of manually exercising and evaluating the operation of the cloaked extensions. The high-fidelity evaluation was inspired by the methodology used by Snyder et al. [39]. This methodology focuses on the extension's operation from the perspective of the user. If the cloaking process introduces an error, but the user does not perceive a difference in the extension's operations, then we deem the extension passes. This method of evaluation exercises much more of the extension's code than the automated tests and it provides an additional metric that evaluates the actual operation of each extension. The high-fidelity experiments were performed by the authors using the testing framework detailed next.

We built a custom framework to methodically follow a four-phase evaluation of each extension and advise the tester on the current step in the process. In phase one, the framework loads the original extension and gives the user five minutes to understand its basic operation (including the time necessary to read the extension's description in the Chrome Store). In phase two, the framework reloads the original extension and the user exercises its functionality for five minutes. In phase three, the framework loads the modified extension and the user spends five minutes completing operations similar to the ones completed in phase two to verify it is still operational. In the last phase, the user records any notes on the evaluation and chooses whether the extension passed or failed.

Similar to the automated tests, we divided the extensions into three groups based on the type of fingerprints they emitted. As a result, the populations for each of the high-fidelity tests were as follows: 13,626 WAR fingerprintable extensions, 2,526 DOM fingerprintable extensions, and 2,727 WAR and DOM fingerprintable extensions.

To create samples for these groups, we created both random and systematic samples containing 25 extensions each. We created the first sample by randomly selecting 25 extensions from the population. We formed the systematic sample by selecting the top 25 most popular extensions based on the number of downloads listed on the Chrome Web Store. Throughout the manual tests, if we could not test an extension because the original version was broken or it was only available in a foreign language, then it was discarded and another one was selected according to the associated sampling method. The resulting samples contained quite a bit of diversity between the extensions. Although we found a few instances of overlapping functionality, we kept these extensions in the samples. However, when we found a duplicate extension, we discarded the duplicate and tested a different extension. Some example extensions included in the test samples included a utility for those who are color blind, a search bar tool, a product search by image, a data extraction tool, and a gesture utility for navigation.

Out of all 150 experiments, 145 of the cloaked extensions retained equivalent functionality (see Table 2). All of the WAR fingerprintable extensions retained their functionality. 96% (48 out of 50 extensions) of the DOM fingerprintable extensions and 94% (47 out of 50 extensions) of the WAR and DOM fingerprintable extensions retained their functionality.

After analyzing the broken extensions, we found three different causes for the broken extensions.

Remote source code using original resource name. The extension loads remote Facebook SDK, which looks for obfuscated ID and class values.

Extension relies on hardcoded values that Droxy alters. An extension relies on hardcoded logic that expects its content scripts to appear in a specific order. However, Droxy must be the first content script, which changes the position of all of the extension's original content scripts, and in one case, it broke the extension.

Droxy implementation limitation. Droxy does not currently support recursive iframe sourcing, *cloneNode*, and some advanced CSS rules that the `cssutils` Python library fails to properly parse.

With engineering improvements to Droxy, we can remediate each of the errors listed above and increase the success rate. For the remote source code, Droxy could intercept the remote source code request and parse it before it is executed. This, of course, would add additional performance overhead. The

hardcoded logic could be rectified by overriding the methods that accesses the content scripts. The implementation limitations can be addressed by adding logic to support them into Droxy.

4.2 Detectability Experiments

The detectability experiments evaluated the efficacy of the cloaking against an extension tracking webpage. In the first experiment, the tracker used anchorprints to detect extensions with either WAR or DOM fingerprints. In the second experiment, the tracker used structureprints to detect the extensions with DOM fingerprints. In the third experiment, we investigated the use of behaviorprints to detect cloaked extensions. Last, we explored different methods for detecting the use of CloakX on an extension.

For the first three experiments, we set the fingerprint matching threshold to three. To meet the matching threshold, the tracker must be able to match the extension's fingerprint to three or fewer extensions in its repository. When the tracker meets the matching threshold, it has successfully detected the extension.

We chose a threshold of three because thresholds higher than three showed a sharp decrease in the tracking benefit gained from an extension detection. The matching threshold represents the number of extensions that match a structureprint. The best threshold depends on the requirements of the web tracker and the resources available. The main purpose of the threshold for our experiments was to balance the search time complexity of the fuzzy searches with the increase in the matching of cloaked extensions. For example, by raising the threshold to 20, the web tracker matches three additional cloaked structureprints (one of which matches 18 extensions).

4.2.1 Detectability Experiment Using Anchorprints

The anchorprint detectability experiments focused on detection using WARs, IDs, and class names. In the first phase of the experiment, we harvested the anchorprints of the extensions. Next, we loaded each of the original extensions and used a tracking webpage to verify that the extensions were detectable using the anchorprint. Finally, we loaded each of the cloaked extensions and used a tracking webpage to evaluate the detectability of the cloaked extensions using its anchorprint. For a successful detection, the tracker must meet the matching threshold.

In our experiment, we found that none of the cloaked extensions were detectable using their WARs, IDs, and class names after cloaking. In the first phase, we harvested 17,833 anchorprints, which includes 16,411 extensions with WAR fingerprints and 1,422 that have DOM fingerprints with IDs and classes. However, we chose to limit the testing to the 17,678 extensions that could be executed after being cloaked and assumed that the 155 broken extensions were detectable (thus providing a lower bound on detectability).

In the second phase, we matched 17,534 of the 17,678 original extensions. The ID and class name functionality of the tracker failed to match 144 extensions because it either failed to trigger the extension's anchorprint or it found too many matching extensions. The ID and class name tracker did not find matches for 26 extensions because those extensions required dynamic triggering and the tracker could not use dynamic triggering and still extract the anchorprint; thus, the extensions did not inject their anchorprint into the webpage. The remaining 118 extensions did not count as a detection because the IDs and class names matched more than three other extensions, which exceeded our threshold for a detection.

Initially, the WAR functionality of the tracker failed to find 956 of the WAR fingerprinted extensions using XMLHttpRequest because none of the WAR declarations in the manifest file existed in the extension. However, we discovered we could reliably match these extensions by timing how long it took for three WAR requests to return. The first request is for the declared but missing resources of the extension. The second request was for the extension's manifest.json, which was not declared as a WAR. The third request was for a randomly generated resource that does not exist in the extension and is not a WAR. If the missing request (i.e., the first) takes the longest to return, then the extension has the resource defined as a WAR but the resource does not exist in the extension. Thus, we improved the tracker such that if the tracker failed to match an extension using any of the WARs, then it performs these three requests for each of the WARs in the 956 extensions and if the first request takes the longest it has detected the extension.

In the third phase, we were able to detect 96 of the cloaked extensions using their anchorprints. After investigating several extensions that were detected, we found that matches occurred because CloakX was not translating the ID and classes for the extensions due to errors introduced through the cloaking process. In other words, the experiment found 96 additional cloaked extensions that did not maintain functionality equivalent to their original versions. Thus, with the additional errors but no actual matches, we found that 98.55% (17,582) of the extensions were undetectable using anchorprints.

4.2.2 Detectability Experiment Using Structureprints

The structureprint experiment tested the detectability of cloaked extensions using exact and fuzzy matching to detect the extensions. In the first phase, we ran each of the 5,311 DOM fingerprintable and WAR and DOM fingerprintable extensions through XHOUND to gather the structureprints. In the next phase, we ran each of the 5,223 cloaked extensions through XHOUND to gather cloaked fingerprints. We considered the extensions that failed the automated tests as detectable. Similar to the WAR detection experiments, we

Table 3: Structureprint Detection Test Results

Structureprint Key Type	Exact Matching		Fuzzy Matching
	Original	*Cloaked*	*Cloaked*
Tags, Attributes, Text	3,756 (71.91%)	91 (1.74%)	217 (4.15%)
Tags and Attribute Values	2,092 (40.05%)	91 (1.74%)	95 (1.82%)
Tags	1,420 (27.19%)	91 (1.74%)	91 (1.74%)

did not test the broken extensions, but we assume that they were detectable. In the last phase, we used the structureprints generated in phase one to match the cloaked fingerprints.

The accuracy and precision of detecting structureprints varies depending on both (1) the DOM elements used to build the structureprint and (2) the matching technique used to identify the extension. Therefore, to explore how CloakX can prevent the detection of various types of structureprints, we ran the last phase several times using three different structureprints (each one representing less information used in the structureprint) and two different matching techniques (one on exact matching and one on fuzzy matching) to ensure CloakX reduced detection for each of them.

The structureprints varied based on the contents used to build the fingerprint. The first type used all the XHOUND data, in other words, each fingerprint included added and changed tags, attribute names, attribute values, and text data. While these are the most accurate, they are also the most brittle; as a result, it is likely that the accuracy will degrade considerably in a real-world environment with dynamic HTML content and visitors that have several extensions installed. The second type of structureprint used only the tags and attribute names, which means the fingerprint did not use the attributes values or text. The third type of structureprint used only the tags.

For detection, the experiment extracted an extension's structureprint and then used exact and fuzzy matching against the structureprint database to identify the extension. Exact matching worked well for detecting uncloaked extensions; however, due to the preciseness required for an exact match, cloaked extensions evaded exact matching. Thus, we also tested using fuzzy matching with a 90% level of confidence. Fuzzy matching was successful when the match was made with a 90% level of confidence. Using either matching technique, if the tracker met the matching threshold (three or fewer matches) using the extension's structureprint then we counted the extension as detected.

Overall, we found that cloaking significantly limited the number of extensions detectable using structureprints. With the full structureprints (tags, attribute names, attribute values, and text) and exact matching, we were able to detect 3,756 of the 5,311 original extensions. The reason that 1,555 extensions were undetectable is because the number of matches made using the extension's structureprint exceeded the matching threshold for a detection (a structureprint must match three or fewer extensions for a successful detection). Using the full structureprints on cloaked extensions, none of the cloaked extensions were detected using exact matching and only 126 extensions were detected using fuzzy matching. Using partial structureprints (attributes and tags), we were able to detect 2,092 of the original extensions; however, the cloaked extensions were undetectable using exact matching and only four were detectable using fuzzy matching. Using the tag only structureprints, we detected 1,420 of the original extensions; however, we were unable to detect any of the cloaked extensions using either matching technique.

4.2.3 Detectability Experiment Using Behaviorprints

To understand the limitations of CloakX, we performed an experiment to test the detectability of cloaked extensions using behaviorprints. We chose ten of the most popular extensions with structureprints and to avoid duplication we excluded all ad-blocking extensions except AdBlock. In addition, we examined ten extensions that we randomly selected from those with structureprints. By analyzing their structureprints, we manually created their behaviorprints from portions of the structureprint that remain constant after cloaking.

For the popular extension sample, six of the extensions added elements to the DOM that made them uniquely identifiable. The extensions LastPass, Pinterest Save Button, and Grammarly all add a base64 encoded image to the DOM that makes them uniquely identifiable. The extensions Ghostery, Evernote, and Skype add a `style` tag to the `head` element with features that made them uniquely identifiable. The extension Turn Off the Lights adds a `data-video` attribute. Although the `data-video` attribute is detectable when the extension is cloaked, CloakX randomly includes this attribute even when the extension is not installed, which increases the attacker's false positive rate and makes it more difficult to correctly detect when the extension is truly installed. Even though the cloaked version of AdBlock was detectable, its behaviorprint was not distinguishable from other popular ad-blocking extensions (e.g., AdBlock Plus, uBlock Origin, and AdGuard AdBlocker) because they all perform the same behavior by deleting ads from the DOM and not injecting any other elements into the DOM. Thus, the detection of ad-blocking extensions exceeds the matching threshold for the identification of a user. Ace Script and Honey added `div` tags with an ID, which means CloakX obfuscated the behaviorprint, and the extensions were not detectable.

For the random sample of ten extensions, five extensions were detectable using behaviorprints and five were undetectable. Similar to popular extensions, five of the ten extensions added elements to the DOM that made them uniquely identifiable. For example, two of them added custom text to the web page. Two of the undetectable extensions performed actions on the DOM, which were duplicated by a number of other extensions. Thus, those extensions exceeded the matching threshold and were undetectable. Finally, the three remaining undetectable extensions only added class names, IDs, and common tags to the DOM, which are obfuscated by CloakX.

4.3 Detectability of CloakX

For our last set of experiments, we evaluated three different techniques meant to determine whether an extension was cloaked by CloakX, thus detecting CloakX. These detection experiments were limited to the 2,447 extensions with structureprints that contained at least one ID or class name.

In the first experiment, we created a method for detecting CloakX after analyzing the lengths of the IDs and class names in cloaked and uncloaked extensions. The IDs and class names generated by CloakX were initially six characters in length and comprised of random alphanumeric characters. However, the IDs and class names in uncloaked extensions averaged 15.1 characters for IDs and 15.9 characters for class names. In addition, we discovered that only 62 uncloaked extensions met the criteria of having all their IDs and class names with a length of six (most of which had only one ID or class name). As a result, we created a method for detecting CloakX that marked an extension as CloakX-enabled if all the IDs and class names injected by the extension were six characters in length. Next, we ran the CloakX detector on 2,447 uncloaked extensions and cloaked extensions, thus evaluating 4,894 extensions. The CloakX detector reported 2,509 cloaked extensions, which means we had 62 false positives, 2,447 true positives, and 2,385 true negatives with an accuracy of 98.7%. As a result, we modified CloakX to randomize the length of the IDs and class names it renames. After making this change, we were no longer able to accurately detect the existence of CloakX based on the length of IDs and class names.

As a follow up, the second experiment attempted to identify cloaked extensions by measuring the entropy of the injected ID and class names. Our hypothesis was that the entropy of randomly generated IDs and class names would be measurably different from those chosen by extension developers. We found that the uncloaked ID and class names exhibited nearly the same amount of entropy as their randomized counterparts. As a result, we could not accurately identify the existence of CloakX using the entropy of ID and class names even though the cloaked values contained randomly generated characters.

In the last CloakX detection experiment, we identified the use of CloakX by exploiting popular extensions that both exhibited a behaviorprint and injected an ID or class name into the webpage. In particular, we found Evernote and Grammarly offered a strong behaviorprint and a related ID. Once we identified the existence of the extension's behaviorprint we looked for the ID or class name, if it did not exist then we determined CloakX was likely installed. For instance, Evernote injects a style tag with unique elements and it uses an ID for the same style tag. When a style tag is found that contains Evernote's elements and the style's ID is not style-1-cropbar-clipper, then the tracker records that it found a cloaked version of Evernote. Similarly, when Grammarly's green icon is detected and the top level html tag does not contain a class starting with gr, the tracker records that it found a cloaked version of Grammarly. We tested this by running the tracker against all 2,447 uncloaked extensions and the two cloaked versions of Evernote and Grammarly. The tracker accurately identified the cloaked versions of both extensions with zero false positives.

4.4 Performance Experiments

CloakX minimally impacts the performance of Chrome in our automated tests. We tested CloakX's performance by randomly selecting 500 extensions that contain structureprints because their cloaking requires more resources. Each individual test loaded Chrome, loaded the extension, and ran a triggering webpage from the local machine, which either triggered a page load event or timed out. We executed the tests ten times on both the original and modified extensions. The tests were performed across 16 cores with each core running at 2.2 Ghz. On average, the original extensions took 12.3128 seconds and used 66,790 KB of memory whereas the modified extensions took 12.3221 seconds and used 67,123 KB of memory. Thus, the average increase in overhead for the cloaked extensions was a .07% increase in execution time (0.0093 seconds per extension) and a .49% increase in memory use (333 KB per extension).

5 Discussion

Using the highest failure rate for each of the fingerprint types and using fuzzy matching, CloakX retained the functionality and hid from detection 96.23% (18,222) of the tested extensions. For anchorprint detectable extensions, CloakX rendered 98.55% (17,574) of the extensions undetectable and with equivalent functionality. For structureprint detectable extensions, the tracker was unable to detect 95.91% (5,094) of the cloaked extensions.

CloakX rendered the detection of extensions using anchorprints significantly less accurate. CloakX increases user's anonymity by diversifying WARs, IDs, class names, and custom attributes used for anchorprints. In our experiments, cloaking the WARs, IDs, and class names destroyed the link between the published extension and the currently installed version. As a result, none of the successfully cloaked extensions could be detected based solely on their WAR, ID, or class name. Although it is possible to cloak custom attributes in a similar fashion, CloakX uses cross extension injection of custom attributes to cloak extensions. For trackers using custom attributes to perform anchorprint detection, CloakX increases the number of matches the tracker makes when evaluating an extension's anchorprint, which causes it to exceed the matching threshold and, thus, not detect the extension.

For detection using structureprints, CloakX obfuscated 95.91% (5,094) of the previously detectable extensions even when fuzzy matching with 90% level of confidence was used.

To prevent structureprint matching, CloakX diversifies the tags and attributes added to the DOM by the extension. While these changes were effective against exact and fuzzy matching, the changes only obfuscate the structureprint. Therefore, it is possible that a tracker could create a more sophisticated matching process (as has been the case in fingerprinting attacks and countermeasures) that limits the search to those DOM modifications that are constant.

For example, cloaked extensions are still sometimes identifiable with behaviorprints. In our experiments with twenty extensions, we were able to manually create unique behaviorprints for eleven of the twenty cloaked extensions.

However, behaviorprinting does not currently scale. First, the creation of behaviorprints requires human intelligence and no recent research has shown how to automatically generate a behaviorprint. Second, consistent human intervention is required to prevent the behaviorprints from going stale and no longer being able to identify the extension. For example, LastPass could update their icon, which would no longer match the saved behaviorprint. Third, due to the dynamic nature of the web ecosystem many of the behaviorprints will likely be difficult to use in practice. Lastly, the more popular an extension is the less value the detection of that extension offers towards the goal of identifying users. As a result, for a tracking website to effectively utilize behaviorprints they need to obtain a large number of behaviorprints from both popular and less popular extensions, which exacerbates the scaling problems.

Protection from behaviorprints is a fundamentally difficult problem because the extensions and the browser share the same view of the DOM. CloakX provides some protection from behaviorprints through its injection of noise into the DOM and with additional features could provide even more protection against behaviorprinting. For example, CloakX can make user identification via behaviorprints even more difficult by adding a feature that randomly injects the behaviorprints of the popular extensions, which increases the false positive detections and further dilutes the user's fingerprint. In addition, it is important to point out that only 3,756 extensions (of the 59,255 extensions we used in our study) have unique enough changes to the DOM to form behaviorprints and many of those are not unique enough to provide a robust means of detecting extensions. Nevertheless, the more complete privacy solution for extension fingerprinting is to modify the browser so that the extension and the website JavaScript see their own views of the DOM.

Although we were able to detect the use of CloakX, detecting the presence of a defense mechanism, like CloakX, is different than defending what the mechanism is explicitly trying to protect against (i.e., the presence of specific browser extensions). The fingerprinting value realized by detecting an extension cloaked with CloakX diminishes with each user that uses cloaked extensions. However, it is unlikely any attackers will try to detect cloaked extensions until CloakX becomes popular enough to warrant the attention. As a result, the fingerprint value of detecting CloakX is limited. However, as CloakX becomes more popular it is possible that malicious or shady websites could deny service to users with CloakX-enabled extensions, but this issue exists with any defensive mechanism (similar to what users experience with ad-blocking extension detection).

Thus, despite the limitations described above, CloakX takes a large step forward towards protecting users from wide-spread automated fingerprinting using anchorprints and structureprints.

5.1 Case Study of Failures

Despite their large size and complexity, CloakX cloaks and retains equivalent functionality of 97.88% of the extensions detectable through their anchorprints and structureprints.

Functional breakage caused by the remote loading of scripts was common in broken extensions. One approach to address this issue is to find droplets that load remote scripts which CloakX could download, cloak, and save inside the extension. While an improvement over our current CloakX prototype, the downside of this approach is that the remote scripts might be dynamically generated and copying them inside the extension would not solve the problem. At a high level, we consider the loading of remote code in browser extensions an open problem because remote code can drastically alter the extension's logic *after* that extension has been vetted by the extension store.

Droxy relies on hash values calculated from a static version of the JavaScript code; however, some extensions dynamically change their inline JavaScript code each time it is produced. As a result, Droxy was unable to find the inline code because the current script would not match the one stored in CloakX's metadata. In most cases, we observed that the differences between the original and live scripts were minor which suggests that alternative search routines that allow for fuzzy-matching would be able to handle most of the observed code-matching issues, such as the one used by Soni et al. [40].

5.2 A New Avenue of Security Exploration

Due to the often-misaligned incentives between extension developers and end users, it is desirable to be able to perform late-stage customizations of browser extensions not only to make extensions less fingerprintable, but to also improve their overall security and privacy. In this work, we showed that despite the complexity of the rewriting process, we were able to automatically modify extensions, without requiring browser changes or changes to the development process of browser extensions. Therefore, our approach could be used in additional contexts, such as removing unnecessary third-party trackers and PII leaks [41, 46] or automatically patching vulnerabilities discovered in browser extensions [17].

6 Related Work

To the best of our knowledge this paper proposes the first client-side countermeasure against the fingerprinting of browser extensions. In this section, we briefly describe prior work on generic browser fingerprinting and the related countermeasures.

Eckersley conducted the first large-scale study that showed browser fingerprinting was sufficient to uniquely identify users without cookies or other stateful identifiers [18]. Since then, researchers studied several related topics including tracking the adoption of fingerprinting in the wild [11, 12, 19, 31, 36], proposing new vectors for browser fingerprinting [32, 33, 37, 38, 42, 45], and describing potential defenses against it [29, 30, 35].

Of all the new vectors proposed for browser fingerprinting, in 2017, researchers discovered three different types of side-channels for detecting the presence of specific browser extensions. Sjösten et al. used WARs to determine whether a browser extension is installed [38]. Using this method, they found unique fingerprints for 12,154 extensions and more than 50% of the 1,000 most popular extensions. With the fingerprint, extension detection is straightforward for an attacker to execute (one check per extension) and works even if the user utilizes incognito mode. To defend against this attack, CloakX dynamically renames all of an extension's WARs and rewrites all references to these WARs from the extension's code. As such, every different installation of the same cloaked extension will now have different WARs.

Starov and Nikiforakis utilized the changes in a webpage's DOM to detect extensions. Similar to the WAR detection technique, the attacker pre-processes all the extensions of interest to extract the DOM fingerprints that can be later used to detect the extension's presence [42]. As with WARs, CloakX dynamically rewrites the IDs and class names of all injected DOM elements, which changes the extension's fingerprint and makes it undetectable.

The last browser extension fingerprinting technique, proposed by Iskander-Rola et al. [37] relies on timing channels to detect the presence of files associated with a browser extension. Their method works regardless of whether the extensions declares the files as web accessible. Similarly, Van Goethem and Joosen propose a variation of the same technique using different timing side channels [45]. Because these attacks abuse the access-control mechanisms of a browser, no amount of extension rewriting can counter them. As such, we consider these attacks as out-of-scope for CloakX because our goal is to counteract the detection techniques without modifying the browser.

7 Conclusion

In this paper, we presented the first client-side countermeasure for defending against the detection of browser extensions. Our system, CloakX, uses the principle of diversification so that two installations of the same extension expose different fingerprintable attributes. CloakX operates in an extension-agnostic fashion by rewriting extensions on the client-side, without requiring any modifications to the web browser. Overall, through a combination of large-scale experiments and manual testing, we showed that our CloakX prototype can successfully handle the majority of browser extensions while causing minimal breakage.

Acknowledgements: We thank the anonymous reviewers for their helpful feedback. This work was supported by the Office of Naval Research (ONR) under grant N00014-17-1-2541, as well as by the National Science Foundation (NSF) under grants CNS-1527086, CNS-1617593 and CNS-1703375.

References

[1] Automatically find and apply coupons. https://chrome.google.com/webstore/detail/honey/bmnlcjabgnpnenekpadlanbbkooimhnj.

[2] Chrome.runtime - getbackgroundpage(). https://developer.chrome.com/extensions/runtime#method-getBackgroundPage.

[3] Content scripts. https://developer.chrome.com/extensions/content_scripts.

[4] Detect adblock – most effective way to detect ad blockers. https://www.detectadblock.com/.

[5] Extension overview. https://developer.chrome.com/extensions/overview.

[6] Github - tajs. http://nicolas.golubovic.net/thesis/master.pdf.

[7] Manifest - web accessible resources. https://developer.chrome.com/extensions/manifest/web_accessible_resources.

[8] Manifest version. https://developer.chrome.com/extensions/manifestVersion.

[9] Sizzle javascript selector. https://sizzlejs.com/.

[10] W3 dom overview. https://www.w3.org/TR/DOM-Level-2-Core/introduction.html.

[11] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz. The Web Never Forgets: Persistent Tracking Mechanisms in the Wild. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, 2014.

[12] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel. FPDetective: Dusting the Web for fingerprinters. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, 2013.

[13] E. Andreasen, A. Feldthaus, S. H. Jensen, C. S. Jensen, P. A. Jonsson, M. Madsen, and A. Møller. Improving tools for javascript programmers. In *Proc. of International Workshop on Scripts to Programs. Beijing, China:[sn]*, pages 67–82, 2012.

[14] E. Andreasen and A. Møller. Determinacy in static analysis for jQuery. *ACM SIGPLAN Notices*, 49(10):17–31, 2014.

[15] E. S. Andreasen, A. Møller, and B. B. Nielsen. Systematic Approaches for Increasing Soundness and Precision of Static Analyzers. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, (June), 2017.

[16] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[17] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *Network and Distributed System Security Symposium (NDSS)*. Citeseer, 2010.

[18] P. Eckersley. How Unique Is Your Browser? In *Proceedings of the 10th Privacy Enhancing Technologies Symposium (PETS)*, pages 1–18, 2010.

[19] S. Englehardt and A. Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1388–1401. ACM, 2016.

[20] Google Chrome Extension. Trump Filter. https://chrome.google.com/webstore/detail/trump-filter/lhondapiaknegjpellpodegmeonigjic.

[21] Google Chrome Extensioon. Hillary Blocker. https://chrome.google.com/webstore/detail/hillary-blocker/kiblhkcoiojbdhhnjaekompfecgelfja.

[22] N. Jagpal, E. Dingle, J.-P. Gravel, P. Mavrommatis, N. Provos, M. A. Rajab, and K. Thomas. Trends and lessons from three years fighting malicious extensions. In *24th USENIX Security Symposium*, 2015.

[23] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the eval that men do. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 34–44. ACM, 2012.

[24] S. H. Jensen, P. a. Jonsson, and A. Møller. Remedying the Eval That Men Do. *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 34–44, 2012.

[25] S. H. Jensen, M. Madsen, and A. Møller. Modeling the html dom and browser api in static analysis of javascript web applications. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 59–69. ACM, 2011.

[26] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for javascript. In *International Static Analysis Symposium*, pages 238–255. Springer, 2009.

[27] S. H. Jensen, A. Møller, and P. Thiemann. Interprocedural analysis with lazy propagation. In *International Static Analysis Symposium*, pages 320–339. Springer, 2010.

[28] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 641–654, San Diego, CA, Aug. 2014. USENIX Association.

[29] P. Laperdrix, B. Baudry, and V. Mishra. Fprandom: Randomizing core browser objects to break advanced device fingerprinting techniques. In *International Symposium on Engineering Secure Software and Systems*, pages 97–114. Springer, 2017.

[30] P. Laperdrix, W. Rudametkin, and B. Baudry. Mitigating browser fingerprint tracking: multi-level reconfiguration and diversification. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 98–108. IEEE Press, 2015.

[31] P. Laperdrix, W. Rudametkin, and B. Baudry. Beauty and the Beast: Diverting modern web browsers to build unique browser fingerprints. In *37th IEEE Symposium on Security and Privacy (S&P 2016)*, San Jose, United States, May 2016.

[32] K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham. Fingerprinting information in javascript implementations. In *Proceedings of W2SP*, volume 2, 2011.

[33] K. Mowery and H. Shacham. Pixel perfect: Fingerprinting canvas in html5. *Proceedings of W2SP*, pages 1–12, 2012.

[34] Nicolas Golubovic. Attacking Browser Extensions, MS Thesis, Ruhr-University Bochum. http://nicolas.golubovic.net/thesis/master.pdf, 2016.

[35] N. Nikiforakis, W. Joosen, and B. Livshits. Privaricator: Deceiving fingerprinters with little white lies. In *Proceedings of the 24th International Conference on World Wide Web*, pages 820–830. International World Wide Web Conferences Steering Committee, 2015.

[36] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *Security and privacy (SP), 2013 IEEE symposium on*. IEEE, 2013.

[37] I. Sanchez-Rola, I. Santos, and D. Balzarotti. Extension breakdown: Security analysis of browsers extension resources control policies. In *26th USENIX Security Symposium*, 2017.

[38] A. Sjösten, S. Van Acker, and A. Sabelfeld. Discovering browser extensions via web accessible resources. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 329–336. ACM, 2017.

[39] P. Snyder, C. Taylor, and C. Kanich. Most websites don't need to vibrate: A cost-benefit approach to improving browser security. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 179–194. ACM, 2017.

[40] P. Soni, E. Budianto, and P. Saxena. The sicilian defense: Signature-based whitelisting of web javascript. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1542–1557. ACM, 2015.

[41] O. Starov and N. Nikiforakis. Extended tracking powers: Measuring the privacy diffusion enabled by browser extensions. In *Proceedings of the 26th International Conference on World Wide Web*, pages 1481–1490. International World Wide Web Conferences Steering Committee, 2017.

[42] O. Starov and N. Nikiforakis. XHOUND: Quantifying the fingerprintability of browser extensions. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 941–956. IEEE, 2017.

[43] M. Stockley. The web attacks that refuse to die. https://nakedsecurity.sophos.com/2016/06/15/the-web-attacks-that-refuse-to-die/.

[44] K. Thomas, E. Bursztein, C. Grier, G. Ho, N. Jagpal, A. Kapravelos, D. McCoy, A. Nappa, V. Paxson, P. Pearce, et al. Ad injection at scale: Assessing deceptive advertisement modifications. In *IEEE Symposium on Security and Privacy (SP)*, 2015.

[45] T. Van Goethem and W. Joosen. One side-channel to bring them all and in the darkness bind them: Associating isolated browsing sessions.

[46] M. Weissbacher, E. Mariconti, G. Suarez-Tangil, G. Stringhini, W. Robertson, and E. Kirda. Ex-ray: Detection of history-leaking browser extensions. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 590–602. ACM, 2017.

[47] X. Xing, W. Meng, B. Lee, U. Weinsberg, A. Sheth, R. Perdisci, and W. Lee. Understanding malvertising through ad-injecting browser extensions. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, pages 1286–1295, 2015.

Less is More: Quantifying the Security Benefits of Debloating Web Applications

Babak Amin Azad
Stony Brook University
baminazad@cs.stonybrook.edu

Pierre Laperdrix
Stony Brook University
plaperdrix@cs.stonybrook.edu

Nick Nikiforakis
Stony Brook University
nick@cs.stonybrook.edu

Abstract

As software becomes increasingly complex, its attack surface expands enabling the exploitation of a wide range of vulnerabilities. Web applications are no exception since modern HTML5 standards and the ever-increasing capabilities of JavaScript are utilized to build rich web applications, often subsuming the need for traditional desktop applications. One possible way of handling this increased complexity is through the process of software debloating, i.e., the removal not only of dead code but also of code corresponding to features that a specific set of users do not require. Even though debloating has been successfully applied on operating systems, libraries, and compiled programs, its applicability on web applications has not yet been investigated.

In this paper, we present the first analysis of the security benefits of debloating web applications. We focus on four popular PHP applications and we dynamically exercise them to obtain information about the server-side code that executes as a result of client-side requests. We evaluate two different debloating strategies (file-level debloating and function-level debloating) and we show that we can produce functional web applications that are 46% smaller than their original versions and exhibit half their original cyclomatic complexity. Moreover, our results show that the process of debloating removes code associated with tens of historical vulnerabilities and further shrinks a web application's attack surface by removing unnecessary external packages and abusable PHP gadgets.

1 Introduction

Despite its humble beginnings, the web has evolved into a full-fledged software delivery platform where users increasingly rely on web applications to replace software that traditionally used to be downloaded and installed on their devices. Modern HTML5 standards and the constant evolution of JavaScript enable the development and delivery of office suites, photo-editing software, collaboration tools, and a wide range of other complex applications, all using HTML, CSS, and JavaScript and all delivered and rendered through the user's browser.

This increase in capabilities requires more and more complex server-side and client-side code to be able to deliver the features that users have come to expect. However, as the code and code complexity of an application expands, so does its attack surface. Web applications are vulnerable to a wide range of client-side and server-side attacks including Cross-Site Scripting [4, 47, 72], Cross-Site Request Forgery [3, 33, 46], Remote Code Execution [18], SQL injection [19, 41], and timing attacks [35, 40]. All of these attacks have been abused numerous times to compromise web servers, steal user data, move laterally behind a company's firewall, and infect users with malware and cryptojacking scripts [43, 49, 74].

One possible strategy of dealing with ever-increasing software complexity is to customize software according to the environment where it is used. This idea, known as *attack-surface reduction* and *software debloating*, is based on the assumption that not all users require the same features from the same piece of software. By removing the features of different deployments of the same software according to what the users of each deployment require, one can reduce the attack surface of the program by maintaining only the features that users utilize and deem necessary. The principle of software debloating has been successfully tried on operating systems (both to build unikernel OSs [53] and to remove unnecessary code from the Linux kernel [51, 52]) and more recently on shared libraries [56, 61] and compiled binary applications [42].

In this paper, we present the first evaluation of the applicability of software debloating for web applications. We focus on four popular open-source PHP applications (phpMyAdmin, MediaWiki, Magento, and WordPress) and we map the CVEs of 69 reported vulnerabilities to the source code of each web application. We utilize a combination of tutorials (encoded as Selenium scripts), monkey testing, web crawling, and vulnerability scanning to get an *objective* and *unbiased* usage profile for each application. By using these methods to stimulate the evaluated web applications in combination with dynamically profiling the execution of server-side code, we can precisely identify the code that was executed during this stimulation and therefore the code that should be retained during the process of debloating.

Equipped with these server-side execution traces, we evaluate two different debloating strategies (file-level debloating and function-level debloating) which we use to remove unnecessary code from the web applications and quantify the security benefits of this procedure. Among others, we discover an average reduction of the codebase of the evaluated web application of 33.1% for file-level debloating and 46.8% for function-level debloating, with comparable levels of reduction in the applications' cyclomatic complexity. In terms of known vulnerabilities, we remove up to 60% of known CVEs and the vast majority of PHP gadgets that could be used in Property Oriented Programming attacks (the equivalent of Return-Oriented Programming attacks for PHP applications).
Overall, our contributions are the following:

- We encode a large number of application tutorials as Selenium scripts which, in combination with monkey testing, crawling, and vulnerability scanning, can be used to objectively exercise a web application. Similarly, we map 69 CVEs to their precise location in the applications' source code to be able to quantify whether the vulnerable code could be removed during the process of debloating.
- We design and develop an end-to-end analysis pipeline using Docker containers which can execute client-side, application stimulation, while dynamically profiling the executing server-side code.
- We use this pipeline to precisely quantify the security benefits of debloating web applications, finding that debloating pays large dividends in terms of security, by reducing a web application's source code, cyclomatic complexity, and vulnerability to known attacks.

To motivate further research into debloating web applications and to ensure the reproducibility of our findings, we are releasing *all* data and software artifacts.

2 Background

In this section, we briefly describe the effect of package managers on software bloat and provide a motivating example for debloating web applications.

2.1 Package managers and software bloat

To ease the development of software, developers reuse third-party libraries, external packages, and frameworks for their applications. This approach enables developers to focus on their applications while relying on proven and tested components. Statistics from popular package managers show that reliance on external packages is a widely adopted practice across many different languages. NPM, the registry hosting NodeJS packages, reports more than 10 billion package downloads a month [73]. Similarly, PyPI, the package manager for Python, reports more than a billion a month [30], while Packagist, the main repository for Composer package manager for PHP, reports the download of 500 million packages each month [29].

At the same time, it is doubtful that *all* the code and features obtained through these packages and frameworks are actually used by the applications that rely on them. For the most part, when developers rely on external dependencies, they include entire packages with no effective way of disabling and/or removing the parts of these packages and frameworks that their applications do not require.

2.2 Motivating web-application debloating

In this study, we look at the bloat of web applications and quantify how debloating can provide concrete security benefits. Even though debloating has been successfully applied in other contexts, we argue that the idiosyncrasies of the web platform (e.g. the ambient authority of cookies and the client/server model which is standard for the web but atypical for operating systems and compiled software) require a dedicated analysis of the applicability of debloating for web applications.

To understand how the bloat of a web application can lead to a critical vulnerability, we use a recent vulnerability of the Symfony web framework (CVE-2018-14773 [28]) as a motivating example. Specifically, the Symfony web framework supported a legacy IIS header that could be abused to have Symfony return a different URL than the one in the request header, allowing the bypassing of web application firewalls and server-side access-control mechanisms. If this type of header was never used by the server, debloating the application would have removed support for it, which ultimately would have prevented anyone from exploiting the vulnerability. Drupal, a popular PHP Content Management System (CMS), was also affected by the same vulnerability since it uses libraries from the Symfony framework to handle parts of its internal logic [26]. Even if Drupal developers were not responsible for the code that leads to the vulnerability, their application could still be exploited since Symfony was an external dependency. Even more interestingly, an analysis of the official Symfony patch on GitHub [27] reveals that the vulnerable lines were derived from yet another framework called Zend [31]. This shows that the structure of web applications can be very complex with code reuse originating from many different sources. Even if developers take all possible precautions to minimize vulnerabilities in their own code, flaws from external dependencies can cascade and lead to a critical entry point for an attacker.

Overall, there are clear benefits that debloating could have on web applications. Assuming that we are able to pinpoint all the code that is required by the users of a given software deployment, all other code (including the code containing vulnerabilities) can be removed from that deployment.

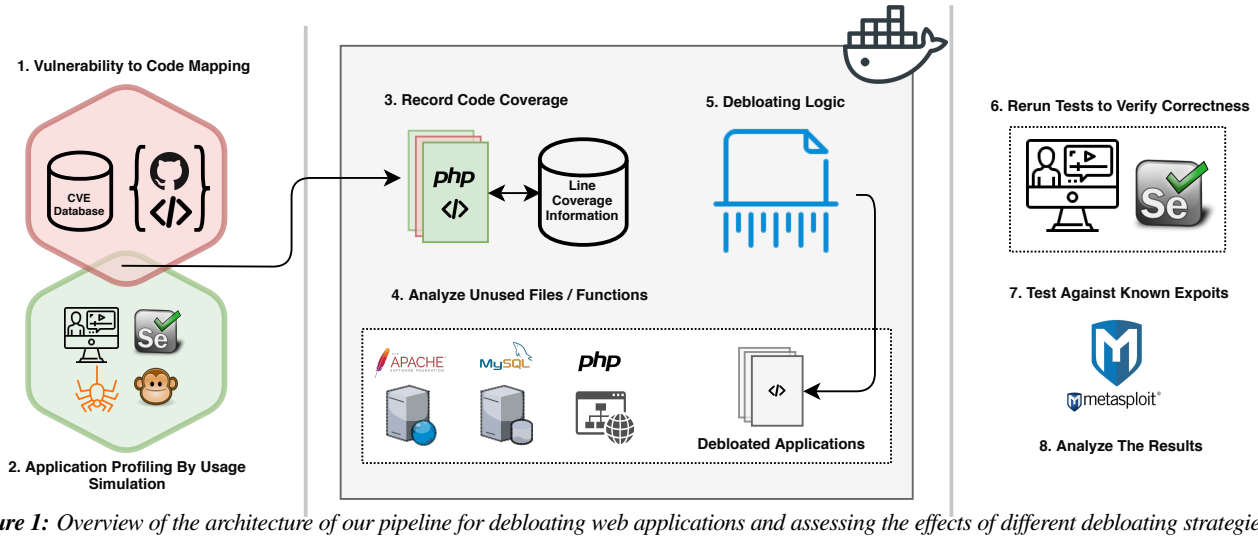


Figure 1: *Overview of the architecture of our pipeline for debloating web applications and assessing the effects of different debloating strategies.*

3 Setup

In this section, we describe the process of gathering information regarding known vulnerabilities (in the form of CVEs) for web applications, designing and executing tests against web applications of interest, and identifying the server-side code that was executed as a result of client-side actions.

3.1 Overview

The setup for our framework is depicted in Figure 1. To debloat target applications, we first collect information about the vulnerabilities of the applications that we analyze in our study. This information includes the files, functions, and line numbers where each vulnerability resides (Step 1, Section 3.3). Then, we simulate usage of the application through a combination of different techniques (Step 2, Section 3.4). Using a PHP profiler tool (`XDebug`), we record the lines, functions, and files, that are triggered during the simulation (Step 3, Section 3.5).

In the middle part of our pipeline, the debloating engine takes both the target applications and coverage information to perform debloating at different levels of granularity, and rewrite parts of the application to remove unused pieces of code based on the debloating strategy being evaluated (Steps 4 and 5, Section 4). Our framework also provides a complete reporting panel to assist human analysts in understanding which vulnerabilities can be removed by the present debloating strategies.

Last, we verify the correctness of our debloating process by running a set of tests against the debloated web applications, and verifying that no removed piece of code is triggered (Step 5). To this end, we utilize assertions in place of the removed code blocks. An absence of error messages from these assertions means that all tests were successfully completed without triggering any missing server-side code. As an final step of verification, we also test the debloated applications against a series of exploits and verify that exploits which abuse any of the vulnerabilities that were removed as part of the debloating process, do not succeed (Step 6, Section 5.6).

To ease integration and facilitate the analysis of new web applications, we adopted a modular architecture that relies on three Docker containers. The *Application* container hosts our web applications. The profiler enabled on its web server is responsible for collecting code coverage information. The *Database* container runs a MySQL server that stores the code coverage information along with the databases of the tested applications. Lastly, the *Debloating* container which includes our debloating logic, analyzes the coverage information and generates debloated versions of applications. It also provides a reporting panel that indicates which vulnerabilities are removed in each application after debloating. To add a new vulnerability, a user simply has to provide the details of the vulnerable file(s) and line(s).

3.2 Analyzed web applications

To understand how the process of debloating increases the security of web applications, we decided against using toy-like web applications. Instead, we focused on established open-source applications with millions of users, and the presence of a sufficient number of known historical vulnerabilities (in the form of CVEs) to allow us to generalize from them. To this end, we selected phpMyAdmin [60], MediaWiki [59], Magento [58], and WordPress [75], which are representative samples of four different types of web applications namely web-administration tools, wikis, online shops, and blogging software. Table 1 shows the versions of these web applications that we utilized, in order to map CVEs to the location of the vulnerability in the source code of each application.

3.3 Vulnerability to source-code mapping

To determine whether debloating web applications can actually remove vulnerabilities, we performed a mapping of known CVEs

Table 1: *Analyzed open-source web applications.*

Web Application	Version	Known CVEs (≥2013)
Magento	1.9.0, 2.0.5	10
MediaWiki	1.19.1, 1.21.1, 1.24.0, 1.28.0	111
phpMyAdmin	4.0.0, 4.4.0, 4.6.0, 4.7.0	130
WordPress	3.9.0, 4.0, 4.2.3, 4.6, 4.7, 4.7.1	131

to the vulnerable lines, functions, and files, that they exploit in each application. This way, by looking at an application after debloating, we can determine if the files, functions, or lines responsible for the vulnerability, are still present or were removed during the debloating process.

Even though there exist multiple databases listing the current and historical CVEs of popular software (including the web applications in question) [36, 37], locating the actual source code containing the vulnerability described in a CVE, is a non-trivial process which requires careful investigation. In some cases, the right patch can be discovered because of a direct reference to a CVE in a commit message, or in a bug report on official public repositories of web applications. For others, the fix is included within numerous commits that have to be carefully analyzed to locate the appropriate lines of code. Since a vulnerability can span over multiple lines, functions, and even multiple files, we record all affected locations in a database so that this information can be later correlated with each evaluated application.

Given the time-consuming nature of mapping CVEs to existing code, for this study, we limited ourselves to, at most, 20 CVEs per application of interest. The complete list of CVEs we mapped for this study can be found in Table 9 in the Appendix. To select these CVEs, we ordered existing vulnerabilities by their CVSS score (thereby selecting the ones that are the most critical) and we did not consider vulnerabilities that were reported before 2013. This focus on fairly recent vulnerabilities (i.e. in the last five years) makes our results more generalizable to the current state of web applications, as opposed to quantifying vulnerabilities in source-code which has since dramatically evolved. Note that, because not all versions of a web application are vulnerable to all evaluated CVEs, we had to map vulnerabilities across a number of different versions, as shown in Table 1.

3.4 Application usage profiling

Modern web applications provide an incredibly wide range of features and options to their users. Even though, from a functional perspective, more features are desirable, from a security perspective, the code that implements new features may contain new vulnerabilities thereby further expanding a program's attack surface. In order for a system to be able to remove code related to unnecessary features, one must first identify which features are necessary for a target set of users.

Given a usage profile, the goal of our framework is to produce debloated versions of web applications which maintain the code and features that are part of that profile but remove the rest. To be as objective as possible with what features are considered "necessary," we utilize four independent sources of web application usage: i) online tutorials describing how to use the applications of interest, ii) web crawlers that autonomously navigate the application, iii) vulnerability scanners that feed malicious content to the application, and iv) monkey testing tools that click on random parts of webpages and type random keystrokes. The combination of all four gives our profiles both breadth (through the crawler and monkey testing) as well as depth (through the user following complicated paths while providing expected inputs and the vulnerability scanner which provides large amounts of malicious inputs trying to exploit the web application).

3.4.1 Tutorials

To simulate common interactions with an application, we use a popular search engine to search for the application's name followed by the word "tutorials" (e.g. "phpMyAdmin tutorials") and follow the tutorials from the first two pages of search results.

Specifically, we map each tutorial to a Selenium script that allows us to both execute the same tutorial multiple times and also assess the correctness of the results (e.g. encode that when we delete a database using phpMyAdmin, the deleted database is no-longer shown in the list of databases). Note that this mapping of tutorials to Selenium scripts is yet another time-consuming process which, occasionally, has to be repeated for different versions of the same web application. One change in a form field or in a selector can break the complete flow of a test suite and we observed a significant number of cases with slight interface changes between two consecutive versions of the same application.

Overall, after fine-tuning the scripts for all our tested versions, we obtained 46 tutorials which translated into 302 use cases scripted as Selenium tests requiring 16,025 lines of code. Given our desire for complete reproducibility of our results, we include the complete list of tutorials in the Appendix (Table 8) along with WebArchive links that will remain available despite potential future domain expirations and linkrot of the original URLs [48].

Below, we provide a non-exhaustive list of actions that were part of the followed tutorials of each web application. Full details are available in the actual tutorials and in the Selenium scripts which we will release together with this paper.

Actions covered by phpMyAdmin tutorials: As a web administration tool, all phpMyAdmin functionality is protected by an authentication mechanism. We followed the actions described by tutorials when logged in as a root user account with full application access. The Selenium-encoded tutorials cover database operations including creating and dropping databases, filling tables with data, querying, table indexes, and importing/exporting data. They also include administration tasks such as adding new user accounts, optimizing databases, checking database server status, obtaining performance metrics, and accessing server settings such as variables, charsets, and engines.

Actions covered by MediaWiki tutorials: MediaWiki provides different features depending on the privileges of the user. Unauthenticated users can only visit and search pages. Registered ones can post and edit content while administrators can perform moderation and management operations. The tutorials that we followed cover all these different use cases. More specifically, actions coded in our tutorials include authentication, creating and renaming pages, importing and exporting content from the wiki, as well as changing settings such as skins, styles, and formatting options.

Actions covered by WordPress tutorials: As a blogging software, WordPress has two distinct entry points, one for normal unauthenticated users to read blogs and post comments, and a separate administration panel accessible to privileged and authenticated users. WordPress tutorials mostly focus on administrative tasks since normal users have limited abilities. The Selenium-encoded tutorials include actions such as creating a new post using HTML for the content, modifying most post options (ranging from visibility and tags to setting featured images), as well as downloading and changing WordPress themes. For the administration panel, the tutorials include exporting content, setting up user accounts, and uploading media. Finally, the tutorials include the visiting of posts and the posting of comments as well as the management of comments, such as approving them, marking them as spam, and deleting them.

Actions covered by Magento tutorials: Magento is the largest evaluated web application in terms of source code and has the most features compared to the other applications. Similar to WordPress, the tutorials mostly target administration tasks which include store settings, advanced product search options, order notification via RSS, product pricing, currencies and tax rules, delivery and payment methods, emails and notifications, reviews and ratings and cache control. Some tutorials go in even more details by covering product and stock management, managing customers and groups configurations, modifying the UI, creating pages, and using widgets. On the customer side, we followed tutorials that included registration of a new account, authentication actions, and purchasing products until checkout.

3.4.2 Monkey testing

Monkey testing is a method for testing software where the simulated user sends random clicks and keystrokes to the target application. This unpredictable behavior can uncover bugs in an application as it can trigger paths and actions that were not anticipated by developers. In our case, we use such a technique to trigger additional code, not covered by tutorials. We observe that this approach adds breadth to the code coverage by reaching easy to access features. In addition, by feeding random key strokes into forms, monkey testing can bring the application in an error state thus exercising error-handling pieces of code.

We rely on the stress-testing library called `gremlins.js` [7] in conjunction with the GreaseMonkey browser extension [6] to inject the library into web application pages. Since this kind of testing can occasionally trigger unwanted actions, we have to take necessary steps to stop them, e.g., prevent the test from leaving the web application and visiting external websites. We also want to prevent `gremlins.js` from getting trapped on a single page as an unexpected JavaScript dialogue box or a dead end page can pause our test execution. An additional issue is that of accidentally logging out a web application by clicking on a logout link. Given that we run monkey-testing under three different usage profiles (public user, logged-in user, and administrator) we took steps to avoid accidental logouts. Overall, we perform the following modifications: i) we remove all links that lead to external pages, ii) we remove logout buttons for applications that require authentication, iii) we override the aforementioned JavaScript functions and iv) we set a timeout to detect when the monkey is stuck and reset it to a known good state. All these actions are done using injected JavaScript on target pages prior to starting the `gremlins.js` library.

To cover a large set of pages from a web application, we run `gremlins.js` for 12 hours for each of the test profiles. To guarantee the reproducibility of our experiment, we choose a fixed seed for each run that will generate the same sequence of pseudo-random actions.

3.4.3 Crawling

Web spiders (also known as crawlers) are a type of bot that follows the links of a web application and optionally submits forms with predefined content. Each newly crawled page is added to a database of the application that the crawler uses to prevent repeated visits to the same pages. For our study, we use BurpSuite Spider v2.0.14beta [2] to crawl our web applications. As a result, we augment the application coverage with code paths that were not triggered, either through the followed tutorials or through monkey testing.

3.4.4 Running vulnerability scanners

Vulnerability scanners are tools that try to detect security flaws in web applications. We use BurpSuite Scanner v2.0.14beta [2] based on the URLs extracted by the spider to look for vulnerabilities in headers, URLs and forms. Notably, the scanner tries different injection mechanisms like SQL injection, XSS, PHP file injection, and path traversal, to trigger errors and reach unwanted states in the application. The vulnerability scanner goes beyond what the crawler and the monkey cover by modifying headers and URL parameters. By inspecting the resulting coverage, we observe that each of these four methods result in exercising server-side code that would not have been exercised through the other methods. We quantify this relationship in Section 5.

3.5 Recording server-side code coverage

Regardless of the method that is used to interact with a web application, in order to be able to successfully remove unused code (i.e. debloat the web application), we must be able to

associate client-side requests with server-side code. To record the files and lines of code that are triggered by user requests, we make use of PHP profilers.

PHP profilers are available as PHP extensions that modify the PHP engine to collect code-coverage information. There exist a number of different profilers, such as, `XDebug` [23], `phpdbg` [16], and `xhprof` [24] all of which require a similar setup to record code coverage. For our framework, we decided to use `XDebug` as it is the most mature profiler and is actively maintained.

3.5.1 Adding coverage support in a web application

Connecting a web application to XDebug. To be able to perform dynamic analysis and record lines of code that are triggered by user requests, our framework must add calls to specific `XDebug` functions in every PHP file of a web application. Specifically, both `xdebug_start_code_coverage()` and `xdebug_get_code_coverage()` functions are called to, respectively, start and receive coverage information. If the "get" function is never called, the coverage information is lost. In the following paragraphs, we describe challenges related to obtaining the code coverage from `XDebug` and how we overcame them.

The case of unrecorded lines. Boomsma and Gross reported on the possibility of removing unused code in a custom PHP application [34]. By performing dynamic analysis, they observed which files were not used and removed them from their application. The authors utilized their own profiler and took advantage of the `auto_append` built-in function of PHP to add the necessary log functions at the very end of all PHP files [1].

For our study, we initially attempted to use the same approach and ran preliminary tests by appending `XDebug` function calls at the end of our tested files. However, we discovered that the coverage was incomplete and that some lines were not properly recorded. Given that any PHP file can call the *exit()* or *die()* function at any time to terminate the current script, our `XDebug` calls which were located at the end of each file, were not always executed thus leading to under-reported code coverage.

3.5.2 Main challenges for getting full coverage

Avoiding early exits. To overcome the coverage problems due to calls to exit functions, we utilized a specific type of PHP callback functions, called *shutdown* functions. When registered, these functions are triggered after all the code on the page has finished running or after either *exit()* or *die()* functions are called. This way, we are able to obtain the desired coverage information even if a PHP script used one of the aforementioned functions. Interestingly, we also discovered that calls to *exit()* inside a shutdown function prevent the execution of other shutdown functions including the call to collect our own code-coverage information. To correct this issue, we statically analyzed the evaluated applications and automatically added calls to collect code coverage before these exit calls (e.g. Line 7 in Listing 1).

Getting correct coverage information of shutdown functions. Another challenge, in terms of recording correct code-coverage information, is to properly record the executed lines of code inside shutdown functions. As mentioned by the PHP manual [12], shutdown functions are called in the order they were registered. This means that if our own shutdown function is registered first, it will also be triggered first, thereby missing any calls to subsequent shutdown functions present in the same PHP file. To get full coverage, we use the following approach: our own shutdown function will perform a late registration of a final shutdown function that will be added at the very end of the execution queue. This way, we can be certain that the very last shutdown function that will be executed in a script will be our own, providing us with the desired coverage information.

Getting correct coverage information of destructors. The final challenge that we faced was to properly record covered lines for all class destructors. PHP uses garbage collection and reference counting to remove objects from memory, whenever they are no longer necessary. However, there is no real way to anticipate when the garbage collector will effectively remove objects during program execution. If objects are destroyed *before* the shutdown functions are executed, our framework has no issue recording them. However, if they are destroyed after, our shutdown functions are incapable of registering the execution of these destructors.

To handle this special case, we rewrote class destructors so that they register themselves while they are executing. Every time a destructor is called, we query the `XDebug` engine to check whether code-coverage recording is currently in progress. This way, we can determine whether the destructor is called before or after shutdown functions. If the destructor is called after shutdown functions, we dynamically decide to start recording all executed lines within the destructor and save the coverage information when it finishes executing.

Summary. As witnessed through the above use cases, collecting the correct code coverage information for a web application is significantly more complicated than one would initially expect. Through the preprocessing of code, and the use of destructors and shutdown functions, we solve the issues that were not even mentioned in prior work and get a precise view of the code that executes at the server side, as a result of user requests. Listing 1 provides an example of concrete modifications in a PHP file. On line 7, we added a code-coverage call before an `exit` which happens inside a shutdown functions to prevent information loss due to early exits. On lines 14 and 17, we wrapped the destructor with code-coverage calls.

```php
<?php
register_shutdown_function("PMA_Response::resp");
class PMA_Response {
  public static function resp() {
    $buffer->flush();
    // Prepend original call to exit:
    collect_code_coverage();
    exit;
  }
}

class TCPDF {
  public function __destruct() {
    // If called after shutdown_functions
    // start recording code coverage
    ...
    // If called after shutdown_functions
    // stop coverage
  }
}
?>
```

Listing 1: *Code rewritten by the debloating framework to ensure correct code coverage of corner cases.*

4 Debloating web applications

In this section, we briefly describe the evaluated debloating strategies and the steps we took to ensure that the debloated applications remain functional.

4.1 Debloating strategies

By combining the simulated usage of a web application (achieved through tutorials encoded in Selenium scripts, web crawlers, monkey testing, and vulnerability scanning) with server-side code profiling, we can identify the code that was executed as part of handling web requests. Consequently, code whose execution was not triggered by any client-side request can presumably be removed since it is not necessary for any of the functionality that is desired by users (as quantified by the utilized usage profiles). In this work, we evaluate the following debloating strategies:

• **File-level debloating:** Given that the source code of web applications spans tens or hundreds of different files, we can completely remove a file, when none of the lines of code in that file were executed during the stimulation of the web application.

• **Function-level debloating:** In function-level debloating, not only can we remove entire files but we can also selectively remove some of the functions contained in other files. This is a more fine-grained approach which allows us to remove more code, than the more conservative, file-level debloating strategy.

More fine-grained approaches are possible, such as, the removal of specific code statements from retained functions which were not exercised during stimulation. However, such changes essentially modify the logic of a function (e.g. removing conditional code blocks) thereby increasing the probability of breaking the resulting program when a minute change of a client-side request would lead the execution into these blocks of code.

4.2 Detecting the execution of removed code

We replace all removed functions and files with placeholders which, if executed, have the following tasks:

• **Exit the application:** If a placeholder happens to be triggered, the PHP application will start its shutdown procedures. This way, the application does not enter an unexpected state that was not planned by the debloating process.

• **Record information about the missing function:** In order to better understand which missing placeholders were triggered and how, our framework logs several pieces of information, such as, the URL that triggered the execution of the removed code, the name of the class and function of the removed code, and the corresponding line numbers.

To ensure that the debloating process has preserved the functionality of the debloated web application, we rerun all the Selenium-mapped tutorials and monkey scripts after the debloating stage. If our placeholder code for removed files and functions executes during this stage, this means that this code should not have been removed.

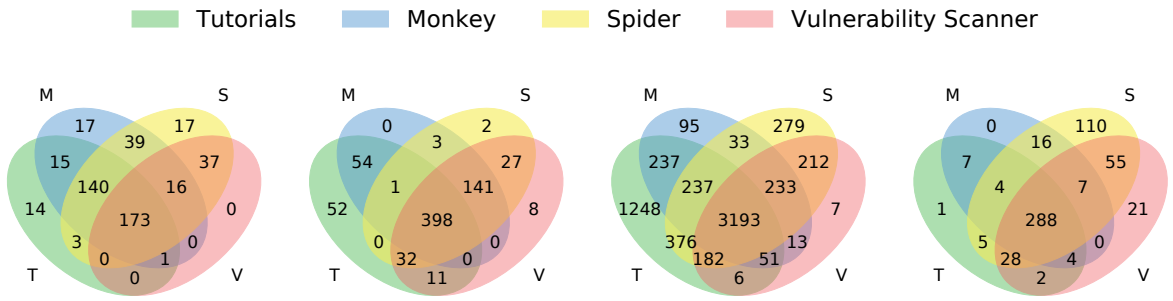
This feedback mechanism proved invaluable during the development of our framework since it helped us identify problems with our coverage logic which in turn revealed the challenges that we described in Section 3.5.2.

5 Results

To assess the impact of debloating web applications, we analyze our results from a number of different perspectives. First, we show the contributions of different application-profiling methods and then compute different metrics to understand the effectiveness of debloating in terms of reducing the attack surface of our tested applications. Next, we focus on CVEs to determine whether debloating can actually remove critical vulnerabilities. Then, we take a closer look at the bloat introduced by external packages along with the security implications that come with using this specific development practice. Finally, we look at what has effectively been removed in debloated applications and test a number of exploits against the original and debloated versions of the evaluated web applications.

5.1 Tutorials vs. Monkey Testing vs. Crawling vs. Vulnerability Scanning

As described in Section 3.4, to ensure that we exercise web applications in an objective and repeatable way, we utilized tutorials, monkey testing, crawlers, and vulnerability scanners. Figure 2 shows the coverage, in terms of server-side files, that each method obtained on the latest version of each web application in our testbed. We can clearly see that all four methods are required, with each method contributing differently for different web applications. For example, tutorials trigger



(a) phpMyAdmin 4.7.0 *(b) MediaWiki 1.28.0* *(c) Magento 2.0.5* *(d) WordPress 4.7.1*

Figure 2: *Venn Diagrams showing covered files during the execution of Tutorials, Crawler, Monkey testing and Vulnerability scanner*

more files in Magento compared to other applications, while Spider covers most unique files in WordPress.

5.2 Debloating by the numbers

To evaluate the effectiveness of our two debloating strategies, we computed different metrics that provide insights into what has actually been removed during the debloating process.

5.2.1 Logical lines of code

The size of a program positively correlates with the number of programming errors (i.e. bugs). According to McConnel [55], the industry average, at least in 2004, was to have between 1 and 25 bugs for every one thousands lines of code. Given the importance of the size of an application to its overall security, we start by estimating the reduction of the attack surface by looking at the Logical Lines Of Code (LLOC, sometimes also called Effective Lines Of Code). LLOC is intended to measure lines of code without comments, empty lines and syntactic structure required by the programming language. LLOC reduction is a robust and precise indicator of how much the volume of the code was reduced. Figure 3 reports on the LLOC for all versions of the applications we debloated.

Number of logical lines over time. Looking at the number of LLOC of the original applications, we can observe two different evolution behaviors. For WordPress, the amount of code is stable and there is even a small decrease of 2% of LLOC between versions 4.7 and 4.7.1. For the other applications, we observe the opposite where the source code in the latest versions spikes, compared to the ones released just before them: 82% LLOC increase for phpMyAdmin, 99% for MediaWiki, and 171% for Magento. By analyzing the code of these newer versions in an attempt to understand their sudden expansion in size, we discovered that these spikes can be attributed to a change in development practices, namely the reliance on external packages. As WordPress does not rely on external packages, it does not

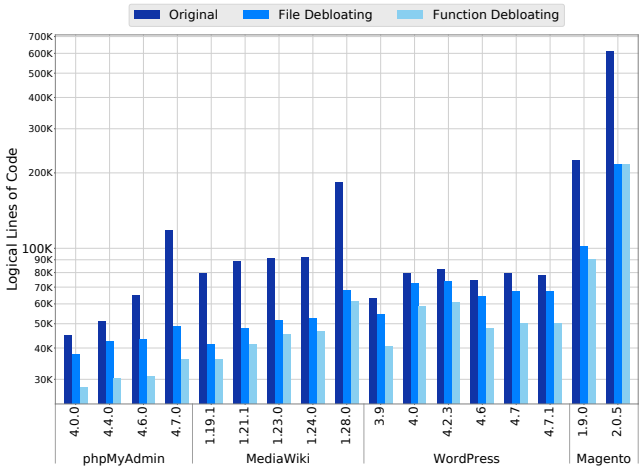


Figure 3: *Logical Lines of Code before and after debloating*

exhibit this kind of behavior. We discuss the issue of relying on external packages in more detail in Section 5.4.

File-level debloating. Overall, file-level debloating, the most conservative of the two evaluated debloating strategies, is already effective in reducing the number of LLOC with an average of 33.1% reduction. The minimum observed in our experiment is 9.2% for WordPress v.4.0 and a maximum of 64.5% for Magento v.2.0.5. For Magento, this reduction represents a removal of 393K lines of code. This number is a clear sign that large web applications encompass many different features that may not be used by all users and therefore result in bloated applications with an unnecessarily large attack surface. At the same time, it is worthwhile repeating that all debloating results presented in this section are conditional to how web applications are used. Therefore, these large levels of debloating cannot be guaranteed for all possible deployments of web applications. We discuss this issue in Section 7.

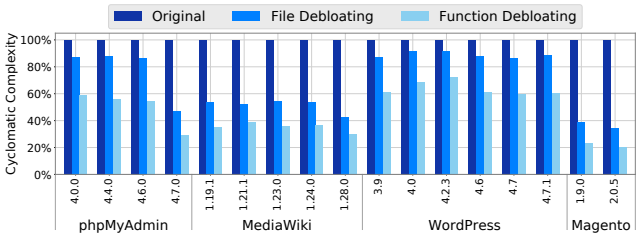


Figure 4: *Evolution of cyclomatic complexity before and after debloating*

Function-level debloating. On average, function-level debloating is able to remove 46.8% of lines of code. For both Magento and MediaWiki, it can remove up to 7% more code over file-level debloating. For phpMyAdmin and WordPress, we observe an increase of debloating capability of up to 24%. This larger reduction (compared to MediaWiki and Magento) is mainly due to the differences in software development practices.

Compared to the other tested applications, phpMyAdmin and WordPress are more monolithic with a smaller number of large source-code files. Since file-level debloating only removes files when none of their functions were executed, the monolithic nature of these two applications resists this kind of coarse-level debloating. Contrastingly, Magento and MediaWiki are developed in a much more modular fashion (many small files each responsible for a small number of well-defined tasks) and therefore lend themselves better to file-level debloating. The more fine-grained, function-level debloating bypasses this issue and can therefore reduce the attack surface of a web application, even for more monolithic web applications.

5.2.2 Cyclomatic complexity

Next, we look at the evolution of cyclomatic complexity (CC). CC is defined as the number of linearly independent paths through the code of an application [54]. A high CC for a single class implies complicated code that is difficult to debug and maintain [39] and therefore more prone to contain vulnerabilities when compared to code with low CC [52, 69].

Figure 4 reports on the evolution of the overall CC for each tested version in our experiment. File-level debloating decreases CC between 5.9% to 74.3% with an average of 32.5%. Function-level debloating decreases the program complexity between 23.8% and 80.2% with an average of 50.3%. These statistics demonstrate that debloating can remove complex instructions and execution paths in addition to simple ones. Moreover, the difference between file-level and function-level debloating shows that code removal through function-level debloating is much more suited to all kinds of web applications as shown earlier through LLOC reduction achieved via function-level debloating.

Table 2: *Number of CVEs removed after application debloating*

Application	Strategy	Total Removed CVEs		Removed Exploitable CVEs	
phpMyAdmin	File Debloating	4/20	20 %	3/19	15.7 %
	Function Debloating	12/20	60 %	11/19	57.8 %
MediaWiki	File Debloating	8/21	38 %	3/16	18.7 %
	Function Debloating	10/21	47.6 %	5/16	31.2 %
WordPress	File Debloating	0/20	0 %	0/20	0 %
	Function Debloating	2/20	10 %	2/20	10 %
Magento	File Debloating	1/8	12.5 %	1/8	12.5 %
	Function Debloating	3/8	37.5 %	3/8	37.5 %

5.3 Analysis of CVEs

In this section, we investigate the number of removed CVEs after debloating along with the effects of debloating on different vulnerability categories.

5.3.1 CVE reduction after debloating

One practical way to measure the security benefits of debloating web applications is to study the effects of debloating on known historical vulnerabilities. If vulnerabilities were part of the core functionality of the program, the evaluated debloating strategies will not be able to remove the code associated with them. However, if some vulnerabilities reside in parts of a web application that are not commonly used, the process of debloating can effectively remove them.

Table 2 compares the effectiveness of debloating strategies by listing the fractions of removed CVEs. We consider a vulnerability to have been successfully removed if all the lines of code and functions associated with that vulnerability were removed during the stage of debloating. This is a conservative approach as one modification performed on a single line could thwart a complete attack. As such, the numbers we report in this section can be interpreted as lower bounds of the actual number of removed CVEs.

In terms of configuration, we selected the default one for each application. However, certain vulnerabilities may not be exploitable under this configuration. For example, there exists 5 CVEs in our dataset for MediaWiki which require file upload functionality to be enabled. Since this option is disabled by default, we make an explicit distinction in the table. "Total Removed CVEs" is the total number of CVEs removed by debloating regardless of whether the vulnerable code is enabled or disabled through a configuration option. "Removed Exploitable CVEs" reports on the CVEs that are reachable under default configurations of target web applications.

On average, we discovered that up to 38 % of vulnerabilities are removed by file debloating whereas 10-60 % are removed by function debloating. As shown in Table 2, function-level debloating can triple (in the case of phpMyAdmin and Magento) the number of removed CVEs, compared to file-level debloating. This behavior can be generalized to web applications that do not have CVE information and demonstrates that the reduction of a web application's LLOC (Section 5.2.1) and its cyclomatic complexity (Section 5.2.2) translates to a reduction of concrete

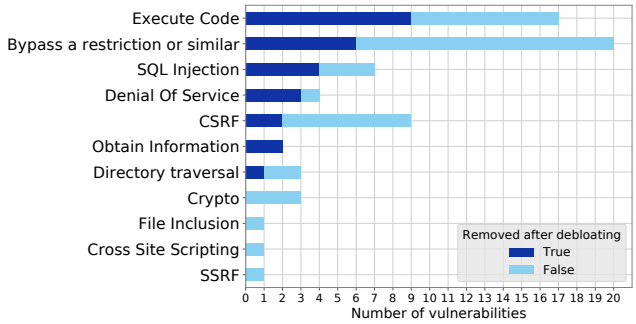


Figure 5: *Vulnerability Categories*

vulnerabilities. Wordpress is a clear negative outlier with only 10% CVE reduction, even through the more flexible function-debloating strategy. As mentioned earlier, WordPress is a relatively monolithic application and most of our mapped CVEs are located in core WordPress code (e.g., Authentication, CSRF tokens, and post/comment-related actions) which cannot be removed by our debloating framework.

5.3.2 Types of CVEs in analyzed web applications

Even though our results demonstrate the ability to remove vulnerabilities from web applications through the use of debloating, one may wonder whether debloating is better suited for some types of vulnerabilities over others. Figure 5 provides details on the categories of the CVEs we removed through debloating.

One can observe that for certain classes of vulnerabilities, such as, Denial-of-Service attacks and Information-Revealing vulnerabilities, debloating can almost completely remove them. For others, such as, restriction bypassing, command execution, and SQL injection, debloating can substantially reduce them. Our interpretation of these findings has to do with the maturity of the evaluated web applications. Specifically, all four web applications have been available for a long period of time, allowing many shallow vulnerabilities to have already been discovered and corrected. The remaining vulnerabilities are likely to be situated in parts of a web application that are less commonly exercised. For example, the code-execution vulnerabilities that can be removed for phpMyAdmin are inside very specific features, such as, the ability to export PHP arrays (CVE-2016-6609), the support of the ZIP extension while importing data (CVE-2016-6633), and the abilities to copy table definitions (CVE-2013-3238) and perform Regex search and replace over table columns (CVE-2016-5734).

Contrastingly, the three cryptography-related vulnerabilities we analyzed are still present in the debloated versions of web applications. One of the CVEs related to this category is about a flaw in the cookie encryption algorithm in phpMyAdmin (CVE-2016-6606). Since every page interacts with user cookies to, at the very least, verify them, vulnerable code cannot be removed. Another vulnerability in this category relates to an insecure random number generator used in cryptographic operations by Magento (CVE-2016-6485). This vulnerability exists in a constructor of the main encryption classes which is widely used throughout the application. When considered together, these findings suggest that cryptography-related vulnerabilities are a core part of web applications and thus unlikely to be removed through the process of debloating.

5.4 External packages

5.4.1 Quantifying the bloat from external packages

In our testbed, phpMyAdmin v.4.7.0, MediaWiki v.1.28.0 and Magento v.2.0.5 rely on external dependencies that can be downloaded via Composer (WordPress does not rely on external packages). As described in Section 2, Composer is a package manager for PHP (similar to the NPM manager for NodeJS applications) which allows web applications to specify which external packages they rely on and have these packages be tracked and updated.

As we briefly discussed in Section 5.2.1, the number of LLOC of these three specific versions dramatically increases (compared to prior versions) because of this dependency on external packages. Table 3 provides statistics on the number of packages pulled by these applications and how much bloat they provide against our usage profiles.

First, one can observe that external packages introduce a large amount of unused code. For all three debloated applications, more than 84% of their code was removed from them. This means that the attack surface is unnecessarily large through the dependency on external packages. The number of removed lines from external packages for Magento is particularly noteworthy with more than 178,000 lines of code removed. Moreover, the number of packages that can be completely removed is also quite large: 84% for phpMyAdmin, 60% for MediaWiki and 81% for Magento. This confirms that most packages are unnecessary for the usage profiles that we recorded. Finally, focusing exclusively on the lines of code, phpMyAdmin is the only application where external packages have more lines than the main application. However, after debloating, this relationship is reversed with the codebase of phpMyAdmin being three times the size of the introduced external packages.

Despite the advantages of using package managers (e.g. the ability to track dependencies and update vulnerable libraries without the need to update the main application), our findings show that these advantages come at a considerable cost in terms of unnecessarily expanding the attack surface of a web application with code that is seldomly executed. As such, developers must take special care to include the bare minimum of external packages, knowing the unwanted side-effects that each external package brings.

5.4.2 Removing POI gadgets

What are POI gadgets? Property Oriented Programing (POP) is an exploitation technique in PHP which works similarly to Return Oriented Programming (ROP) [67] and is used to exploit

Table 3: *Statistics on the external packages included in web applications and the effects of debloating in terms of reducing their LLOC.*

Application	Before debloating			After function-level debloating					
	# lines in main application	*# lines in packages*	*# packages*	*# lines in main application*	*# lines in packages*	*# packages completely removed*	*# packages where a given % lines were removed*		
							>70%	*<70% and >30%*	*<30%*
phpMyAdmin 4.7.0	35,739	82,604	45	26,377 (-26.2%)	9,653 (-88.3%)	38 (84.4%)	2	1	4
MediaWiki 1.28.0	133,019	50,898	40	54,827 (-58.8%)	6,285 (-87.7%)	24 (60.0%)	2	2	12
Magento 2.0.5	396,448	212,906	71	181,696 (-54.2%)	34,038 (-84.0%)	58 (81.7%)	6	5	2

PHP Object Injection (POI) vulnerabilities [11]. In this technique, the attacker creates exploit gadgets from available code in the applications. By chaining multiple gadgets within the application, an attacker can usually run arbitrary code, write to arbitrary files, or interact with a database. Dahse et al. have studied the automatic generation of such gadget chains for PHP applications [38].

PHP unsafe deserialization. The PHP language gives developers the ability to serialize arbitrary objects in order to store them as text, or transfer them over the network. Deserialization reverses this process, generating PHP objects from serialized data. This mechanism can be abused by an attacker to load specific classes in the application and build a gadget chain. Practical examples of this vulnerability are when `unserialize` is called on a database field or value of a field within a cookie that can be manipulated by the users.

Historically, this attack was very difficult to successfully execute. Attackers could only build gadgets with the classes that were present in the context of the vulnerable file. They needed insights into how the application was built in order to know which classes could be abused for gadgets. However, starting from PHP 5, the `__autoload()` magic function [10] was introduced and unintentionally made exploitation of deserialization vulnerabilities easier. This new loading feature was beneficial for PHP developers who did not have to manually include all the files they wanted to use at the very top of each of their PHP files. It also helped the adoption of package managers like Composer, as any external dependency could be easily called from anywhere in the application. The downside of this new function was that it also allowed attackers to instantiate any PHP class across the entire application thereby enabling the easier construction of gadget chains.

In order to build a chain, attackers use these so-called "magic" functions [13] that form the basis of their gadget chain. One of the functions that is widely used in POI exploits is the *destruct* function. In Section 3.5, we detailed the challenges in getting complete coverage of destructors in our tested applications. Accurate coverage of destructors also allows us to precisely analyze the impact of debloating on gadget creation.

Can debloating remove gadgets from external packages? Given the increased footprint of web applications due to their reliance on package managers and external dependencies, one may wonder about the possibility of abuse of these packages for the creation of gadgets. To measure the effect of debloating on Property-Oriented-Programming (POP) gadgets, we utilized the PHPGGC [17] library. PHPGGC (which stands for PHP Generic Gadget Chains) contains a list of known gadgets in popular PHP packages such as Doctrine, Symfony, Laravel, Yii and Zend-Framework. When a vulnerable PHP application includes any of the packages listed in PHPGGC, the attackers can generate gadget chains to achieve RCE, arbitrary file writes, and SQL injections.

Table 4: *List of packages with known POP gadget chains*

Application	Package	Removed by Debloating	
		File	*Function*
phpMyAdmin 4.7.0	Doctrine	✔	✔
	Guzzle	✔	✔
MediaWiki 1.28.0	Monolog	✔	✔
Magento 2.0.5	Doctrine	✔	✔
	Monolog	✖	✔
	Zendframework	✖	✔

We analyzed the available gadget chains in PHPGGC and checked whether any of our tested PHP applications included these chains. Table 4 summarizes the presence of each gadget and whether debloating removes them or not. WordPress is not included in this table because it does not rely on external packages. This does not make WordPress immune to POI attacks, but universally known gadget chains in popular external packages can not be used to exploit WordPress. For the affected applications, file-level debloating removes 4/6 gadgets while function debloating removes 6/6 available gadget chains. This again demonstrates the power of debloating which can not only remove some fraction of vulnerabilities but also make the exploitation of the remaining ones harder by removing the gadgets that attackers could abuse during a POI attack.

5.4.3 Utilizing development packages in production

During our analysis of external packages, we identified yet another source of bloat in new versions of web applications. When declaring external dependencies through Composer, two options are available: "require" and "require-dev". The first option indicates packages that are mandatory for the application to run properly. The second lists packages that should only be used in development environments, such as, packages providing

support for unit testing, performance analysis, and profiling. We discovered that applications downloaded from official websites often include these development packages. As such, when these packages are used to deploy web applications in production mode, they will contain unnecessary development libraries. This does not only increase the attack surface by having unnecessary code bloating the application, but can also lead to exploitation for misconfigured applications.

CVE-2017-9841 presents one example of such a vulnerability [25]. Specifically, this CVE refers to an RCE attack in specific versions of the PHPUnit library, which is a popular unit testing library for PHP. By default, Composer places all external packages under "vendor" directory. If this specific directory happens to be accessible through a misconfiguration of the server, PHPUnit files are then accessible and can be exploited to conduct an RCE attack.

The four web applications that we evaluated for this study, present different behaviors with respect to development packages. WordPress does not rely on external packages downloaded through Composer. MediaWiki never included development packages in its releases. phpMyAdmin had them in version 4.7.0 but stopped including them in version 4.8.3 (the latest at the time of writing). Magento started including them from version 2.0 and still includes them today. We have reached out to Magento and informed them about this issue.

5.5 Qualitative analysis of the removed code

In the previous sections, we analyzed the effects of debloating on the source code of applications from a software-engineering perspective (i.e. LLOC and Cyclomatic Complexity reduction) as well as from a security standpoint (i.e. number of CVEs and gadgets removed). At the same time, one may wonder what exactly was removed from each application during the process of debloating.

Given that thousands of files were removed, manually analyzing each file does not scale. As such, we turn to NLP techniques that allow us to cluster the removed files together and provide us with hints about the nature of each cluster. Specifically, we use the k-means clustering algorithm based on text vectors extracted from removed file names and file paths. Each file path includes directories that indicate which library or package, the file belongs to. For most modern web applications, this allows for a reasonable separation of files across different application plugins and modules. To end up with meaningful clusters, we tuned TFIDF vectorizer parameters along with the number of k-means clusters. We used the TFIDF maximum frequency limit to ignore common terms appearing in more than 50% of the files. Depending on the size and modularity of the application, 10 to 20 clusters yielded the most instructive grouping of files.

Table 5 shows the categories of the three largest removed clusters from each web application. Across all four applications, we observe the removal of source code related to external packages (e.g. Symfony for phpMyAdmin, Elastica for MediaWiki, and Zendframework1 for Magento), followed by localization/theme files (e.g. twentyfourteen theme for WordPress), and unused database drivers. We provide more application-specific details of removed features in the next paragraphs.

Table 5: *Features and external packages with the most removed files after file debloating (removed features are marked in italic). Entries marked with ∗ are packages that are indirectly pulled by other "require-dev" packages (not used by core application) for the purpose of test coverage reporting and coding standard enforcement.*

Applications	Features/Packages with most files removed
phpMyAdmin 4.7.0	1) Guzzle [8]: "Generating API HTTP response" * 2) Symfony [20]: "Parsing configuration files" * 3) PHP_CodeSniffer [15]: "Enforcing coding standards" *
MediaWiki 1.28.0	1) *Messages & Languages* 2) Less.php [9]: "Generating CSS code" 3) Elastica [5]: "Elastic search interface used by extensions"
WordPress 4.7.1	1) Twentyfourteen theme [21] 2) Twentytwelve theme [22] 3-4) Also theme related 5) *Multi-site administration*
Magento 2.0.5	1) Zendframework1 [14]: "Generating web pages and database operations" 2) *Sales, Orders & Credit Memo* 3) *Internal framework filters & Views*

phpMyAdmin's removed features include the uploading of plugins, GIS visualizations, and unused file formats used in import/export (such as, Dia, EPS, PDF, SVG, and ZIP). In addition, debloating removed unused plugins and external packages which make up the top 3 features removed from this web application as shown in Table 5. phpMyAdmin version 4.6.0 and 4.7.0 include unit tests which are also removed by our system. The LLOC for the removed test files is less than 2% of the whole code base of the application.

MediaWiki provides an API to interact with the wiki which is separate from the regular web interface that users interact with. Most actions within this API, including queries, file upload, and non-default output formats for this API were removed. Top categories of removed files consist of localization of messages and language files in addition to external dependencies (Lines 2 and 3) as listed in Table 5. The debloating process also removes file-upload modules which are disabled, by default, in MediaWiki. It is important to note that even if a module is "disabled," the code still resides on the server and could be abused by specific types of attacks. For example, in a recent attack against a WordPress plugin, the vulnerability could be exploited even if that plugin was disabled [32]. Debloating *removes* the source code of disabled and unused features and therefore does not suffer from this type of attack. Finally, the process of debloating, removed unused extensions of Mediawiki (e.g. citation, input box, pdf handler, poem and syntax highlighting). Mediawiki 1.19.1 and 1.28.0 include unit tests, and they measure less than 1.5% of LLOC in the whole code base of their respective versions.

WordPress takes a slightly different approach where the core functionality is concentrated in a relatively small number

Table 6: Verifying exploitability of vulnerabilities by testing exploits against original & debloated web applications

CVE	Target Software	Exploit Successful?	
		Original	Debloated
CVE-2013-3238	phpMyAdmin 4.0.0	✔	✔
CVE-2016-5734	phpMyAdmin 4.4.0	✔	✖
CVE-2014-1610	MediaWiki 1.21.1	✔	✔
CVE-2017-0362	MediaWiki 1.28.0	✔	✖
CVE-2018-20714	WordPress 3.9	✔	✔
CVE-2015-5731	WordPress 4.2.3	✔	✔
CVE-2016-4010	Magento 2.0.5	✔	✖
CVE-2018-5301	Magento 2.0.5	✔	✖

of large PHP files. The removed features of WordPress include installation files, unused modules (FTP, multi-site, user registration), disabled themes and update files (note that we could not exercise update files during our tests because this would change the version of the evaluated web application and create inconsistencies in our analysis of removed CVEs). In terms of testing, the installation files that we obtained from the WordPress website do not contain any unit tests.

Magento consists of both external packages and internal modules. We observed that various internal modules were removed, including an XML API for mobile, wishlists, ratings, and specific payment modules (such as, Paypal). Since many packages and internal modules include the terms "sales," "orders," and "tax," these individual files across multiple modules were clustered into the same category by k-means. Finally, Magento 1.9.0 does not include unit tests while the test files included in Magento 2.0.5 and its external packages measure up to 15% of its code base. For Magento 2.0.5, Zendframework1 which is an external dependency has most of its files removed by debloating.

5.6 Testing debloated web applications against real exploits

To ensure the correct mapping of CVEs to source code and the ability of debloating to stop real attacks, we collected 4 exploits available in the Metasploit framework and augmented them with 4 POCs that we developed based on public bug-tracker records and vulnerability details. After verifying that we can successfully exploit the original versions of the evaluated web applications, we tested the same exploits on the debloated versions. Half of the previously successful exploits failed because the vulnerable code was removed during the process of debloating. Table 6 lists the tested exploits against original and debloated applications.

As before, this demonstrates that while debloating is not a panacea against all possible issues, it can substantially improve the security of web applications. Finally, we present a demonstration of CVE-2016-4010 on Magento 2.0.5 in the following video: https://vimeo.com/328225679.

Table 7: Measurements of the execution time, the CPU and memory consumption for the tested web applications with XDebug and Code Coverage (CC) and without XDebug. The reported values for the CPU and memory correspond to the average for each application.

Application		Execution (s)	CPU (%)	Memory (%)
Magento 2.0.5	*Without XDebug*	317	21.7	10.7
	With CC	584 (x1.85)	56.9 (x2.62)	11.82 (x1.10)
MediaWiki 1.2.8	*Without XDebug*	36	30.7	5.2
	With CC	121 (x3.38)	79.3 (x2.58)	6.9 (x1.31)
phpMyAdmin 4.7.0	*Without XDebug*	102	3.7	5.7
	With CC	116 (x1.14)	31.5 (x8.47)	5.6 (x0.97)
WordPress 4.7.1	*Without XDebug*	68	8.2	8.2
	With CC	170 (x2.50)	42.6 (x5.22)	12.5 (x1.53)

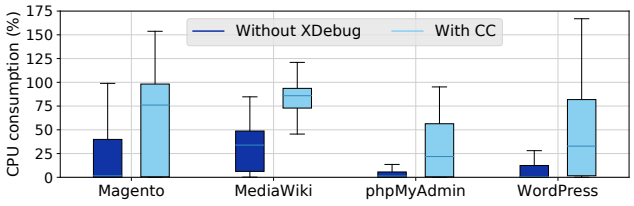


Figure 6: Measurement of the CPU consumption for the tested web applications. 100% corresponds to the use of a single CPU core.

6 Performance analysis

It is known that code-coverage tools impose a non-negligible overhead on web applications [65]. In this section, we report on the results of conducting all the Selenium tests with and without `XDebug` (our chosen PHP profiler) while measuring execution time, and recording server-side CPU usage and memory consumption. Table 7 presents the overall results and Figure 6 focuses on CPU consumption.

First, looking at the execution time, we can see that code coverage has a varying impact on the tested web applications. On one hand, phpMyAdmin is lightly affected with a 14% increase. On the other hand, the time it takes to run all tests for MediaWiki has tripled. For CPU consumption, the overhead is noticeable and all applications at least double their use of resources when code coverage is active. phpMyAdmin is exhibiting the biggest performance hit with a reported average almost 9 times higher than the one from the base application. Figure 6 shows that all median values are higher for applications with `XDebug` and most applications, at some point, require a second core with values above 100%. Finally, in terms of memory consumption, the server-side code profiler incurs a relatively modest increase for most applications. The worst overhead is observed when evaluating WordPress with an increase of 4.3% of the total device memory (16GB), i.e., an additional 700MB of RAM.

Even though our results show that the overall overhead is substantial, it is important to note that this overhead is not the overhead of the debloated web applications. Debloated web applications do not require code-coverage statistics and will therefore execute in the exact same environment as the original application (i.e. without `XDebug`). Depending on how code-coverage infor-

mation is obtained, this overhead may or may not be an issue. For example, if the coverage is calculated in an offline fashion where traces of application usage are replayed against a testing system, this overhead will have no impact on the real production systems. To allow for the online computation of code coverage (using real-time user traffic), we need more optimized code profilers. For example, `XDebug` currently overloads 43 opcodes to obtain line-level code-coverage information that is more fine-grained than required by our debloating techniques and incurs an unnecessary performance overhead [64]. We leave the development and evaluation of faster code profilers for future work.

7 Limitations and future work

In this study, we set out to precisely quantify the security benefits of debloating, when applied to web applications. Through a series of experiments, we demonstrated that debloating web applications has a number of very concrete advantages. We showed that debloating can, on average, decrease an application's code base by removing hundreds of thousands of lines of code, reduce its cyclomatic complexity by 30-50% and remove code associated with up to half of historical CVEs. Moreover, even for vulnerabilities that could not be removed, debloating can remove gadgets that makes their exploitation significantly harder. Next, we discuss some of the inherent and technical limitations of our approach and future direction.

Lack of available exploits: The number of exploits publicly available compared to the total number of registered CVEs is low. At the same time, the effort to study vulnerability reports, find the relevant patch or bug report, and track the actual vulnerability down to source code level takes a non-negligible amount of manual labor. This lack of available exploits limits our ability to test the exploitability of vulnerabilities before debloating since certain vulnerabilities might only be exploitable under specific configurations. For example the set of five file-upload-related vulnerabilities in our MediaWiki dataset (marked as gray in Table 9) require access to file upload functionality which is disabled by default. A maintained set of automated, replayable exploits against popular web applications similar to "BugBox" introduced by Nilson et al. in 2013, could substantially help researchers at this step [57].

To address this issue, we mapped the CVEs to features within those applications. This is done by studying the architecture of target applications based on documentation within the code and available on their websites. We marked a CVE as unexploitable if the underlying feature is disabled by default, and online tutorials in our dataset do not require users to enable that functionality. This limitation only applies to reported numbers on removed CVEs and does not affect our results on POI gadgets since their mere existence is enough for them to be used in gadget chains.

Our approach results in lower bounds for CVE removal since disabling modules through application configuration does not guarantee removal of all code paths that trigger those modules. Taking CVE-2019-6703 as an example, a vulnerability was discovered in the WordPress "Total Donations" plugin [32] and disabling this plugin did not prevent attackers from invoking the vulnerable end point and running their exploits.

Dynamic code coverage: Given our reliance on dynamic code-coverage techniques, it is clear that the success of debloating a web application is tightly related to its usage profile. Even though we constructed profiles in a way that is reproducible and unbiased (i.e. by relying on external popular tutorials, monkey testing, crawlers, and vulnerability scanners), we cannot claim that real web users would not trigger code that was removed during the stage of debloating, while they are interacting with a debloated web application.

More specifically, our modeled usage profiles do not cover all possible benign states of target web applications as we assume that users do not use all available features. Our intuition behind debloating proves to be successful to a large degree since removing unnecessary features brings clear security improvements. At the same time, our current usage model may not cover deep error states (e.g. logical errors in multi-stage form submissions, or the invalid structure of uploaded files). As such, we intend to follow-up this work with crowd sourcing and user studies to understand how administrators, developers, and regular users utilize the evaluated web applications and whether their usage profiles would allow for similar levels of debloating.

Due to nature of our approach, we can not take advantage of standard static-analysis techniques, since we aim to remove the features that are not useful for a given set of users, not those that are not reachable by other code. Using static analysis would greatly overestimate the code that needs to be maintained through the process of debloating and the resulting web application would contain code (and therefore vulnerabilities) that is not useful to all users. Going forward, we envision a hybrid approach where dynamic analysis is used as a first step to identify the core features that are useful for a specific set of users. These features can then be used as a starting point for a follow-up static analysis phase to ensure that all code related to these features is maintained when debloating a web application.

Handling requests to removed code: A separate issue is that of handling requests to removed code. Our current prototype utilizes assertions to log these requests so that we can investigate why the corresponding server-side code was not captured by our coverage profiler. When real users utilize debloated web applications, one must decide how these failures (i.e. client-side requests requiring server-side code that was removed) will be handled. Assuming that cleanly exiting the application and showing an error to the user is not sufficient, we need methods to authenticate the user's request, determine whether the request is a benign one (and not a malicious request that aims to exploit the debloated web application) and potentially re-introduce the removed code. The client/server architecture of web applications lends itself well to this model since the web server can decide to re-introduce debloated code and handle the user's request, without any knowledge of this happening from the side of

the user. All of this, however, requires server-side systems to introduce the code at the right time and for the appropriate users. We leave the design of such systems for future work.

Metrics to measure debloating effectiveness: In this paper, we use Cyclomatic Complexity (CC), Logical Lines of Code (LLOC), reduction in historical CVEs, and POP gadget reduction as four metrics to measure the effects of debloating on different web applications. However, not every line of code contributes equally to a program's attack surface. For example, 15% of removed files from Magento 2.0.5 are test files for external packages and the core of the application. Such code may not be directly exploitable or used in a POP chain unless there is a misconfiguration (e.g., autoloading including these files, or the directories being publicly accessible). As such, the resulted reduction in source code metrics (CC and LLOC) may also reflect the code that does not contribute to the attack surface. Contrastingly, the reduction of exploitable CVEs draws a more realistic picture of real world attacks. The drawback of this metric is its unavailability for proprietary software and the manual effort required to map CVEs to source code and verify their exploitability.

Debloating effectiveness: Through our debloating experiments we discovered that, in terms of debloating, not all applications are "equal." Modular web applications debloat significantly better than monolithic ones (such as Wordpress). We hope that our findings will inspire different debloating strategies that lend themselves better to monolithic web applications which resist our current function-level and file-level debloating strategies.

8 Related work

Over the years, different approaches that target very different parts of the software stack have been studied in the context of software debloating.

8.1 Debloating for the web

Despite the importance of the web platform, there has been very little work that attempts to apply debloating to it. Snyder et al. investigated the costs and benefits of giving websites access to all available browser features through JavaScript [70]. The authors evaluated the use of different JavaScript APIs in the wild and proposed the use of a client-side extension which controls which APIs any given website would get access to, depending on that website's level of trust. Schwarz et al. similarly utilize a browser extension to limit the attack surface of Chrome and show that they are able to protect users against microarchitectural and side-channel attacks [66]. These studies are orthogonal to our work since they both focus on the client-side of the web platform, whereas we focus on the server-side web applications.

Boomsma et al. performed dynamic profiling of a custom web application (a PHP application from an industry partner) [34]. The authors measured the time it takes for their dynamic profile system to get complete coverage and the percentage of files that they could remove. Since the application was a custom one, the authors were not able to report specifics in terms of the reduction of the programs attack surface, as that relates to CVEs. Contrastingly, by focusing on popular web applications, and utilizing function-level as well as file-level debloating, we were able to precisely quantify the reduction of vulnerabilities, both in terms of known CVEs as well as gadgets for PHP object-injection attacks.

8.2 Debloating in other platforms

Regehr et al. developed *C-Reduce* which is a tool that works at the source code level [63]. It performs reduction of C/C++ files by applying very specific program transformation rules. Sun et al. designed a framework called *Perses* that utilizes the grammar of any programming language to guide reduction [71]. Its advantage is that it does not generate syntactically invalid variants during reduction so that the whole process is made faster.

Heo et al. worked on *Chisel* whose distinguishing feature is that it performs fine-grained debloating by removing code even on the functions that are executed, using reinforcement learning to identify the best reduced program [42].

All three aforementioned approaches are founded on Delta debugging [76]. They reduce the size of an application progressively and verify at each step if the created variant still satisfies the desired properties.

Sharif et al. proposed *Trimmer*, a system that goes further than simple static analysis [68]. It propagates the constants that are defined in program arguments and configuration files so that it can remove code that is not used in that particular execution context. However, their system is not particularly well suited for web applications where we remove complete features. Our framework goes beyond this contextual analysis by mapping what is actually executed by the application.

Other works include research that revolves mainly around static analysis to remove dead code. Jiang et al. looked at reducing the bloat of Java applications with a tool called *JRed* [45]. Jiang et al. also designed *RedDroid* to reduce the size of Android applications with program transformations [44]. Quach et al. adopted a different approach by bringing dead-code elimination benefits of static linking to dynamic linking [61].

Rastogi et al. looked at debloating a container by partitioning it into smaller and more secure ones [62]. They perform dynamic analysis on system-call logs to determine which components and executables are used in a container, in order to keep them. Koo et al. proposed configuration-driven debloating [50]. Their system removes unused libraries loaded by applications under a specific configuration. They test their system on Nginx, VSFTPD, and OpenSSH and show a reduction of 78% of code from Nginx libraries is possible based on specific configurations.

9 Conclusion

In this paper, we analyzed the impact of removing unnecessary code in modern web applications through a process called

software debloating. We presented the pipeline details of the end-to-end, modular debloating framework that we designed and implemented, allowing us to record how a PHP application is used and what server-side code is triggered as a result of client-side requests. After retrieving code-coverage information, our debloating framework removes unused parts of an application using file-level and function-level debloating.

By evaluating our framework on four popular PHP applications (phpMyAdmin, MediaWiki, Magento, and WordPress) we witnessed the clear security benefits of debloating web applications. We observed a significant LLOC decrease ranging between 9% to 64% for file-level debloating and up to an additional 24% with function-level debloating. Next, we showed that external packages are one of the primary source of bloat as our debloating framework was able to remove more than 84% of unused code in versions that used Composer, PHP's most popular package manager. By quantifying the removal of code associated with critical CVEs, we observed a reduction of up to 60% of high-impact, historical vulnerabilities. Finally, we showed that the process of debloating also removes instructions and classes that are the primary sources for attackers to build gadgets and perform POI attacks.

Our results demonstrate that debloating web applications provides tangible security benefits and therefore should be seriously considered as a practical way of reducing the attack surface of web-applications deployments.

Acknowledgements: We thank our shepherd Giancarlo Pellegrino and the anonymous reviewers for their helpful feedback. This work was supported by the Office of Naval Research (ONR) under grants N00014-16-1-2264 and N00014-17-1-2541, as well as by the National Science Foundation (NSF) under grants CNS-1813974 and CMMI-1842020.

10 Availability

The main purpose of our work is to quantify the security benefits of debloating web applications, allowing the community to have informed discussions about the advantages of debloating, without the need of vague references to attack-surface reduction. To ensure the repeatability of our findings and to motivate more research in this area, *all* developed code and data artifacts are publicly available at: https://debloating.com.

References

[1] Automatically append or prepend files in a PHP script. https://www.php.net/manual/en/ini.core.php#ini.auto-append-file.

[2] Burp Suite web vulnerability scanner. https://portswigger.net/burp.

[3] Cross-Site Request Forgery (CSRF) - OWASP. https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF).

[4] Cross-site Scripting (XSS) - OWASP. https://www.owasp.org/index.php/Cross-site_Scripting_(XSS).

[5] Elastica: Elasticsearch client. https://github.com/ruflin/Elastica.

[6] Greasemonkey. https://www.greasespot.net/.

[7] gremlins.js. https://github.com/marmelab/gremlins.js.

[8] Guzzle: PHP HTTP client. https://github.com/guzzle/guzzle.

[9] less.js ported to PHP. https://github.com/oyejorge/less.php.

[10] PHP autoload built-in function. http://php.net/manual/en/language.oop5.autoload.php.

[11] PHP Object Injection Vulnerability. https://www.owasp.org/index.php/PHP_Object_Injection.

[12] PHP: register_shutdown_function - Manual. https://secure.php.net/manual/function.register-shutdown-function.php.

[13] PHP wakeup built-in function. http://php.net/manual/en/language.oop5.magic.php#object.wakeup.

[14] PHP Zend Framework 1. https://github.com/zendframework/zf1.

[15] PHP_CodeSniffer is a PHP package that tokenizes PHP, JavaScript and CSS files and detects violations of a defined set of coding standards. https://github.com/squizlabs/PHP_CodeSniffer.

[16] phpdbg PHP Debugger. https://github.com/krakjoe/phpdbg.

[17] PHPGGC: PHP Generic Gadget Chains. https://github.com/ambionics/phpggc.

[18] Remote Code Execution Vulnerability — Netsparker. https://www.netsparker.com/blog/web-security/remote-code-evaluation-execution/.

[19] SQL Injection: OWASP. https://www.owasp.org/index.php/SQL_Injection.

[20] Symfony PHP framework. https://github.com/symfony/symfony.

[21] WordPress Twenty Fourteen theme. https://wordpress.org/themes/twentyfourteen/.

[22] WordPress Twenty Twelve theme. https://wordpress.org/themes/twentytwelve/.

[23] XDebug Debugger and Profiler Tool for PHP. https://xdebug.org/.

[24] xhprof function-level hierarchical profiler for PHP. https://github.com/phacility/xhprof.

[25] NVD - CVE-2017-9841 (PHPUnit vulnerability). https://nvd.nist.gov/vuln/detail/CVE-2017-9841, 2017.

[26] Drupal Core - 3rd-party libraries -SA-CORE-2018-005 — Drupal.org. https://www.drupal.org/SA-CORE-2018-005, 2018.

[27] [HttpFoundation] Remove support for legacy and risky HTTP headers - Symfony framework on GitHub. https://github.com/symfony/symfony/commit/e447e8b92148ddb3d1956b96638600ec95e08f6b#diff-9d63a61ac1b3720a090df6b1015822f2R1694, 2018.

[28] NVD - CVE-2018-14773 (Symfony vulnerability). https://nvd.nist.gov/vuln/detail/CVE-2018-14773, 2018.

[29] Packagist statistics. https://packagist.org/statistics, 2018.

[30] PyPI Stats. https://pypistats.org/packages/__all__, 2018.

[31] Security Advisory: URL Rewrite vulnerability (Zend Framework). https://framework.zend.com/security/advisory/ZF2018-01, 2018.

[32] WordPress sites under attack via zero-day in abandoned plugin — ZDNet. https://www.zdnet.com/article/wordpress-sites-under-attack-via-zero-day-in-abandoned-plugin/, 2019.

[33] Barth, A., Jackson, C., and Mitchell, J. C. Robust Defenses for Cross-site Request Forgery. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (NY, USA, 2008), CCS, ACM.

[34] Boomsma, H., Hostnet, B. V., and Gross, H. Dead code elimination for web systems written in PHP: Lessons learned from an industry case. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)* (Sept 2012).

[35] Brumley, D., and Boneh, D. Remote Timing Attacks Are Practical. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12* (Berkeley, CA, USA, 2003), SSYM'03, USENIX Association.

[36] CVE Details: The ultimate security vulnerability datasource. https://www.cvedetails.com/.

[37] NIST: National Vulnerability Database. https://nvd.nist.gov/.

[38] DAHSE, J., KREIN, N., AND HOLZ, T. Code Reuse Attacks in PHP: Automated POP Chain Generation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2014), CCS '14, ACM.

[39] GILL, G. K., AND KEMERER, C. F. Cyclomatic complexity density and software maintenance productivity. *IEEE transactions on software engineering 17*, 12 (1991).

[40] GOETHEM, T. V., JOOSEN, W., AND NIKIFORAKIS, N. The Clock is Still Ticking: Timing Attacks in the Modern Web. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)* (2015).

[41] HALFOND, W. G., VIEGAS, J., ORSO, A., ET AL. A classification of SQL-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering* (2006), IEEE.

[42] HEO, K., LEE, W., PASHAKHANLOO, P., AND NAIK, M. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), ACM.

[43] HONG, G., YANG, Z., YANG, S., ZHANG, L., NAN, Y., ZHANG, Z., YANG, M., ZHANG, Y., QIAN, Z., AND DUAN, H. How You Get Shot in the Back: A Systematical Study About Cryptojacking in the Real World. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), CCS '18.

[44] JIANG, Y., BAO, Q., WANG, S., LIU, X., AND WU, D. RedDroid: Android Application Redundancy Customization Based on Static Analysis. In *Proceedings of the 29th IEEE International Symposium on Software Reliability Engineering (ISSRE18)* (2018).

[45] JIANG, Y., WU, D., AND LIU, P. JRed: Program Customization and Bloatware Mitigation Based on Static Analysis. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1.

[46] JOVANOVIC, N., KIRDA, E., AND KRUEGEL, C. Preventing cross site request forgery attacks. In *Securecomm and Workshops* (2006), IEEE.

[47] KIRDA, E., KRUEGEL, C., VIGNA, G., AND JOVANOVIC, N. Noxes: A Client-side Solution for Mitigating Cross-site Scripting Attacks. In *Proceedings of the 2006 ACM Symposium on Applied Computing* (New York, NY, USA, 2006), SAC '06, ACM.

[48] KOEHLER, W. A longitudinal study of Web pages continued: a consideration of document persistence. *Information Research 9*, 2 (2004).

[49] KONOTH, R. K., VINETI, E., MOONSAMY, V., LINDORFER, M., KRUEGEL, C., BOS, H., AND VIGNA, G. MineSweeper: An In-depth Look into Drive-by Cryptocurrency Mining and Its Defense. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), CCS '18.

[50] KOO, H., GHAVAMNIA, S., AND POLYCHRONAKIS, M. Configuration-driven software debloating. In *Proceedings of the 12th European Workshop on Systems Security* (New York, NY, USA, 2019), EuroSec '19, ACM.

[51] KURMUS, A., SORNIOTTI, A., AND KAPITZA, R. Attack surface reduction for commodity os kernels: Trimmed garden plants may attract less bugs. In *Proceedings of the Fourth European Workshop on System Security* (2011), EUROSEC '11.

[52] KURMUS, A., TARTLER, R., DORNEANU, D., HEINLOTH, B., ROTHBERG, V., RUPRECHT, A., SCHRÖDER-PREIKSCHAT, W., LOHMANN, D., AND KAPITZA, R. Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring. In *Proceedings of Network and Distributed Systems Security (NDSS)* (2013).

[53] MADHAVAPEDDY, A., AND SCOTT, D. J. Unikernels: Rise of the virtual library operating system. *Queue 11*, 11 (2013).

[54] MCCABE, T. J. A complexity measure. *IEEE Transactions on software Engineering*, 4 (1976).

[55] MCCONNELL, S. *Code complete*. Pearson Education, 2004.

[56] MISHRA, S., AND POLYCHRONAKIS, M. Shredder: Breaking Exploits through API Specialization. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)* (2018).

[57] NILSON, G., WILLS, K., STUCKMAN, J., AND PURTILO, J. Bugbox: A vulnerability corpus for PHP web applications. In *Presented as part of the 6th Workshop on Cyber Security Experimentation and Test* (Washington, D.C., 2013), USENIX.

[58] Magento: eCommerce Platform. https://magento.com/.

[59] MediaWiki: Free and Open Source Software Wiki . https://www.mediawiki.org/wiki/MediaWiki.

[60] phpMyAdmin: MySQL web administration. https://phpmyadmin.net/.

[61] QUACH, A., PRAKASH, A., AND YAN, L. K. Debloating Software through Piece-Wise Compilation and Loading. *Proceedings of USENIX Security* (2018).

[62] RASTOGI, V., DAVIDSON, D., DE CARLI, L., JHA, S., AND MCDANIEL, P. Cimplifier: Automatically Debloating Containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2017), ESEC/FSE 2017, ACM.

[63] REGEHR, J., CHEN, Y., CUOQ, P., EIDE, E., ELLISON, C., AND YANG, X. Test-case Reduction for C Compiler Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2012), PLDI '12, ACM.

[64] RETHANS, D. Code Coverage: The Present. https://derickrethans.nl/code-coverage.html.

[65] RETHANS, D. Xdebug's Code Coverage speedup. https://derickrethans.nl/xdebug-codecoverage-speedup.html.

[66] SCHWARZ, M., LIPP, M., AND GRUSS, D. JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks. *Ndss*, February (2018).

[67] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (2007), ACM.

[68] SHARIF, H., ABUBAKAR, M., GEHANI, A., AND ZAFFAR, F. TRIMMER: Application Specialization for Code Debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (NY, USA, 2018), ASE 2018, ACM.

[69] SHIN, Y., AND WILLIAMS, L. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement* (2008), ACM.

[70] SNYDER, P., TAYLOR, C., AND KANICH, C. Most Websites Don'T Need to Vibrate: A Cost-Benefit Approach to Improving Browser Security. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2017), CCS '17, ACM.

[71] SUN, C., LI, Y., ZHANG, Q., GU, T., AND SU, Z. Perses: Syntax-guided Program Reduction. In *Proceedings of the 40th International Conference on Software Engineering* (New York, NY, USA, 2018), ICSE '18, ACM.

[72] VOGT, P., NENTWICH, F., JOVANOVIC, N., KIRDA, E., KRUEGEL, C., AND VIGNA, G. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *NDSS* (2007), vol. 2007.

[73] VOSS, L. The State of JavaScript Frameworks. https://www.npmjs.com/npm/the-state-of-javascript-frameworks-2017-part-2-the-react-ecosystem, 2018.

[74] WANG, W., FERRELL, B., XU, X., HAMLEN, K. W., AND HAO, S. SEISMIC: SEcure In-lined Script Monitors for Interrupting Cryptojacks. In *European Symposium on Research in Computer Security* (2018), Springer.

[75] WordPress: OpenSource Content Management System. https://wordpress.com/.

[76] ZELLER, A., AND HILDEBRANDT, R. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng. 28*, 2 (Feb. 2002).

Table 8: *Comprehensive list of tutorials collected from the first two pages of Google search results*

phpMyAdmin	
A	https://www.siteground.com/tutorials/phpmyadmin/
A	https://www.reg.ca/faq/PhpMyAdminTutorial.html
A	https://www.w3resource.com/mysql/administration-tools/phpmyadmin-tutorial.php
A	https://code.tutsplus.com/tutorials/installing-and-using-phpmyadmin-for-web-development–cms-21947
A	https://www.homeandlearn.co.uk/php/php12p2.html
A	https://www.wpbeginner.com/beginners-guide/beginners-guide-to-wordpress-database-management-with-phpmyadmin/
A	http://members.ipage.com/knowledgebase/read_article.bml?kbid=5923
A	https://www.digitalocean.com/community/tutorials/how-to-install-and-secure-phpmyadmin-on-ubuntu-16-04
A	https://www.fastwebhost.com/tutorials/knowledge-base/phpmyadmin-tutorial-administration-2/
A	https://www.tutorialspoint.com/cpanel/cpanel_phpmyadmin.htm
A	https://www.w3schools.com/php/php_mysql_intro.asp
A	https://pimylifeup.com/raspberry-pi-mysql-phpmyadmin/
A	https://www.webhostface.com/kb/knowledgebase/mysql-search-replace/
A	https://www.eukhost.com/web-hosting/phpmyadmin.php
MediaWiki	
A	https://www.siteground.com/tutorials/mediawiki/
A	http://helpwiki.evergreen.edu/wiki/index.php/Mediawiki_Tutorial
A	https://lifehacker.com/5396832/customize-mediawiki-into-your-ultimate-collaborative-web-site
A	https://hepmdb.soton.ac.uk/wiki/images/0/0b/Open4a-Getting-Started-with-mediawiki.pdf
A	https://www.fastwebhost.com/tutorials/cat/mediawiki-tutorial/
A	https://www.semantic-mediawiki.org/wiki/Help:Getting_started
A	https://www.inmotionhosting.com/support/edu/mediawiki/getting-started-mediawiki
A	https://www.hostknox.com/tutorials/mediawiki/installation
A	https://www.digitalocean.com/community/tutorials/how-to-install-mediawiki-on-ubuntu-14-04
A	https://computers.tutsplus.com/tutorials/how-to-build-your-own-wiki–cms-19772
A	https://www.tmdhosting.com/tutorials/mediawiki/how-to-backup-mediawiki.html
Magento	
A	https://www.tutorialspoint.com/magento/
A	https://www.siteground.com/tutorials/magento/
A	https://blog.magestore.com/magento-tutorial/
A	https://www.cminds.com/the-ultimate-beginners-guide-to-magento/
A	https://code.tutsplus.com/articles/from-beginner-to-advanced-in-magento-introduction-installation–cms-21969
A	https://www.simicart.com/blog/best-magento-tutorial-resources-beginner/
A	https://www.cloudways.com/blog/magento/
A	https://magenticians.com/
A	https://www.mageplaza.com/kb/magento-2-tutorial/
A	https://devdocs.magento.com/guides/m1x/magefordev/mage-for-dev-1.html
A	https://u.magento.com/
A	https://stuntcoders.com/magento-tutorials/magento-tutorial-for-beginners/
WordPress	
A	https://codex.wordpress.org/WordPress_Lessons
A	https://www.000webhost.com/wordpress-tutorial
A	https://wpapprentice.com/wordpress-tutorial/
A	https://premium.wpmudev.org/blog/a-wordpress-tutorial-for-beginners-create-your-first-site-in-10-steps/
A	https://ithemes.com/tutorial/category/wordpress-101/
A	https://easywpguide.com/wordpress-manual/
A	https://www.siteground.com/tutorials/wordpress/
A	https://www.tutorialspoint.com/wordpress/
A	https://www.hostinger.com/tutorials/wordpress/

Table 9: *Comprehensive list of mapped CVEs and whether vulnerable files, functions or lines were triggered based on our usage profiles. Grey rows indicate CVEs located in modules that are, by default, disabled.*

#	CVE	Ver.	Vulnerability Triggered			Affected Functionality
			Files	Functions	Lines	
phpMyAdmin						
1	CVE-2013-3238	4.0.0	✔	NA	✖	Rename table using Regex
2	CVE-2013-3240	4.0.0	✔	✔	✔	Plugins
3	CVE-2014-8959	4.0.0	✖	✖	✖	GIS Editor
4	CVE-2016-6609	4.0.0	✔	✖	✖	Export as phparray
5	CVE-2016-6619	4.0.0	✔	✖	✖	Recent tables
6	CVE-2016-6620	4.0.0	✖	✖	✖	Table tracking
7	CVE-2016-6628	4.0.0	✖	✖	✖	Create charts
8	CVE-2016-6629	4.0.0	✖	✖	✖	Configuration option
9	CVE-2016-6631	4.0.0	✖	✖	✖	Create transform plugins
10	CVE-2016-6633	4.0.0	✔	✖	✖	Import ESRI shape file
11	CVE-2016-9866	4.0.0	✔	NA	✖	User preferences
12	CVE-2016-5703	4.4.0	✔	✖	✖	Central columns
13	CVE-2016-5734	4.4.0	✔	✖	✖	Table search using Regex
14	CVE-2016-6616	4.4.0	✖	✖	✖	User groups
15	CVE-2017-1000017	4.4.0	✔	✔	✖	Replication
16	CVE-2016-6606	4.6.0	✔	✔	✔	Authentication cookies
17	CVE-2016-6617	4.6.0	✔	✖	✖	Export templates
18	CVE-2016-9849	4.6.0	✔	✔	✔	Authentication
19	CVE-2016-9865	4.6.0	✔	NA	✖	Core deserialization
20	CVE-2017-1000499	4.7.0	✔	✔	✔	Navigation tree
MediaWiki						
21	CVE-2013-2114	1.19.1	✔	✖	✖	File upload from chunks
22	CVE-2013-6453	1.21.1	✔	✖	✖	Verify uploaded file
23	CVE-2014-1610	1.21.1	✔	✖	✖	PDF Upload
24	CVE-2014-2243	1.21.1	✔	✔	✖	User settings
25	CVE-2014-5241	1.21.1	✔	✖	✖	JSON Output formatter
26	CVE-2014-9277	1.21.1	✔	✖	✖	Flash policy output
27	CVE-2014-9276	1.23.0	✔	✔	✔	Expand templates
28	CVE-2015-2936	1.24.0	✔	✔	✔	Authentication
29	CVE-2015-2937	1.24.0	✖	✖	✖	XMP data reader
30	CVE-2015-6728	1.24.0	✔	✖	✖	Get watchlists through API
31	CVE-2015-8002	1.24.0	✔	✖	✖	File upload from chunks
32	CVE-2015-8003	1.24.0	✔	✖	✖	File upload API
33	CVE-2015-8623	1.24.0	✖	✖	✖	User object
34	CVE-2015-8624	1.24.0	✖	✖	✖	User object
35	CVE-2017-0370	1.24.0	✔	✔	✔	Markup parser (blacklist)
36	CVE-2017-0362	1.28.0	✔	✔	✔	Track pages
37	CVE-2017-0363	1.28.0	✔	✔	✔	Search
38	CVE-2017-0364	1.28.0	✔	✔	✔	Search
39	CVE-2017-0367	1.28.0	✔	✔	✔	Localization cache
40	CVE-2017-0368	1.28.0	✔	✔	✔	System messages
41	CVE-2017-8809	1.28.0	✔	✔	✔	APIs and RSS
Magento						
42	CVE-2015-1397	1.9.0	✔	✔	✔	Prepare SQL condition
43	CVE-2015-1398	1.9.0	✔	✔	✖	OAuth & XML modules
44	CVE-2015-1399	1.9.0	✔	✔	✔	Actions predispatch
45	CVE-2015-8707	1.9.0	✔	✖	✖	Password reset
46	CVE-2016-2212	1.9.0	✔	✖	✖	Order status RSS
47	CVE-2016-4010	2.0.5	✔	✔	✔	Shopping cart
48	CVE-2016-6485	2.0.5	✔	✔	✔	Cryptography functions
49	CVE-2018-5301	2.0.5	✖	✖	✖	Delete customer address
WordPress						
50	CVE-2014-5203	3.9	✔	✔	✖	Widget customization
51	CVE-2014-5204	3.9	✔	✔	✔	CSRF token verification
52	CVE-2014-5205	3.9	✔	✔	✔	CSRF token verification
53	CVE-2018-12895	3.9	✔	✔	✔	Delete post thumbnail
54	CVE-2015-2213	4.0	✔	✔	✔	Untrash comment
55	CVE-2017-14723	4.0	✔	✔	✔	Prepared queries
56	CVE-2014-9033	4.0	✔	✔	✖	Password reset
57	CVE-2014-9037	4.0	✔	✔	✔	Password hashing library
58	CVE-2016-6635	4.0	✔	✖	✖	Ajax compression test
59	CVE-2014-9038	4.0	✔	✔	✔	HTTP request API
60	CVE-2015-5731	4.2.3	✔	✔	✖	Admin panel
61	CVE-2016-7169	4.6	✔	✔	✖	Sanitize uploaded file name
62	CVE-2017-17091	4.6	✔	NA	✖	Create new user
63	CVE-2017-5492	4.7	✔	✔	✔	Admin screen API, widgets
64	CVE-2017-9064	4.7	✔	✔	✔	Admin file system operations
65	CVE-2018-10101	4.7	✔	✔	✔	HTTP request API
66	CVE-2018-10100	4.7	✔	NA	✖	Login
67	CVE-2017-6815	4.7	✔	✔	✔	Redirect URL validation
68	CVE-2017-5611	4.7.1	✔	✔	✔	Query helper
69	CVE-2017-16510	4.7.1	✔	✖	✖	Prepared queries

The Web's Identity Crisis: Understanding the Effectiveness of Website Identity Indicators

Christopher Thompson, Martin Shelton, Emily Stark,
Maximilian Walker, Emily Schechter, Adrienne Porter Felt
Google

Abstract

Users must understand the identity of the website that they are visiting in order to make trust decisions. Web browsers indicate website identity via URLs and HTTPS certificates, but users must understand and act on these indicators for them to be effective. In this paper, we explore how browser identity indicators affect user behavior and understanding. First, we present a large-scale field experiment measuring the effects of the HTTPS Extended Validation (EV) certificate UI on user behavior. Our experiment is many orders of magnitude larger than any prior study of EV indicators, and it is the first to examine the EV indicator in a naturalistic scenario. We find that most metrics of user behavior are unaffected by its removal, providing evidence that the EV indicator adds little value in its current form. Second, we conduct three experimental design surveys to understand how users perceive UI variations in identity indicators for login pages, looking at EV UI in Chrome and Safari and URL formatting designs in Chrome. In 14 iterations on browsers' EV and URL formats, no intervention significantly impacted users' understanding of the security or identity of login pages. Informed by our experimental results, we provide recommendations to build more effective website identity mechanisms.

1 Introduction

To use the web safely, users must be able to understand the identity of the website that they are visiting. Without understanding a website's identity, users cannot make an informed decision about whether to provide it with their personal information or trust its content. Such misunderstandings result in common attacks like phishing and social engineering [36].

Web browsers use two mechanisms to communicate website identity to users. The first is the URL displayed in the browser address bar, along with a padlock icon to indicate an authenticated connection. For example, before a user types their Google password into a webpage, the user should verify that the domain in the browser address bar is "google.com" and that the padlock icon is present. Second, some HTTPS connections are authenticated with an Extended Validation (EV) certificate. An EV certificate associates a website with a legal entity, whose name and jurisdiction is typically displayed alongside the URL in the address bar (Figure 1). Users can check the EV indicator to verify that a website is associated with an established legal entity that they trust.

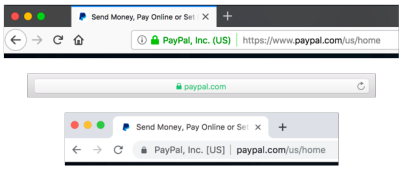


Figure 1: Examples of EV certificate UI in different web browsers (from top to bottom: Firefox, Safari, and Chrome).

Prior work suggests that neither URLs nor EV indicators work very well as indicators of website identity. Many users do not look at the URL even when primed to try to identify fraudulent sites [12]. EV indicators also do not help users identify fraudulent sites [24]. Even if users did notice the EV indicator, EV certificates can contain misleading information that limits their usefulness [10, 11].

However, much of the work that studies the effectiveness of website identity indicators is dated, testing browser UIs that are no longer in common use. For example, most EV indicator research is ten years old and studied the first browser EV UI from Internet Explorer 7. Browser security indicators and the web security landscape have changed dramatically since these studies were conducted, with widespread adoption of HTTPS [15] and constant evolution of browser UI [16]. Further, prior work does not examine how users react to website identity indicators in the wild.

In this paper, we examine the effectiveness of browser identity indicators – URLs and EV UI – from several angles. We focus primarily on how users react to the EV indicator, since it is designed to provide human-meaningful identity information, and we also investigate whether browser UIs can be tweaked to make the URL more human-meaningful. Our

goal is to study the effectiveness of modern browser identity indicators at a much larger scale than previous work.

First, we analyzed a large-scale field experiment (Section 3) from the Google Chrome web browser. We examined a suite of user behavior measurements with and without the EV indicator present on websites that serve EV certificates. This experiment simulates a situation in which a user visits an attack website that mimics a victim website exactly but does not possess an EV certificate for the victim site. We find little evidence that the absence of the EV indicator affects how users interact with the site. We do find, however, that the EV indicator itself draws clicks.

Second, we conducted a series of survey experiments (Section 4) to investigate follow-up questions about EV UI that we could not answer with field data. Our surveys, with over 1,800 total participants from the U.S. and U.K., sought to answer two questions: (1) Would a recent proof-of-concept attack on EV certificates [11] be effective on real users? (2) How do users react to other browsers' EV UIs? In these surveys, we find no evidence that the EV UI in either Chrome or Safari impact how comfortable users feel when logging into a webpage.

Finally, having found little evidence that EV indicators influence user behavior, we consider whether URLs can be more effective identity indicators (Section 5). We surveyed over 1,000 users to assess reactions to different variations on Chrome's URL display. Each variation was designed to draw users' attention to the domain name, in hopes that they would notice that the webpage was a phishing site. We found no significant differences among any of the variations, leading us to believe that a more radical redesign is necessary for URLs to effectively communicate website identity to users.

Our results, along with the body of existing work, suggest that modern browser identity indicators are not effective. We use these results to provide recommendations for building better website identity mechanisms. Based on promising results from prior research [7, 14], we argue that negative, active indicators (such as full-page warnings) are a more promising avenue than passive, positive indicators (such as the padlock icon or EV indicator in the address bar). We further recommend that user research should be incorporated into the design phase for future browser identity indicators.

2 Background

2.1 URLs and website identity

The fundamental identity indicator of the web is the URL. All major web browsers display the URL of the page in order to convey the website's identity. Figure 2 shows how different web browsers display URLs to users.

2.1.1 HTTPS and certificates

In the URL bar, the presence of the "https://" scheme and/or a padlock icon indicate that the identity of the site has been verified through a cryptographic certificate. Most websites use *domain-validated (DV) certificates*. A website owner can obtain a DV certificate by proving control over a domain to a certificate authority (CA) [1]. DV certificates can be obtained easily and inexpensively from a number of CAs.

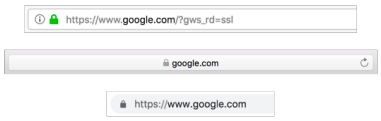


Figure 2: Examples of different browser URL displays (from top to bottom: Firefox 64, Safari 12.0.3, and Chrome 72).

2.1.2 Registrable domain

In most situations, a user who is making a security decision should pay attention to the *registrable domain* [6] rather than the full URL. The registrable domain is composed of the high-level domain name suffix under which internet users can register names [5], plus the DNS label immediately preceding it. For example, "google.com" is the registrable domain in the URL "https://accounts.google.com/", and "google.co.uk" in "https://accounts.google.co.uk/".

The registrable domain is typically the identity indicator of interest because when an organization controls a registrable domain, the same organization can typically control all the subdomains and paths within that domain.

2.2 Extended Validation (EV) certificates

EV certificates are a type of HTTPS certificate in which a domain owner undergoes additional validation with a CA to tie their domain to a legal entity. Most major web browsers display the legal entity name and jurisdiction in the URL bar alongside the URL, as shown in Figure 1. Notably, Safari recently stopped displaying the legal entity name and simply colors the domain green when an EV certificate is present.[1]

Section 2 of the CA/Browser Forum guidelines for EV certificates [3] specifically states that one primary purpose of EV certificates is to enable a browser to inform the user about the specific legal identity of the business with which they are interacting when using a website. A secondary purpose given by the guidelines is that EV certificates can be considered to combat phishing and other malicious web activity.[2] If an attack website is impersonating a victim business, the attack site is not supposed to be able to obtain an EV certificate for the victim business. A user visiting the attack website might notice that there is no EV indicator for the victim business and thereby conclude that the website is not the legitimate website for the victim business.

EV certificates are typically more cumbersome to obtain than DV certificates. Domain owners pay a premium for

[1] Apple did not mention this change in their release notes, however it was discussed on Twitter [13, 30] and technical blogs [23].

[2] Microsoft [18] and Mozilla [21] gave similar motivations for their introduction of browser EV UI.

EV certificates and undergo a days- or weeks-long validation process.

2.2.1 Weaknesses of EV certificates

EV certificates suffer from a number of security and usability weaknesses. These weaknesses have led to a vigorous debate in the security community about whether browsers should continue to display EV certificate UI [22, 23, 32]. Our work seeks to inform this debate with up-to-date, large-scale, in situ data about how users react to EV indicators.

Malicious EV certificates. EV is not intended to verify that the holder of the certificate is law-abiding, trustworthy, or safe [3]. Researchers have shown that EV certificates can be obtained for misleading names that could be useful in an attack. We describe these attacks in Section 7.1.2.

Usability issues. A body of work from the mid-2000s indicates that EV certificates are not an effective phishing defense because users do not pay attention to the EV UI in web browsers. We survey this work in Section 7.1.1.

Further, the legal entity names in EV certificates are not always intuitive or understandable, because the legal entity does not always match the company's user-visible brand. For example, the personal finance management site `mint.com` has a legal entity name of "Intuit Inc."

3 EV field experiment

To understand whether browser EV UI has an effect on user behavior in the wild, we analyze data from a large-scale field experiment. In this experiment, the EV UI was disabled for a subset of Google Chrome users. We compare a variety of metrics representing users' interactions with EV websites in the experimental and control groups. We do not find evidence that the EV UI impacts user behavior significantly for most metrics. The exception is that users who see the EV UI are more likely to open and interact with the Page Info bubble, which is anchored to the connection security indicator chip (Figure 3). Additionally, we examined the effects of removing the EV UI on a set of 20 top EV sites. We found a small negative impact on navigations to one of these sites.

3.1 Methodology

3.1.1 Dataset

We analyze data from Chrome's user metrics program. Chrome collects metrics in the form of enums, booleans, counts, and times. Our dataset comes from the Stable channel, which has the largest set of users and is the default installation release channel. Stable channel is considered the most representative for measurement and experimentation purposes.

Chrome metrics reports are pseudonymous, containing client information such as the operating system and country, but no personal information (e.g., age or gender). The user metrics program is enabled by default for consumer installs. Users may opt out during installation or in browser settings.

A subset of metrics are keyed by the URL on which the metric is recorded. URLs are only provided for users who have also opted to sync their browsing data with Google servers [20]. We use these URL-keyed metrics to check for changes in user behavior on specific well-known sites with EV certificates. We analyze a subset of our metrics on each of the top 20 EV sites, as visited by Chrome users during our experimental period. We report these results with the domains blinded.

Our dataset includes metrics collected from January 15, 2019 to January 28, 2019. Chrome 71 was fully rolled out to the Stable channel during this time period.[3]

3.1.2 Experimentation framework

Chrome contains an experiment framework in which users can be randomly assigned to experiment groups. Metrics reports are then tagged with the user's group. In our dataset, 1% of Stable channel users were assigned to an experimental group in which the EV certificate UI was disabled, and 1% to a control group. In the experimental group, users who visited EV sites saw a padlock but no additional HTTPS UI.

3.1.3 Metrics

In our study, we analyzed a set of user behaviors that we hypothesized might be affected by a user's perception of the security and identity of a site.

Our selection of metrics is informed by a review of related work (Section 7). We sought to measure behaviors in situ that previous work measured via lab studies or surveys. Below we describe each metric and why we included it.

- **Navigations.** Do users navigate to different sites, or navigate away from EV sites more in the experimental group? We measure:
  - The number of navigations to EV pages, normalized by the total number of navigations.
  - The median time spent on each page visit (not including time spent in the background).[4]
  - The number of times that users left EV pages, by closing the tab, using Back/Forward functionality, or reloading. We normalize by the total number of navigations to EV pages.

  We are interested in navigations because prior work on browser security indicators surveyed users on whether they would leave the page if a particular security indicator appeared in their browser [16].
- **Form submissions.** A metric is recorded when a user submits an HTML form. We consider the number of

[3] During this period, our dataset included millions of clients in each group for our main analysis. For our per-origin analysis, we had tens of thousands of clients on average in each group for each origin.

[4] Due to a bug in our initial data collection for this metric, we instead use data from May 15–28 (after a fixed version reached Chrome's Stable channel) for this metric.

form submissions that occur on pages with EV certificates, normalized by the total number of navigations to EV pages. Previous studies have surveyed users on their willingness to enter login [31] or credit card [16] details, both of which typically involve submitting a form.

- **Autofill interactions.** Chrome's autofill feature saves credit card details that the user enters and provides suggestions when users fill out payment forms. We analyze the number of times a suggestion was selected normalized by the number of times a suggestion was shown. As with form submissions, these metrics provide insight into whether users in the experimental group are less comfortable providing credit card details to the page.
- **Page Info interactions.** Chrome's Page Info bubble is the dialog that appears when a user clicks on the main connection security indicator in the address bar (Figure 3). A metric is recorded every time a user opens the Page Info bubble and every time they use its functionality (e.g., opening the certificate details dialog or inspecting cookies). We analyze Page Info behavior because previous lab studies have examined users' interactions with the equivalent dialog in other browsers [33]. We normalize the number of times the Page Info bubble was opened by the total number of navigations to EV pages. We normalize the number of different actions within the Page Info bubble by the total number of times the Page Info bubble was opened.
- **Downloads.** Downloading a file, particularly an executable, may represent a trust decision for users. We record the number of downloads initiated from EV pages, normalized by the total number of EV navigations.
- **Site Engagement.** Chrome records an aggregate metric called Site Engagement (SE) that approximately measures how much active time a user spends on a site.[5] Each web origin receives a SE score between 0 and 100. It goes up as a user clicks, scrolls, performs keypresses, or plays media on a site, and decays over time as a user does not interact with the site. We compare SE scores with and without the EV UI to see if there might be effects related to user engagement that are not captured by our other metrics. We analyze this metric on a per-site basis. For each user on each origin, we compute the average SE score per visit as well as the average change in SE score over each visit.

3.1.4 Analysis

To see if there are statistically significant effects on any of our metrics of user behavior between our control group and our experimental group, we perform a Welch's t-test for unequal sample variances (as our sample sizes and variances are not guaranteed to be equal between our treatment and control groups) for each metric of interest.

For each metric, we report the difference between the experimental group and the control group, the 95% confidence interval for the difference, and the p-value of the t-test.

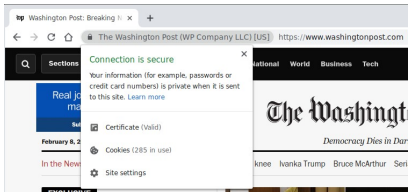


Figure 3: The Page Info bubble in Chrome.

3.1.5 Limitations

Incompletely capturing user reactions. It is possible that users in the experimental group reacted differently than the control group in ways that we did not measure. For example, perhaps when the EV UI is disabled, users use a throwaway password or deny all permission prompts. Though we cannot feasibly measure all such user reactions, we feel that our study still provides value by (a) measuring a wide variety of user behaviors that one could reasonably expect to be influenced by a security or identity indicator, and (b) studying user behaviors in a naturalistic scenario that have previously been studied only in labs or surveys.

Limited insight into per-site effects. Some, but not all, of our metrics are keyed by URL (Section 3.1.1). For metrics which are not URL-keyed, we can draw conclusions only about user behavior in aggregate over all sites. It is possible that the absence of the EV indicator influenced these metrics on particular sites but did not have a significant effect when aggregated over all EV sites.

Incomplete simulation of attack scenarios. Our study analyzes whether users react to a missing EV indicator on a website that does not otherwise look suspicious. This simulates an attack scenario in which an attack website mimics a victim website exactly except for the EV certificate. This attack scenario could arise if an attacker obtains a homograph domain (one that looks nearly or exactly identical to a victim domain)[6] or an ordinary domain-validated certificate for the victim site. However, EV certificates might significantly influence user behavior in other attack scenarios that we did not study. For example, consider a website that spoofs `paypal.com` but is hosted at an obviously incorrect URL. In this scenario, a missing EV indicator might prompt a user to inspect the URL and thereby detect the attack. While our experiment does not cover such attack scenarios, we feel that our experiment's attack scenario is of particular interest: the purpose of EV indicators are to provide human-meaningful identity information on the premise that other signals, such

[5] `https://www.chromium.org/developers/design-documents/site-engagement`

[6] `https://en.wikipedia.org/wiki/IDN_homograph_attack`

as the URL, are not sufficient tools for identifying websites. We explore an additional attack scenario of practical interest via a survey experiment, as described in Section 4.

Dataset selection. Our dataset does not come from a truly random sample of Chrome's user population: users can opt out of the user metrics program, and users must specifically opt in to browsing data syncing to report URLs (Section 3.1.1). However, we still believe this data is valuable in light of its scale and naturalistic observation.

3.1.6 Ethical considerations

Although our institution is not subject to IRB approval, the EV experiment went through an internal review process before launching, including security and privacy reviews. As discussed in Section 3.1.1, Chrome metrics reports are pseudonymous [20].

The experiment rolled out in several release channels before Stable, per Chrome's usual release process. The experiment was monitored as it rolled out, and had any problems been detected, it could have been disabled at any time.

For users in the experimental group, the Chrome developer console contained a message explaining the experiment. This message was intended to inform site owners why their EV certificate UI might not be showing.[7]

Changes to browser security and identity indicators come with the risk that users feel safer on malicious sites and take actions that they wouldn't otherwise take (for example, a user might enter credit card details on a scam site because the UI change made them believe it was safe). Our approach is similar to other field studies on browser security UI, such as exploring new security indicator icons [16], and more conservative than default feature rollouts in Chrome, as the experiment targeted only a small percentage of users and could have been disabled at any time had there been unexpected effects indicating that the experiment put users at risk. In this case, we expected the experiment to, at most, make users act more cautiously on legitimate sites, since we were only modifying positive security UI (compared to, for example, prior work experimenting with full page connection security warnings [14]).

We note that Brave Browser, which is based on the Chromium project, has opted in to not showing the EV UI using our experimental feature [2]. Brave's previous implementation (based on Muon) also intentionally did not show any EV UI [4]. Our dataset only includes data from official Chrome clients.

[7] The developer console is a default-hidden UI intended for web developers, where many technical warnings about the page are printed (e.g., identifying specific mixed content subresources, or the use of deprecated APIs). In the Stable channel, the console was opened by 2% of clients over the 14-day period of our study. We believe that this indicates that the console warning would not be a potential source of priming for a vast majority of participants. We did not see a significant difference in how often the console was opened (normalized by page loads) between our experimental and control groups (p=0.57, 95% CI: [-0.000052,+0.000028]).

3.2 Results

3.2.1 Summary

We did not see any significant differences in user behavior in our navigation or on-page metrics between our experimental group and our control group. Table 1 summarizes the results of our statistical analysis for each of our metrics.

3.2.2 Page Info interactions

Users in the control group, who saw EV UI, were significantly more likely to open the Page Info bubble. However, users in the experimental group, who did not see EV UI, were more likely to take an action in the Page Info bubble after opening it.

The experimental group opened the Page Info bubble on 0.02% of EV page loads, compared to 0.25% in the control group. Additionally, participants in our experimental group were much more likely to take an action in the Page Info bubble after opening it, across all Page Info action types.

To investigate further, we performed an additional analysis where we normalized the number of Page Info actions by the total number of EV page loads, to see if the overall number of Page Info actions taken went down in the experimental group. Table 2 shows the results of this analysis. While some Page Info actions were more common per page load in the control group, the effect sizes were very small. That is, hiding the EV UI did not make users substantially less likely to perform actions in the Page Info bubble.

Applying a Bonferroni correction for multiple testing with $m = 19$ (for each of the tests in Tables 1 and 2), the corrected significance level would instead be $\alpha = {}^{0.05}/_{m} = 0.002$. This implies that the significant results in Table 2 may be due to chance only.[8]

One possible explanation for this finding is that the large size of the EV indicator draws accidental clicks, leading users to open the Page Info bubble but not actually use it. Another hypothesis is that users notice and are curious about the EV indicator, even if it does not influence their security decisions (consistent with prior work that found that users noticed identity indicators but did not use them in their decision-making processes [28, 33, 40]). We cannot conclusively differentiate between these two hypotheses.

3.2.3 Per-site metrics

We analyzed three URL-keyed metrics (navigations, Site Engagement score, and change in Site Engagement score) on each of the top 20 EV sites. For 14 of the 20 most-visited EV origins, there were no differences with $p < 0.05$. The remaining 6 origins are shown in Table 3, each with one metric with $p < 0.05$. Five of these are very small to small negative effects on the number of navigations to the site, while one

[8] Using a Bonferroni correction allows us to control for Type I errors. We show results significant at both the $p < 0.05$ level and the multiple testing-corrected level, as the Bonferroni correction can be too conservative in cases of correlated tests.

	Control (σ)	Experiment (σ)	Δ	95% CI	p-value	Cohen's d
EV Navigations	6.18% (0.13%)	6.18% (0.13%)	-0.00	(-0.03, +0.03)	0.80	0.00
Time on EV pages (s)	2609.58 (47724.51)	2621.61 (47816.15)	+12.03	(–156.06, +180.12)	0.89	0.00
Page Ended With Tab Closed	31.64% (0.27%)	31.58% (0.27%)	-0.05	(-0.15, +0.03)	0.20	0.00
Page Ended With Back/Forward	3.61% (0.08%)	3.61% (0.08%)	+0.00	(-0.02, +0.01)	0.51	0.00
Page Reloaded	0.96% (0.05%)	0.96% (0.05%)	+0.00	(-0.02, +0.01)	0.51	0.00
Download started	3.77% (2.61%)	3.44% (1.30%)	-0.33	(-1.05, +0.38)	0.36	0.00
Form submitted	43.45% (0.97%)	43.49% (0.98%)	+0.04	(-0.30, +0.37)	0.83	0.00
CC filled	55.52% (0.79%)	55.91% (0.79%)	+0.39	(-1.21, +1.99)	0.63	0.00
Page Info opened	0.25% (0.04%)	0.02% (0.009%)	-0.23	(-0.24, -0.22)	0.00	0.09
Cookies dialog opened	0.54% (0.66%)	3.48% (0.17%)	+2.94	(+2.31, +3.57)	0.00	0.36
Changed permissions	1.26% (0.12%)	10.78% (0.34%)	+9.52	(+8.25, +10.78)	0.00	0.63
Certificate dialog opened	0.74% (0.11%)	5.52% (0.22%)	+4.78	(+3.97, +5.58)	0.00	0.39
Connection help opened	0.21% (0.04%)	1.51% (0.11%)	+1.29	(+0.89, +1.69)	0.00	0.26
Site settings opened	0.83% (0.08%)	5.80% (0.22%)	+4.97	(+4.17, +5.76)	0.00	0.49

Table 1: Summary of statistical tests for our EV field experiment metrics. The only differences that were significant at the $p < 0.05$ level were for Page Info behavior (highlighted).

	Control (σ)	Experiment (σ)	Δ	95% CI	p-value	Cohen's d
Cookies dialog opened	0.0019% (0.0026%)	0.0010% (0.0017%)	-0.001	(-0.0017, -0.00002)	0.01	0.004
Changed permissions	0.0031% (0.0028%)	0.0025% (0.0024%)	-0.0006	(-0.0015, +0.0003)	0.18	0.002
Certificate dialog opened	0.0022% (0.0026%)	0.0017% (0.0026%)	-0.0005	(-0.0014, +0.0004)	0.28	0.002
Connection help opened	0.0009% (0.0017%)	0.0005% (0.0014%)	-0.0004	(-0.0009, +0.0002)	0.17	0.002
Site settings opened	0.0028% (0.0028%)	0.0014% (0.0014%)	-0.0014	(-0.0023, -0.0006)	0.007	0.006

Table 2: Summary of our followup analysis of Page Info behavior, with counts of actions normalized by the number of EV page navigations instead. The highlighted rows were significant at the $p < 0.05$ level, but the effect sizes are negligible.

is a very small *positive* effect on the per-visit change in the Site Engagement score. However, if we apply a Bonferroni correction with $m = 60$ (three metrics checked across 20 origins), then we should instead consider a significance level of $\alpha = {}^{0.05}/_{m} = 0.0008$. With the correction, only one origin had a significant difference in user behavior: Origin 15 had 4.26 (95% CI: 2.20 to 6.32, $d = 0.24$) fewer navigations on average per user in the experimental condition.

We note that our navigation metric used here is not normalized due to limitations of the URL-keyed metrics dataset, so these results may be affected by natural variations in browsing volume between users.

4 EV survey experiments

In our EV field study, we failed to find evidence that the absence of the EV indicator influences most user behaviors. In this section, we examine two follow-up questions that were infeasible to answer via field experiment:

1. **Does the EV UI help users detect cross-jurisdiction collision attacks?** We were particularly interested in cross-jurisdiction collisions due to a recent high-profile proof of concept [11]. In this attack, two EV certificates are registered with the same legal entity name in different jurisdictions. We studied this question via survey because a field experiment would have required the browser to display incorrect information.
2. **How do users react to EV UI in modern browsers other than Chrome?** We focused on the Apple Safari browser because it recently made a significant change to its EV UI, removing the legal entity name and simply showing the domain in green (Figure 5). Because we did not have access to Safari field data, we instead conducted a survey experiment.

4.1 Methodology

We ran two online survey experiments, corresponding to the two research questions described above.

4.1.1 Questions

The surveys showed participants a login screen for a well-known financial webpage in their respective countries: PayPal in the U.S. and HSBC in the U.K. We asked participants three questions, displayed underneath the screenshot.

First, in a five-point Likert scale, we asked participants to rate their comfort level logging into the webpage: *Would you feel comfortable logging in on this website? Very comfortable / Somewhat comfortable / Neither comfortable nor uncomfortable / Somewhat uncomfortable / Very uncomfortable*

To avoid leading participants' responses, we intentionally left this question up to their interpretation and allowed them to elaborate. We next asked participants for open-ended de-

		Control	Experiment	Δ	95% CI	p-value	Cohen's d
Origin	Metric						
3	Navigations	18.55	14.78	-3.77	(-6.78, -0.76)	0.014	0.17
4	Navigations	25.80	23.08	-2.72	(-4.73, -0.72)	0.0078	0.10
10	Navigations	13.37	11.10	-2.27	(-4.33, -0.21)	0.031	0.14
14	Navigations	20.40	15.65	-4.74	(-8.48, -1.01)	0.013	0.21
15	Navigations	18.57	14.31	-4.26	(-6.32, -2.20)	0.00005	0.24
18	Δ Site Engagement	0.88	1.40	+0.52	(+0.09, +0.95)	0.017	0.13

Table 3: Summary of our (blinded) per-origin analysis from our UKM dataset. The included rows are the origin/metric pairs that were significant at the $\alpha = 0.05$ level. The highlighted row is the only significant result after applying a Bonferroni correction ($\alpha = 0.0008$). All the differences have at most a small effect size (Cohen's d).

tails about their reasoning: *Can you tell us why you feel that way? (If there's nothing to add, leave blank.)*

The final question appeared with the same login page screenshot, allowing users to click on it up to three times to mark the relevant sections: *Click the item(s) on the screen that make you feel that way.*

4.1.2 Participants

We recruited U.S. participants through Mechanical Turk and U.K. participants through Clickworker. We selected the U.S. and U.K. because EV usage was common in these countries (based on our dataset from Section 3), and we were unable to recruit enough participants in other countries where EV usage is common. Participants received a $.40 or €.35 incentive for participation. Our cross-jurisdiction collision survey ran from January 29 to February 3, 2019, with 592 U.S. participants and 650 U.K. participants. Our Safari EV survey ran from January 29 to February 1, 2019, with 290 U.S. participants and 305 U.K. participants.

Demographics. In both surveys, U.S. participants skewed slightly older than U.K. participants, who were overrepresented in the 18-24 age range. In the cross-jurisdiction attack survey, U.S. participants skewed slightly male (55%) and U.K. participants skewed slightly female (55%). Full demographic details can be found in the Appendix.

4.1.3 Experimental conditions

Cross-jurisdiction collision survey. In this survey, we randomly assigned participants to see one of five conditions with a screenshot of the login page, each manipulating the country code displayed in the EV indicator, as shown in Figure 4. One condition omitted the country code entirely, one showed the correct country code (US or GB), and three showed incorrect country codes (MX, RU, and BR).

Safari EV UI survey. Safari changed its EV display in macOS 10.14 to no longer display the legal entity name. In this survey, we randomly assigned participants to one of two conditions. In the first, users saw the login webpage with the EV display used in macOS 10.13, and in the second condition, users saw the EV display from macOS 10.14 (Figure 5).

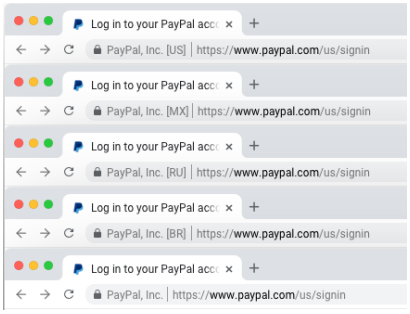


Figure 4: Five conditions shown to U.S. participants, manipulating only country code.



Figure 5: Two conditions shown to U.K. participants, manipulating display of EV to include the site's registrable domain (macOS 10.14) or EV legal entity name (as in macOS 10.13).

4.1.4 Data coding

Two researchers coded the qualitative responses on users' comfort level, with one team member (the *codemaster*) open coding the initial coding rounds, and the other iteratively providing feedback to the codemaster. In the final round of iteration, both researchers coded all responses for both surveys. Cohen's κ, a measure of inter-rater reliability, was 0.974 in the cross-jurisdiction survey (with 95.3% agreement), and 0.949 (with 97.6% agreement) in the Safari EV formatting survey, both indicating strong consistency between coders. The codemaster resolved the remaining conflicts.

4.1.5 Limitations

Artificial scenario. As with previous lab and survey studies about browser identity indicators, our surveys are an artificial scenario. This approach has limited ecological validity, as participants are not tasked with signing into a real website, nor with their real credentials, and thus they may feel less concerned than usual. However, in a more naturalistic scenario, we would expect that users would also pay less

	Cnd 1	Cnd 2	Cnd 3	Cnd 4	Cnd 5
U.S.					
Very comfortable	63%	63%	61%	56%	68%
Somewhat comfortable	30%	24%	25%	28%	21%
Neither comfortable nor uncomfortable	2%	4%	5%	3%	3%
Somewhat uncomfortable	3%	7%	6%	6%	7%
Very uncomfortable	2%	3%	3%	8%	2%
n	121	120	115	117	119
U.K.					
Very comfortable	48%	56%	46%	44%	56%
Somewhat comfortable	31%	33%	36%	39%	35%
Neither comfortable nor uncomfortable	10%	5%	3%	8%	5%
Somewhat uncomfortable	6%	4%	12%	7%	3%
Very uncomfortable	5%	2%	3%	3%	2%
n	125	132	128	132	133

Table 4: Users' comfort levels logging into a webpage with different EV country codes. Cnd 1 is the topmost variation shown in Figure 4 and Cnd 5 is the bottommost.

overall attention to security concerns because no one would ask them about their comfort level before they logged in. We therefore consider our results to describe upper bounds on how EV indicators influence user behavior.

Demographics. Since we only surveyed U.S. and U.K. participants, our results may not generalize to other contexts and cultures.

4.2 Results

Across surveys and conditions, we found that most users felt comfortable logging into each webpage, regardless of the EV UI. In nearly all cases, we found no differences among users' self-reported comfort levels with each login page.

4.2.1 Cross-jurisdiction collision survey

We found no evidence that the country code displayed in the EV indicator helps users detect a cross-jurisdiction attack.

Quantitative results. In both the U.S. and U.K., participants were most likely to say they felt "Very comfortable" logging into the webpage, regardless of the country code presented. We conducted a Kruskal-Wallis test, and in both the U.S. ($\chi^2 = 1.1783, df = 4, p = 0.8817$) and U.K. ($\chi^2 = 2.4994, df = 4, p = 0.6447$), we found no significant differences among users' comfort levels in each condition. Table 4 shows the full results.

Reasons for comfort or discomfort. When asked to identify why they felt "somewhat" or "very comfortable", participants were more likely to refer to cues in the content area, rather than Chrome UI.

Responses varied somewhat in each region. U.S. participants were most likely to describe feeling familiar with the webpage (e.g., *"PayPal is well known so it makes me feel somewhat comfortable."*), while U.K. participants most commonly pointed to an HTTPS indicator (e.g., *"the https along with the padlock in the address bar"*) but not EV-specific UI.

Participants referred to cues in the content area such as:

- familiarity with the webpage
- the page's simplicity or ease of use (e.g., *"I feel very comfortable because it is easy to understand..."*)
- the page's general design (e.g., *"A comfortable amount of white space without the page feeling empty"*)
- the page looking normal or expected (e.g., *"The sign in system here has followed a standard sign in page and gives all necessary help"*)

When referring to cues in the browser itself, participants most commonly referred to the HTTPS indicator, specifically identifying the padlock icon (e.g., *"Mainly because of the padlock on the top search bar makes me think it's secure enough to use safely"*). Participants also noted that the URL looked normal or expected (e.g., *"... the link web address doesn't look abnormal"*). They were far less likely to refer to EV UI specifically (e.g., *"The site displays that it is secure with a registered identity, PayPal Inc.... "*).

As many as 3% of U.S. participants and 14% of U.K. participants in each condition referred to the site as safe or secure, without describing their reasoning (e.g., *"It's a secure bank login page"*).

Few noticed oddities in the page's country code (no more than 8% in any U.S. condition and 5% in the U.K). Even when participants did notice, it did not necessarily make them uncomfortable (e.g., *"I never noticed the MX on a PayPal page, but it seems legit."*).

Table 5 shows a subset of results of our open-ended question about why users felt comfortable or uncomfortable.

Items on the page. When asked to "click item(s) on the page that make you feel that way", participants were most likely to click the HTTPS indicator (but not EV UI specifically), parts of the URL, or page logos. Figure 6 displays an example heatmap for these clicks. The other heatmaps can be found in the Appendix.

These results suggest that many users do use HTTPS security indicators and site URLs to determine the legitimacy of a website. However, in both qualitative and quantitative responses, almost no participants appear to notice EV UI. Additionally, these results suggest a cross-jurisdiction attack could be viable in part because users infer the legitimacy of a website from the presence of HTTPS indicators.

4.2.2 Safari EV UI survey

We found no evidence that the change in Safari's EV format affected users' comfort logging in to a webpage.

Quantitative results. In both the U.S. and U.K., in both conditions, participants were most likely to say they felt "Somewhat comfortable" or "Very comfortable" logging into the webpage. We conducted a Kruskal-Wallis test, and in both the U.S. ($\chi^2 = 0.0808, df = 1, p = 0.7762$) and U.K.

	U.S. Cnd 1	Cnd 2	Cnd 3	Cnd 4	Cnd 5	U.K. Cnd 1	Cnd 2	Cnd 3	Cnd 4	Cnd 5
n	92	120	93	93	115	83	91	81	83	74
Comfortable reasons										
I'm familiar with this website	33%	26%	31%	40%	33%	10%	7%	6%	7%	14%
I see an HTTPS indicator	32%	16%	23%	19%	17%	27%	25%	21%	23%	35%
URL looks normal	8%	8%	15%	9%	10%	1%	4%	2%	4%	4%
Page looks simple / easy to use	9%	7%	9%	10%	7%	18%	16%	9%	16%	15%
Page looks well-designed	2%	2%	0%	3%	0%	4%	8%	14%	12%	3%
I see an EV certificate	1%	1%	2%	1%	1%	1%	0%	1%	1%	1%
Uncomfortable reasons										
Country code looks strange	0%	6%	5%	8%	0%	0%	1%	5%	0%	0%
Page does not look normal	1%	1%	2%	4%	3%	1%	1%	0%	7%	3%
Page looks bland	1%	1%	4%	1%	3%	10%	2%	1%	5%	1%
URL looks odd	0%	1%	0%	1%	1%	1%	2%	2%	2%	3%
Page looks poorly-designed	0%	0%	0%	0%	0%	6%	7%	9%	7%	4%

Table 5: Sample results of the open-ended question "Can you tell us why you feel that way?" when participants were asked how comfortable they were logging in to a site. Cdn 1 is the topmost condition shown in Figure 4 and Cdn 5 is the bottommost. Full results are shown in the Appendix.



Figure 6: Example click heatmap, displaying what U.K. participants say made them feel comfortable or uncomfortable on a webpage with an RU country code in the EV indicator.

	Cnd 1	Cnd 2
U.S.		
Very comfortable	50%	47%
Somewhat comfortable	32%	30%
Neither comfortable nor uncomfortable	4%	2%
Somewhat uncomfortable	8%	16%
Very uncomfortable	6%	5%
n	142	148
U.K.		
Very comfortable	43%	42%
Somewhat comfortable	46%	39%
Neither comfortable nor uncomfortable	3%	7%
Somewhat uncomfortable	3%	11%
Very uncomfortable	4%	1%
n	152	153

Table 6: Users' comfort levels logging into a webpage with different Safari EV UIs. Cnd 1 is the variation with the site's registrable domain and Cnd 2 is the EV legal entity name.

($\chi^2 = 0.50313, df = 1, p = 0.4781$), we found no significant differences in users' comfort levels across conditions. Table 6 shows the full results.

Reasons for comfort or discomfort. Similar to the results from our cross-jurisdiction attack survey, U.S. participants were most likely to say they felt comfortable logging in because they are familiar with the webpage, while U.K. respondents were more likely to say they felt comfortable because they saw an HTTPS indicator. However, most participants in both conditions also referred to content area cues, such as the page looking as expected, or the page being simple or well-designed. Table 7 shows the full results.

Once again, as much as 6% in the U.S. and 9% in the U.K. said the website they saw is "safe" or "secure" without mentioning whether the browser or content area made them feel that way.

Participants said they felt uncomfortable logging in for several reasons, varying by region. In the U.S., participants were most likely to say they felt uncomfortable logging in because they could not see the URL (e.g., *"There's no web address present, so it could be a spoofed page"*). In the U.K. participants were most likely to say they felt uncomfortable because something in the content area was poorly-designed (e.g., *"The page looks very cold and sterile"*). Overall, however, participants were uncomfortable for very similar reasons in each region. When referring to the browser UI, they cited issues with the appearance or (in)visibility of the URL. When referring to issues with the content area, participants said the page looks bland or poorly designed.

Participants were split as to whether the EV indicator made them feel comfortable or uncomfortable, with many stating they wanted to be able to see the full URL (e.g.,

	U.S. Cnd 1	U.S. Cnd 2	U.K. Cnd 1	U.K. Cnd 2
n	115	118	95	98
Comfortable reasons				
I'm familiar with this website	40%	28%	7%	10%
I see an HTTPS indicator	25%	23%	27%	33%
Page looks simple / easy to use	8%	11%	5%	6%
Page looks normal (unclear)	7%	8%	17%	14%
It's safe / secure (unclear)	6%	2%	9%	8%
I see an EV certificate	2%	4%	2%	1%
URL looks normal	4%	0%	2%	0%
Page looks well-designed	0%	1%	12%	11%
Uncomfortable reasons				
I can't see the URL	4%	13%	3%	6%
I'm not sure if it's safe / secure (unclear)	7%	5%	3%	5%
Page looks bland	3%	7%	5%	5%
The URL looks odd	2%	1%	3%	2%
I do not see an HTTPS indicator	2%	0%	1%	0%
Page looks poorly-designed	0%	1%	9%	5%
Unclear or other	5%	12%	8%	9%

Table 7: Results of the open-ended question "Can you tell us why you feel that way?" when participants were asked how comfortable they were logging in to a site. Cdn 1 is the top condition shown in Figure 5 and Cdn 2 appears below.

"Looks like the genuine page but I'd like more reassurance of this, like being able to see the URL").

As many as 7% of U.S. participants and 5% of U.K. participants said they weren't sure if the site was safe or secure, but were unclear how (e.g., *"It doesn't look secure"*).

Items on the page. When asked to "click item(s) on the page that make you feel that way", participants were most likely to click the HTTPS indicator, as well as the page logo. Figure 7 displays a heatmap for these clicks in one condition. The other heatmaps can be found in the Appendix.

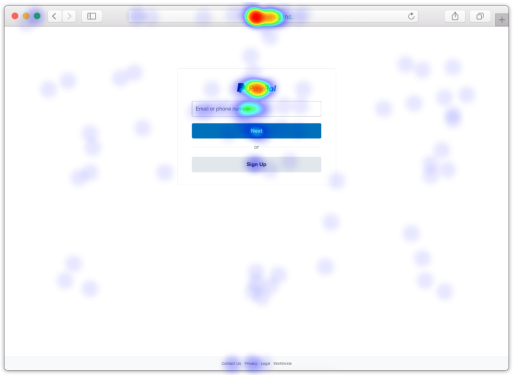


Figure 7: An example of a click heatmap from U.K. participants. This condition displayed the EV legal entity rather than a registrable domain.

5 URL highlighting survey experiment

As with the EV indicator, prior research has found that users often do not notice URLs or do not use them to make security decisions [12, 28, 40]. We conducted a survey experiment to learn whether more pronounced URL formatting changes in the browser address bar would draw attention to the URL and help users understand its security properties, but we found that these URL formatting changes were not effective.

5.1 Methodology

In this survey, we showed users a screenshot of a Google login page with a suspicious URL in the browser address bar (`accounts.google.com.amp.tinyurl.com` instead of `accounts.google.com`). We asked users to identify the website and then asked them if they would be comfortable entering their login credentials on the site.

5.1.1 Questions

The first question in the survey asked participants to identify the website in an open-ended response: *Before we move ahead, please identify the above website.* The subsequent questions asked users how comfortable they were logging in to the website and why. These questions were identical to Section 4.1.1 except that we did not ask participants to "click the item(s) on the page that make you feel that way."

5.1.2 Participants

Our survey ran from November 20 to November 21, 2018. We recruited 1,180 U.S. participants from Mechanical Turk who were paid a $.40 incentive.

Demographics. Similar to our previous U.S. surveys, the sample skewed slightly male (53%), with adults 55 and older underrepresented. Full demographic details can be found in the Appendix.

5.1.3 Experimental conditions

This survey showed participants a Google sign-in page with an incorrect URL (`accounts.google.com.amp.tinyurl.com`), simulating a phishing attack. We randomly assigned participants to one of seven conditions (Figure 8). Condition 1 (the control) used the Chrome 69 address bar UI, while other conditions attempted to draw attention to the registrable domain (`tinyurl.com`) in various ways.[9]

5.1.4 Data coding

Because there was almost no ambiguity in participant responses to our first question about the website's identity, only one researcher coded these responses. For all other questions, we coded the data as in Section 4.1.4. Based on a subsample of 100 responses coded by two security researchers,

[9] We chose these particular URL highlighting formats as we wanted to examine variants that we believed would (1) give emphasis to the registrable domain by manipulating color and spatial layout, (2) be noticeably distinguishable from the existing format but (3) not overtly distracting from browsing, so each variant could viably be deployed in the real world.

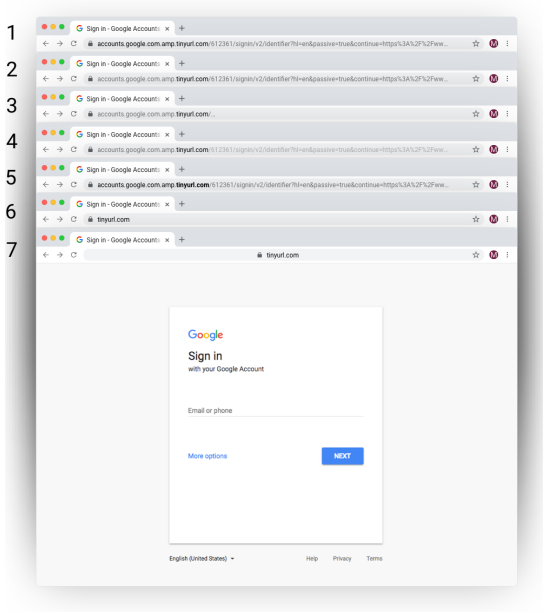


Figure 8: Conditions shown to U.S. participants, manipulating the URL display to emphasize the registrable domain.

Cohen's κ was 0.946, indicating strong agreement, with the two coders in agreement 95.4% of the time. The codemaster resolved the remaining conflicts.

5.1.5 Limitations

This survey suffers the same limitations as in Section 3.1.5: namely, an artificial scenario and limited generalizability beyond the U.S. Additionally, in this survey, participants may have responded to the novelty of the URL format, and not just the URL content, making it difficult for us to isolate the impact of the URL format alone. However, this did not appear to significantly impact our results because we did not detect any significant differences across variations.

5.2 Results

5.2.1 Website identification

Few participants noticed anything strange about the website when asked to identify it. 85% of all participants said the website was Google, when in fact, the address said `tinyurl.com`. 13% of participants correctly identified the website by its URL. 1% described both Google and TinyURL, and 1% provided a different response.

5.2.2 Comfort logging in

In all conditions, participants were most likely to say they felt comfortable logging into the webpage, despite the suspicious URL. Across the seven conditions, we found no significant differences ($\chi^2 = 2.847, df = 6, p = 0.8278$). Table 8 shows the coded results of our question about why users felt comfortable or uncomfortable logging in.

When asked why users reported feeling "somewhat" or "very comfortable", the majority of responses described looking at cues in the content area, citing that the website looked familiar (e.g., *"Because it's familiar. I've seen it plenty of times."*), or that they trust the website that appeared in the content area (e.g., *"Google is a secure company"*).

When describing discomfort, participants most commonly cited oddities with the URL (e.g., *"It seems to be an attempt to spoof Google on tinyurl"*). Relatively few participants mentioned concerns with feeling unsure how they would have navigated to this site (e.g., *"Because I have no idea how or why I'm here"*), while some described feeling unsure about the general security or safety of the site, but did not specify why (e.g., *"It's an imposter"*).

Notably, even in open-ended responses where participants appear to have been looking at the URL, they did not necessarily notice any oddities. For example, one participant reported feeling "Very comfortable" with the `tinyurl.com` URL: *"Because the URL looks like a Google page should."*

Condition 6, which showed only the registrable domain on the left of the address bar, stood out as the most distinct, with users citing oddities in the URL and generalized safety concerns at a disproportionate rate. However, the differences in comfort level between the control and this condition were not statistically significant ($\chi^2 = 0.4541, df = 1, p = 0.5004$).

6 Discussion

6.1 Summary of results

In this paper, we used large-scale field data and surveys to corroborate past results on browser identity indicators and to contribute new findings.

Our EV field experiment (Section 3) found that removing the EV UI has no effect on most user behavior metrics. However, removing EV UI did cause users to open the Page Info bubble (Figure 3) less often, and it caused a small decrease in navigations for one of the top 20 EV sites. Our experiment corroborates prior work suggesting that EV UI does not help users detect attacks [24], but at a much larger scale, with naturalistic data, and with up-to-date browser UIs. The effect on Page Info is also consistent with prior findings that users may notice EV UI but not use it in their security decisions [33].

Our EV surveys (Section 4) are the first to study cross-jurisdiction collisions and Safari's recent EV UI change. In all conditions across both surveys, EV UI did not appear to affect users' comfort levels when logging into a webpage. Our qualitative data corroborates past results that users use the content area rather than browser UI to make trust decisions [12] and that connection security indicators can be mistaken to mean that the site is safe [16]. We contribute new findings that EV indicators are likely ineffective against cross-jurisdiction collision attacks and that Safari's old and new EV UIs have similar impacts on users' comfort levels.

	Cnd 1	Cnd 2	Cnd 3	Cnd 4	Cnd 5	Cnd 6	Cnd 7
n	132	127	130	124	128	132	137
Comfortable reasons							
Looks familiar	36%	33%	35%	35%	38%	23%	32%
I trust Google	20%	17%	12%	15%	16%	16%	15%
Page looks simple / easy to use	8%	3%	8%	4%	5%	4%	4%
Site is secured or safe	5%	6%	6%	5%	6%	5%	4%
Page looks normal (unspecified)	2%	1%	0%	2%	2%	2%	1%
URL looks normal	2%	2%	0%	1%	2%	0%	0%
Uncomfortable reasons							
The URL looks funny	23%	27%	33%	27%	30%	32%	33%
I'm not sure the site is safe (unspecified)	2%	7%	2%	7%	2%	13%	4%
I'm unsure where I came from / where I am	3%	3%	2%	0%	2%	3%	1%
Unclear or other	3%	6%	3%	6%	2%	5%	9%

Table 8: Coding results of the open-ended question "Can you tell us why you feel that way?" when participants were asked how comfortable they were logging in to a site. Cdn 1 is the topmost condition shown in Figure 8 and Cdn 7 is the bottommost.

Finally, we surveyed users to determine if variations on Chrome's URL display can make it a more effective identity indicator (Section 5). None of our variations appeared to make users uncomfortable to log in to a phishing webpage. This survey corroborated prior studies showing that URLs are ineffective identity indicators [12, 28, 40], and extended them to show that several variations on browser URL display are ineffective as well. There were small but statistically insignificant differences among our variations; while a larger sample size might yield statistically significant differences, we think they are unlikely to be large effects.

6.2 Ineffectiveness of identity indicators

Removing the EV indicator did not affect most user behaviors, suggesting that an EV certificate does not provide a good defense against phishing or social engineering. While the EV UI did cause users to open Page Info more often, users did not use its functionality substantially more often. We therefore believe that users may notice the EV indicator, but do not appear to use it in making security decisions. Moreover, our survey results suggest that recent proof-of-concept attacks against EV [11] would likely be effective, and that simple UI tweaks do not make URLs an effective identity indicator either. We conclude that browser vendors should pursue more radical redesigns of their current website identity indicators if they want them to be more effective.

6.3 Guidance for designing identity indicators

Based on our experimental results and our review of prior work (Section 7), we provide the following recommendations for the design of identity indicators:

- **Prefer active, negative indicators to passive indicators.** Our UI changes failed to make the URL an effective identity indicator. Prior work has seen some success in redesigning EV indicators to make them more noticeable [33] or more understandable [8], but not better able to help users detect attacks. In contrast, active warnings like SSL errors have been successfully redesigned to reduce clickthrough rates [14]. We therefore recommend that the security community focus on triggering active warnings when a website's identity is suspicious (for example, when a domain is suspiciously similar to a popular domain), rather than relying on users to notice and act on passive identity indicators.
- **Prominent UI is an opportunity for user education.** Removing the EV indicator caused users to open the Page Info bubble less (Section 3.2.2). This effect suggests that prominent browser UI can be an opportunity to draw users' attention and educate them about the browser's identity indicators. For example, the Page Info bubble could explain the site's identity and how users should take action on it. However, we saw that in both our control and experimental groups the typical user never opened the Page Info bubble (4.65% of users in the control group opened Page Info, while 0.45% of users in the experimental group did). It is unclear if this is due to a lack of user understanding or a mismatch between users' goals and the controls provided by Page Info. Additionally, prior attempts at user education about identity indicators have been only marginally effective (e.g., [24, 28, 40]). Combined, we believe this indicates that more work is needed to understand if this approach is viable.
- **Incorporate user research in identity indicator design.** We recommend that browser vendors undergo extensive user research before launching new identity indicators, via both browser telemetry and user studies. As our work shows, both types of user research provide value: telemetry from field experiments can measure aggregate or per-site effects over large numbers of users in naturalistic settings, whereas user studies can provide insight into users' thought processes.

7 Related work

In this section, we survey related work on browser identity indicators and EV certificates.

7.1 EV effectiveness

7.1.1 User studies

Detecting fraudulent sites. In the 2000s, a number of studies analyzed how users react to EV indicators, finding that they were not effective in helping users detect phishing.

Jackson et al. [24] surveyed 27 participants about Internet Explorer 7's new EV UI. They concluded that it did not help users detect two types of phishing attacks (picture-in-picture and homograph attacks), even after receiving education about the UI.

Sobey et al. [33] analyzed Firefox 3's EV indicator as well as their own new EV design. In a lab study of 28 participants, they found that users did not notice Firefox 3's new EV indicator, but half did notice their new design. However, only a small number of participants seemed to use the newly designed indicator for decision-making.

These studies provide evidence that browser EV indicators are not effective, but they study only a small number of participants in an artificial lab scenario. Moreover, they study the very earliest EV indicators; little work has been done recently to study EV in modern browser UIs. Our work updates and expands these studies by providing large-scale in situ browser telemetry data, as well as survey data from over 1,000 participants, using modern browser UIs.

Designing EV for reassurance and understanding. Biddle et al. [8] studied Internet Explorer 7's EV indicator, comparing it to a new EV indicator of the researchers' design. Surveying 40 participants, the researchers found that their new design improved users' confidence, ease of finding information, and ease of understanding. However, they did not evaluate whether the new design helped users identify the attacks we considered. It remains an open question whether a redesigned EV indicator can effectively prevent phishing and social engineering attacks.

7.1.2 Attack proofs of concept

Researchers have recently demonstrated flaws in the EV validation procedures. The researchers obtained misleading certificates that can undermine the effectiveness of EV.

One researcher obtained a certificate for a company named "Identity Verified" [10]. This demonstrated that a malicious website could abuse the EV indicator's privileged position in browser UI to make the attack website seem more legitimate.

Another researcher obtained an EV certificate for a company named "Stripe, Inc.", mimicking the payments company but incorporated in a different state [11]. This demonstrated that EV certificates are subject to cross-jurisdiction collisions in which a user may not be able to distinguish two identical company names (one legitimate and one malicious) incorporated in different jurisdictions.

Our work is complementary to these attacks. We are primarily concerned with whether users notice and understand the EV indicator, rather than with how it can be attacked and abused. However, we do lend credence to the cross-jurisdiction collision demonstration by evaluating whether users notice cross-jurisdiction collisions (Section 4).

7.2 URL comprehension

Our work analyzes whether simple tweaks to browser URL display can help users identify fraudulent sites. Several prior studies have examined whether users understand URLs and can use them to detect attacks.

Lin et al. [28] asked 22 participants to identify fraudulent sites with and without explicit instruction to look at the browser address bar. While their user education effort was successful to an extent, it was not effective for many users and cannot be relied upon as a sole defense. Similarly, Wu et al. [39] and Dhamija et al. [12] found that neither browser address bars nor various supplemental security toolbars helped users detect phishing. In a lab study with a think-aloud protocol, Jakobsson et al. [25] concluded that users look at URLs in the process of determining whether a website is authentic, but they can be easily fooled by tricky URLs.

Xiong et al. [40] expanded Lin et al.'s work to include a control condition that did not highlight the domain in the UI, as well as a larger, more representative participant group and eye-tracking data. They found that instructing participants to look at the address bar led to a modest improvement in their ability to detect fraudulent sites, but the domain highlighting in the browser UI had no detectable effect. Their eye-tracking data suggested that explicit instructions about the browser address bar can draw users' attention to the URL, but does not give them the information or understanding that they need to draw accurate security conclusions from it.

Our work extends Xiong et al.'s study by testing multiple UI variations. Our URL formatting survey (Section 5) corroborates the existing findings that drawing users' attention to the URL bar does not help them make accurate security decisions. We contribute new findings that various UI modifications do not succeed in the goal of making the URL more noticeable and comprehensible.

7.3 Other web security UIs

Other security UIs on the web have been examined through user studies, browser telemetry, surveys, and eye-tracking.

7.3.1 Connection security indicators

Research results on browser connection security indicators have been mixed. While some studies have found that many users look at and understand them [19,37], others have found that they do not affect user behavior [12, 31]. Felt et al. [16] surveyed thousands of users to redesign connection secu-

rity indicators that met modern design constraints and better communicated the intended semantics.

Multiple studies have investigated user understanding of connection security and HTTPS, finding that users, especially those without technical backgrounds, do not have well articulated mental models for how the Internet works [26], and often conflate HTTPS and the lock icon with site security rather than connection security [38]. Krombholz et al. [27] expanded this prior work by exploring end user and administrator mental models of HTTPS, finding many misconceptions about the benefits and threat models of HTTPS among both groups. Particularly relevant for our work here, they found general distrust in HTTPS as a protocol and that security indicators are rarely part of users' mental models.

Our work contributes to this body of evidence that browser identity indicators, like connection security indicators, do not help users make security decisions. While we do not attempt to redesign identity indicators in this paper, the techniques used by Felt et al. to redesign connection security indicators could be useful for redesigning identity indicators.

7.3.2 Browser warnings and prompts

A large body of work has examined users' reactions to browser security warnings and prompts. Malkin et al. [29] and Bravo-Lillo et al. [9] conducted Mechanical Turk studies to evaluate UI changes for HTTPS warnings and plugin installation prompts, respectively. Browser security warnings have been found to have high clickthrough rates in lab studies (e.g., [12, 34, 35]), but lower in the wild [7, 14].

7.3.3 Website credibility and authenticity

Websites themselves contain security UI, including security and identity indicators. Fogg et al. [17] performed an online study to understand what makes users perceive a website as credible, and Jakobsson et al. [25] conducted a lab study to examine how users determine whether a website is authentic. These studies found that various aspects of a webpage, such as its language and spelling, can contribute to whether users perceive it as credible and/or authentic. Our survey experiments also find that users pay more attention to the website content than to browser UI when making trust decisions.

8 Conclusion

Browser identity indicators, including URLs and EV certificates, are supposed to help users identify phishing, social engineering, and other attacks, but prior lab studies and surveys suggested that older browser identity UIs are not effective security tools. In this paper, we sought to understand whether users would act on modern browser identity indicators. We provide naturalistic large-scale data about how users react to the EV indicator. We then survey thousands of users to understand the effects of recent developments in the EV ecosystem, and whether simple tweaks to browsers' URL displays can help users understand URLs better as identity indicators. We conclude that modern browser identity indicators are not effective. To design better identity indicators, we recommend that browsers consider focusing on active negative indicators, explore using prominent UI as an opportunity for user education, and incorporate user research into the design phase.

9 Acknowledgments

Thanks to Devon O'Brien, Jim Bankoski, Parisa Tabriz, Ryan Sleevi, Andrew Whalley, and Chelsea Tanaka for their support and feedback on this work.

References

[1] Domain-validated certificate. `https://en.wikipedia.org/wiki/Domain-validated_certificate`.

[2] Extended validation SSL certificate is not indicated in browser URL field. `https://github.com/brave/brave-browser/issues/3860`.

[3] Guidelines for the issuance and management of extended validation certificates. `https://cabforum.org/wp-content/uploads/CA-Browser-Forum-EV-Guidelines-v1.6.8.pdf`.

[4] Prominently show validated legal identity jurisdiction to users. `https://github.com/brave/browser-laptop/issues/791`.

[5] Public suffix list. `https://publicsuffix.org/`.

[6] URL living standard. `https://url.spec.whatwg.org/#host-registrable-domain`.

[7] AKHAWE, D., AND FELT, A. P. Alice in Warningland: A large-scale field study of browser security warning effectiveness. In *Proceedings of the 22nd USENIX Security Symposium* (2013).

[8] BIDDLE, R., VAN OORSCHOT, P. C., PATRICK, A. S., SOBEY, J., AND WHALEN, T. Browser interfaces and extended validation SSL certificates: An empirical study. In *Proceedings of the ACM Workshop on Cloud Computing Security* (2009).

[9] BRAVO-LILLO, C., KOMANDURI, S., CRANOR, L. F., REEDER, R. W., SLEEPER, M., DOWNS, J., AND SCHECHTER, S. Your attention please: Designing security-decision UIs to make genuine risks harder to ignore. In *Proceedings of the 9th Symposium on Usable Privacy and Security* (2013).

[10] BURTON, J. First part of phishing with EV. `https://www.typewritten.net/writer/ev-phishing/`.

[11] CARROLL, I. G. Extended validation is broken. `https://stripe.ian.sh/`.

[12] DHAMIJA, R., TYGAR, J. D., AND HEARST, M. Why phishing works. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2006).

[13] ESCOBAR, A. Safari Technology Preview now hides the company name (or legal entity) when showing an Extended Validation (EV) certificate, but still displays a green padlock. Progress, both in security and usability. Tweet. https://twitter.com/andrewe/status/1037737841558728706, September 2018.

[14] FELT, A. P., AINSLIE, A., REEDER, R. W., CONSOLVO, S., THYAGARAJA, S., BETTES, A., HARRIS, H., AND GRIMES, J. Improving SSL warnings: Comprehension and adherence. In *Proceedings of the 33rd Conference on Human Factors in Computing Systems* (2015).

[15] FELT, A. P., BARNES, R., KING, A., PALMER, C., BENTZEL, C., AND TABRIZ, P. Measuring HTTPS adoption on the web. In *Proceedings of the 26th USENIX Security Symposium* (2017).

[16] FELT, A. P., REEDER, R. W., AINSLIE, A., HARRIS, H., WALKER, M., THOMPSON, C., ACER, M. E., MORANT, E., AND CONSOLVO, S. Rethinking connection security indicators. In *Proceedings of the 12th Symposium on Usable Privacy and Security* (2016).

[17] FOGG, B. J., MARSHALL, J., LARAKI, O., OSIPOVICH, A., VARMA, C., FANG, N., PAUL, J., RANGNEKAR, A., SHON, J., SWANI, P., AND TREINEN, M. What makes web sites credible?: A report on a large quantitative study. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2001).

[18] FRANCO, R. IE7 and High Assurance at RSA Europe. https://blogs.msdn.microsoft.com/ie/2006/10/20/ie7-and-high-assurance-at-rsa-europe/, October 2006.

[19] FRIEDMAN, B., HURLEY, D., HOWE, D. C., FELTEN, E., AND NISSENBAUM, H. Users' conceptions of web security: A comparative study. In *SIGCHI Extended Abstracts on Human Factors in Computing Systems* (2002).

[20] GOOGLE LLC. Google Chrome privacy whitepaper. https://www.google.com/chrome/privacy/whitepaper.html.

[21] HECKER, F. CAs, certificates, and the SSL/TLS UI. http://hecker.org/mozilla/ssl-ui, November 2005.

[22] HELME, S. Are EV certificates worth the paper they're written on? https://scotthelme.co.uk/are-ev-certificates-worth-the-paper-theyre-written-on/.

[23] HUNT, T. Extended Validation Certificates are Dead. https://www.troyhunt.com/extended-validation-certificates-are-dead/, September 2018.

[24] JACKSON, C., SIMON, D. R., TAN, D. S., AND BARTH, A. An evaluation of extended validation and picture-in-picture phishing attacks. In *Proceedings of the International Conference on Financial Cryptography and Data Security* (2007).

[25] JAKOBSSON, M., TSOW, A., SHAH, A., BLEVIS, E., AND LIM, Y.-K. What instills trust? A qualitative study of phishing. In *Financial Cryptography and Data Security* (2007).

[26] KANG, R., DABBISH, L., FRUCHTER, N., AND KIESLER, S. "My Data Just Goes Everywhere": User mental models of the Internet and implications for privacy and security. In *Proceedings of the 11th Symposium on Usable Privacy and Security*.

[27] KROMBHOLZ, K., BUSSE, K., PFEFFER, K., SMITH, M., AND VON ZEZSCHWITZ, E. "If HTTPS were secure, I wouldn't need 2FA": End user and administrator mental models of HTTPS. In *Proceedings of the 40th IEEE Symposium on Security & Privacy* (May 2019).

[28] LIN, E., GREENBERG, S., TROTTER, E., MA, D., AND AYCOCK, J. Does domain highlighting help people identify phishing sites? In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2011).

[29] MALKIN, N., MATHUR, A., HARBACH, M., AND EGELMAN, S. Personalized security messaging: Nudges for compliance with browser warnings. In *Proceedings of the 2nd European Workshop on Usable Security* (2017).

[30] PERKINS, N. Testing out #Safari in both #iOS12 and #macOSMojave and it appears that they removed the company name in the EV trust indicator and replaced it with just the URL. @iangcarroll wonder if they saw your website? Tweet. https://twitter.com/HelferNick/status/1003842702553899009, June 2018.

[31] SCHECHTER, S. E., DHAMIJA, R., OZMENT, A., AND FISCHER, I. The emperor's new security indicators. In *Proceedings of the IEEE Symposium on Security and Privacy* (2007).

[32] SIMKO, C. Why EV SSL is here to stay. https://www.globalsign.com/en/blog/why-ev-ssl-is-here-to-stay/.

[33] SOBEY, J., BIDDLE, R., VAN OORSCHOT, P. C., AND PATRICK, A. S. Exploring user reactions to new browser cues for extended validation certificates. In *Proceedings of the European Symposium on Research in Computer Security* (2008).

[34] SOTIRAKOPOULOS, A., HAWKEY, K., AND BEZNOSOV, K. On the challenges in usable security lab studies: Lessons learned from replicating a study on SSL warnings. In *Proceedings of the 7th Symposium on Usable Privacy and Security* (2011).

[35] SUNSHINE, J., EGELMAN, S., ALMUHIMEDI, H., ATRI, N., AND CRANOR, L. F. Crying wolf: An empirical study of SSL warning effectiveness. In *Proceedings of the 18th USENIX Security Symposium* (2009).

[36] VERIZON. 2018 data breach investigations report. https://enterprise.verizon.com/resources/reports/dbir/.

[37] WHALEN, T., AND M. INKPEN, K. Gathering evidence: Use of visual security cues in web browsers. In *Proceedings of Graphics Interface* (2005).

[38] WU, J., AND ZAPPALA, D. When is a tree really a truck? Exploring mental models of encryption. In *Proceedings of the 14th Symposium on Usable Privacy and Security*.

[39] WU, M., MILLER, R. C., AND GARFINKEL, S. L. Do security toolbars actually prevent phishing attacks? In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2006).

[40] XIONG, A., PROCTOR, R. W., YANG, W., AND LI, N. Is domain highlighting actually helpful in identifying phishing web pages? *Human Factors 59*, 4 (2017), 640–660.

Appendix

A Survey demographics

At the end of each survey we asked participants for information about their age and gender. Table 9, Table 10, and Table 11 show the demographic information for each of the three surveys.

B Full EV survey results

Figure 9 shows the full set of heatmaps for the cross-jurisdiction EV survey. Figure 10 shows the full set of heatmaps for the Safari EV survey.

Table 12 shows the full results of our open-ended coding for the cross-jurisdiction EV survey.

Gender	U.S.	U.K.
Male	55%	44%
Female	44%	55%
Other	0%	0%
Decline to answer	1%	1%
Age		
18-24	15%	31%
25-34	41%	32%
35-44	25%	20%
45-54	12%	11%
55-64	7%	4%
65+	1%	1%
Decline to answer	0%	0%
n	592	650

Table 9: Participant makeup for Chrome cross-jurisdiction EV formatting survey.

Gender	U.S.	U.K.
Male	50%	47%
Female	50%	52%
Other	0%	1%
Decline to answer	0%	0%
Age		
18-24	14%	23%
25-34	39%	32%
35-44	24%	24%
45-54	14%	11%
55-64	7%	8%
65+	2%	2%
Decline to answer	0%	0%
n	290	305

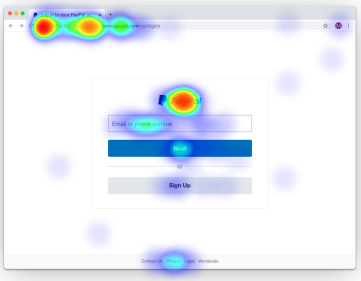
Table 10: Participant makeup for Safari EV formatting study.

Gender	
Male	53%
Female	46%
Other	1%
Decline to answer	1%
Age	
18-24	13%
25-34	42%
35-44	24%
45-54	12%
55-64	7%
65+	2%
Decline to answer	1%
n	1180

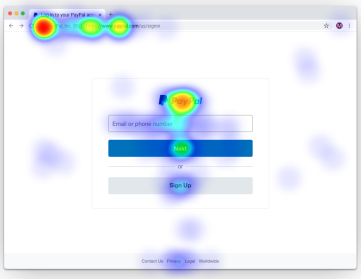
Table 11: Participant makeup for URL formatting study.



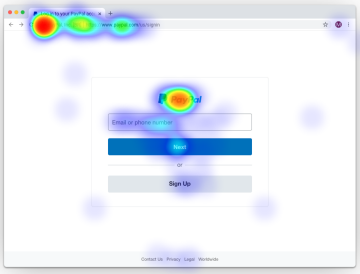
(a) US Cnd1: [US]



(b) US Cnd2: [MX]



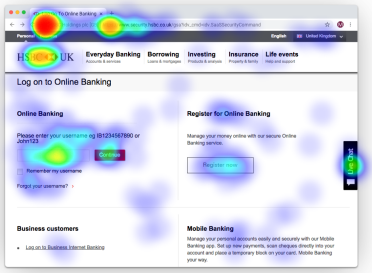
(c) US Cnd3: [RU]



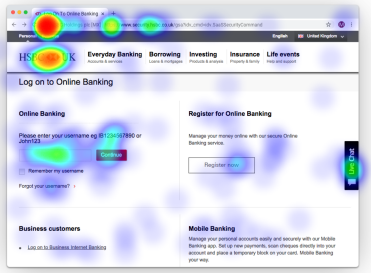
(d) US Cnd4: [BR]



(e) US Cnd5: No CC



(f) UK Cnd1: [GB]



(g) UK Cnd2: [MX]



(h) UK Cnd3: [RUBR]

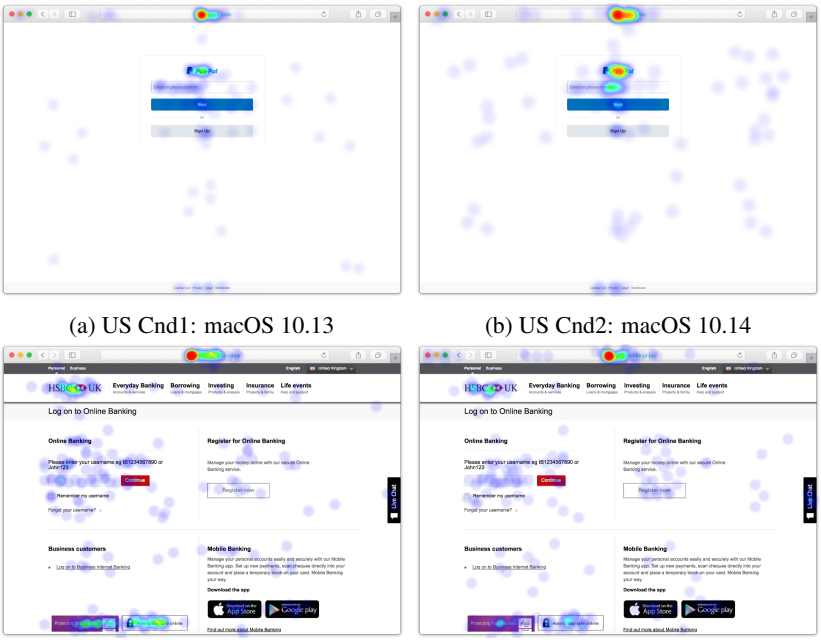


(i) UK Cnd4: [BR]



(j) UK Cnd5: No CC

Figure 9: Heatmaps for Chrome cross-jurisdictional EV surveys.



(a) US Cnd1: macOS 10.13 (b) US Cnd2: macOS 10.14

(c) UK Cnd1: macOS 10.13 (d) UK Cnd2: macOS 10.14

Figure 10: Heatmaps for Safari EV UI survey.

	U.S. Cnd 1	Cnd 2	Cnd 3	Cnd 4	Cnd 5	U.K. Cnd 1	Cnd 2	Cnd 3	Cnd 4	Cnd 5
n	92	120	93	93	115	83	91	81	83	74
Comfortable reasons										
I'm familiar with this website	33%	26%	31%	40%	33%	10%	7%	6%	7%	14%
I see an HTTPS indicator	32%	16%	23%	19%	17%	27%	25%	21%	23%	35%
Page looks normal (unclear)	12%	10%	10%	6%	11%	8%	10%	11%	7%	24%
URL looks normal	8%	8%	15%	9%	10%	1%	4%	2%	4%	4%
Page looks simple / easy to use	9%	7%	9%	10%	7%	18%	16%	9%	16%	15%
"It's safe / secure" (unclear)	5%	4%	8%	9%	3%	11%	16%	9%	11%	7%
Page looks well-designed	2%	2%	0%	3%	0%	4%	8%	14%	12%	3%
I see an EV certificate	1%	1%	2%	1%	1%	1%	0%	1%	1%	1%
Not asking for sensitive information	2%	3%	0%	0%	0%	5%	2%	0%	1%	0%
Uncomfortable reasons										
Country code looks strange	0%	6%	5%	8%	0%	0%	1%	5%	0%	0%
Page does not look normal	1%	1%	2%	4%	3%	1%	1%	0%	7%	3%
Page looks bland	1%	1%	4%	1%	3%	10%	2%	1%	5%	1%
Not sure if it's safe / secure (unclear)	3%	1%	1%	1%	3%	5%	1%	6%	4%	5%
Page asks for sensitive information	2%	1%	0%	3%	2%	0%	0%	0%	0%	0%
I do not see an HTTPS indicator	0%	0%	0%	3%	0%	1%	1%	0%	0%	0%
URL looks odd	0%	1%	0%	1%	1%	1%	2%	2%	2%	3%
Page looks poorly-designed	0%	0%	0%	0%	0%	6%	7%	9%	7%	4%
Lack of green security indicator	0%	0%	0%	0%	0%	0%	0%	1%	5%	0%
Unclear	3%	0%	0%	5%	0%	1%	0%	0%	4%	0%
Other	5%	1%	4%	1%	1%	2%	5%	5%	2%	1%

Table 12: Results of the open-ended question "Can you tell us why you feel that way?" when participants were asked how comfortable they were logging in to a site with different EV country code. Cdn 1 is the topmost condition shown in Figure 4 and Cdn 5 is the bottommost.

RAZOR: A Framework for Post-deployment Software Debloating

Chenxiong Qian*, Hong Hu*, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, Wenke Lee

Georgia Institute of Technology

Abstract

Commodity software typically includes a large number of functionalities for a broad user population. However, each individual user usually only needs a small subset of all supported functionalities. The *bloated* code not only hinders optimal execution, but also leads to a larger attack surface. Recent works have explored *program debloating* as an emerging solution to this problem. Unfortunately, these works require program source code, limiting their real-world deployability.

In this paper, we propose a practical debloating framework, RAZOR, that performs code reduction for deployed binaries. Based on users' specifications, our tool customizes the binary to generate a functional program with minimal code size. Instead of only supporting given test cases, RAZOR takes several control-flow heuristics to infer complementary code that is necessary to support user-expected functionalities. We evaluated RAZOR on commonly used benchmarks and real-world applications, including the web browser FireFox and the close-sourced PDF reader FoxitReader. The result shows that RAZOR is able to reduce over 70% of the code from the bloated binary. It produces functional programs and does not introduce any security issues. RAZOR is thus a practical framework for debloating real-world programs.

1 Introduction

"Entities are not to be multiplied without necessity."
— Occam's Razor

As commodity software is designed to support more features and platforms to meet various users' needs, its size tends to increase in an uncontrolled manner [16, 39]. However, each end-user usually just requires a small subset of these features, rendering the software *bloated*. The bloated code not only leads to a waste of memory, but also opens up unnecessary attack vectors. Indeed, many serious vulnerabilities are rooted in the features that most users never use [31, 35]. Therefore, security researchers are beginning to explore software *debloating* as an emerging solution to this problem.

*The two lead authors contributed equally to this work.

Unfortunately, most initial works on software debloating rely on the availability of program source code [40, 15, 44], which is problematic in real-world use. First, most users do not have access to the source code, and even if they do, it is challenging for them to rebuild the software, diminishing the intended benefits of software bloating. Moreover, users may use the same software in drastically different ways, and thus the unnecessary features to be removed will accordingly vary from user to user. Therefore, to obtain the most benefits, the debloating process should take place after software deployment and should be tailored for each individual user.

Making such a *post-deployment* approach beneficial and usable to end-users creates two challenges: 1) how to allow end-users, who have little knowledge of software internals, to express which features are needed and which should be removed and 2) how to modify the software binary to remove the unnecessary features while keeping the needed ones.

To address the first challenge, we can ask end-users to provide a set of sample inputs to demonstrate how they will use the software, as in the CHISEL work [15]. Unfortunately, programs debloated by this approach only support given inputs, presenting a rather unusable notion of debloating: if the debloated software only needs to support an apriori, fixed set of inputs, the debloating process is as simple as synthesizing a map from the input to the observed output. However, from our experiments, we find that even processing the same input multiple times will result in different execution paths (due to some randomization factors). Therefore, the naive approach will not work even under simplistic scenarios.

In order to practically debloat programs based on user-supplied inputs, we must identify the code that is necessary to completely support required functionalities but is not executed when processing the sample inputs, called *related-code*. Unfortunately, related-code identification is difficult. In particular, it is challenging for end-users (even developers) to provide an input corpus that exercises all necessary code that implements a feature. Furthermore, if the user provides some description of all possible inputs (*e.g.*, patterns), it is still hard to identify all reachable code for those inputs. Thus, we be-

lieve that any debloating mechanism in the post-deployment setting will be based on *best-effort heuristics*. The heuristics should help identify the related-code as much as possible, and meanwhile include minimal functionally unrelated code. Note that techniques like dead code elimination [23, 22] and delta debugging [49, 42] do not apply to this problem because they only focus on either removing static dead code or preserving the program's behavior on a few specific inputs.

We design four heuristics that infer related-code based on the assumption that code paths with more significant divergence represent less related functionalities. Specifically, given one executed path p, we aim to find a different path q such that 1) q has no different instructions, or 2) q does not invoke new functions, or 3) q does not require extra library functions, or 4) q does not rely on library functions with different functionalities. Then, we believe q has functionalities similar to p and treat all code in q as related-code. From 1) to 4), the heuristic includes more and more code in the debloated binary. For a given program, we will gradually increase the heuristic level until the generated program is stable. In fact, our evaluation shows that even the most aggressive heuristic introduces only a small increase of the final code size.

Once all the related-code is identified, we develop a binary-rewriting platform to remove unnecessary code and generate a debloated program. Thanks to the nature of program debloating, our platform does not face the symbolization problem from general binary-rewriting tools [51, 53, 52, 5]. Specifically, a general binary-rewriting tool has to preserve all program functionalities, which is difficult without a reliable disassembling technique and a complete control-flow graph (CFG) [2]. For debloating, we preserve only the functionalities related to the sample inputs, where the disassembling and CFG are available by observing the program execution.

We designed the RAZOR framework to realize the post-deployment debloating. The framework contains three components: **`Tracer`** monitors the program execution with the given sample inputs to record all executed code; **`PathFinder`** utilizes our heuristics to infer more related-code from the executed ones; **`Generator`** generates a new binary based on the output of `Tracer` and `PathFinder`. In the RAZOR framework, we implemented three tracers (two based on dynamic binary instrumentation and one based on a hardware tracing feature), four path finding heuristics, and one binary generator.

To understand the efficacy of RAZOR on post-deployment debloating, we evaluated it on three sets of benchmarks: all SPEC CPU2006 benchmarks, 10 coreutils programs used in previous work, and two real-world large programs, the web browser `Firefox` and the closed-sourced PDF parser `FoxitReader`. In our evaluation, we performed tracing and debloating based on one set of training inputs and tested the debloated program using a different set of functionally similar inputs. Our results show that RAZOR can effectively reduce 70-80% of the original code. At the same time, it introduces only 1.7% overhead to the new binary. We compared RAZOR with CHISEL on debloating 10 coreutils programs and found that CHISEL achieves a slightly better debloating result (smaller code size), but it fails several programs on given test cases. Further, CHISEL introduces exploitable vulnerabilities to the debloated program, such as buffer overflows resulting from the removed boundary checks. RAZOR does not introduce any security issues. We also analyzed the related-code identified by our path finder and found that different heuristics effectively improve the program robustness.

In summary, we make the following contributions:

- **New approach**. We proposed a practical post-deployment debloating framework that works on program binaries. Besides given test inputs, our system supports more inputs of the required functionalities.
- **Open source**. We designed RAZOR as an end-to-end system to produce a minimal functional executable. We implemented our system on an x86-64 Linux system and will open source RAZOR at https://github.com/cxreet/razor.
- **Practical and ready-to-use**. We evaluated RAZOR on real-world programs such as `Firefox` and `FoxitReader` and showed that these programs can be significantly debloated, resulting in better security.

2 Problem

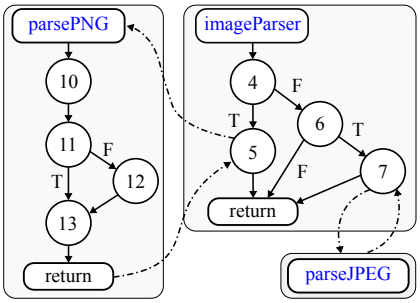
2.1 Motivating Example

Figure 1a shows a bloated program, which is designed to parse image files in different formats. Based on the user-provided `options` (line 4 and 6), the program invokes function `parsePNG` to parse PNG images (line 5) or invokes function `parseJPEG` to handle JPEG images (line 7). In function `parsePNG`, the code first allocates memory to hold the image content and saves the memory address in `img` (line 10). Then it makes sure `img` is aligned to 16-bytes with the macro `ALIGN` (line 11 and 12). Finally, it invokes function `readToMem` to load the image content from file into memory for further processing. Function `parseJPEG` has a structure similar to `parsePNG`, so we skip its details.

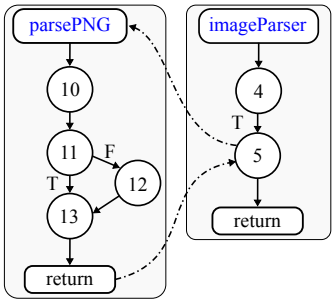
Although the program in Figure 1a merely supports two image formats, it is still bloated if the user only uses it to process `PNG` files. For example, screenshots on iPhone devices are always in PNG format [27]. In this case, the code is bloated with the unnecessary `JPEG` parser, which may contain security bugs [18]. Attackers can force it to process malformed `JPEG` images to trigger the bug and launch remote code execution. In real-world software ecosystem, we can easily find document readers (*e.g.*, Preview on MacOS) that support obsolete formats (*e.g.*, PCX, Sun Raster, TGA). We can debloat these programs to reduce their code sizes and attack surfaces.

```
1  #define MAX_SIZE 0xffff
2  #define ALIGN(v,a) (((v+a-1)/a)*a)
3  void imageParser(char *options, char *file_name) {
4    if (!strcpy(options, "PNG"))
5      parsePNG(file_name);
6    else if (!strcpy(options, "JPEG"))
7      parseJPEG(file_name);
8  }
9  void parsePNG(char *file_name) {
10   char * img = (char *)malloc(MAX_SIZE + 16);
11   if ((img % 16) != 0)
12     img = ALIGN(img, 16);
13   readToMem(img, file_name);
14 }
15 void parseJPEG(char *file_name) { ... }
```

(a) A bloated image parser.



(b) Original control-flow graph.



(c) Debloated control-flow graph

Figure 1: Debloating an image parser. (a) shows the code of the bloated image parser, where the program invokes different functions to handle `PNG` or `JPEG` files based on the options. The control-flow graphs before and after debloating are shown in (b) and (c).

2.2 Program Debloating

In this paper, we develop techniques to remove user-specified unnecessary functionalities from bloated programs. Given a program P that has a set of functionalities $\mathscr{F} = \{F_0, F_1, F_2, ...\}$ and a user specification of necessary functionalities $\mathscr{F}_u = \{F_i, F_j, F_k, ...\}$, our goal is to generate a new program P' that only retains functionalities in $\mathscr{F}_u$ and gracefully refuses requests of other functionalities in $\mathscr{F} - \mathscr{F}_u$.

The program in Figure 1a has two high-level functionalities: parsing `PNG` images and parsing `JPEG` images, while the user specification only requires the first functionality. In this case, the goal of debloating is to generate minimal code that only supports parsing `PNG` files while exiting gracefully if the given images are in other formats. From the simple code we can easily tell that code in the yellow background (*i.e.*, line 6, 7 and 15) is not necessary, so we remove such code in a safe manner: function `parseJPEG` will be simply removed; for line 6 and 7, we should replace the code with fault-handling code to prompt warnings and exit gracefully.

In this paper, we focus on reducing functionalities from software binaries. Specifically, the program P is given as a binary, while the source code like Figure 1a is not available. Instead, we construct the control-flow graph (CFG) from the executable and use it to guide the binary debloating. Figure 1b and Figure 1c show CFGs of the bloated binary and the debloated one, respectively. Black arrows represent intra-procedural jumps, while dotted arrows stand for inter-procedural calls and returns. Originally, function `imageParser` can execute lines 6 and 7 and invoke function `parseJPEG`. In the debloated binary, these lines and functions are not reachable, and the CFG is simplified to Figure 1c. For the vulnerability in the removed code, the new binary prevents attackers from triggering them in the first place.

2.3 Challenges and Solutions

From the previous example, we can find the gap between the user specification and the code removed: users specify that the functionality of parsing `PNG` files is necessary (*i.e.*, others are unnecessary), while we finally remove line 6, line 7, and function `parseJPEG`. However, mapping high-level functionalities to low-level code manually is challenging, especially for large programs. Specifically, this leads to two general challenges of program debloating:

C1. How to express unnecessary functionalities;

C2. How to map functionalities to program code.

One possible solution is to rely on end-users to provide a set of test cases for each necessary/unnecessary functionality so we can inspect the program execution to learn the related program code. Our problem can be rephrased as follows: given the program binary P_b and a set of test cases $T = \{t_i, t_j, t_k, ...\}$, where each test case t_i triggers some functionalities of P_b, we will create a minimal program P_b' that supports and only supports functionalities triggered by the test cases in T.

Test cases help us address challenges **C1** and **C2**. However, it is impossible to provide test cases that cover all related-code of the required functionalities. In this case, some related-code will not be triggered. If we simply remove all never-executed code, the program functionality will be broken. For example, the code at lines 11 and 12 of Figure 1a will make sure the pointer `img` is aligned to 16. Based on the concrete execution context, the return value of `malloc` (at line 10) may or may not satisfy the alignment requirement. If the execution just passes the check at line 11, the simple method will delete line 12 for the minimal code size. However, if the later execution expects an aligned `img`, the program will show unexpected behavior or even crash. Our evaluation in §5.2 shows that simply removing all non-executed code introduces many bugs, even exploitable ones, to the debloated program. Therefore, a test-case-based debloating system faces the following challenge.

C3. How to find more related-code from limited test cases.

To address challenge **C3**, we propose control-flow-based heuristics to infer more related-code that is necessary to support the required functionalities but was missed during our

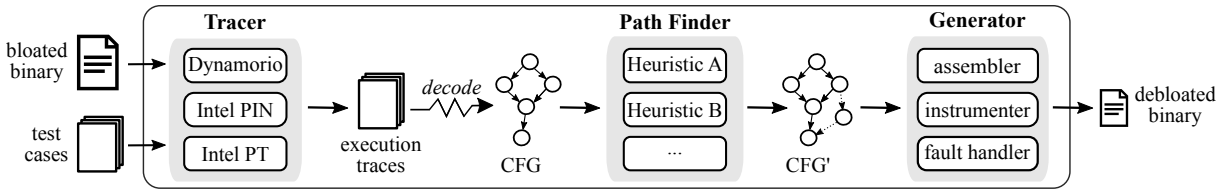


Figure 2: Overview of RAZOR. It takes in the bloated program binary and a set of test cases and produces a minimal, functional binary. `Tracer` collects the program execution traces with given test cases and converts them into a control-flow graph (CFG). `PathFinder` utilizes control-flow-based heuristics to expand the CFG to include more related-code. Based on the new CFG, `Generator` generates the debloated binary.

inspection. Suppose the test cases in T only trigger the execution of instructions in $\mathscr{I} = \{i_0, i_1, i_2, ...\}$, our heuristic will automatically infer more code that is related to the functionalities covered by T. Specifically, we identify a super set $\mathscr{I}' = \mathscr{I} \cup \{i_x, i_y, i_z, ...\}$ and keep all instructions in $\mathscr{I}'$ while removing others to minimize the code size. When debloating the code in Figure 1a, the execution of given test cases does not cover line 12. However, with our heuristics, we will include this line in the debloated program. The evaluation in §5.3 shows that our heuristic is effective in finding related-code paths and introduces only a small increase in code size.

```
Executed Blocks
[0x4005c0,0x4005f2]
[0x400596,0x4005ae]
...

Conditional Branches
[0x4004e3: true]
[0x4004ee: false]
[0x400614: true & false]
...

Indirect Calls/Jumps
[0x400677: 0x4005e6#18,0x4005f6#6]
...
```

Figure 3: A snippet of the collected trace. It includes the range of each executed basic block, the taken/non-taken of each condition branch, and the concrete target of indirect jumps/calls. We also record the frequency of each indirect jump/call target (after #).

3 System Design

Figure 2 shows an overview of our post-deployment debloating system, RAZOR. Given a bloated binary and a set of test cases that trigger required functionalities, RAZOR removes unnecessary code and generates a debloated binary that supports all required features with minimal code size. To achieve this goal, RAZOR first runs the binary with the given test cases and uses `Tracer` to collect execution traces (§3.1). Then, it decodes the traces to construct the program's CFG, which contains only the executed instructions. In order to support more inputs of the same functionalities, `PathFinder` expands the CFG based on our control-flow heuristics (§3.2). The expanded CFG contains non-executed instructions that are necessary for completing the required functionalities. In the end, with the expanded CFG, `Generator` rewrites the original binary to produce a minimal version that only supports required functionalities (§3.3).

3.1 Execution Trace Collection

`Tracer` executes the bloated program with given test cases and records the control-flow information in three categories: (1) executed instructions, including their memory addresses and raw bytes; (2) the taken or non-taken of conditional branches, like `je` that jumps if equal; (3) concrete targets of indirect jumps and calls, like `jmpq *%rax` that jumps to the address indicated by register `%rax`. Our `Tracer` records the raw bytes of executed instructions to handle dynamically generated/modified code. However, instruction-level recording is inefficient and meanwhile most real-world programs only contain static code. Therefore, `Tracer` starts with basic block-level recording that only logs the address of each executed basic block. During the execution, it detects any dynamic code behavior, like both writable and executable memory region (*e.g.*, just-in-time compilation [13]), or overlapped basic blocks (*e.g.*, legitimate code reuse [26]), and switches to the instruction-level recording to avoid missing instructions. A conditional branch may get executed multiple times and finally covers one or both targets (*i.e.*, the fall-through target and the jump target). For indirect jump/call instructions, we log all executed targets and count their frequencies.

Figure 3 shows a piece of collected trace. It contains two executed basic blocks, one at address `0x4005c0` and another at `0x400596`. The trace also contains three conditional branch instructions: the one at `0x4004e3` only takes the `true` target; the one at `0x4004ee` only takes the `false` target; the one at `0x400614` takes both targets. One indirect call instruction at `0x400677` jumps to target `0x4005e6` for 18 times and jumps to target `0x4005f6` for six times. As the program only has static code, `Tracer` does not include the instruction raw bytes.

We find that it is worthwhile to use multiple tools to collect the execution trace. First, no mechanism can record the trace completely and efficiently. Software-based instrumentation can faithfully log all information but introduces significant overhead [7, 25, 6]. Hardware-based logging can record efficiently [20] but requires particular hardware and may not guarantee the completeness (*e.g.*, data loss in `Intel PT` [17]). Second, program executions under different tracing environ-

ments will show divergent paths. For example, `Dynamorio` always expands the file name to its absolute path, leading to different executed code in some programs (*e.g.*, `vim`). Therefore, we provide three different implementations (details in §4.1) with different software and hardware mechanisms. End-users can choose the best one for their requirement or even merge traces from multiple tools for better code coverage.

CFG construction. With the collected execution traces, RAZOR disassembles the bloated binary and constructs the partial control-flow graph (CFG) in a reliable way. Different from previous works that identify function boundaries with heuristics [52, 51, 3, 4, 45], RAZOR obtains the accurate information of instruction address and function boundary from the execution trace. For example, we can find some of all possible targets of indirect jumps and calls.

Starting from such reliable information, we are able to identify more code instructions [47]. For conditional branch instructions, both targets are known to us. Even if one target is not executed, we can still reliably disassemble it. For indirect jumps, we can identify potential jump tables with specific code patterns [53]. For example, `jmpq *0x4e65a0(,%rdx,8)` indicates a jump table starting from address `0x4e65a0`. By identifying more instructions, we are able to include them in the binary if our heuristic treats them as related-code.

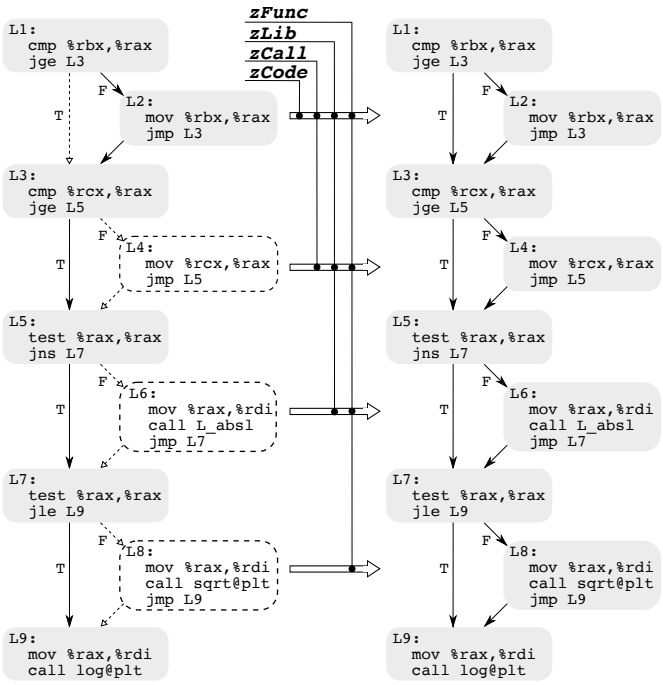


Figure 4: Identifying related-code with different heuristics. Dashed branches and blocks are not executed and thus are excluded from the left CFG, while others are executed.

3.2 Heuristic-based Path Inference

Considering the challenge of generating test cases to cover all code, we believe no perfect method can completely identify all missed related-code. As the first work trying to mitigate the problem, we adopt the *best-effort heuristic* approach to include more related-code. Next, we present these heuristics one by one, from the conservative one (including less code) to the aggressive one (including more code):

(1) Zero-code heuristic (zCode). This heuristic adds new edges (*i.e.*, jumps between basic blocks) into the CFG. For conditional branch instructions that only have one target taken (the fall-through target or the jump target), `PathFinder` checks whether the non-taken target is already in the CFG (*i.e.*, reached through other blocks). If so, `PathFinder` permits the jump from this instruction to the non-taken target. This heuristic does not add any new instructions and thus will not affect the code reduction.

Figure 4 shows an example of related-code identification with heuristics, with the original CFG on the left and the expanded CFG on the right. The code is designed to calculate `log(sqrt(absl(max(rax,rbx,rcx))))`. Dashed branches and blocks are not executed during tracing, while others are executed. The original execution path is `L1`→`L2`→`L3`→`L5`→`L7`→`L9`. Blocks `L4`, `L6`, `L8`, and the branch `L1`→`L3` are missed in the original CFG. With the zCode heuristic, `PathFinder` adds branch `L1`→`L3` into the new CFG, as `L3` is the non-taken branch of the conditional jump `jge L3` in `L1` and it is already reached from `L2` in the current CFG.

(2) Zero-call heuristic (zCall). This heuristic includes alternative execution paths that do not trigger any function call. With this heuristic, `PathFinder` starts from the non-taken target of some conditional branches and follows the control-flow information to find new paths that finally merge with the executed ones. If such a new path does not include any `call` instructions, `PathFinder` includes all its instructions to the CFG. When `PathFinder` walks through non-executed instructions, we do not have the accurate information for stable disassembling or CFG construction. Instead, we rely on existing mechanisms [53, 3] to perform binary analysis. When applying the zCall heuristic on the example in Figure 4, `PathFinder` further includes block `L4`, and path `L3`→`L4`→`L5`, as this new path merges with the original one at `L5` and does not contain any `call` instruction.

(3) Zero-libcall heuristic (zLib). This heuristic is similar to zCall, except that `PathFinder` includes the alternative paths more aggressively. The new path may have `call` instructions that invoke functions within the same binary or external functions that have been executed. However, zLib does not allow calls to non-executed external functions. In Figure 4, with this heuristic, `PathFinder` adds block `L6` and path `L5`→`L6`→`L7` to the CFG, as that path does not have any call to non-executed external functions.

(4) Zero-functionality heuristic (zFunc). This heuristic further allows including non-executed external functions as long as they do not trigger new high-level functionalities. To correlate library functions with functionalities, we check their

Algorithm 1: Path-finding algorithm.

Input: CFG - the input CFG; libcall_groups - the library call groups.
Output: CFG′ - the expanded CFG
CFG′ ← CFG
/* iterate over each conditional branch */
1 **for** *cnd_br* ∈ *CFG*:
2 nbb = get_non_taken_branch(cnd_br)
3 **if** *nbb* == *NULL*: **continue**
4 **if** *heuristic* >= *zCode* ***and*** *nbb* ∈ *CFG*:
5 CFG′ = CFG′ ∪ {cnd_br→nbb}
6 paths = get_alternative_paths(CFG′, nbb)
7 **for** *p* ∈ *paths*:
8 include = false
9 **if** *heuristic* == *zCall*: include = !has_call(p)
10 **elif** *heuristic* == *zLib*: include = !has_new_libcall(p)
11 **elif** *heuristic* == *zFunc*:
12 include = !has_new_func(CFG′, p, libcall_groups)
13 **if** *include*:
14 CFG′ = CFG′ ∪ p

descriptions and group them manually. For `libc` functions, we classify the ones that fall into the same subsection in [32] to the same group. For example, `log` and `sqrt` are in the subsection `Exponentiation and Logarithms`, and thus we believe they have similar functionalities. With this heuristic, `PathFinder` includes block `L8` and path `L7`→`L8`→`L9`, as `sqrt` has a functionality similar to the executed function `log`.

Algorithm 1 shows the steps that `PathFinder` uses to find related-code that completes functionalities. For each conditional branch in the input CFG (line 1), the algorithm invokes the function `get_non_taken_branch` to get the non-taken branch (line 2). If both branches have been taken, the algorithm proceeds to the next conditional branch (line 3). Otherwise, `PathFinder` starts to add code depending on the given heuristic (line 4 to 14). If the non-taken branch is reachable in the current CFG (line 4), zCode enables the new branch in the output CFG (line 5). If the heuristic is more aggressive than zCode, `PathFinder` first gets all alternative paths that start from the non-taken branch and finally merges with some executed code (line 6). Then, it iterates over all paths (line 7) and calls corresponding checking functions (*i.e.*, `has_call`, `has_new_libcall`, and `has_new_func`) to check whether or not the path should be included (line 9 to 12). In the end, `PathFinder` adds the path to the output CFG if it satisfies the condition (line 14).

3.3 Debloated Binary Synthesization

With the original bloated binary and the expanded CFG, `Generator` synthesizes the debloated binary that exclusively supports required functionalities. First, it disassembles the original binary following the expanded CFG and generates a pseudo-assembly file that contains all necessary instructions. Second, `Generator` modifies the pseudo-assembly to create a valid assembly file. These modifications symbolize basic blocks, concretize indirect calls/jumps, and insert fault handling code. Third, it compiles the assembly file into an object file that contains machine code of the necessary instructions. Fourth, `Generator` copies the machine code from the object file into a new code section of the original binary. Fifth, `Generator` modifies the new code section to fix all references to the original code and data. Finally, `Generator` sets the original code section non-executable to reduce the code size. We leave the original code section inside the debloated program to support the potential read from it (*e.g.*, jump tables in code section for implementing `switch` [11]). We discuss this design choice in §6.

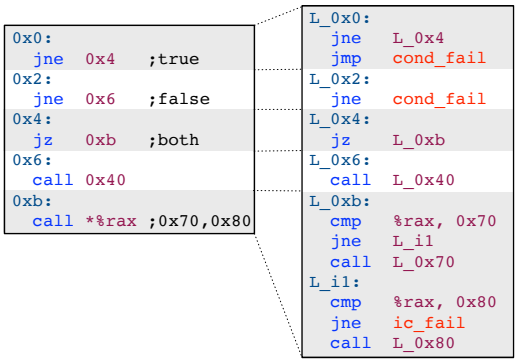


Figure 5: Synthesize debloated assembly file. Each basic block is assigned a unique label; indirect calls are expanded with comparisons and direct calls; fault handling code is inserted.

3.3.1 Basic Block Symbolization

We assign a unique label to each basic block and replace all its references with the label. Specifically, we create the label `L_addr` for the basic block at address `addr`. Then, we scan all direct jump and call instructions and replace their concrete target addresses with corresponding labels. In this way, the assembler will generate correct machine code regardless of how we manipulate the assembly file. Figure 5 shows an assembly file before and after the update, illustrating the effect of basic block symbolization. Before the update, all call and jump instructions use absolute addresses, like `jne 0x6` in basic block `0x0`. After the symbolization, the basic block at `0x6` is assigned the label `L_0x6`, while instruction `jne 0x6` is replaced with `jne L_0x6`. Similarly, instruction `call 0x40` in block `0x06` is replaced with `call L_0x40`. One special case is the conditional branch `jne 0x6` in basic block `0x2`. In the extended CFG, it only takes the fall-through branch, which means that jumping to block `0x6` should not be allowed in the debloated binary. Therefore, instead of replacing `0x6` with symbol `L_0x6`, we redirect the execution to the fault handling code `cond_fail` (will discuss in §3.3.3). Note that basic block symbolization only updates explicit use of basic block addresses, *i.e.*, as direct call/jump targets. We handle the implicit address use, like saving function address into memory for indirect call, with the indirect call/jump concretization.

3.3.2 Indirect Call/Jump Concretization

Indirect call/jump instructions use implicit targets that are loaded from memory or calculated at runtime. We have to make sure all possible targets point to the new code section. For the sake of simplicity, we use the term `indirect call` to cover both indirect calls and indirect jumps.

With the execution traces, `Generator` is able to handle indirect calls in two ways. The first method is to locate constants from the original binary that are used as code addresses and replace them with the corresponding new addresses, as in [52, 51]. However, this method requires a heavy tracing process that records all execution context and a time-consuming data-flow analysis. Therefore, it is impractical for large programs. The second method is to perform the address translation before each indirect call, as in [53]. In particular, we create a map from the original code addresses to the new ones. Before each indirect call, we map the old code address to the new one and transfer the control-flow to the new address.

Our `Generator` takes a method similar to the second one, but with different translations for targets within the same module (named local targets) and targets outside the module (named global targets). For local targets, we define a concrete policy for each indirect call instruction. Specifically, we replace the original call with a set of compare-and-call instructions, one for each local target that is executed by *this* instruction at tracing. Then, we call the new address of the matched old addresses. Global targets have different addresses in multiple runs because of the address space layout randomization (ASLR). We use a per-module translation table to solve this problem. Different from previous work that creates a translation table for all potential targets in the module [53], our translation table contains only targets that are ever invoked by other modules. At runtime, if the target address is outside the current module, we use a global translation function to find the correct module and look up its translation table to get the correct new address to invoke.

Figure 5 gives an example of indirect call concretization. In the execution trace, instruction `call *%rax` in block `0xb` transfers control to function at `0x70` and `0x80`. Our concretization inserts two `cmp` instructions, one to compare with the address `0x70` and another to compare with `0x80`. For any successful comparison, `Generator` inserts a direct call to transfer the control-flow to the corresponding new address.

Security benefit. Our design achieves a stronger security benefit on control-flow protection over previous methods. For example, the previous work binCFI [53] uses a map to contain all valid code addresses, regardless of which instruction calls them. Thus, any indirect call instruction can reach all possible targets, making the protection vulnerable to existing bypasses [12, 43, 9]. Our design is functionally equivalent to creating one map for each indirect call, which contains both the targets obtained from the trace and the targets inferred by our `PathFinder`. For inter-module indirect calls, we limit the targets to a small set that is ever invoked by external modules. In this way, attackers who try to change the control flow will have fewer choices, and the debloated binary will be immune to even advanced attacks.

Frequency-based optimization. Depending on the number of executed targets, we may insert many compare-and-call instructions that will slow the program execution. For example, one indirect call instruction in `perlbench` benchmark of SPEC CPU2006 has at least 132 targets, and each target is invoked millions of times. To reduce the overhead, we rank all targets with their execution frequencies and compare the address with high-frequent targets first. The targets inferred from heuristics have a frequency of zero. With this optimization, we can reduce the overhead significantly.

3.3.3 Fault Handling

Running a debloated binary may reach removed code or disabled branches for various reasons, such as a user's temporal requirement for extra functionalities or malicious attempts to run unnecessary code. We redirect any such attempt to a fault handler that exits the execution and dumps the call stack. Specifically, for conditional jump instructions with only one target taken, we intercept the branch to the non-taken target to hook any attempt of the invalid jump. Similarly, for indirect call instructions, if no allowed target matches the runtime target, we redirect the execution to the fault handler.

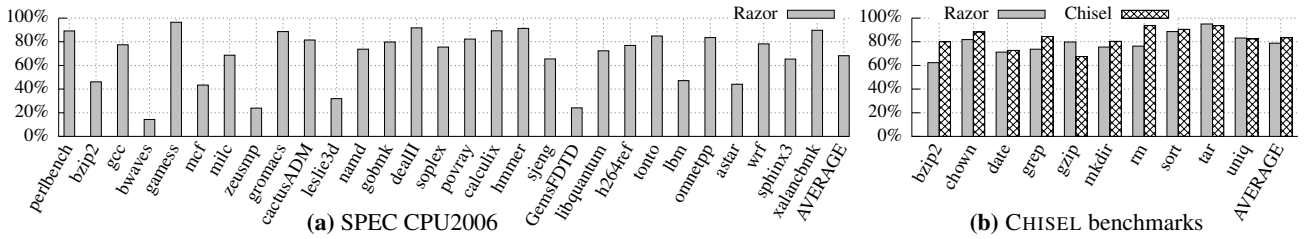
Figure 5 includes examples of hooking failed conditional jumps and indirect calls. For instruction `jne 0x4` in block `0x0`, we insert `jmp cond_fail` to redirect the branch to the fall-through target to the fault handler `cond_fail`. Similarly, we update instruction `jne 0x6` with `jne cond_fail` to prevent jumping to the non-executed target. For conditional branch `jz 0xb` which has both targets taken, we do not insert any code. For instruction `call *%rax`, we insert code `jne ic_fail` in the case that all allowed targets are different from the real-time one.

4 Implementation

We implement a prototype of RAZOR with 1,085 lines of C code, 514 lines of C++ code, and 4,034 lines of python code, as shown in Table 1. The prototype currently supports x86-64 ELF binaries. Our design is platform-agnostic and we plan to support other binary formats from different architectures. We tried our system on system libraries (*e.g.*, libc.so, libm.so) and report our findings in §6.

4.1 `Tracer` Implementations

As we discussed in §3.1, each tracing method has different benefits and limitations, such as the tracing efficiency and completeness. We provide three different implementations of



(a) SPEC CPU2006 **(b)** CHISEL benchmarks

Figure 6: Code size reduction on two benchmarks. We use RAZOR to debloat both SPEC CPU2006 benchmarks and CHISEL benchmarks without any path finding and achieve 68.19% and 78.8% code reduction. CHISEL removes 83.4% code from CHISEL benchmarks.

Component	Tracer	PathFinder	Generator	Total
C	1,085	0	0	1,085
C++	514	0	0	514
Python	218	743	3,073	4,034

Table 1: Implementation of different RAZOR components.

`Tracer` in RAZOR so that users can choose the best one for their purpose. In our evaluation, we use software-based instrumentation to collect complete traces for simple programs, and use a hardware-based method to efficiently get trace from large programs.

Tracing with software instrumentation. We use the dynamic instrumentation tools `Dynamorio` [7] and `Pin` [25] to monitor the execution of the bloated program. Both tools provide instrumentation interfaces at function level, basic block level, and instruction level. We implement three instrumentation passes to collect control-flow information. First, at the beginning of each basic block we record its start address; second, for each conditional jump instruction, we insert two pieces of code between the instruction and its two targets to log the taken information; third, before each indirect call and jump instruction, we record the concrete target for each invocation. At runtime, we remove the basic block instruction immediately after its first execution to avoid unnecessary overhead. Similarly, we remove the instrumentation of conditional branches once that branch has been taken. However, we keep the instrumentation of indirect call and jump instructions, as we do not know the complete set of targets.

Tracing with hardware feature. Considering the overhead of software instrumentation, we provide an efficient `Tracer` built on Intel Processor Trace (`Intel PT`) [20]. `Intel PT` records the change of flow information in a highly compressed manner: the `TNT` packet describes whether one conditional branch is taken or non-taken; the `TIP` packet records the target of indirect branches, like indirect call and return. As `Intel PT` directly writes the trace to physical memory without touching the page table or memory cache, it achieves the most efficient tracing. Our `Tracer` decodes the traces from `Intel PT` to get necessary control-flow information. We can use other hardware features available on different platforms to implement efficient `Tracer`, like branch trace store (BTS) on Intel CPUs or program flow trace (PTM) on ARM CPUs.

4.2 Update ELF Exception Handler

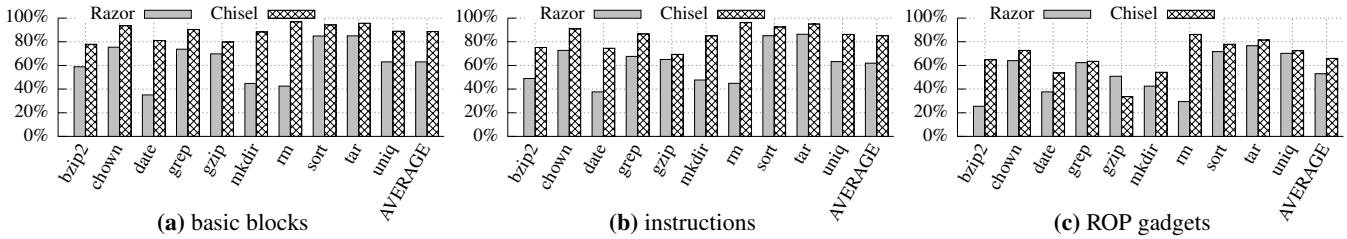
ELF binaries generated by gcc and clang adopt the table-based exception handling [46] to provide stack unwind and exception handler information. Specifically, ELF keeps a table in the `.eh_frame_hdr` section, one entry per function. Each entry indicates the location of a frame description entry (FDE) in the `.eh_frame` section, which further specifies the location of the language-specific data area (LSDA). The LSDA region in the `.gcc_except_table` section contains the concrete address of exception handlers, called landingpad.

We have to replace the old value of all landingpads in `.gcc_except_table` with the new ones. However, the challenge is that the value in `.gcc_except_table` is encoded in the LEB128 format – a variable-length encoding that may have different lengths for different values. Since we update the old address with a different one, the encoding of the new address may take more bytes and thus cannot be put into the original location. To solve this problem, we update the section layout of the binary to create more space for the new address. Specifically, we shrink the table inside the `.eh_frame_hdr` section to exclude entries of non-executed functions. Recall that the given test cases only trigger part of the functionalities, and the non-executed functions will not be included in the debloated binary. Then we shift `.eh_frame` and `.gcc_except_table` sections to get more space for our update of landingpad values.

5 Evaluation

In this section, we perform extensive evaluation in order to understand RAZOR regarding the following aspects:

- **Code reduction.** How much code can RAZOR reduce from the original bloated binary? (§5.1)
- **Functionality.** Does the debloated binary support the functionalities in given test cases? (§5.2) How effective is `PathFinder` in finding complementary code? (§5.3)
- **Security.** Does RAZOR reduce the attack surface of the debloated binaries? (§5.4)



(a) basic blocks **(b)** instructions **(c)** ROP gadgets

Figure 7: Reduction of basic blocks, instructions, and ROP gadgets, debloated by RAZOR and CHISEL from CHISEL benchmarks.

- **Performance.** How much overhead does RAZOR introduce into the debloated binary? (§5.5)
- **Practicality.** Does RAZOR work on commonly used software in the real world? (§5.6)

Experiment setup. We set up three sets of benchmarks to evaluate RAZOR: 29 SPEC CPU2006 benchmarks, including 12 C programs, seven C++ programs, and 10 Fortran programs; 10 coreutils programs used in the CHISEL paper[1] [15]; the web browser `Firefox` and the close-source PDF reader `FoxitReader`. We use the software-based tracing tools that rely on `Dynamorio` and `Pin` to collect the execution traces of SPEC and CHISEL benchmarks, to get accurate results; for the complicated programs `Firefox` and `FoxitReader`, we use the hardware-based tracing tool (relying on `Intel PT`) to guarantee the execution speed to avoid abnormal behaviors. We ran all the experiments on a 64-bit Ubuntu 16.04 system equipped with Intel Core i7-6700K CPU (with eight 4.0GHz cores) and 32 GB RAM.

5.1 Code Reduction

We applied RAZOR on SPEC CPU2006 benchmarks and CHISEL benchmarks to measure the code size reduction. For SPEC benchmarks, we treated the `train` dataset as the user-given test cases. For CHISEL benchmarks we obtained test cases from the paper's authors. We did not apply any heuristics of path finding for this evaluation. As RAZOR works on binaries, we cannot measure the reduction of source code lines. Instead, we compare the size of the executable memory region before and after the debloating, specifically, the program segments with the executable permission. Figure 6a shows the code reduction of SPEC benchmarks debloated by RAZOR. Figure 6b shows the code reduction of CHISEL benchmarks, debloated by CHISEL and RAZOR.

On average, RAZOR achieves 68.19% code reduction for SPEC benchmarks and 78.8% code reduction for CHISEL benchmarks. Especially for `dealII`, `hmmer`, `gamess`, and `tar`, RAZOR removes more than 90% of the original code. For `bwaves`, `zeusmp`, and `GemsFDTD`, RAZOR achieves less than 30% code reduction. We investigated these exceptions and found that these programs are relatively small and the `train` datasets already trigger most of the code.

Meanwhile, CHISEL achieves 83.4% code reduction on CHISEL benchmarks. For seven programs, CHISEL reduces more code than RAZOR, while RAZOR achieves higher code reduction than CHISEL for the other three programs. CHISEL tends to remove more code as long as the execution result remains the same. For example, variable initialization code always gets executed at the function beginning. CHISEL will remove it if the variable is not used in the execution, while RAZOR will keep it in the debloated binary. Although CHISEL performs slightly better than RAZOR on code reduction, we find that the debloated binaries from CHISEL suffer from robustness issues (§5.2) and security issues (§5.4).

Other reduction metrics. We also measured RAZOR's effectiveness on reducing basic blocks (Figure 7a) and instructions (Figure 7b) from CHISEL benchmarks and compared these results with those achieved by CHISEL. On average, RAZOR removes 53.1% of basic blocks and 63.3% of instructions from the original programs, while CHISEL reduces 66.0% of basic blocks and 88.5% of instructions from the same set of programs. This result is consistent with the code size reduction, where RAZOR reduces less code, as it can neither remove any executed-but-unnecessary blocks or instructions, nor utilize compiler to aggressively optimize the debloated code.

5.2 Functionality Validation

We ran the debloated binaries in CHISEL benchmarks against given test cases to understand their robustness. For each benchmark, we compiled the original source code to get the original binary and compiled the debloated source code from CHISEL to get the CHISEL binary. Then, we used RAZOR to debloat the original binary with given test cases, generating the RAZOR binary. Next, we ran the original binary, the CHISEL binary, and the RAZOR binary again with the test cases. We examine the execution results to see whether the required functionalities are retained in the debloated binaries.

Table 2 shows the validation result. RAZOR binaries produce the same results as those from the original binaries for all test cases of all programs (the last column), showing the robustness of the debloated binaries. Surprisingly, CHISEL binaries only pass the tests of three programs (*i.e.*, `chown`,

[1] We appreciate the help of CHISEL authors for sharing the source code and their benchmarks.

Program	Version	# of Tests	Failed by Chisel				Failed by Razor
			W	I	C	M	
bzip2	1.0.5	6	**2**	–	**2**	–	– (zLib)
chown	8.2	14	–	–	–	–	– (zFunc)
date	8.21	50	**5**	–	**3**	–	– (zLib)
grep	2.19	26	–	–	–	**6**	– (zLib)
gzip	1.2.4	5	–	**1**	–	–	– (zLib)
mkdir	5.2.1	13	–	–	–	**1**	– (zLib)
rm	8.4	4	**2**	–	–	–	– (zFunc)
sort	8.16	112	–	–	–	–	– (zCall)
tar	1.14	26	**3**	–	–	**4**	– (zCall)
uniq	8.16	16	–	–	–	–	– (zCall)

Table 2: Failed test cases by RAZOR binaries and CHISEL binaries. CHISEL failed some tests with different reasons: **W**rong operations, **I**nfinite loop, **C**rashes, and **M**issing output. For RAZOR binaries, we show the heuristic that makes the program pass all tests.

`sort`, and `uniq`) and trigger some unexpected behaviors for the other seven programs. Considering that CHISEL verifies the functionality of the debloating binary, such a low passing rate is confusing. We checked these failed cases and the verification process of CHISEL and found four common issues.

Wrong operation. The debloated program performs unexpected operations. For examples, `bzip2` should decompress the given file when the test case specifies the `-d` option. However, the binary debloated by CHISEL always decompresses the file regardless of what option is used. We suspect that CHISEL only uses one test case of decompression to debloat the program and thus removes the code that parses command line options.

Infinite loop. CHISEL may remove loop condition checks, leading to infinite loops. For example, `gzip` fails one test case because it falls into a loop in which CHISEL drops the condition check. We believe the reason is that the test case used by CHISEL only iterates the loop one time. The verification step of CHISEL should identify this problem. However, we found that the verification script adopts a small timeout (*e.g.*, 0.1s) and treats any timeout as a successful verification. Therefore, it cannot detect any infinite loops.

Crashes. The debloated binary crashes during execution. For example, `date` crashes three test cases because CHISEL removes the check on whether the parameters of `strcmp` are `NULL`. `bzip2` crashes three test cases for the same reason.

Missed output. CHISEL removes code for printing out on stdout and stderr, leading to missed results. For example, `grep` fails six test cases, as the binary does not print out any result even through it successfully finds matched strings. We find that in the verification script of CHISEL, all output of the debloated binaries is redirected to the `/dev/null` device. Therefore, it cannot detect any missing or inconsistent output.

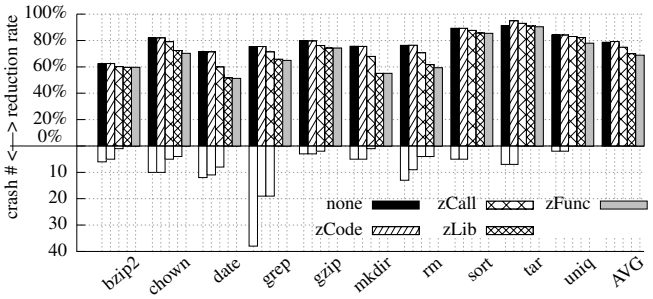


Figure 8: Path finding on CHISEL benchmarks with different heuristics. The top part is the code reduction, while the bottom part is the number of crashes. 'none' means no heuristic is used.

5.3 Effectiveness of Path Finding

We use two sets of experiments to evaluate the effectiveness of `PathFinder` on finding the related-code of required functionalities. First, we use RAZOR to debloat programs with different heuristics, from the empty heuristic to the most aggressive zFunc heuristic, aiming to find the least aggressive heuristic for each program. Second, we perform N-fold cross validation to understand the robustness of our heuristic. In this subsection, we focus on the first experiment and leave the N-fold cross validation in §5.6.1.

We tested RAZOR on CHISEL benchmarks as follows: (1) design training inputs and testing inputs that cover the same set of functionalities; (2) trace programs with the training inputs and debloat them with none, zCode, zCall, zLib, and zFunc heuristics; (3) run debloated binaries on testing inputs and record the failed cases. The setting of evaluating `PathFinder` is given in Table 7 of Appendix A. We use the same options for training inputs and testing inputs to make sure that the debloated binaries are tested for the same functionalities as those triggered by the training inputs. The difference is the concrete value for each option or the file to process. For example, when creating folders with `mkdir`, we use various parameters of the option `-m` for different file mode sets. For program `bzip2` and `gzip`, we use different files for training and testing.

Figure 8 presents our evaluation result, including the code reduction (the top half) and the number of failed test cases (the bottom half) under different heuristics. We can see that debloating with a more aggressive heuristic leads to more successful executions. All binaries generated without any heuristic fail on some testing inputs. `grep` fails on all 38 testing inputs, while `chown` and `rm` fail more than half of all tests. The zCode heuristic helps mitigate the crash problem, like making `grep` work on 19 test cases. However, all generated binaries still fail some inputs. The zCall heuristic further improves the debloating quality. For program `sort`, `tar`, and `uniq`, it avoids all previous crashes. With the zLib heuristic, only two programs (*i.e.*, `chown` and `rm`) still have a small number of failures. In the end, debloating with the zFunc heuristic

```
1 int fillbuf(...) { ...
2   if (minsize <= maxsize_off)
3     if (...) ...
4   newalloc = newsize+ ...;
5 }
```

Figure 9: A crash case reduced by applying zCode heuristic.

```
1 int fts_safe_changedir(..,){
2   if (dir) {
3     tmp=strcmp(dir,".."); ...
4   } ...
5 }
```

Figure 10: A crash case reduced by applying zFunc heuristic.

```
1 int compare(line *a,line *b) {
2   alen = a->length - 1UL;
3   blen = b->length - 1UL;
4   if (alen == 0UL) {
5     diff = -(blen != 0UL);
6   } else {
7     if (blen == 0UL) {
8       diff = 1;
9     } else { ... }
10 }}
```

Figure 11: A crash case reduced by applying zCall heuristic.

```
1 int main(...) { ...
2   fail = make_dir(..);
3   if (!fail) {
4     if (!create_parents) {
5       if (!dir_created) {
6         tmp_7=gettext("error");
7         error(0,17,tmp_7,tmp_6);
8         fail = 1;
9         ...
10 }}}}
```

Figure 12: A crash case reduced by applying zLib heuristic.

Program	CVE	Orig	Chisel	Razor
bzip2-1.0.5	CVE-2010-0405	✓		
	CVE-2011-4089*	✗		
	CVE-2008-1372	✗	✔	
	CVE-2005-1260	✗	✔	
chown-8.2	CVE-2017-18018*	✓	✘	✘
date-8.21	CVE-2014-9471*	✓	✘	
grep-2.19	CVE-2015-1345*	✓	✘	✘
	CVE-2012-5667	✗	✔	
gzip-1.2.4	CVE-2005-1228*	✓	✘	✘
	CVE-2009-2624	✓		
	CVE-2010-0001	✓	✘	✘
mkdir-5.2.1	CVE-2005-1039*	✓		
rm-8.4	CVE-2015-1865*	✓		
sort-8.16	CVE-2013-0221*	✗		
tar-1.14	CVE-2016-6321*	✓	✘	
uniq-8.16	CVE-2013-0222*	✗		

Table 3: Vulnerabilities before and after debloating by RAZOR and CHISEL. ✓ means the binary is vulnerable to the CVE, while ✗ mean it is not vulnerable. CVEs with ∗ are evaluated in [15].

reduces all crashes in all programs.

Interestingly, although aggressive heuristics introduce more code to the debloated binary (shown in the top of Figure 8), they do not significantly decrease the code reduction. Without any heuristic, the average code reduction rate of 10 programs is 78.7%. The number is reduced by −0,4%, 3.8%, 8.8%, and 12.6% when applying zCode, zCall, zLib, and zFunc heuristics, respectively. Therefore, even with the most aggressive zFunc heuristic, the code reduction does not decrease heavily. At the same time, all crashes are resolved, showing the benefits of applying heuristics. Note that the zCode heuristic slightly increases the code reduction over the no heuristic case, as it enables more branches of conditional jumps, which in turn reduces the instrumentation of failed branches.

We investigated the failed cases mitigated by different heuristics and show some case studies as follows:

(1) The **zCode** heuristic enables the non-taken branch for executed conditional jumps. Figure 9 shows part of the function `fillbuf` of program `grep` that fails if we do not use the zCode heuristic. The training inputs always trigger the `true` branch of the condition at line 2 and jump to line 3, which in turn reach line 4. However, in the execution of testing inputs, the conditional at line 2 takes the false branch (i.e., `minsize > maxsize_off`) and triggers the jump from line 2 to line 4. This branch is not allowed from execution traces. The zCode heuristic enables this branch, as line 4 has been reached in the previous execution.

(2) The **zCall** heuristic includes alternative paths that do not trigger any `call` instructions. Figure 11 shows an example where the zCall heuristic helps include necessary code in the debloated binary. Function `compare` in program `sort` uses a sequence of comparisons to find whether two text lines are different. Since the training inputs have no empty lines, the condition at line 4 and line 7 always fails. However, the testing inputs contain empty lines, which makes these two conditional jumps take the `true` branches. The zCode heuristic adds lines 5 and 8 and related branches to the debloated program, which effectively avoids this crash.

(3) The **zLib** heuristic allows extra calls to native functions or library functions if they have been used in traces. It helps avoid a crash in program `mkdir` when we use the debloated binary to change the file mode of an existing directory. Figure 12 shows the related code, which crashes because of the missing code from line 6 to line 9. Since `mkdir` does not allow changing the file mode of an existing directory, the code first invokes function `gettext` to get the error message and then calls library function `error` to report the error. The zLib heuristic includes this path in the binary because both `gettext` and `error` are invoked by some training inputs.

(4) The **zFunc** heuristic includes alternative paths that invoke similar library functions. Figure 10 shows the code that causes `rm` to fail without this heuristic. When `rm` deletes a folder that contains both files and folders, it triggers the code at line 3 to check whether it is traversing to the parent directory. Since the training inputs never call `strcmp`, the debloated binary fails even with the zLib heuristic. However, the training inputs ever invoke function `strncmp`, which has the functionality similar to `strcmp` (*i.e.*, string comparison). Therefore, the zFunc heuristic adds this code in the debloated binary.

The results show that `PathFinder` effectively identifies related-code that completes the functionalities triggered by training inputs. It enhances the robustness of the debloated binaries while retaining the effectiveness of code reduction.

5.4 Security Benefits

We count the number of reduced bugs to evaluate the security benefit of our debloating. For each program in the CHISEL benchmark, we collected all its historical vulnera-

bilities, including the ones shown in the current version and the ones only in earlier versions. For the former bugs, we check whether the buggy code has been removed by the debloating process. If so, the debloating process helps avoid related attacks. For the latter bugs, we figure out whether their patches are retained in the debloated binary. If not, the debloated process makes the program vulnerable again. Table 3 shows our evaluation result, including 16 CVEs related to CHISEL benchmarks. 13 bugs are shown in the current version, and 10 of them are evaluated in [15] (followed by *). Three bugs only exist in older versions (*i.e.*, CVE-2010-0405, CVE-2009-2624, and CVE-2010-0001).

RAZOR successfully removes four CVEs from the original binaries and does not introduce any new bugs. Specifically, CVE-2017-18018 in `chown`, CVE-2015-1345 in `grep`, CVE-2005-1228 and CVE-2010-0001 in `gzip` are removed in the debloated binaries. Six vulnerabilities from `bzip`, `date`, `gzip`, `mkdir`, `rm`, and `tar` remain, as the test cases execute related vulnerable code. Another six vulnerabilities are not caused by the binary itself. For example, CVE-2011-4089 is caused by the race condition of the bash script `bzexe`, not by the `bzip2` binary. Therefore, RAZOR will not disable such bugs.

With a more aggressive code removal policy, CHISEL disables two more CVEs than RAZOR, but unfortunately brings three old bugs to the debloated binaries. Specifically, CHISEL removes the vulnerable code of CVE-2014-9471 from `date` and the code of CVE-2016-6321 from `tar`. Meanwhile, it removes the patches of CVE-2008-1372 and CVE-2005-1260 in `bzip2`, and CVE-2012-5667 in `grep`, rendering the debloated binaries vulnerable to these already-fixed bugs.

Compared to CHISEL, RAZOR removes the bloated code in a conservative way. Although such strategy may hinder removing more bugs, but it also helps avoid new bugs in the debloated binary. This result is consistent with our findings in §5.2, where CHISEL achieves higher code reduction but fails some expected functionalities.

Reduction of ROP gadgets. We also measured the reduction of ROP gadgets. Once the attacker is able to divert the control-flow, the number of reusable ROP gadgets indicate the vulnerability of the program to control-flow hijacking attacks. Figure 7c show that RAZOR reduces 61.9% ROP gadgets, while CHISEL reduces 85.1% ROP gadgets. Although RAZOR achieves less ROP gadget reduction, this result is expected. In the design of RAZOR, we intentionally pay more attention on preventing forward-edge control-flow attacks, where attackers corrupt function pointers, instead of return addresses, to diver the control-flow. As shadow stack technique are getting deployed in compilers [24] and even hardware [19], our technique of indirect call/jump concretization (§3.3.2) complements existing practical return-protections to achieve complete control-flow integrity.

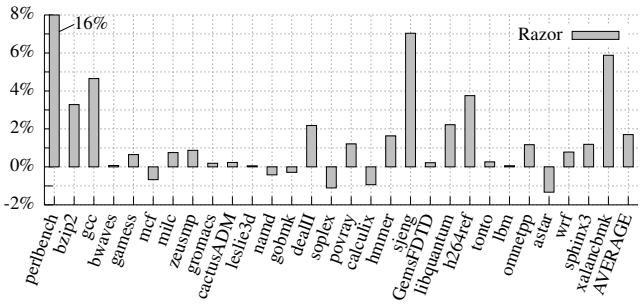


Figure 13: Performance overhead by RAZOR on SPEC CPU®2006 benchmarks. The average overhead is 1.7%.

5.5 Performance Overhead

Efficient Debloating. On average, RAZOR takes 1.78 seconds to debloat CHISEL benchmarks, 8.51 seconds for debloating `Firefox`, and 50.42 seconds to debloat `FoxitReader` As a comparison, CHISEL has to spend one to 11 hours to debloat the relatively small CHISEL benchmarks. Therefore, RAZOR is a practical debloating tool.

Runtime Overhead. We measured the performance overhead introduced by RAZOR to SPEC benchmarks and show the result in Figure 13. On average, RAZOR introduces 1.70% overhead to debloated programs, indicating its efficiency for real-world deployment. The highest overhead occurs on the debloated `perlbench` binary, which slows the execution by 16%. We inspected the debloated programs and confirmed that the indirect call concretization is the main source of the performance overhead. With the indirect call concretization, one indirect call instruction is replaced by several comparison and direct calls. For `perlbench`, some indirect call instructions have more than 100 targets. Correspondingly, RAZOR introduces a large number of `if-else` there, leading to a high performance overhead. We deployed the frequency-based optimization and reduced the overhead from over 100% to the current 16%. We plan to use binary search to replace current one-by-one matching in order to further reduce the overhead.

5.6 Debloating Real-world Programs

To evaluate the practicality, we used RAZOR to debloat two widely used software programs – the web browser `Firefox` and the closed-sourced PDF reader `FoxitReader`. For `Firefox`, we ran RAZOR to load the top 50 Alexa websites [28]. We randomly picked 25 websites as the training inputs and used the other 25 websites as the testing inputs. For `FoxitReader`, we ran RAZOR to open and scroll 55 different PDF files that contain tables, figures, and JavaScript code. We randomly picked 15 of them as the training inputs and used the other 40 files as the testing inputs.

Code reduction and functionality. Table 4 shows the code reduction rate and the number of failed cases of debloated binaries with different path-finding heuristics. Both `Firefox`

Heuristic	FireFox		FoxitReader	
	crash-sites	reduction	crash-PDFs	reduction
none	13	67.6%	39	89.8%
zCode	13	68.0%	10	89.9%
zCall	2	63.1%	5	89.4%
zLib	0	60.1%	0	87.0%
zFunc	0	60.0%	0	87.0%

Table 4: Debloating `Firefox` and `FoxitReader` with RAZOR, together with diffferent path-finding heuristics.

and `FoxitReader` require at least the zLib heuristic to obtain crash-free binaries, with 60.1% and 87.0% code reduction, respectively. Without heuristics, `Firefox` fails on 13 out of 25 websites and `FoxitReader` fails on 39 out of 40 PDF files. The zCode heuristic helps reduce `FoxitReader` crashes to 10 PDF files and increases the code reduction by avoiding fault-handling instrumentation. The zLib and the zFunc heuristic eliminate all crashes. Compared with the non-heuristic debloating, the zLib heuristic only decreases the code reduction rate by 7.5% for `Firefox` and by 2.8% for `FoxitReader`. Therefore, it is worth using this heuristic to generate robust binaries.

Performance overhead. We ran the debloated `Firefox` (with zLib) on several benchmarks and found that RAZOR introduces −2.1%, 1.6%, 0%, and 2.1% overhead to Octane [33], SunSpider [34], Dromaeo-JS [30], and Dromaeo-DOM [29] benchmarks. For `FoxitReader`, we did not find any standard benchmark to test the performance. Instead, we used the debloated binaries to open and scroll the testing PDF files and did not find any noticeable slowdown.

Application – per-site browser isolation. As one application of browser debloating, we can create minimal versions that support particular websites, effectively achieving per-site isolation [38, 21, 48] . For example, the bank can provide its clients a minimal browser that only supports functionalities required by its website while exposing the least attack surface. To measure the benefit of the per-site browser, we applied RAZOR on three sets of popular and security-sensitive websites: banking websites, websites for electronic commerce, and social media websites. Table 6 shows the debloating result, the used path-finding heuristic and the security benefits over the general debloating in Table 4. As we can see, the banking websites can benefit with at least 5.0% code reduction for the per-site minimal browser. The E-commerce websites will have around 3.0% extra code reduction, a little less because of its high requirement on user interactions. Surprisingly, social media websites can benefit by up to 8.5% extra code reduction and at least 4.2% when supporting all three websites. We believe the minimal web browser through binary debloating is a practical solution for improving web security.

Train/Test	ID	#Failed	Reduction	failed websites
20/30	T10	1	59.3%	wordpress.com
	T11	0	59.3%	
	T12	1	59.3%	wordpress.com
	T13	1	59.3%	twitch.tv
	T14	1	59.3%	wordpress.com
	T15	1	59.5%	wordpress.com
	T16	2	59.5%	twitch.tv, wordpress.com
	T17	1	59.3%	twitch.tv
	T18	1	59.3%	twitch.tv
	T19	2	59.6%	wordpress.com, twitch.tv
25/25	T00	0	59.3%	
	T01	2	59.1%	wordpress.com, twitch.tv
	T02	2	59.3%	wordpress.com, twitch.tv
	T03	2	59.1%	wordpress.com, twitch.tv
	T04	0	59.2%	
	T05	1	59.1%	aliexpress.com
	T06	0	59.2%	
	T07	0	59.1%	
	T08	2	59.3%	wordpress.com, twitch.tv
	T09	0	59.1%	

Table 5: N-fold validation of zLib heuristic on `Firefox`. First, we randomly split Alexa's Top 50 websites into five groups, and select two groups (20 websites) as the training set and others (30 websites) as the test set for 10 times. Second, we randomly split the 50 website into 10 groups, and select five groups (25 websites) as the training set, and others (25 websites) as the test set for 10 times.

5.6.1 N-fold Cross Validation of Heuristics

To further evaluate the effectiveness of our heuristics, we conducted N-fold cross validation on `Firefox` with the zLib heuristic, as it is the least aggressive heuristic that renders `Firefox` crash-free.We performed two sets of evaluations and show the result in Table 5. First, we randomly split Alexa's Top 50 websites into five groups, 10 websites per group. We picked two groups (20 websites) for training and used the remaining 30 websites for testing. We performed this evaluation 10 times. The result in the table shows that during one test with ID `T11`, the debloated `Firefox` successfully loads and renders 30 testing websites. The debloated `Firefox` fails two websites (6.7%) seven times and fails one website (3.3%) two times. Second, we randomly split Alexa's Top 50 websites into 10 groups, five websites per group. We randomly picked five groups (25 websites) for training and used the others (25 websites) for testing. We performed this evaluation 10 times. The result shows that, in five times, the debloated `Firefox` loads and successfully renders the tested 25 websites. The debloated `Firefox` fails one (4%) website one time and fails two websites (8%) four times. The code size reduction is consistently round 60%. These results show that our heuristics are effective for inferring non-executed code with similar functionalities of training inputs. Among all the tests, only three websites trigger additional code and the program gracefully exits with warning information. We plan to check these websites to understand the failure reasons.

We also manually checked what code of `Firefox`

Type	Site	Reduction	Heuristic	Benefits
Banking	bankofamerica.com	69.4%	zCall	+6.3%
	chase.com	69.6%	zCall	+6.5%
	wellsfargo.com	68.8%	zCall	+5.7%
	all-3	68.1%	zCall	+5.0%
E-commerce	amazon.com	71.4%	none	+3.8%
	ebay.com	70.7%	none	+3.1%
	ikea.com	70.6%	none	+3.0%
	all-3	70.4%	none	+2.8%
Social Media	facebook.com	70.8%	zCall	+7.7%
	instagram.com	71.6%	zCall	+8.5%
	twitter.com	74.0%	none	+6.4%
	all-3	71.8%	none	+4.2%

Table 6: Per-site browser debloating

was removed. We find that code related to features such as record/replay, integer/string conversion, compression/decompression are removed.

6 Discussions

Best-effort path inference. Mapping high-level functionalities to low-level code is known to be challenging, especially when source code is unavailable. RAZOR empirically adopts control-flow-based heuristics to infer more related-code with its best effort. We understand that such a heuristic cannot guarantee the completeness or soundness of the path inference, and the debloated binary may miss necessary code (*i.e.*, code for handling different environment variables) or include unnecessary ones (like some initialization code). However, we noticed that the heuristic-based method has been widely used in binary analysis and rewriting [53, 52]. With the execution trace, RAZOR is able to mitigate some limitations of these works, such as finding indirect call targets. Further, the evaluation result demonstrates that our control-flow-based heuristics are practically effective.

CFI and debloating. Control-flow integrity (CFI) enforces that each indirect control-flow transfer (*i.e.*, indirect call/jump and return) goes to legitimate targets [1]. It prevents malicious behaviors that are unexpected by program developers. In contrast, software debloating removes benign-but-unnecessary code based on users' requirements. For example, if function `A` is designed to be a legitimate target of an indirect call `i`, CFI will allow the transfer from `i` to `A`. However, if the user does not need the functionality in `A`, software debloating will disable the transfer and completely remove the function code. In fact, CFI and debloating are complementary to each other. On the one hand, debloating achieves a coarse-grained CFI where an attacker can only divert the control-flow to remaining code. It also simplifies the analysis required by some CFI works [50, 37] because of a smaller code base. On the other hand, existing CFI works provide fundamental platforms for enforcing debloating. For example, RAZOR makes use of several binary analysis techniques developed in binCFI [53] for optimization.

Library debloating. We tried to use RAZOR to debloat system libraries for each program. Our tool works well on some libraries (*e.g.*, `libm.so` and `libgcc.so`), but fails on others. For example, the debloated `libc.so` triggers a different execution path even if we aggressively include more related-code with the zFunc heuristic. After inspecting the failure cases on `libc.so`, we found that its execution path is very sensitive to the change of the execution environment. One reason is that `libc.so` contains a lot of highly optimized code for memory or string operations (*e.g.*, `memcmp`), which, based on the argument value, choose the most efficient implementation. For example, function `strncmp` implements 16 different subroutines to process strings with different alignments. Another reason is that it performs different executions according to the process status. For example, for each memory allocation, `malloc` searches a set of cached chunks and picks up the first available one. Inputs with different sizes may cause `malloc` to walk through a complete non-executed path. From such a preliminary result, we plan to develop library-specific heuristics to handle environment-sensitive executions. For example, we can perform debloating on the function level instead of the current basic block level. We also plan to explore existing library debloating solutions that work on source code [40] and port them into binaries if necessary.

Removing original code. The current design of RAZOR keeps the original code section inside the debloated program and changes its permission to read-only to reduce the attack surface. This design simplifies the handling of potential data inside the code section, which the program may read for special purposes. For example, LLVM will emit jump tables in the code section to support efficient `switch` statements [11], and the indirect jump instruction will obtain its targets by reading the table. To further reduce the program size and memory usage, we can completely remove the original code section as follows: 1) during the execution tracing, we set the original code section to execute-only [11] so that any read from the code section will trigger the exception and can be logged by `Tracer`; 2) we perform backward data-flow analysis to identify the source of the data pointer used for each logged memory access; 3) during the binary synthesization, we relocate the data from the original code section to a new data section and update the new code to visit the new location. In this way, we are able to handle the challenging problem of data relocation during binary rewriting. In fact, we performed a study to understand the prevalence of these problems and found that for all the programs tested in the paper, none of them ever reads any data from the code section, given the test cases we used. In these cases, we can simply remove the original code section to minimize the file size and memory footprint.

Future work. We will release the source code of RAZOR. We plan to extend the platform to support binaries in more

formats and architectures, including shared libraries, 32-bit binaries, Windows PE programs, MacOS March-O programs, and ARM binaries. At the same time, we will design more security-related heuristics to make RAZOR support various real-world situations.

7 Related Work

Library debloating. Program libraries are designed to support a large number of functionalities for different users. Library debloating customizes the general code base for each program and leads to significant code reduction. Mulliner *et al*. propose `CodeFreeze` to remove the unnecessary functionalities from Windows shared libraries [36]. They start from per-library control-flow analysis to identify the code dependency of each exported function. Then they check the program binary to find all required library functions. By stitching program required functions and per-library CFG, they rewrite the library to remove unreachable code region. Similarly, Quach *et al*. [40] present library debloating through piece-wise compilation and loading. Instead of customizing the library for each program, they split the large library into small groups based on the control-flow dependency. At runtime, they use a customized loader to rewrite the library code to remove unnecessary functions. Jiang *et al*. [23, 22] propose to remove dead code from Android Apps, Java Runtime Environment, and SDKs. Our system is different from library debloating in two ways. First, previous work performs the binary rewriting at the beginning of each process, leading to performance overhead for each execution, while RAZOR generates the debloated binary through static binary rewriting, which is only performed once and used forever. Second, library debloating utilizes static analysis to find the unused code and has to conservatively keep all potentially useful code. In contrast, our system relies on a dynamic execution trace to locate the code that is executed during tracing or inferred with our heuristic and removes all others.

Delta debugging. Delta debugging is proposed to minimize bug-triggering inputs. For example, Regehr *et al*. [42] propose `C-Reduce` to generate a smaller test cases efficiently. Sun *et al*. [49] present `Perses`, which exploits formal syntax to generate smaller and functionally equivalent program in a timely manner. Recently, Heo *et al*. [15] proposed CHISEL to use reinforcement learning for further speeding up the delta debugging process. However, the programs generated by delta debugging only support given test cases, while real-world software usually has an infinite number of test cases for certain functionalities. Instead, RAZOR takes control-flow-based heuristics to infer more related-code that is necessary to complete the required functionalities.

Source code debloating. Several recent works use program analysis to debloat programs. Bu *et al*. [8] propose a bloat-ware design paradigm that analyzes Java source code to optimize object allocations to avoid memory usage bloating at runtime. Sharif *et al*. [44] propose Trimmer, which propagates a user-provided configuration to program code and utilizes the compiler optimization to reduce code size. These systems, as well as [42, 49, 15], rely on the complicated analysis of program source code, which is not always available for deployed programs. In contrast, RAZOR only requires program binaries, making it more practical for deployment.

Container Debloating. Containers are becoming more popular, and their code base is bloated. Guo *et al*. [14] proposed a method to monitor the program execution to identify necessary resources and create a minimal container for the traced program. Rastogi *et al*. [41] developed Cimplifier, which uses dynamic analysis to collect resource usages for different programs and partitions the original container into a set of smaller ones based on user-defined policies. The resulting containers only have resources to run one or more executable programs. The design of RAZOR is also applicable for debloating containers or other systems. For example, `Intel PT` supports tracing operating systems.

Hardware Debloating. Nowadays, hardware devices are also bloated. For example, general-purpose processors are overly designed for specific applications, such as implantables, wearables, and IoT devices. Cherupalli *et al*. propose an approach that automatically removes unused gates from the design of a general-purpose processor to generate a bespoke processor for a specific application [10]. On average, the approach can reduce the area by 62% and the power by 50% from the general processor. Currently, software debloating and hardware debloating are performed separately. An interesting direction is to consider both hardware devices and software programs to find more debloating space.

8 Conclusion

In this paper, we presented RAZOR, a framework for practical software debloating on program binaries. It utilizes a set of test cases and control-flow-based heuristics to collect necessary code to support user-expected functionalities. The debloated binary has a reduced attack surface, improved security guarantee, robust functionality, and efficient execution. Our evaluation shows that RAZOR is a practical framework for debloating real-world programs.

Acknowledgment

We thank the anonymous reviewers, and our shepherd, Michael Bailey, for their helpful feedback. This research was supported in part by the DARPA Transparent Computing program under contract DARPA-15-15-TC-FP006, by the ONR under grants N00014-17-1-2895, N00014-15-1-2162 and N00014-18-1-2662. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA and ONR.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, 2005.

[2] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *Proceedings of the 25th USENIX Security Symposium (USENIX)*, 2016.

[3] Dennis Andriesse, Asia Slowinska, and Herbert Bos. Compiler-Agnostic Function Detection in Binaries. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy*, 2017.

[4] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, 2014.

[5] Erick Bauman, Zhiqiang Lin, and Kevin Hamlen. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium*, 2018.

[6] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the 2005 USENIX Annual Technical Conference*, 2005.

[7] Derek Bruening and Saman Amarasinghe. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2004.

[8] Yingyi Bu, Vinayak Borkar, Guoqing Xu, and Michael J. Carey. A Bloat-aware Design for Big Data Applications. In *Proceedings of the 2013 International Symposium on Memory Management*, 2013.

[9] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-Flow Integrity: Precision, Security, and Performance. *ACM Comput. Surv.*, 2017.

[10] Hari Cherupalli, Henry Duwe, Weidong Ye, Rakesh Kumar, and John Sartori. Bespoke Processors for Applications with Ultra-low Area and Power Constraints. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.

[11] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, 2015.

[12] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of Control: Overcoming Control-Flow Integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, 2014.

[13] Google. V8 JavaScript Engine. https://chromium.googlesource.com/v8/v8.git.

[14] Philip J. Guo and Dawson Engler. CDE: Using System Call Interposition to Automatically Create Portable Software Packages. In *Proceedings of the 2011 USENIX Annual Technical Conference*, 2011.

[15] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[16] Gerard J. Holzmann. Code Inflation. *IEEE Software*, 32(2), Mar 2015.

[17] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing Unique Code Target Property for Control-Flow Integrity. In *Proceedings of the 25th ACM Conference on Computer and Communications Security*, 2018.

[18] ImageTragick. ImageMagick Is On Fire: CVE-2016-3714. https://imagetragick.com/.

[19] Intel. Control-Flow Enforcement Technology Preview. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.

[20] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, volume 3 (3A, 3B, 3C & 3D): System Programming Guide. November 2018.

[21] Yaoqi Jia, Zheng Leong Chua, Hong Hu, Shuo Chen, Prateek Saxena, and Zhenkai Liang. The Web/Local Boundary Is Fuzzy: A Security Study of Chrome's Process-based Sandboxing. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[22] Y. Jiang, D. Wu, and P. Liu. JRed: Program Customization and Bloatware Mitigation Based on Static Analysis. In *2016 IEEE 40th Annual Computer Software and Applications Conference*, 2016.

[23] Yufei Jiang, Qinkun Bao, Shuai Wang, Xiao Liu, and Dinghao Wu. RedDroid: Android Application Redundancy Customization Based on Static Analysis. In *Proceedings of the 29th IEEE International Symposium on Software Reliability Engineering*, 2018.

[24] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, 2014.

[25] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[26] Haoyu Ma, Kangjie Lu, Xinjie Ma, Haining Zhang, Chunfu Jia, and Debin Gao. Software Watermarking Using Return-Oriented Programming. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, 2015.

[27] John Martellaro. Why Your iPhone Uses PNG for Screen Shots and JPG for Photos. https://www.macobserver.com/tmo/article/why-your-iphone-uses-png-for-screen-shots-and-jpg-for-photos.

[28] The Top 500 Sites on the Web. https://www.alexa.com/topsites.

[29] Dromaeo-DOM. http://dromaeo.com/?dom.

[30] Dromaeo-JS. http://dromaeo.com/?dromaeo.

[31] The Heartbleed Bug. http://heartbleed.com/.

[32] Function and Macro Index. https://www.gnu.org/software/libc/manual/html_node/Function-Index.html.

[33] Octane. https://chromium.github.io/octane.

[34] SunSpider. https://webkit.org/perf/sunspider-1.0.2/sunspider-1.0.2/driver.html.

[35] CVE-2014-0038: Privilege Escalation in X32 ABI. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0038, 2014.

[36] Collin Mulliner and Matthias Neugschwandtner. Breaking Payloads with Runtime Code Stripping and Image Freezing. In *Black Hat USA Briefings (Black Hat USA)*, Las Vegas, NV, August 2015.

[37] Ben Niu and Gang Tan. Per-Input Control-Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[38] The Chromium Projects. Site Isolation. https://www.chromium.org/Home/chromium-security/site-isolation.

[39] Anh Quach, Rukayat Erinfolami, David Demicco, and Aravind Prakash. A Multi-OS Cross-Layer Study of Bloating in User Programs, Kernel and Managed Execution Environments. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*, 2017.

[40] Anh Quach, Aravind Prakash, and Lok Yan. Debloating Software through Piece-Wise Compilation and Loading. In *Proceedings of the 27th USENIX Security Symposium*, 2018.

[41] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. Cimplifier: Automatically Debloating Containers. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[42] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case Reduction for C Compiler Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

[43] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, 2015.

[44] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. TRIMMER: Application Specialization for Code Debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[45] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing Functions in Binaries with Neural Networks. In *Proceedings of the 24th USENIX Conference on Security Symposium*, 2015.

[46] Igor Skochinsky. Compiler Internals: Exceptions and RTTI. http://www.hexblog.com/wp-content/uploads/2012/06/Recon-2012-Skochinsky-Compiler-Internals.pdf, 2012.

[47] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, 2013.

[48] Peter Snyder, Cynthia Taylor, and Chris Kanich. Most Websites Don't Need to Vibrate: A Cost-Benefit Approach to Improving Browser Security. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[49] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. Perses: Syntax-guided Program Reduction. In *Proceedings of the 40th International Conference on Software Engineering*, 2018.

[50] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-edge Control-Flow Integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[51] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making Reassembly Great Again. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium*, 2017.

[52] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable Disassembling. In *Proceedings of the 24th USENIX Conference on Security Symposium*, 2015.

[53] Mingwei Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.

Appendix

A Settings for Evaluating PathFinder

Program	Training Set Size	Testing Set Size	Options
bzip2	10	30	`-c`
chown	6	17	`-h, -R`
date	22	33	`-date, -d, -rfc-3339, -utc`
grep	19	38	`-a, -n, -o, -v, -i, -w, -x`
gzip	10	30	`-c`
mkdir	12	24	`-m, -p`
rm	10	20	`-f, -r`
sort	12	28	`-r, -s, -u, -z`
tar	10	30	`-c, -f`
uniq	24	40	`-c, -d, -f, -i, -s, -u, -w`

Table 7: Settings for evaluating `PathFinder` on the CHISEL benchmarks. We use the training set to debloat the binary, and run the generated code with the testing set. The last column is the options we pass to the binaries during training and testing.

Back to the Whiteboard: a Principled Approach for the Assessment and Design of Memory Forensic Techniques

Fabio Pagani
EURECOM

Davide Balzarotti
EURECOM

Abstract

Today memory analysis plays a fundamental role in computer forensics and is a very active area of research. However, the field is still largely driven by custom rules and heuristics handpicked by human experts. These rules describe how to overcome the semantic gap to associate high level structures to individual bytes contained in a physical memory dump. Structures are then traversed by following pointers to other objects, and the process is repeated until the required information is located and extracted from the memory image.

A fundamental problem with this approach is that we have no way to *measure* these heuristics to know precisely how well they work, under which circumstances, how prone they are to evasions or to errors, and how stable they are over different versions of the OS kernel. In addition, without a method to measure the quality and effectiveness of a given heuristic, it is impossible to compare one approach against the others. If a tool adopts a certain heuristic to list the sockets associated to a program, how do we know if that is the only possible way to extract this information? Maybe other, even better, solutions exist, just waiting to be "discovered" by human analysts.

For this reason, we believe we need to go back to the drawing board and rethink memory forensics from its foundations. In this paper we propose a framework and a set of metrics we can use as a basis to assess existing methodologies, understand their characteristics and limitations, and propose new techniques in a principled way. The memory of a modern operating system is a very large and very complex network of interconnected objects. Because of this, we argue that automated algorithms, rather than human intuition, should play a fundamental role in evaluating and designing future memory forensics techniques.

1 Introduction

Computer forensics is often considered an art, as the analyst proceeds by formulating hypothesis about the cause of an incident and uses inductive reasoning to reinforce or discard them based on clues and artifacts collected from the target system. The way these artifacts are extracted and analyzed is largely based on the experience and on the set of heuristics encoded into the available tools. Memory forensics, i.e. the field focusing on the analysis of snapshots of the physical memory of a machine, is no exception to this rule. Memory forensic tools needs to recover the high-level semantic associated with sequences of raw bytes – thus reconstructing the internal state of the operating system (OS) and its applications at the time the memory was acquired. Unfortunately, modern OSs are very complex software whose memory often contains millions or tens of millions of individual objects at any moment in time. Even worse, both the fields and the layout of these objects can change when the kernel is updated or recompiled, and the connections among them evolves very rapidly – with a considerable amount of links and pointers that change every few milliseconds.

Currently memory forensics techniques rely on a large number of rules and heuristics that describe how to navigate through this giant graph of kernel data structures to locate and extract information relevant to an investigation. For example an analyst can use rules — commonly known as *plugins* in the field terminology — to retrieve the list of processes running at acquisition time (including their name, starting time, process ID, and other related information) or the list of open sockets. The result of the analysis depends on the number and accuracy of these rules. However, the field today is still in its infancy and each individual technique is manually written by researchers and practitioners. As a result, it is often unclear why a particular exploration strategy has been chosen, except for the fact that some developers found it reasonable based on their experience. Even worse, how accurate a given heuristic is and how we can compare it with other candidates to decide which strategy is more suitable for a given investigation remains an open question.

In fact, we still do not even know how to properly characterize the accuracy of a technique, as its quality depends on the metric we use to evaluate it, which in turn depends on the goal of the analyst. For instance, in an adversarial environment in which the analyst is investigating a sophisticated attack, a good heuristic would be one that is difficult to evade for an attacker. In a different investigation in which there is no risk of tampered

kernel data, a heuristic that only traverses closely-related (in physical memory) data structures may be preferable as pages acquired far apart may otherwise contain inconsistent information if the dump was not acquired atomically.

Contribution: The goal of this paper is to introduce a more principled way to approach the problem of memory analysis and forensics. Our plan is articulated around three main points. The first intuition is that heuristics used to extract information from memory dumps should be automatically generated by computers and not handpicked by humans. As we will show in our experiments, the graph of kernel objects is tightly connected and there are tens of millions of different ways to reach a given structure by starting from a global symbol. The second point is the fact that it is very important to be able to quantitatively measure the properties of each heuristic, so that different options can be compared against one another and an analyst can decide which technique is more appropriate for her investigation. Finally, the analyst should be able to obtain some form of guarantee about the results, to ensure that once a given quality metric has been chosen, a certain technique is the *optimal* solution to navigate the intricacies of runtime OS data structures.

As a step towards these goals, we constructed a complete graph of the internal data structures used at runtime by the Linux kernel. In our graph, nodes represent kernel objects and edges a pointer from one object to another. We chose Linux as the availability of its source code simplifies the creation of our model. However, a similar graph was also extracted in the past by Microsoft for the Windows kernel [4] and could therefore be reused for our purpose. The resulting map of the memory is a giant network (containing over a million nodes) with a very dynamic topology that is constantly reshaped as new data structures get allocated and deallocated.

Memory forensics tools adopts rules to navigate through the data structures present in a memory dump, and these rules can therefore be represented as paths in our kernel graph. Nodes and edges can then be decorated with additional pieces of information that capture different properties an analyst can find important in an analysis routine. In our study we model this phase by introducing and discussing five different metrics: *Atomicity*, *Stability*, *Generality*, *Reliability*, and *Consistency*. We used these metrics to compute a score associated to each path, and therefore to existing memory analysis techniques, as well as to compute the optimal solution according to a chosen set of criteria. We then discuss the intricacies of identifying such optimal paths by performing experiments with 85 different kernel versions and 25 individual memory snapshots acquired at regular time intervals.

Building a map of the kernel memory is a very tedious and time-consuming process. However, we believe this map can have many interesting uses in computer security beyond memory forensics – including virtual machine introspection (VMI), kernel hardening, and rootkit detection. Since both the code and the results of the previous attempts to build this graph [4,17,19] are not publicly available, we decided to release all our data – hoping it will help other researchers to considerably reduce the time required to investigate and validate techniques that require information about the content of a running kernel.

2 Motivation

Being quite in its infancy, memory forensics has still many open problems, which have been recently summarized by Case and Richard [6]. The authors divided them in two categories, depending on whether they are related to the *acquisition* or the *analysis* of a memory dump. More precisely, the first category contains all the practical issues of acquiring memory from a device under investigation while the second one deals with the capabilities of memory forensics, such as malware detection and evidence extraction.

One of the main issues belonging the first category is *page smearing*, which is a consequence of the fact that while the acquisition is performed the underlying system is not frozen and thus the dump may contain inconsistent information [12]. While the term was coined in 2004 [5], its actual implications are still unclear to the community. For instance, a recent study from Le Berre [18] pointed out that in real investigations more than the 10% of memory dumps suffer from this problem and thus can not be properly analyzed with existing tools. Our work can help mitigating this issue by assessing how existing techniques are affected by non-atomic acquisitions, and help design new heuristics which are more robust against the presence of inconsistent information.

The second category focuses instead on challenges related to memory analysis. For example, as of today, forensics practitioners lack the necessary tooling for extracting a number of interesting information such as Powershell activity and evidences related to Office applications and private browsing sessions, or to analyze sophisticated userland malware. Finally, a vast range of technologies did not receive any forensics coverage: Apple iOS, Chromebooks, and IoT devices are still out of scope when it comes to memory forensics analysis.

While these issues are very different from one another, most of them share the same underlying assumption: kernel objects must be located, traversed and interpreted by a set of rules. Our approach enables forensics practitioners and researchers to evaluate, under different constraints, the quality of these rules and provide them with a framework to compare and discover new sets of rules.

3 Approach

In this section we describe the four-step approach we propose to precisely measure and improve the quality of existing memory forensics techniques. The first step consists of building a precise representation of all data structures that exists in a running kernel and of the way these structures are connected

① `[init_task].tasks.prev` ↻ → **`task_struct`**`.mm` → `mm_struct.mmap` → `vm_area_struct.vm_next` ↻ → `vm_area_struct`

② `[root_cpuacct].css.cgroup` → `cgroup_root.cgrp.e_csets[2].next` → `css_set.tasks.next` → **`task_struct`**`.mm` → `mm_struct.mmap` → `vm_area_struct.vm_next` ↻ → `vm_area_struct`

Figure 1: Two different paths that reach the same `vm_area_struct` object.

to one another. The challenges and the process we followed to build this kernel graph are described in details in Section 4. In the second phase we map existing forensic analysis techniques into our model, by representing their algorithms as paths through the kernel graph. We then color the graph according to different properties that are relevant for a forensic investigation, and we employ graph-based algorithms to assess the characteristics of the previously-identified paths and find new ones that may exhibit better properties. Finally, in the fourth and final phase of our methodology we translate our findings back to the memory forensic space by generating improved analysis plugins, thus increasing the number and quality of the rules that are used today to analyze memory dumps.

3.1 Memory Forensics as a Graph Exploration Problem

The goal of memory forensics is to bridge the semantic gap between the raw bytes that constitute a physical memory's snapshot and the high-level abstractions provided by modern operating systems. This task requires the forensic tool to be able to correctly translate virtual to physical memory addresses, as well as to identify the data structures that contain the required information (e.g., the name of the files opened by a given process). The latter is typically achieved in two phases. First, the system locates a known object – either because it resides at a fixed or predictable location, by using symbols information generated by the compiler when the kernel was built, or by carving a particular data structure based on a set of known properties and invariants. Starting from this entry point, the analysis then traverses different memory regions, moving from one data structure to the next by following pointers, until it reaches the required piece of information. For example, we assume the analyst found a suspicious process and she wants to extract its executable code for further analysis. On Linux, this analysis starts from extracting the position of the global kernel variable `init_task` of type `task_struct`. This is one of the most important kernel object in terms of Linux memory forensics since every kernel thread and user space process has its own and it serves as a hub to reach several other relevant pieces of information. After locating `init_task`, the processes list is walked until the `task_struct` belonging to the suspicious process is found. From here, the `mm_struct` is reached by dereferencing the `mm` field. Finally, the list of `vm_area_struct`, each of which defines a virtual memory area, is retrieved — first by following the `mmap` pointer, then by using the `vm_next` field. With this information, the analyst can find the executable regions of the process and can proceed to save their content to disk.

This procedure can be naturally represented as a path on a graph in which every node is a kernel object, and every link a pointer. While the final node is dictated by a given forensic task, both the first and the intermediate nodes are often the result of handcrafted routines based on the experience and expert judgment of the developers of the forensic tool.

In our graph, the previously presented analysis would correspond to the path ① in Figure 1. The path contains the names of the structures and fields that need to be traversed (in square brackets when they refer to global symbols in the kernel) as well as the type of transition (→: follow a pointer reference, ↻: visit multiple structures of the same type linked together). For simplicity, we report inner structures in our paths as names in the edge and not explicitly as standalone nodes. Also, note that in the example ①, since the suspicious process was freshly spawned, the shortest path in our graph traverses the process list *backwards* — contrarily to the more common *forward* walking.

On top of the previous solution, our approach shows that a stunning 2.5 million different sequences of vertices exist in the kernel graph to reach the very same target object starting from a global variable, *only* counting the paths with no more than 10 edges. For example, path ② in Figure 1 begins from the little-known global symbol `root_cpuacct`, passes through a number of *cgroup*-related objects, before finding the `task_struct` of the suspicious process.

The previous two "rules" are both capable of locating a given process structure in a memory dump. The first is certainly more intuitive and it may also traverse a lower number of data structures. However, this is purely a *qualitative* assessment, and it is unclear if the first solution actually has any clear advantage or whether it provides any better guarantee then the second.

3.2 Path Comparison

As we saw in the previous example, if we want to assess the quality of a given solution, we first need to define what "quality" means in our context. In other words, when two paths exist to reach the same target data structure, we need to define a metric that can tell us which one is better to follow from a forensic perspective.

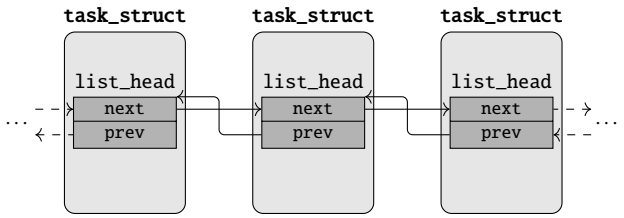


Figure 2: `task_struct`s organized in a doubly linked list.

A developer may favor the shortest path, as it is simpler to implement and may appear to be more robust according to the intuition that the fewer the data structures that need to be parsed, the less likely it is that something can go wrong while doing that. However, this approach raises another important issue about today's approach for memory analysis: its ad hoc nature and lack of a scientific foundation. In fact, it is not clear today how different exploration techniques can be compared and how they can be evaluated against one another in a precise and measurable way.

A first important observation is that there is not a single, absolute metric that defines the quality of a memory exploration rule. It all depends on the goal of the analyst, the conditions under which the memory snapshot was acquired, and the type of threat that is investigated. For example, in the common case in which a memory snapshot is acquired non-atomically, the analyst may prefer to adopt an approach that only traverses structure closely located in memory, thus minimizing the chances of inconsistencies. On the opposite case in which the memory was acquired atomically in a lab from a virtual machine used to investigate a possible rootkit, the analyst would certainly favor a different approach that traverses structures whose values cannot be tampered with by the attacker. In yet another scenario, an investigator may try to analyze a dump for which she was not able to retrieve a correct OS profile, and therefore she might be interested in paths that traverse structures that have changed very rarely across different kernels, to maximize her probability of success.

Therefore, it is the analyst who needs to select the more appropriate fitness function to compare paths according to any combination of desired properties. And once this function has been chosen, it is possible to use it to compute the optimal path (and therefore the optimal exploration strategy) to traverse the kernel graph. In this paper we explore different possible scenarios by proposing several metrics to enrich the graph (more details about this process are presented in Section 5) and then use this information to evaluate existing approaches and discuss other, non-conventional solutions that can provide better guarantees for the analyst.

4 Graph Creation

The first step of our methodology consists in building a model of the operating system kernel, that we can later use to compare different memory forensic approaches. The model we chose for our analysis is a graph of kernel objects, in which nodes represent kernel data structures and edges represent relationships between objects (for example a pointer from one structure to another).

The core idea is simple and relies on two crucial pieces of information extracted from the kernel debugging symbols. The first one is the layout, in terms of the exact type and offset of each field, of all the `struct` defined and used by the kernel code. The second information is instead related to the address, name, and type of global kernel variables that play the role of entry points for our graph exploration. Starting from these global pointers, our algorithm can recursively traverse other structures, each time following a pointer and casting the target memory to the appropriate type. While this process may seem straightforward at first, there are many special cases that make the construction of a kernel graph a complex procedure that requires multiple phases and several dedicated components.

In the rest of the section we discuss in more details some of these problems and the way we handled them in our study: abstract data types (and the issue with non-homogeneous circular lists), opaque pointers, and the presence of uninitialized or invalid data.

4.1 Abstract Data Types

Over the years, to maintain a reasonable quality over its code base, the Linux kernel developers have adopted several design patterns [20]. In particular, the kernel exports a rich set of APIs to manipulate and create complex data structures, such as double-linked lists and trees of various types, thus relieving kernel developers from the burden of reinventing the wheel every time they need to store and organize multiple objects. For this reason, the existing APIs are not tied to a specific type of kernel object but rely instead on predefined data types that can be included in more complex `struct` objects, and in a number of macros to manipulate them.

Figure 2 shows one of the most common example of this pattern, in which several `task_struct` are organized in a doubly linked list using the `list_head` type. While this provides a simple and efficient way to organize data structures, it unfortunately poses a serious challenge to the automated exploration of kernel objects. In fact, if the leftmost `task_struct` in the figure was already identified by other means (for example because it was pointed to from a global variable), simply following the `next` pointer would result in the discovery of the inner `list_head` structure, but *not* of the outer `task_struct`.

In fact, this operation is performed in the source code by using dedicated macros. In the case of the previous example, a developer would invoke:

```
container_of(var, struct task_struct, task)
```

that the compiler pre-processor translates to a snippet of code required to cast the target `list_head` variable `var` to the requested type based on the current offset inside it (as specified

by the field `task`). However, in our analysis we cannot simply mimic the same behavior by subtracting the offset of the list field from `next` pointer and to cast the result to the correct type to obtain a reference to the outer object. In fact, there are many cases in which this approach would lead to wrong results and it is not sufficient to look at the field type or at its value to distinguish these problematic cases. One example is the list rooted in the field `children` of a `task_struct`. While the field points to another `task_struct`, it does so by reaching it at a different offset (in the `sibling` field). Because of this and other similar problems (explained in more details later in the paper) it is not possible to systematically apply the "subtract and cast" strategy.

For each pointer in a data structure we need to know where — in terms of object type and offset in the target structure — it points to. Other works that built a map of the Linux kernel [1, 24, 35] solved the problem by manually annotating the source code. While this was doable for old kernel versions (e.g., 2.4), it would take many weeks of tedious work to annotate a recent kernel – which today uses more than 6000 different data structures and more than a thousand instances of `list_head`s. Moreover, manual annotations are error prone and are tailored to one specific code base, thus requiring to be verified and modified whenever a new kernel version is released. The compiler community has also already extensively studied the points-to problem [9, 14, 15, 22, 30, 34]. Unfortunately, the techniques they proposed are not suitable to our work as they tend to favor speed (an important factor at compile-time) over precision [4] (a more important factor for our analysis). Only four previous studies automatically extracted a type graph of a kernel [4, 17, 19, 29]. However, none of their systems is available: in one case because the authors relied on the internal source code of the Microsoft Windows operating system [4], and in the other because the entire work was lost [17].

For this reason, we decided to implement our own points-to analysis – which consists of a `clang` plugin that reasons on the Abstract Syntax Tree (AST) of each kernel compilation unit. Contrary to standard points-to analysis, our approach focuses only on the *type* information. More precisely, traditional solutions are designed to identify where each pointer points to, while in our case we only need to extract the target structure, and the offset inside that structure. The result is a *type graph* of the kernel under analysis. To extract this information we take advantage of the fact that the information we need can be inferred by analyzing the source code of the kernel that is in charge of manipulating the data structure in question. Our plugin explores the AST until it finds a call to a kernel API related to data structure management. At this point it analyzes the parameters and resolves their structure type and field name. An example of API call and respective AST is given in Figure 3. In the example, a call to the API `list_add` is used to append the `new` task at the beginning of the list rooted at `head->tasks`. This give us the information that the field `tasks` of `task_struct` indeed points to the very same type. Our plugin current supports `list_head`s, `hlist_head`s (used in the implementation of hash tables), and `rb_root` (used in the implementation of red-black trees).

Except for those, the most common type that is still not supported by our prototype is `radix_tree`, which however is only used 8 times in the entire kernel code base.

As we will show in Section 4.6, our approach is very effective and was able to resolve the type pointed by 250 global lists and by more than 1110 unique object fields in the Linux kernel 4.8, compiled with the Ubuntu 16.04 kernel configuration. Moreover, while our approach is tailored to the Linux kernel, it can be adapted to work on any other operating system, given the availability of its source code. Finally, since the parameter resolution routine does not perform complex analyses, our analysis does not introduce any significant overhead at compilation time.

Circular Lists of Non Homogeneous Elements

As we already discussed in the previous section, certain linked list can chain together object of different types. Since the code must have a way to determine to which type the target element belongs to, this pattern is only present in the form of a "root" object which is the first element of a circular list of otherwise homogeneous objects.

As a consequence, these lists can *only* be traversed starting from their root node, as traversing the loop from an intermediary objects can result into unexpectedly reaching the root node (of a different type) when dereferencing one of the `next` pointers. For example, other than the already cited **children** field of `task_struct`, also the `thread_node` field of the same structure points inside a `signal_struct` object.

To avoid this problem, our analysis classifies every `list_head` field in one of the following three categories: *root* pointer, *intermediate* pointer or *homogeneous* pointer. The first two are used to mark `list_head` fields that belong to lists that contain mixed types, while the latter describes the more common case of homogeneous list. For instance, `task_struct.children` is a root pointer, `task_struct.sibling` an intermediate and `task_struct.tasks` a homogeneous one. This classification can be automatically derived from the type graph: whenever two objects of different types are involved we label the first as *root* and the second as *intermediate*, while all the other objects are labeled as *homogeneous*. During the exploration phase, depending on the type of the pointer, we adopt a different strategy:

- *homogeneous* pointers can be explored by our algorithm in any order.
- *root* pointers require instead our algorithm to immediately walk and retrieve the objects of the entire circular list.
- *intermediate* pointers are ignored since we do not know if they point to another intermediate element or to a

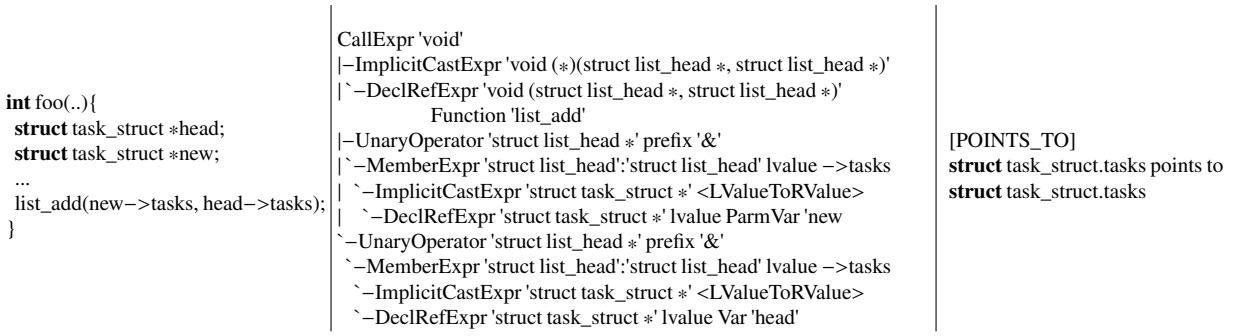


Figure 3: On the left a call to `list_add`, in the center its simplified AST representation, and on the right the plugin output.

root head. This case happens when we enter a circular list from one of its middle elements. This pointer will eventually be explored when the corresponding root node will be visited.

This classification works for every list encountered during the exploration phase, *except* for global `list_head` variables which are always marked as *root* node. In this case, during the very first part of the exploration, these lists are walked entirely and their elements appended to the worklist.

4.2 Uninitialized and Invalid Data

During our data structure exploration, there are cases that could potentially introduce false nodes to our graph. This is due to pointers that contain valid memory addresses but are not yet initialized or that were not valid at the time the snapshot was acquired. One common cause for these errors is the fact that most of the memory management kernel APIs do not initialize to zero the allocated memory. As a result, if an object contains an array of pointers there is not way to tell if one element points to an initialized object (except if the pointer has an invalid value). Another source of false-positives comes from the *non quiescent* state in which the kernel might be when the snapshot is taken [16]. In other words, this means that the kernel could have been in the middle of updating a data structure, leaving dangling pointers in the snapshot. Finally, even if very rare, kernel bugs can contribute to the generation of similar errors.

For these reasons we implemented two sets of heuristics to check if an object is valid or not. The first *soft* rule checks that the number of valid pointers in a kernel object is greater or equal than the number of invalid ones (after removing null pointers and the pointers which normally point to userspace memory, such as the ones contained in `struct sigaction`). The second, more precise, heuristic immediately flags an object as invalid if certain conditions are not verified (such as kernel objects that contain a negative spinlock, or those with function pointers that do not point in the executable sections of the kernel). Finally, we require that, whenever present, a `list_head` has to be valid, i.e. its `next` and `prev` pointer must point to addressable memory. If these rules are not met, we consider the object invalid and discard it from our analysis.

4.3 Opaque Pointers

Opaque pointers, as represented by `void*` fields or by `long long` integers that contain at runtime the address of other objects, are traditionally one of the hardest obstacle to build a complete map of kernel objects. Luckily, this is not the case in our particular scenario. Since we are interested in using our graph to analyze and improve existing memory forensic techniques, opaque pointers play a very marginal role (if any at all) in this space. As they can point to potentially any structure, and the actual target type can change over time, traversing these pointers can be unpredictable during a post-mortem analysis. Even if none of the heuristics we encountered in our experience make use of them, we decided to include them in our graph. After the exploration ends, in case the target of an opaque pointer was discovered by other means, we create the resulting edge, clearly marking it. In this way, we are able to detect if any of these edges are traversed during our experiments. Finally, it is important to understand that these limitations cannot lead to "wrong" results (since they cannot create erroneous paths in the graph), but nonetheless restrict the guarantees of optimality we discuss in the next sections to the constructed graph.

4.4 Limitations and Manual Fixes

Like all previous attempts to build a map of the kernel memory, two particular limitations also affect our solution: unions, and dynamically allocated arrays. Handling the latter case would require more sophisticated code analysis techniques to identify the variable number of elements contained in the arrays, which are beyond the scope of this paper. Nevertheless, we identified few cases of dynamically allocated arrays that contain information that can be relevant for memory forensics and we decided to handle them by hardcoding a custom logic. The first cases are global hash tables where often the size is not inferable from the hash table itself. For example, the `pid_hash` hash table, used by the kernel to quickly locate a process given its

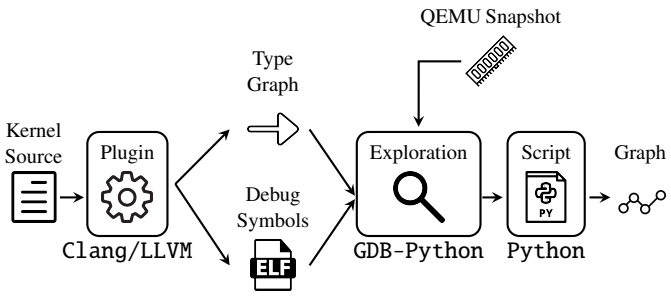


Figure 4: System Overview.

process id, is implemented by using a dynamically allocated array where the size is specified in another global variable (`pidhash_shift`). The second cases are instead dynamically allocated arrays pointed by a kernel object. For example, the file descriptor table associated with each process, which contains the files opened by a process (field `fd` of `struct fdtable`). Once again, this is a dynamically allocated array of `struct file` pointers, and the size can be retrieved from the field `max_fds` of the same structure.

Finally, the handling of `per_cpu` pointers was also hard-coded in our implementation. These are special pointers that, thanks to a double indirection mechanism when dereferenced, give to each processor a different copy of the same variable.

However, we want to stress that this limitation does not invalidate our findings since the graph extracted by our approach is *not* incorrect, but only potentially incomplete.

4.5 Implementation

Our final system is illustrated in Figure 4. It consists of an LLVM compiler plugin to perform the *points-to analysis* on the kernel code at compile-time and a set of python `gdb` extensions that combine the information extracted in the previous step with the information provided by kernel debug symbols to identify all kernel objects contained in a memory snapshot acquired using the QEMU emulator. The kernel exploration routine starts by loading a QEMU snapshot, parsing the type graph, and appending the global object symbols to an internal worklist. At this point the real exploration begins: an object is fetched from the worklist and analyzed using the heuristics we adopted to identify invalid or uninitialized memory. If it is well-formed, each of its field are processed to identify structures, pointers to other structures, or arrays of either type. All them are retrieved and appended to the worklist – paying attention to implement the techniques described above to handle abstract data types. These objects are then processed by a separate component responsible to build the final kernel graph that we will later use to carry out our experiments.

4.6 Final Kernel Graph

We built our kernel graph using *graph-tool* [23], a python library designed to handle large networks. To reduce the size of the graph, we chose to represent with one vertex each *outer* structure identified during the exploration. In other terms we decided to group together, in a single vertex, all the nested structures (but we keep the nesting information as it is needed when we need to move from the graph space back to the memory analysis heuristics). This transformation also makes the graph directed, and result in only one type of edges that represent pointers from a structure to another. As we will thoroughly discuss in Section 5 we assign a number of different weights to each node and edge to allow for several comparisons among different paths.

Figure 5 shows a kernel graph counting 109,000 nodes and 846,000 edges, plotted using Gephi [2]. This graph contains more than 41,000 strongly connected components with the vast majority (95%) containing only one node. On the other hand, the largest one contains 53% of the vertices and has a diameter of 272 nodes. As we will discuss in Section 6, this has important consequences for memory analysis, as it results in a multitude of available paths to move from one node to almost anything else in the kernel memory. The vertex with the highest in-degree is of type `super_block`, pointed by more than 11,000 `inode`s and 11,000 `dentry`s. If we exclude the file system, the node with the highest degree is a `vm_operations_struct`, pointed by more than 4200 `vm_area_struct`s.

In the picture, the size of labels and node is adjusted according to the betwenees centrality of a node. This type of centrality counts how many shortest path between every pair of nodes pass through a node. In other terms, the larger the size the more often a node is present inside every shortest path. The node color depends instead on the kernel subsystem the object belongs to. By using the name of the file where the object is defined we were able to classify them in roughly 7 classes, from file system to object related to memory or process management.

5 Metrics

In the previous section we described how we extracted a global map of a running kernel that can serve as basis for our analysis. However, without any further information, the only way we can compare two paths on the graph is by looking at their *length*, computed by counting either the total number of nodes or the total number of unique structures that need to be traversed. In fact, this simple approach may resemble the one adopted today by most of the memory forensic tools, where the most straightforward path is often chosen by the developers. However, this solution does not tell anything about the *quality* of a given path, nor about the presence of better options to solve the same problem. To get a solid foundation on which we can compare different techniques we need therefore to define a metric. And since the idea of having an absolute metric is

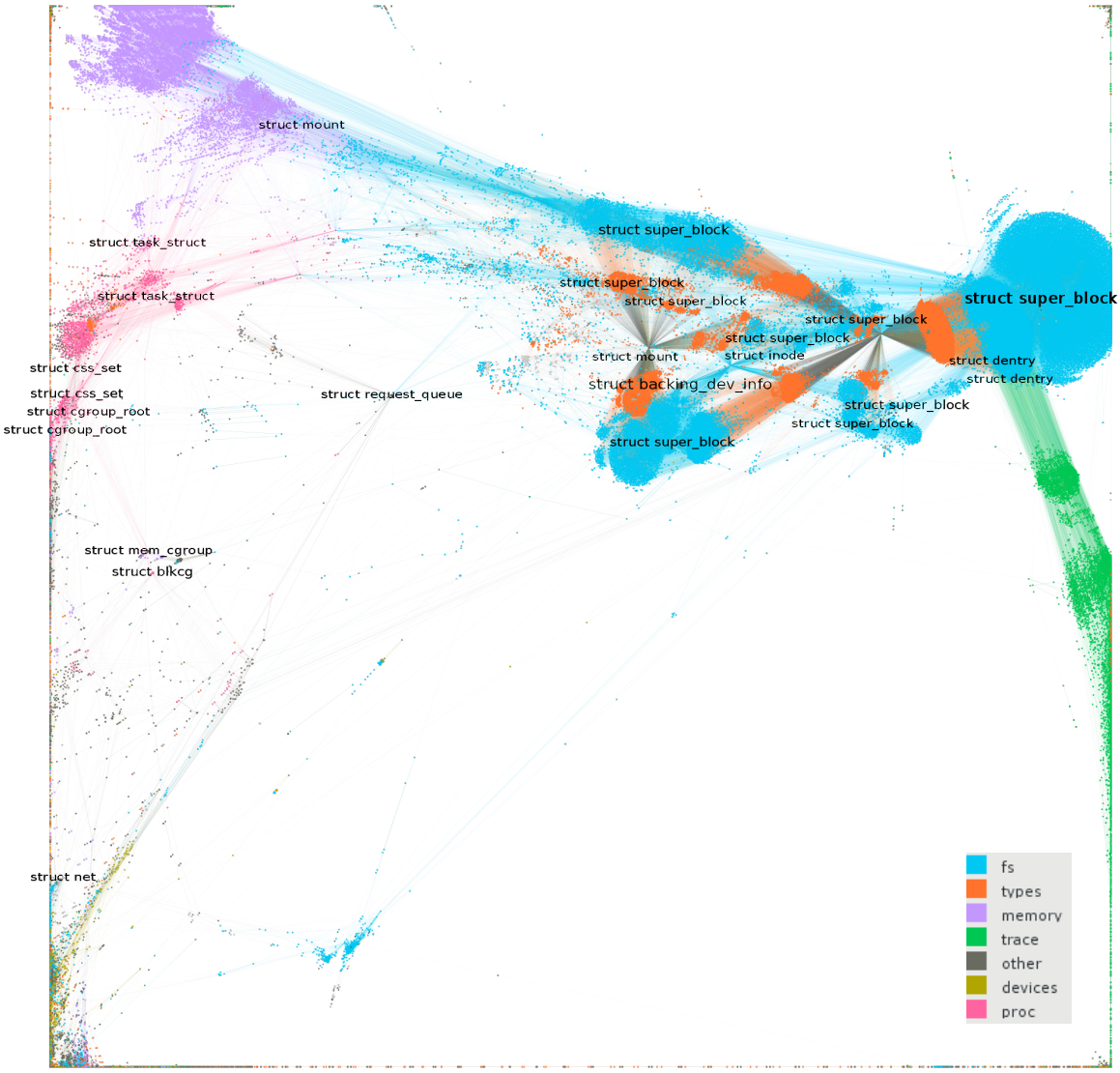


Figure 5: Kernel Graph

unrealistic, multiple different metrics can be plugged on our graph to study the characteristics of each path.

For our experiments we decided to investigate and add to our graph three numerical and two boolean weights, related to the atomicity, stability, generality, reliability, and consistency of a path. As described below, all of them capture different but important aspects of what an analyst may expect from a memory analysis routine.

Atomicity (numerical)

This weight express the distance in physical memory between two interconnected kernel objects. While this metric is expressed in terms of distance among physical pages, for an easier interpretation we often express it in seconds (as distance in time between the acquisition of the two pages). The atomicity is a very important aspect in most of today's investigation that rely on *non-atomic* dumps. In fact, moving across objects located far apart in memory - and thus acquired far apart on the time scale - can introduce inconsistencies. Intuitively, by using this metric the best path between a pair of nodes is the one which minimize the time-delta among all visited structures, thus passing only thorough objects acquired very close in time. More precisely, we can adopt three distinct ways to measure Atomicity:

- `Acquisition Window` (AW) – this is the total window

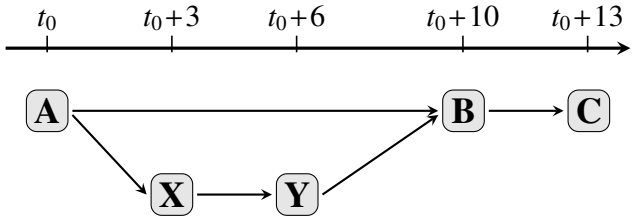


Figure 6: Time acquisition of nodes belonging to two paths.

that covers all data structures traversed in the path. E.g., one path may walk fifteen objects, all of which were acquired in a period of 23 seconds.

- `Cumulative Time Gap` (CTG) – this is the sum of the time difference of each edge traversed in the path. For instance, if a path visits three consecutive nodes (*A*, *B*, and *C*) and the difference between the acquisition time of the pointer in *A* and the content of *B* was 7 seconds and the difference between *B* and *C* was 3, the CTG would be 10 seconds.
- `Maximum Time Gap` (MTG) – this just takes into account the longest "jump" in a path. In the previous example, this would be 7 seconds.

All three measures are related to the Atomicity, but they capture different aspects. If it is important than none of the visited structures have changed during the acquisition, AW is the best metric. CTG gives instead a cumulative probability that things can go wrong by following links. The more edges are traversed, and the more far apart are the objects on the end of those edges, the more likely it is than a link can be corrupted due to the non-atomicity of the dump. Finally, MTG provides an estimation of the single most fragile edge in a path. This can be an important information, as traversing 10 edges each one a second apart can be a better option than traversing a single link with a nine seconds delay in the acquisition.

This can lead to some counter-intuitive results. For example, let suppose our graph analysis identifies two paths to reach a certain target structure *C* namely `{A` → `B` → `C}` and `{A` → `X` → `Y` → `B` → `C}` (for simplicity we ignore the name of the pointers). Both paths start from a structure `A` but the first traverses a single node `B` before reaching the destination while the second takes a detour through two other intermediate data structures `Y` and `Z` before re-joining the first path. Figure 6 shows the two paths on a time scale, that represent at which time the memory containing each data structure was collected.

While the second path is obviously a longer variation of the first, and therefore seems logical to believe that has nothing better to offer, it is very well possible that the detour reduces the probability of incurring in broken links. The pointer `A` → `B` was in fact collected 10 seconds before the object `B`, while the longest path decreases these time gaps to a maximum of four seconds. Whether this is an advantage or not depends on how often those pointers are modified in a running kernel, which we capture with our next metric.

Stability (numerical)

This weight expresses the stability over time of a given node or edge on the graph. Some structures are allocated at boot time and are never modified afterwards, while other parts of the graph are very ephemeral and contain structures that get allocated and de-allocated multiple times per second. By computing a heat-map of the stability of each edge (extracted by processing a number of consecutive snapshots), this weight can provide a valuable information on how the kernel map evolves over time, on which paths are more stable, and on which are instead more ephemeral and may only exists for short periods of time.

We measure Stability by computing the `Minimum Constant Time` (MCT) of all links in a path. The MCT can tell, for instance, that over a certain number of memory images all edges traversed by a certain heuristic remain constant for a minimum time of 30 seconds. In our experiments, we computed this metric by taking a snapshot of the same system at seconds 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60, 100, 200, 350, 700, 1000, 3000, 5000, 8000 and 12000.

Surprisingly we found that the 81% of edge are stable, i.e., they never change across our experiments. The majority of them are objects related to the file system (`inode` and `dentry`), which the kernel caches for performance reasons. But this does not mean the graph does not evolve, actually quite the opposite. For example, we saw an increase of more than 60% of both nodes and edges between the graph built at $t = 0$ and the one built at $t = 700$.

Moreover, if we exclude the filesystem subsystem and the paths that remained stable over all our experiments, 11% of the edges changed in less than 10 seconds, 12.5% in less than a minute, and 97% in the first hour.

Generality (numerical)

This weight captures another important problem of memory forensics: the constant change in the layout of kernel objects. This is due to several factors. First of all the kernel is always under active development which means that fields are continuously added to and removed from kernel objects definitions. Moreover, the layout is also influenced by the configuration options chosen at compile time. Existing tools mitigate this problem by requiring additional compile-time information (part of what it is normally called an *OS Profile*). Unfortunately, there are cases in which this information is not available, which today greatly complicate (if not completely preclude) the ability of analyzing a particular memory dump. Therefore, it would be interesting to compute analysis paths that traverse structures which change very rarely across different distributions, kernel versions, and enabled kernel options.

For this reason we downloaded 85 kernels from the Ubuntu repository, spanning from version `4.4.0-21` to `4.15.0-20`. For every object defined in each of these kernels we extracted the offset of the fields required for navigation – such as structure pointers or array of structures. We aggregated this information in a single `Kernels Counter` (KC) weight computed by counting over how many of the 85 kernels an entire path would remain constant (i.e., all its traversed link were present at the same offsets in their corresponding structures).

Reliability (boolean)

This is a very important aspect in memory forensics and captures how tamper-resistant is a given path on the graph, assuming an attacker is capable of reading and writing arbitrary kernel memory. Some paths are very easy for an attacker to modify, and therefore cannot be trusted by an analyst whenever she suspects the attacker might have gained admin privileges on the machine. On the other hand, other paths are more robust, as breaking them would make the system unstable. This can potentially result in programs malfunction or termination and, in the worst case, in a crash of the entire operating system. The robustness of individual data structures has already been studied in the past by several works [1, 10, 25]. But here we are instead interested in the reliability of a path, i.e., not in the fact that individual fields (such as a file name) can be modified, but whether an attacker can tamper with the edges that need to be traversed to prevent a certain heuristic to reach its destination (to the best of our knowledge, this problem has never been addressed in the literature). Being able to compute a path on the graph that only traverses tamper-resistant edges may have a great impact on memory forensics. While today we still do not have enough information to color the entire graph according to this metric, we can still compute the reliability on demand. This means that we cannot compute the optimal solution according to its reliability, but once we have a candidate solution we can perform experiments to verify it.

Consistency (boolean)

As a final property in this list we want to show how metrics can also be aggregated to capture more complex properties of a path. For this example we chose to combine the *stability* and *atomicity* of a path in a single measure that captures how likely it is for a given path to traverse consistent information. Intuitively, traversing a path whose nodes were acquired over a period of 20 seconds may be acceptable if those structures change very rarely, but completely unacceptable if its links are modified every few milliseconds. We capture this aspect by consider a path *consistent* if and only if the acquisition gap of each edge is lower than the minimum change time of the edge as computed in all our snapshots.

6 Experiments

We now discuss how our graph-based framework can be used in different scenarios, in which we investigate existing techniques used by Volatility [33], we discuss the intricacies of computing optimal paths, and we discover new solutions to reach all processes running in a system. In any case, these are only examples of what can be achieved by adopting a more systematic approach to memory forensics, and many more applications can benefit from our framework.

All experiments were conducted on a QEMU machine equipped with 2GB of RAM and 4 virtual CPUs, running `wordpress` on top of a `LAMP` stack. Before and between the acquisitions, we generated some activity by visiting the CMS pages and performing basic system administration task, such as logging in via `ssh` and updating the list of packages.

6.1 Scenario 1

In the first scenario we want to apply our methodology to study the quality of current memory forensics techniques. For our example we selected seventeen Volatility plugins that explore different subsystems (process, network, and filesystem) and mapped them as *paths* in our kernel graph. To achieve this we manually analyzed each plugin and extracted which global variables and kernel objects are traversed. With this information we were able to write a python script which automatically extracts these paths from our graph. Note that many plugins traverse similar kernel structures (e.g `linux_pslist` and `linux_pstree`) so, to avoid duplicates, we only report results for a subset that rely on different information. The final list of the plugins we analyzed is reported in Table 1.

Before looking at the individual metrics, we wanted to investigate to which degree the structures traversed by these heuristics are interconnected. The total number of unique objects used by this heuristics depends on the size of the graph. In our experiments they vary from 20 to hundreds of thousands. As we already introduced in Section 4.6, by averaging over the 25 graphs we created, more than 96% of the nodes used by the heuristics belong to a single giant *strongly* connected component that contains on average 53% of all the nodes in the graph. By combining this information with the nodes visited by Volatility, we found that this component contains all the information related to running processes, such as their mapped memory and open files, but also the information related to the `arp` table and the `tty`s. The remaining 4% of the nodes used by the Volatility rules are instead scattered among several other components. The biggest one, which contains only 0.5% of the heuristics nodes, contains the information related to the installed modules and, more in general, to the `kobject`s subsystem. Finally, the rest of the nodes belong to components containing only a single node. These are the nodes representing, for example, the global `pid_hashtable` and its associate `hlist_head`s.

Table 1: Comparison of Volatility plugins implemented as paths in our graph

Name	Description	# Nodes	Atomicity AW	Atomicity CTG	Atomicity MTG	Stability MCT	Generality KC	Consistency *Fast*	Consistency *Slow*
`linux_arp`	Prints the ARP table	13	16.24	53.25	16.24	12,000	50/85	✓	✓
`linux_check_afinfo`	Verifies the function pointers of network protocols	24	16.27	44.55	16.05	700	85/85	✓	✓
`linux_check_creds`	Checks processes that share credential structures	248	16.34	453.92	16.24	2	29/85	✓	✓
`linux_check_fop`	Check file operation structures for rootkit modifications	16099	16.38	142,856.15	16.38	0	29/85	✗	✗
`linux_check_modules`	Compares module list to sysfs info, if available	151	16.27	54.06	16.23	700	85/85	✓	✓
`linux_check_tty`	Checks `tty` devices for hooks	13	16.26	17.52	15.69	30	85/85	✓	✓
`linux_find_file`	Lists and recovers files from memory	14955	16.33	35,627.45	16.32	0	85/85	✗	✗
`linux_ifconfig`	Gathers active interfaces	12	16.25	44.19	16.25	12,000	50/85	✓	✓
`linux_iomem`	Provides output similar to `/proc/iomem`	7	16.70	50.09	16.70	12,000	50/85	✓	✓
`linux_lsmod`	Lists loaded kernel modules	12	16.23	44.27	16.05	700	85/85	✓	✓
`linux_lsof`	Lists file descriptors and their path	821	16.33	19,885.52	16.26	0	29/85	✗	✗
`linux_mount`	Lists mounted fs/devices	495	16.33	8488.13	16.32	10	85/85	✓	✗
`linux_pidhashtable`	Enumerates processes through the PID hash table	469	16.67	451.87	16.67	30	31/85	✓	✗
`linux_proc_maps`	Gathers process memory maps	4722	16.27	2629.19	16.24	0	31/85	✗	✗
`linux_proc_maps_rb`	Gathers process maps through the mappings rb-tree	4722	16.27	3310.69	16.24	0	31/85	✗	✗
`linux_pslist`	Lists active tasks by walking task_struct→task list	124	16.27	189.41	16.24	30	31/85	✓	✓
`linux_threads`	Prints threads of processes	157	16.27	280.68	16.24	30	31/85	✓	✓

This is an important finding, as it means that the vast majority of the information needed for forensic purposes is interconnected and reachable from one another. Translated in practical terms, the presence of this giant connected component means that is enough to locate a single kernel object to reach all the other interesting ones only by dereferencing pointers. This might be beneficial in scenarios where the position of global kernel objects is not available to the analyst. In such cases, one entry point can often be located by memory carving and then used as starting point for every other analysis.

By looking at the atomicity metrics (columns four-to-six in Table 1) the first thing that stands out is that the values for the Acquisition Window (`AW`) and the Maximum Time Gap (`MTG`) are very similar and relatively constant across all commands. After further investigation we discovered that this is due to the fact that, when compiled with normal configurations, Linux kernel global variables are located in the low part of the physical memory while kernel objects are allocated in the higher end. Since all heuristics start from global symbols, the very first edge already accounts for the maximum gap between two consecutive kernel objects. This also influences the acquisition window, since one of the two farthest objects is always the global variable from where the heuristic starts from. On the other hand, the Cumulative Time Gap (`CTG`) shows more variations as it is also influenced by the number of traversed objects.

To better understand this phenomenon, in Figure 7 we plotted the content of the physical memory as a Hilbert curve. In the graph, each pixel represents a physical page and its color shows if the page is traversed by the Volatility heuristics (red in the graph) or if it contains at least one node of our connected component of the kernel graph (green). It is clear that the relevant data structures are not spread equally on the entire physical memory. Instead, they clearly aggregated around three main clusters, which we marked respectively as C1, C2, and C3. In our experiments the kernel global variables are all located in C1 while other information are often stored in C2 and C3. Therefore, most heuristics start from C1 and

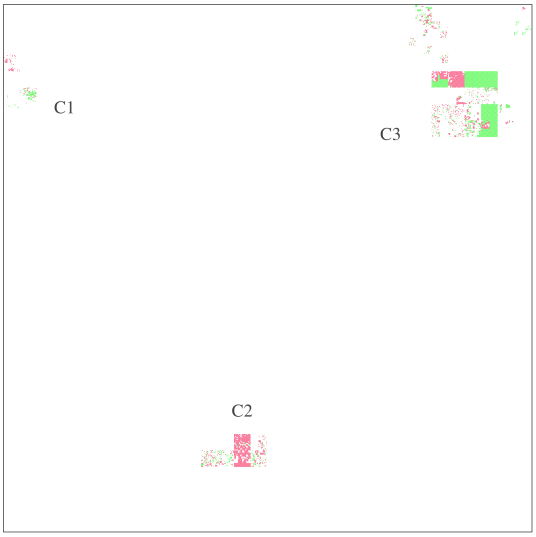


Figure 7: View as Hilbert curve of physical memory.

then eventually traverse an edge towards one of the other regions - which alone is responsible for the entire `AW` and `MTG` metrics. This physical distribution is also very important for the third scenario presented in Section 6.3, where we will encounter heuristics that need to hop back and forth from the three clusters, significantly impacting the atomicity metrics.

The second surprising result of this first scenario is the fact that the Kernel Counters (`KC`) of six plugins *never* changed across all the different kernel versions we used in our analysis. This means that, even when fields were added or removed from these object, the offsets of the fields used by the plugins remained constant. This has important implications for current memory forensics tools where a *profile* of the kernel is needed to analyze a memory dump. Our experiment suggests that, at least for locating certain information, a generic structure layout can be used across almost 100 kernel versions, released

as far as 2 years apart.

Another important propriety we evaluated in this first scenario is the consistency of the selected techniques. This is especially useful to better understand how the continuous modifications of kernel objects might impact memory dumps taken in a non-atomic fashion. This was recently listed by Case et al. [6] as "*one of the most pressing issues*" of memory forensics. While Case focused on page smearing (an inconsistency between the page tables and the referred physical memory), with this experiment we show that this problem does not affect only page tables but also references among kernel objects. The most important variable that influence the consistency of the memory is the duration of the acquisition process. To align with real world scenarios we run two different tests, by setting the acquisition ratio respectively to the fastest and to the slowest tool as reported by McDown et al. [21]. In that study, the authors compared seven different memory acquisition tools, chosen from a survey conducted over 41 companies specialized in memory forensics.

Interestingly, out of the 17 plugins we tested, three have a stability of 12,000 seconds, which means that none of the links they traversed *ever* changed over a period of more than three hours. At the other end of the spectrum, eleven plugins walked links that remained stable for less than a minute (and in five cases even less than one second). In this case, this may result in wrong pointers depending on how far in the physical memory were the page containing the link and the page containing the linked object. In fact, the last column of Table 1 shows that our analysis found inconsistencies in five (when the fastest tool to acquire the memory was used) or seven (in the case of the slowest solution was used) plugins. The affected plugins interest different parts of the kernel, but they can be divided to three distinct categories: *Memory* (`linux_proc_maps`, `linux_proc_maps_rb`), *File system* (`linux_check_fop`, `linux_find_file`, `linux_lsof`, `linux_mount`) and *Process* (`linux_pidhashtable`)

In the Memory category we found respectively 33 inconsistencies that affected the connections among `vm_area_struct` of a process, which are kept both in a linked list and in a red-black tree. These errors affected five instances of `apache`, one of `systemd-login` and one of `agetty`. The filesystem category included 40 unique inconsistencies in the hierarchy of dentries (fields `d_subdirs` and `d_child`) 53 in the mapping from a `dentry` to an `inode` (field `d_inode`). The latter object was also involved in 43 cases of inconsistency towards its `file_operations` object (field `i_fop`), while 23 `file` object had inconsistent edges pointing to their `dentry` and its `mount` objects. (field `f_path.dentry` and `f_path.mnt`). The most interesting cases of inconsistencies in this category – 10 in total – involved the array containing the pointers to the `file`s opened by a process. This array belonged to three distinct instances of `apache`, one of `systemd` and one of the `mysql` database. In the process category, we only detected one case of inconsistent edge between a `struct pid` and the pointed `task_struct`.

To systematically understand if these inconsistent paths can be avoided, we used once again our kernel graph – this time by filtering out *all* the 5,000 inconsistent edges, and searching for alternative paths to reach the same objects used by the affected plugins. Our graph exploration was able to discover alternative paths for 107 out of 213 inconsistent edges. For example, in the case of inconsistent array of opened files for the `systemd` process the alternative path — which traversed 11 additional nodes — was able to reach the target `file` by first locating the `task_struct` of the same process, then accessing its corresponding `files_struct` and from here reaching the `file` via the `fd_array` field (an array only used when the process opens less than 64 files). While these detours were sufficient in our experiments to retrieve the missing information, more experiments are required to understand if those alternative paths can be generalized to other scenarios. In any case, they show once more that the giant connected component that hosts most of the relevant data structures may allow analyst to find alternatives solution to mitigate the presence of wrong pointers and inconsistent information. Sadly, almost 50% of the affected pointers did not allow for an alternative path, thus emphasizing again the severe consequences that the lack of atomicity can have on memory analysis.

6.2 Scenario 2

In our second case study we want to understand if we can employ the kernel graph to find new heuristics for common forensics tasks. In particular, we focus on the starting point of many forensics investigation: listing the processes running at the acquisition time. Currently Volatility implements three different plugins[1] to list the processes, respectively by walking the process list, by using the `pidhash` hashtable, and by parsing the kernel memory allocator. However, the latter is only applicable if the kernel uses the `SLAB` allocator. Unfortunately, many distributions, such as Ubuntu and Debian, ships by default with the `SLUB` allocator, which is not supported by Volatility and which does not keep track of full *slabs* – thus making this technique not applicable anymore.

The main reason for looking for alternative solutions is that previous research already pointed out that rootkits are already capable of removing a process from the process list, but also to unlink a process from the pid hashtable [19,26,27] thus leaving the forensic analyst without a reliable method to list processes. Moreover, as we already discussed in the previous scenario, the lack of atomicity of a memory dump can introduce inconsistencies and result in broken pointers also in the list of running processes. For these reason, it is important to find new ways to locate processes, so that their output can be compared with other techniques to spot inconsistencies or hidden processes.

[1] Volatility also includes a plugin to carve `task_struct` objects by using a signature, but this is a parallel approach that does not require exploring memory but relies instead on pattern-matching.

Table 2: Comparison between different heuristics used to find processes

Category	Root Node	New	# Nodes	# task_struct	Atomicity AW	Atomicity CTG	Atomicity MTG	Stability MCT	Generality KC	Reliability	Consistency
scheduling	runqueues	✓	9	4	16.71	20.08	16.70	0.00	34/85	—	✗
	root_task_group	✓	10	4	16.65	21.14	16.27	0.00	18/85	—	✗
cgroup	css_set_table	✓	172	156	16.27	433.32	16.24	10.00	29/85	✗	✗
	cgrp_dfl_root	✓	186	156	16.30	369.10	16.30	10.00	29/85	✗	✓
memory/fs	dentry_hashtable	✓	58383	23	16.31	58120.38	16.30	0.00	36/85	✗	✗
	inode_hashtable	✓	14999	23	16.32	31594.48	16.31	1.00	36/85	✗	✗
workers	wq_workqueues	✓	427	69	16.68	1727.89	16.24	200.00	39/85	✗	✓
process	init_task(linux_pslist)	✗	124	124	16.27	189.41	16.24	30.00	31/85	✗	✓
	init_task(linux_threads)	✗	156	156	16.27	280.68	16.24	30.00	31/85	✗	✓
	pid_hash(linux_pidhashtable)	✗	469	156	16.67	451.87	16.67	30.00	30/85	✗	✓

This scenario is also interesting as it is harder to translate into a graph exploration problem. In fact, since we are looking for techniques to list all (or a part of) the running processes, this is equivalent to a *collection* of, possibly not homogeneous, paths. As a result, listing all processes is not simply equivalent to a path, but more to an *algorithm* to explore the graph.

Our approach to find new heuristics is the following. First, we discarded all the global roots that do not have a path to reach all the `task_struct`s in *every* graph we created. As a result, we were left with 621 global roots (out of more than 8000 we started with). Second, we modified the graph to remove the edges already used by known techniques, such as the `tasks` field. This helps removing all those paths that would just find a different way to reach a single process, and then walk the list like the existing plugins already do. While not useless per se, our goal is to find *new* solutions and not variations of the existing ones.

By only considering the *shortest* paths from every root node to every task structure, our system found more than 100 million distinct paths, generated from a set of more than 966,000 sequences of vertices. This is possible because, as we later discovered, the graph contained many parallel edges connecting the same nodes. In fact, by putting things in perspective, on average every sequence of vertices from a root node to a target object generates more than 100 unique paths. The good news is that this makes extremely difficult for attackers to modify all edges required to completely hide a process. On the other hand though, this also makes very hard the task of identifying interesting patterns in this multitude of options. For simplicity, we first decided to filter out all *similar* edges – i.e., parallel edges that shares the same metrics (and that therefore are equivalent for our purpose). This operation removed more than 300,000 edges, some of which played an important role in the path explosion. For example, many entries of the array `e_cset_node` of the `css_set` object pointed multiple times to the same vertex. After this operation the number of different paths decreased to about 7.5 millions paths.

We then merged similar paths into *templates*, constructed by keeping only the type of the objects present in the path, and by also removing adjacent nodes with the same type (which capture the ⟳ link discussed in Section 3). Finally, we removed templates that were subset of other templates, resulting in a final set of 4067 path templates.

By manually exploring these options, we soon realized that they belong to only four main families, depending on the kernel subsystem they live in. The first one is related to the *cgroup* subsystem, the second to the *memory* subsystem through the `mm_struct` structure, the third passes through the *work queues* to reach kernel workers, and the last traverses the `struct rq` and follows the `curr` field, a per-cpu runqueue. The results are summarized in Table 2.

Unfortunately, there are no alternative paths that can improve the atomicity. In fact, the bulk of the time gap (16.24 seconds) is due to the difference in the acquisition time of the global entry points (located in C1 in Figure 7) and the first task structure (located in C2 and C3). However, all these edges are very stable and in only one case (for the `css_set_table`) the value of this first connection ever changed during our memory acquisition.

The memory-based heuristics walked a red-black tree (`i_mmap`) that is very ephemeral and, while exploring it, we found more than 30 edges that could be inconsistent if the memory dump is not taken atomically. A similar problem affects the scheduler, whose structures also contain links that change very rapidly. We observed an interesting phenomenon in the cgroup-related heuristics. The first is inconsistent as it traverses a pointer with a very large time gap. However, the second avoid this problem by reaching the same `css_set` structures by taking a detour through several intermediate objects which act as a bridge to lower the time gap. This is an example of the counter-intuitive behavior we introduced in Section 5 (Figure 6), where we predicted that the most direct path might not always be the best in term of consistency. The worker-related approach was the best in terms of stability, consistency, and generality. However, its goal is to list all active kernel workers and therefore this heuristic is unable to capture normal userspace processes. Finally, the two heuristics in the process category, which represent the Volatility plugins `linux_pslist` and `linux_threads`, had both a stability of 30 seconds. This is strange, as several processes should have started during this

time frame. However, new processes were all appended to the tail of the process list without altering the intermediate nodes.

To test the Reliability of the heuristics we wrote a kernel module that tries to hide an userspace process by unlinking it from the path required by each heuristic. As a result, each case required a custom hiding technique. For the cgroup heuristics we deleted the processes from the `cg_list` linked list. For the memory we first found every non-anonymous, i.e. backed by a file, `vm_area_structs`. We then delete all this structures from the red black tree rooted in the `inode`, which keeps track of all the `vm_area_struct` which are currently mapping this file. For the first two process heuristics, we removed the process from the process list (by unlinking `task_struct.tasks`), while for the `pid_hash` we removed the `struct pid` from the hashtable. For the workqueue we instead created a custom workqueue and queued a simple work function that mimicked the behavior of the userspace process we used in our test. We then proceeded by unlinking the worker from the linked list rooted at `worker_pool.workers`.

In all the cases our program continued to run without observable side-effects – showing that each path we listed so far can be tampered with by a properly written rootkit. As we also discussed in Section 5, we believe that more experiments are needed to improve the assessment of a path's reliability. While it is true that our program continued to run, there can be a multitude of events (e.g. the kernel starting to swap memory) that might compromise the stability of the altered system.

6.3 Scenario 3

In the third scenario we show how we can compute optimal paths, with respect to the different metrics we proposed in this work. As running example, we picked this time the problem of finding the files opened by a given process (identified by its `task_struct`).

To run our experiments we collected all the `task_struct` and all the associated `file` objects and analyzed the paths Volatility would take to move from the first to the second. However, we immediately run into a strange behavior, as the metrics were returning very different results for different files. To understand the reason we had to look closer at how the physical pages were assigned to the different kernel objects.

Figure 7 explains very well the three classes of behavior we identified in our experiments. Since the clusters (C1, C2 and C3) are located far apart in memory (and therefore they can be acquired far apart in time), whenever a heuristic moves from one structure contained in one cluster to another contained in a different one, it needs to take a "jump" with associated a considerable time gap. If a `task_struct` and all the intermediate objects needed to reach the open files are located inside the same cluster, then time gaps are extremely small and path are always consistent. In this case paths are already optimal and there is no much room for improvement. If they are instead located in two different clusters, then the atomicity increase by almost nine seconds. However, the picture shows that also in this case it is not possible to find better alternatives, as all paths would need to cross the gap between the clusters– incurring in the same penalty. Finally, there are examples in which the `task_struct` and the `file` objects were located in the same cluster, but the intermediate structures traversed by Volatility resided in the other one. In this case the Volatility heuristic needs to jump across clusters twice, incurring twice in the risk of inconsistent links. But in this third case it might be possible to use our graph to find an alternative path that is fully contained in the same cluster.

An example of each of these three cases is shown in Table 3, along with the metrics computed on the Volatility heuristic and those computed on the optimal paths extracted from our graph. Regarding the cumulative time gap (CTG), our insight was correct and only paths belonging to the third category could be considerably improved. In fact, the table shows that from more than 17 seconds in the Volatility case, the optimal path had a CTG of less than 0.01 seconds. Accordingly, also the MTG decreased with the same magnitude. As we discussed in the previous scenario, finding a consistent path for this particular problem is sometimes possible. Indeed, when this is the case, we were able to find a path that remained stable for all our experiments. Interestingly, for the second case, one of the paths with maximum stability has also higher generality than the one used by Volatility but, since it passes through more nodes, it has an higher CTG. On the other hand, maximizing the generality of a path has a serious impact to its consistency and stability. In fact, while we were able to find paths which are constant over 50 kernels, none of them was consistent, independently to the speed of acquisition.

7 Discussion and Future Directions

The goal of our work is to provide a principled way to think about memory forensics as a graph-related optimization task. This way of modeling the problem opens the door to a multitude of different possibilities to evaluate and compare existing techniques, design algorithms to compute new alternative solutions, validate the consistency of kernel structures, or propose heuristics customized to different experiments setup and acquired dump.

We tried to discuss some of these opportunities through our experiments, but we are aware that many questions are still open and new research is needed to shed light to each individual use case. For this reason, we decided to release our code and data to other researchers, hoping that this will facilitate new experiments in this field and accelerate new findings based on our methodology.

In this paper we focused on the analysis of traditional computers. This choice was simply dictated by the fact that this is the area where memory forensics is more mature and for which most of the heuristics have been designed so far. Nevertheless, we believe that our system could be used to help researchers to better design and implement future forensic

Table 3: Optimal paths compared with Volatility paths

Name	# Nodes	Atomicity AW	Atomicity CTG	Atomicity MTG	Stability MCT	Generality KC	Consistency *Fast*	Consistency *Slow*
File A – all structures in one cluster								
`Volatility`	4	0.01	0.01	0.01	700	29/85	✓	✓
`Opt-MTG`	4	0.01	0.01	0.01	700	29/85	✓	✓
`Opt-CTG`	4	0.01	0.01	0.01	700	29/85	✓	✓
`Opt-MCT`	4	0.54	0.54	0.54	**12000**	29/85	✓	✓
`Opt-KC`	4	0.01	0.01	0.01	700	29/85	✓	✓
File B – structures located in two clusters								
`Volatility`	4	8.72	8.72	8.72	12000	29/85	✓	✓
`Opt-MTG`	4	8.72	8.72	8.72	12000	29/85	✓	✓
`Opt-CTG`	4	8.72	8.72	8.72	12000	29/85	✓	✓
`Opt-MCT`	11	16.23	72.84	16.21	12000	36/85	✓	✓
`Opt-KC`	8	9.71	46.15	9.71	0	**50/85**	✗	✗
File C – structures located in one cluster, with intermediate steps in the other								
`Volatility`	4	8.73	17.45	8.73	12000	29/85	✓	✓
`Opt-MTG`	3	0.003	0.003	**0.003**	12000	29/85	✓	✓
`Opt-CTG`	3	0.003	**0.003**	0.003	12000	29/85	✓	✓
`Opt-MCT`	3	16.23	82	16.20	12000	36/85	✓	✓
`Opt-KC`	10	9.71	55.56	9.71	0	**50/85**	✗	✗

frameworks tailored to emerging technologies such as mobile devices and the Internet of Things (IoT).

Main findings: our experiments show that a large part of the kernel graph belongs to a giant connected component. This means there are thousands, or even millions of possible paths that allow an analyst to move from one node to another. It also means that it is very difficult for an attacker to completely hide some piece of information from all possible paths.

Another consequence of the interconnected topology of the graph is that it is hard for an analyst to simply inspect all possible paths, looking for new techniques to implement in memory forensic tools. We tried to do this in our second scenario, and run into a path explosion problem even by considering only all shortest paths. However, this effort allowed us to discover two new promising techniques (one based on `cgroups` and one on `workrqueues`) that can complement those used today by Volatility [2].

Sadly, the problem of finding an optimal path turned out to be very delicate and dependent on multiple factors. In fact, the exact memory layout when the snapshot is acquired may affect the metrics associated to different links (e.g., one path may be optimal for one dump but poor in another). This may suggest that maybe, instead of relaying on a single solution, new techniques should try to explore the graph by following many parallel paths.

Moreover, we are aware that some of the metrics we proposed in this paper turned out to be ineffective in the evaluation. However, we decided to include them anyway in the paper for two reasons. First, because we did not know in advance that (for example the Maximum Time Gap) would be irrelevant in the analysis of common Linux kernels. This has nothing to do with the heuristic itself, but with the fact that the kernel allocates global variables (entry points) very far from other objects. We believe this fact to be an interesting finding which came as a consequence of applying our framework. Second, while this is true in our experiments, it is probably not the case on other operating systems or OS kernels. So, we believe it is still interesting to implement and discuss those ineffective metrics in our framework.

Finally, we want to stress the fact that our main contribution is not the discovery of new technique, but the introduction of a model that can be used to *reason* about memory analysis, explore its complexity, and perform quantitative measurements.

Future Work: In this paper we discuss a number of metrics an analyst can use to compare different solutions. However, the list is certainly not exhaustive and we expect more to be defined in the future. More work is also needed to understand which metric is better at capturing certain aspects of an investigation.

Reliability is certainly one of the most important characteristic of an analysis technique. Unfortunately, it is also the only one we discussed that cannot be extracted with automated experiments. More research is needed to fill this gap and enable to compute the reliability of a large amount of links among kernel objects.

Finally, to be useful in practice, our prototype should be applied to a larger number of memory dumps taken from different systems. This could help generalize the results and customize the analysis to an environment that resemble the one under investigation.

[2] We implemented both as Volatility plugins

8 Related Work

The analysis of kernel objects and their inter-dependencies has attracted the interest of both the security and the forensics community. While the common goal, namely ensuring the integrity of the kernel against malware attacks using the inter-dependencies between kernel object, is shared by the majority of works on this topic, the methods and the tools used to achieve it are often different. Several research papers have also been published on reconstructing and analyzing data structure graphs of user-space applications [3, 7, 28, 31]. However, most of these techniques are not directly applicable to kernel-level data structures, because they do not take in account the intricacies present in the kernel, such as resolution of ambiguous pointers to handle custom data structures. For this reason they will not be discussed in this Section.

To better highlight the different approaches, we decided to divide them in two distinct categories. The first one covers approaches which presented the analysis of a *running* kernel. The second category is focusing instead on *static* approaches, which require only a memory snapshot or the OS binary.

Dynamic Analysis

One of the first example of dynamic kernel memory analysis was presented by Rhee et al. in 2010 [26]. The tool, named `LiveDM`, places hooks at the beginning and at the end of every memory-related kernel functions, to keep track of every allocation and deallocation event. When these hooks are triggered, the hypervisor notes the address and the size of the allocated kernel object and the call site. The latter information is used, along with the result of an offline static analysis of the kernel source code, to determine the type of the allocated object. A work built on top of `LiveDM` is `SigGraph` [19]. In this paper the authors generate a signature for each kernel object, based on the pointers contained in the data structure. These signatures are then further refined during a profiling phase, where problematic pointers - such as null pointers - are pruned. The result can then be used by the final user to search for a kernel object in a memory dump. The major concern about signature-based scheme is their uniqueness, that avoids problems related to isomorphism of signatures. The authors found that nearly the 40% of kernel object contains pointers and - among this objects - nearly 88% have unique `SigGraph` signatures. Unfortunately the uniqueness was reported *prior* the dynamic refinement, so it is unclear the percentage of non-isomorphic signatures. For this reasons a kernel graph built using the approach adopted by `SigGraph` would only retrieve a *partial* view of the entire memory graph.

Another work focused on signatures to match kernel objects in memory was done by Dolan-Gavitt et al in 2009 [10]. The main insight of this work is that while kernel rootkits can modify certain fields of a structure - i.e. to unlink the malicious process from the process list - other fields (called *invariants*) can not be tampered without stopping the malicious behavior or causing a kernel crash. The invariant are determined in a two steps approach. During the first one every access to a data structure is logged, using the stealth breakpoint hypervisor technique [32]. Then, the most accessed field identified in the previous step are fuzzed and the kernel behavior is observed. If the kernel crashes then there are high chances that this field can not be modified by a rootkit. On the other hand, if no crash is observed, than the field is susceptible to malicious alteration. The direct results of these two phases is that highly accessed field which result in a crash when fuzzed are good candidates to be used as *strong* signatures. While this approach looks very promising is not easily adaptable to our context since, as also noted by the authors, creating a signature for small structures can be difficult. Furthermore, generating a signature requires to locate at least one *instance* of a structure in memory, which might not be straightforward.

Xuan et al. [35] proposed *Rkprofiler*. This tool combine a trace of read and write operations of malicious kernel code with a pre-processed kernel type graph, to identify the tampered data. The problem of ambiguous pointers is overcome by annotating the type graph with the real target of a list pointer. While it is not clearly stated in the paper, it seems the annotation was manually done. As we discussed in Section 4 the Linux kernel uses a large amount of ambiguous pointers, thus making the manual annotation approach not feasible anymore.

While more focused on kernel integrity checks, `OSck` [16] uses information from the kernel memory allocator (*slab*) to correctly label kernel address with their type. The integrity check are run in a kernel thread, separated by the hypervisor. This two components allow OSck to write custom checkers that are periodically run. While this approach seems promising, only a small subset of frequently-used structures are allocated using *slab* (such as `task_struct` or `vm_area_struct`), and thus is unsuitable for our needs.

Another approach to create a Windows kernel object graph is *MACE* [11]. Using a pointer-constrain model generated from dynamic analysis on the memory allocation functions and unsupervised learning on kernel pointers, MACE is able to correctly label kernel objects found in a memory dump. Once again, while the output of the work is a kernel object graph for Windows, the application only focuses on rootkit detection.

Static Analysis

One of the most prominent work in this field is KOP [4], and its subsequent refinement MAS [8]. Very similarly to our approach, the authors use a combination of static and memory analysis techniques respectively on the kernel code and on a memory snapshot. In the first step they build a precise *field-sensitive* points-to graph, which is then used during the memory analysis phase to explore and build the kernel objects graph. Contrary to this solution, ours does not make *any* assumption about the kernel memory allocator. While the Windows kernel has only one allocator, the Linux kernel has three different ones (`slab`, `slub`, and `slob`). Moreover, only a predefined subset of kernel objects are allocated in custom slabs, while the vast majority is sorted in generic slabs based on their size.

Gu et al. [13] presented `OS-Sommelier+`, a series of tech-

niques to fingerprint an operating system from a memory snapshot. In particular, one of these techniques is based on the notion of *loop-invariants*: a chain of pointers rooted at a given kernel object that, when dereferenced, points back to the initial object. Once a ground truth is generated from a set of known kernels, this loop invariant *signatures* can be used to fingerprint unknown kernels. Unfortunately the paper does not mention the problem of ambiguous pointers, and we believe our graph generation approach could improve `OS-Sommelier`$^+$ detection.

Finally, as we already discussed, none of the tools to automatically build a graph of kernel objects was publicly available.

9 Conclusion

Memory forensics focuses on locating and extracting artifacts from a memory snapshots, using a broad set of custom rules. However, the quality of the existing heuristics is difficult to measure and it is largely based on the experience of the researchers who wrote them. As a result, analysts are left without any clear guidelines on how to compare and evaluate different approaches and how to assess the results they produce.

For these reasons, in this paper we proposed a method to study memory forensics techniques in a principled way. Our solution is based on a graph representation that captures the relationships between all kernel objects, enriched with a set of metrics that covers different aspects of memory forensics. We believe that our framework can help researchers to measure the quality of existing memory forensics techniques, but also to extract qualitatively better heuristics.

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 771844 – BitCrumbs).

References

[1] Baliga, A., Ganapathy, V., and Iftode, L. Automatic inference and enforcement of kernel data structure invariants. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual* (2008), IEEE, pp. 77–86.

[2] Bastian, M., Heymann, S., Jacomy, M., et al. Gephi: an open source software for exploring and manipulating networks. *Icwsm 8* (2009), 361–362.

[3] Bursztein, E., Hamburg, M., Lagarenne, J., and Boneh, D. Openconflict: Preventing real time map hacks in online games. In *Security and Privacy (SP), 2011 IEEE Symposium on* (2011), IEEE, pp. 506–520.

[4] Carbone, M., Cui, W., Lu, L., Lee, W., Peinado, M., and Jiang, X. Mapping kernel objects to enable systematic integrity checking. In *Proceedings of the 16th ACM conference on Computer and communications security* (2009), ACM, pp. 555–565.

[5] Carvey, H. Digital forensics of the physical memory.

[6] Case, A., and Richard III, G. G. Memory forensics: The path forward. *Digital Investigation 20* (2017), 23–33.

[7] Cozzie, A., Stratton, F., Xue, H., and King, S. T. Digging for data structures. In *OSDI* (2008), vol. 8, pp. 255–266.

[8] Cui, W., Peinado, M., Xu, Z., and Chan, E. Tracking rootkit footprints with a practical memory analysis system. In *USENIX Security Symposium* (2012), pp. 601–615.

[9] Das, M. Unification-based pointer analysis with directional assignments. *Acm Sigplan Notices 35*, 5 (2000), 35–46.

[10] Dolan-Gavitt, B., Srivastava, A., Traynor, P., and Giffin, J. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM conference on Computer and communications security* (2009), ACM, pp. 566–577.

[11] Feng, Q., Prakash, A., Yin, H., and Lin, Z. Mace: High-coverage and robust memory analysis for commodity operating systems. In *Proceedings of the 30th annual computer security applications conference* (2014), ACM, pp. 196–205.

[12] Gruhn, M., and Freiling, F. C. Evaluating atomicity, and integrity of correct memory acquisition methods. *Digital Investigation 16* (2016), S1–S10.

[13] Gu, Y., Fu, Y., Prakash, A., Lin, Z., and Yin, H. Multi-aspect, robust, and memory exclusive guest os fingerprinting. *IEEE Transactions on Cloud Computing 2*, 4 (2014), 380–394.

[14] Hardekopf, B., and Lin, C. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *ACM SIGPLAN Notices* (2007), vol. 42, ACM, pp. 290–299.

[15] Heintze, N., and Tardieu, O. Ultra-fast aliasing analysis using cla: A million lines of c code in a second. In *ACM SIGPLAN Notices* (2001), vol. 36, ACM, pp. 254–263.

[16] Hofmann, O. S., Dunn, A. M., Kim, S., Roy, I., and Witchel, E. Ensuring operating system kernel integrity with osck. In *ACM SIGARCH Computer Architecture News* (2011), vol. 39, ACM, pp. 279–290.

[17] Ibrahim, A. S., Hamlyn-Harris, J., Grundy, J., and Almorsy, M. Digger: Identifying os kernel objects for runtime security analysis. *International Journal on Internet and Distributed Computing Systems 3*, 1 (2013), 184–194.

[18] Le Berre, S. From corrupted memory dump to rootkit detection. https://exatrack.com/public/Memdump_NDH_2018.pdf, 2018.

[19] Lin, Z., Rhee, J., Zhang, X., Xu, D., and Jiang, X. Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *NDSS* (2011).

[20] LWN. Linux kernel design patterns - part 2. https://lwn.net/Articles/336255/, 2009.

[21] McDown, R. J., Varol, C., Carvajal, L., and Chen, L. In-depth analysis of computer memory acquisition software for forensic purposes. *Journal of forensic sciences 61* (2016), S110–S116.

[22] Pearce, D. J., Kelly, P. H., and Hankin, C. Efficient field-sensitive pointer analysis of c. *ACM Transactions on Programming Languages and Systems (TOPLAS) 30*, 1 (2007), 4.

[23] Peixoto, T. P. The graph-tool python library. *figshare* (2014).

[24] Petroni Jr, N. L., and Hicks, M. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM conference on Computer and communications security* (2007), ACM, pp. 103–115.

[25] Prakash, A., Venkataramani, E., Yin, H., and Lin, Z. Manipulating semantic values in kernel data structures: Attack assessments and implications. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on* (2013), IEEE, pp. 1–12.

[26] Rhee, J., Riley, R., Xu, D., and Jiang, X. Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory. In *International Workshop on Recent Advances in Intrusion Detection* (2010), Springer, pp. 178–197.

[27] Riley, R., Jiang, X., and Xu, D. Multi-aspect profiling of kernel rootkit behavior. In *Proceedings of the 4th ACM European conference on Computer systems* (2009), ACM, pp. 47–60.

[28] Saltaformaggio, B., Gu, Z., Zhang, X., and Xu, D. Dscrete: Automatic rendering of forensic information from memory images via application logic reuse. In *USENIX Security Symposium* (2014), pp. 255–269.

[29] Schneider, C., Pfoh, J., and Eckert, C. Bridging the semantic gap through static code analysis. *Proceedings of EuroSec 12* (2012).

[30] Steensgaard, B. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1996), ACM, pp. 32–41.

[31] Urbina, D., Gu, Y., Caballero, J., and Lin, Z. Sigpath: A memory graph based approach for program data introspection and modification. In *European Symposium on Research in Computer Security* (2014), Springer, pp. 237–256.

[32] Vasudevan, A., and Yerraballi, R. Stealth breakpoints. In *Computer security applications conference, 21st Annual* (2005), IEEE, pp. 10–pp.

[33] Walters, A. The volatility framework: Volatile memory artifact extraction utility framework, 2007.

[34] Wilson, R. P., and Lam, M. S. *Efficient context-sensitive pointer analysis for C programs*, vol. 30. ACM, 1995.

[35] Xuan, C., Copeland, J. A., and Beyah, R. A. Toward revealing kernel malware behavior in virtual execution environments. In *RAID* (2009), vol. 9, Springer, pp. 304–325.

Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences

Kangjie Lu, Aditya Pakki, and Qiushi Wu
University of Minnesota

Abstract

Missing a security check is a class of semantic bugs in software programs where erroneous execution states are not validated. Missing-check bugs are particularly common in OS kernels because they frequently interact with external untrusted user space and hardware, and carry out error-prone computation. Missing-check bugs may cause a variety of critical security consequences, including permission bypasses, out-of-bound accesses, and system crashes. While missing-check bugs are common and critical, only a few research works have attempted to detect them, which is arguably because of the inherent challenges in the detection—whether a variable requires a security check depends on its semantics, contexts and developer logic, and understanding them is a hard problem.

In this paper, we present CRIX, a system for detecting missing-check bugs in OS kernels. CRIX can scalably and precisely evaluate whether any security checks are missing for critical variables, using an inter-procedural, semantic- and context-aware analysis. In particular, CRIX's modeling and cross-checking of the semantics of conditional statements in the peer slices of critical variables infer their *criticalness*, which allows CRIX to effectively detect missing-check bugs. Evaluation results show that CRIX finds missing-check bugs with reasonably low false-report rates. Using CRIX, we have found 278 new missing-check bugs in the Linux kernel that can cause security issues. We submitted patches for all these bugs; Linux maintainers have accepted 151 of them. The promising results show that missing-check bugs are a common occurrence, and CRIX is effective and scalable in detecting missing-check bugs in OS kernels.

1 Introduction

Security checks are a class of conditional statements that validate program execution states. Security checks play an important role in ensuring the security of OS kernels. Not only do OS kernels accept arbitrary untrusted inputs, but they also perform complicated tasks such as concurrent resource management and multi-user/capability access control. Therefore, OS kernels often enter into erroneous states and require security checks to capture them.

A missing-check bug exists when an intended security check is not enforced for a critical variable. Examples of such critical variables include the ones used to indicate potential erroneous execution states, e.g., the return value of `kmalloc()`, and the ones used in critical operations, e.g., the size variable in `memcpy()`. Figure 1 shows a concrete example of missing-check bugs. `ib_get_client_data()` may fail and return `NULL`. Since `smcibdev` is not checked, the following uses of it may cause multiple problems—NULL-pointer dereferences, failures in removing and unregistering devices, and memory leaks. To fix the problem, a security check should be enforced between lines 6 and 8 to ensure that `smcibdev` is not NULL.

```
1  /* Linux: net/smc/smc_ib.c */
2  static void smc_ib_remove_dev(struct ib_device *ibdev...)
3  {
4    struct smc_ib_device *smcibdev;
5      /* ib_get_client_data may fail and return NULL */
6    smcibdev = ib_get_client_data(ibdev, &smc_ib_client);
7      // ERROR1: NULL-pointer deference
8    list_del_init(&smcibdev->list);
9      /* ERROR2: device cannot be removed or unregistered */
10   smc_pnet_remove_by_ibdev(smcibdev);
11   ib_unregister_event_handler(&smcibdev->event_handler);
12     /* ERROR3: memory leak */
13   kfree(smcibdev);
14     /* No return value: caller cannot know the errors */
15 }
```

Figure 1: Example: A new missing-check bug found by CRIX. The missed check against variable `smcibdev` will cause multiple problems, as annotated in the code.

Missing-check bugs may cause critical security impacts because security checks are a main means for OS kernels to ensure their security and reliability. To understand the importance of security checks, we first studied recently reported security vulnerabilities in the National Vulnerabilities Database (NVD). We found that 59.5% security vulnerabilities stem

from missing-check bugs, which were all fixed by inserting security checks. We then investigated these vulnerabilities and found that at least 52% (excluding denial-of-service cases) of them will cause severe security impacts such as permission bypass, memory corruption, system crashes/hangs.

Although missing-check bugs are critical and prevalent, only a few research works have attempted to detect them in OS kernels and have several limitations. In particular, Vanguard [32] assumes that some critical operations should always be checked. It however detects missing-check bugs for only four specified critical operations such as arithmetical division and array indexing. Some other approaches (e.g., Chucky [46], Juxta [23], Kremenek et al. [19], and Dillig et al. [7]) employ cross-checking, inconsistency analysis, or machine learning to reduce false positives in detecting bugs. These approaches have non-trivial limitations. First, the manual specification for critical variables covers only a small set of critical variables. This leads to significant false negatives. Second, most of these approaches are not semantic- or context-aware. For example, they tend to treat any conditional statement (i.e., an `if` or a `switch` statement) as a security check. In fact, whether a variable requires a security check highly depends on its semantics and contexts, without considering which, the detection would suffer from high false-negative and false-positive rates.

The lack of effective research in detecting missing-check bugs is arguably because of several inherent challenges. (1) Critical variables that require security checks take diverse forms. For example, a critical variable can be a parameter of a critical function (e.g., the size variable in `memcpy()`), a global variable, or a return value of a function call that is used in only security checks but not others such as arithmetic operations (this case is missed by Vanguard [32]). Therefore, generally checking for different kinds of critical variables is hard.

(2) Identifying security checks requires semantic understanding. Treating any conditional statement as a security check will cause both significant false positives and false negatives. In fact, according to our study §6, the majority (about 70%) of conditional statements are not security checks but some normal selectors in which both branches of the conditional statements lead to normal execution. (3) Missing-check bugs are context dependent, and the detection should be context aware. For example, an error code may not require a security check at all if it is used in a debugging function. As such, missing-check detection should be context aware. (4) Last but not least, OS kernels are extremely large and complex. Checking every variable will not scale, and corner cases such as hand-written assembly will make the analysis error-prone.

In this paper, we present CRIX (Criticalness and constraints Inferences for detecting missing checks), a system that overcomes the aforementioned challenges to effectively detect missing-check bugs in OS kernels. At a high level, CRIX first employs an automated approach to identify critical variables as the analysis targets. For each critical variable, CRIX constructs *peer* slices that share similar semantics and contexts. After that, CRIX models constraints of conditional statements in each slice. By cross-checking the modeled constraints of the peer slices of a critical variable, CRIX finally identifies deviations as potential missing-check bugs and reports them for further confirmation.

While the high-level idea of CRIX is intuitive, it entails overcoming multiple technical challenges. We thus have developed multiple new techniques to tackles these challenges. (1) We first propose a two-layer type analysis to identify indirect-call targets, which serves as a foundation of our data-flow analysis engine. In addition to the function-type analysis (the first layer) employed by traditional control-flow integrity (CFI) techniques [4, 25, 40], the two-layer type analysis further uses struct-type analysis, which is also employed by Ge et al. [10], to refine indirect-call targets. (2) We then develop an automated analysis that identifies security-checked variables as potential critical variables, which not only narrows down the analysis scope and thus scales the detection to OS kernels, but also significantly reduces false reports by filtering out non-critical variables. (3) We further propose peer-slice construction to collect slices of a critical variable that share similar semantics and contexts. The set of peers enables effective cross-checking for potential missing check cases of the critical variable. (4) At last, to precisely detect missing-check cases, we construct constraints from the conditional statements in the peer slices and model them based on their semantics (e.g., the condition type in conditional statements). The modeled constraints allow CRIX to cross-check slices for detecting missing-check bugs, in a semantic-aware manner.

With the new techniques, CRIX's analysis is scalable, semantic- and context-aware, and the data-flow analysis engine in CRIX is inter-procedural, flow-, context-, and field-sensitive. By focusing on the small set of automatically identified critical variables, CRIX can scale to large programs like the Linux kernel. The peer-slice construction allows CRIX to reason about potential missing-check cases in a semantic- and context-aware manner, and the constraint modeling and cross-checking enable CRIX to infer the criticalness of critical variables. As a result, CRIX is able to effectively and precisely detect missing-check bugs in complex and large system software such as the Linux kernel.

We have implemented CRIX on top of LLVM as multiple static-analysis passes. We chose the Linux kernel as the experimental target given its prevalence and complexity (more than 25 million SLOC). CRIX finished the analysis for the whole Linux kernel in about one hour and reported many missing-check cases. By manually investigating the top 804 missing-check cases reported by CRIX, we confirmed 278 new missing-check bugs. We also submitted patches for all of them to the Linux maintainers. Out of these patches, 151 have been accepted, with 134 applied to the mainline Linux kernel and 17 confirmed. The results show that CRIX is highly scalable and effective in finding missing-check bugs. We also

discuss CRIX's portability in §6 and believe that CRIX can be easily extended to detect missing-check bugs in other system software.

We make the following contributions in this paper.

- **A new system for missing-check bug detection.** Missing-check bugs constitute the root cause of the majority (59.5%) of recent security vulnerabilities. We propose a semantic- and context-aware approach to scalably and effectively detect missing-check bugs in OS kernels. The resulting system, CRIX, is open sourced [1] .

- **Multiple new general techniques.** We propose multiple new general techniques in CRIX, which would benefit other research. In particular, the peer-slice construction identifies code paths that share similar semantics and contexts, which is useful for general differential analysis. The automated critical-variable inference finds a small set of targets that deserve precise analysis and protection, which narrows down target scope and could improve the performance for techniques such as fuzzing and data-flow integrity. The two-layer type analysis refines indirect-call targets, without introducing false negatives, which is also useful for inter-procedural static analysis, control-flow integrity, and program debloating.

- **Numerous new bugs in the Linux kernel.** With CRIX, we found a large number of new missing-check bugs in the Linux kernel, which may cause critical security and reliability issues to the Linux kernel used by billions of devices. We reported these new bugs and have worked with Linux maintainers to fix many of them.

The rest of this paper is organized as follows. We present the study on missing check bugs in §2, the design of CRIX in §4, implementation of CRIX in §5, evaluation of CRIX in §6. We further discuss the extension and limitations of CRIX in §7. We present related work in §8, and conclude in §9.

2 Missing Checks in OS Kernels

To propose an effective approach to finding missing-check bugs, we first study the characteristics of previously reported missing-check bugs.

Bug-set collection. We collect previously reported missing-check bugs from NVD [26]. We first selected the recent 200 vulnerabilities that were reported during 2017 and 2018. Out of them, we then selected the ones fixed by enforcing security checks, which returned us 119 (59.5%) vulnerabilities. We finally took the missing-check bugs leading to these vulnerabilities as the bug set for our study.

[1] https://github.com/umnsec/crix/

2.1 Impact of Missing-Check Bugs

To assess the impact of missing-check bugs, we investigated (1) what percent of security vulnerabilities are caused by missing-check bugs, (2) common classes of security vulnerabilities caused by missing-check bugs, and (3) severe security impact of missing check–related vulnerabilities.

Percent of missing check–related vulnerabilities. As we mentioned in the bug-set collection, a majority (59.5%) of recent security vulnerabilities were caused by missing-check bugs. This is expected because common vulnerabilities such as out-of-bound access and access-control errors are typically fixed with security checks.

Common classes of missing-check impact. We then classified the security impact of the 119 missing-check bugs based on the classification provided by CVEDetails [48]. We found that missing-check bugs can introduce at least ten classes of vulnerabilities. Table 1 shows the six most common classes. In particular, more than half of the missing-check bugs may result in denial-of-services, and more the 52% of them may result in other severe impacts. Some missing-check bugs may have multiple impacts, so the total number is $> 100\%$.

DoS	Over flow	Bypass privi.	Info leak	Memory corrupt	Code exec.
51.2%	16.0%	14.3%	11.7%	6.7%	3.4%

Table 1: Common security impacts of missing-check bugs.

Severe security impact. We also looked into the most severe vulnerabilities from 2017 to 2018 that have a CVSS (Common Vulnerability Scoring System) score 10 (the highest severity level) from the Linux kernel. We in total found 15 such vulnerabilities. Specifically, we found that 11 of these vulnerabilities are caused by missing-check bugs. The targets of the missing checks in these vulnerabilities include buffer length, function return value, pointer value, and permissions. Correspondingly, the missing-check bugs will cause severe impacts, including buffer overflow, use-after-free, memory corruption, permission pass, which will finally result in data losses, information leaks, and even attackers control of the whole system. Figure 2 shows an example of a severe missing-check bug (CVE-2017-18017) with a CVSS 10. The attacker-controllable `len` and `tcp_hdrlen` are used as a loop-termination condition for memory access. Missing the security checks for these variables will result in denial of service, information leak, and memory corruption.

2.2 Targets of Security Checks

According to our analysis of the missing-check bugs, generally, security checks have two classes of targets: state variables and critical-use variables.

```
/* Linux: net/netfilter/xt_TCPMSS.c (CVE-2017-18017) */
static int tcpmss_mangle_packet(struct sk_buff *skb,
        unsigned int tcphoff, ...) {
    tcph = (struct tcphdr *)(skb_network_header(skb) + tcphoff);
    tcp_hdrlen = tcph->doff * 4;
    /* Security checks for both "len" and "tcp_hdrlen" */
    if (len < tcp_hdrlen || tcp_hdrlen < sizeof(struct tcphdr))
        return -1;
}
```

Figure 2: A missing-check bug causing multiple severe security impacts: denial of service, information leak, and memory corruption. The bug is assigned with ID CVE-2017-18017.

State variables. State variables indicate if the current execution is in an erroneous state, e.g., if an operation is successful or not. According to the C programming convention, a return value of a function often serves as an indicator of execution states. State variables are prevalent in OS kernels because kernel operations are error-prone. OS kernels have to frequently use and check state variables to ensure that an operation is successful. A special feature of state variables is that they are often used in only security checks but not any other function-related operations such as arithmetic operations. Line 4 in Figure 3 is an example of checking the state variable `ret`, which is used only in the security check at line 4. In function `btrfs_search_slot()`, different error codes are returned for indicating various erroneous states, which should be checked in callers.

Critical-use variables. Variables used in critical operations are another common class of check targets. Intuitively, variables should be checked before being used in a critical operation. Common critical-use variables include pointers in dereferencing, offsets in array indexing, operands of binary operations such as arithmetic division, and parameters in critical functions (e.g., `memcpy()`) that may cause security issues. Line 9 in Figure 3 is an example of checking the variable `tx_out` before it is being used in the critical function `dmaengine_submit()` which internally dereferences `tx_out`.

```
/* Linux: fs/btrfs/inode-map.c */
ret = btrfs_search_slot(NULL, root, &key, path,0,0);
/* "ret" is a state variable for the search operation */
if(ret < 0)
    goto out;

/* Linux: drivers/crypto/omap-des.c */
/* "tx_out" is checked before "dmaengine_submit" uses it */
if(!tx_out)
    return-EINVAL;
dmaengine_submit(tx_out);
```

Figure 3: Examples of check targets. `ret` is a state variable checked in line 4, and `tx_out` is checked at line 9 because it is used by the critical function `dmaengine_submit()`.

3 Overview of CRIX

The goal of CRIX is to detect missing-check bugs in OS kernels. To this end, CRIX automatically infers whether a variable in the target OS kernel requires a security check. The detection of missing-check bugs, in general, is to answer the following questions: (1) does a variable require a security check, (2) if possible, what security check should be enforced for the variable, and (3) is such a security check present. Answering these questions requires the understanding of the contexts and semantics of the code, which is challenging. Prior research [41] has shown the promise of statistical inferences in finding bugs. Such inferences identify inconsistent cases as potential bugs, which avoids the hard problem of understanding contexts and semantics. It makes sense that a deviation from common patterns is often problematic and thus is likely a potential bug, given that a majority of the code is correct. In CRIX, we also employ the general idea of statistical inference to find missing-check bugs. However, compared to the previous detection, CRIX is context and semantic aware.

Figure 4 shows the overview of CRIX. CRIX consists of three phases: (1) preprocessing phase which prepares a global call graph, control-flow graph, and alias results; (2) analysis phase which performs the key analyses to identify critical variables, construct peer slices for them, and construct constraints for peer slices, and (3) postprocessing phase which cross-checks constraints of peer slices and reports missing-check bugs.

In the first phase, given the LLVM IR (intermediate representation), CRIX constructs a precise global call graph, which is not only foundational to all the following data-flow analysis, but also enables the peer-slice construction, as will be presented in §4.3. Since LLVM does not provide targets of indirect calls, CRIX employs a technique, namely two-layer type analysis, to precisely find indirect-call targets.

In the second phase, CRIX first identifies critical variables. Because a large number of variables are non-critical in OS kernels, conservatively checking all variables would cause significant scalability and false-positive issues. CRIX therefore first identifies critical variables (see §2.2). The intuition behind the critical-variable identification is that security-checked variables are typically critical. Therefore, CRIX identifies the security-checked variables as critical variables, which however requires CRIX to first identify security checks. Since the majority of conditional statements are not security checks [43], CRIX employs an approach to identify security checks, as presented in §4.2.1.

Since the critical variables identified through security checks have already been checked in the current code paths, CRIX instead tries to identify missing-check bugs in the peer code paths. To this end, CRIX constructs peer slices that share similar semantics and contexts with the current code path checking the critical variable. To find substantial peer slices, given a critical variable, CRIX identifies the sources and uses of the critical variable, and employs data-flow analysis to find slices for each source and each use (see §4.3).

A slice of a source or a use of a critical variable may or may not contain a security check. A naive approach is to iden-

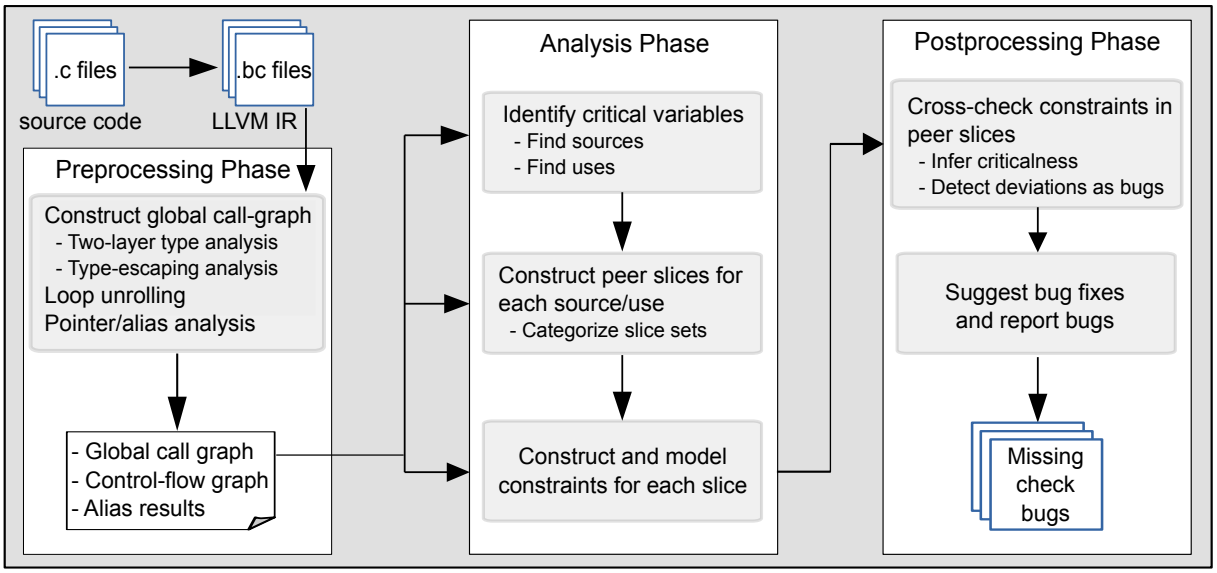


Figure 4: The overview of CRIX. CRIX has three phases. It takes as input LLVM IR and produces missing-check bug reports.

tify any slices that do not have a security check as potential missing-check bugs. This will however introduce significant false positives because (1) the source or the use may not be very "critical"; (2) even if security checks are present in some slices, they may not be semantically equivalent. To address this problem, CRIX first extracts the constraints from the conditional statements in the peer slices and models them in a special way (see §4.4) that can both preserve the semantics and facilitate the following bug detection.

With the modeled constraints extracted from conditional statements, in the last phase, CRIX cross-checks (statistical analysis) them to infer the "criticalness" of the source or the use based on how common the constraints are, i.e., how frequently the source or use is checked in its peer slices. If the criticalness is significant, not having a constraint would be identified as a deviation, and a slice that does not have the constraint would be identified as a potential missing-check bug. In the end, CRIX suggests bug fixes based on the constraints in the peer slices and reports the details for further manual confirmation.

4 Design of CRIX

In this section, we present the design of the key techniques in CRIX, including the identification of indirect-call targets, construction of peer slices, construction and modeling of constraints, and statistical analysis of constraints for reporting missing-check cases. Other techniques such as alias analysis and loop unrolling will be presented in the implementation section (§5).

4.1 Identifying Targets of Indirect Calls

A precise call graph serves as a foundation for a variety of program analyses and security defense mechanisms. In particular, any inter-procedural data- and control-flow analysis requires a precise call graph. Control-flow integrity (CFI) [1, 9, 10, 25] and software debloating [28] techniques also require a precise call graph. Unfortunately, in large programs, constructing a precise call graph is an open problem in general because of the challenge of finding the targets of indirect calls. At compilation time, it is hard to know which address-taken functions would be valid targets of an indirect call.

Existing approaches for finding the targets of indirect calls can be classified into two categories: pointer analysis [2, 3, 8, 22, 36, 37] and type analysis [4, 9, 25, 40, 42]. Pointer analysis–based approaches aim to find the point-to relationships between dereferenced function pointers and address-taken functions. Such approaches have fundamental limitations. While unsound pointer analysis will miss valid function targets, sound pointer analysis often introduces a large number of false positives—many unrelated functions are included as potential targets of an indirect call. Further, the pointer analysis *itself* requires a precise call graph. Whenever the pointer analysis encounters an indirect call, an expensive recursive analysis must be employed to find the targets.

Due to the limitations with pointer analysis–based approaches, recent CFI research opted for type analysis. Type analysis–based approaches try to match the number and types of arguments of an address-taken function with the ones of an indirect call. Matched functions are considered potential targets of the indirect call. Such approaches have been used in practice. For example, LLVM-CFI [4] employs such a type analysis. Type analysis–based approaches are conservative in

that all possible targets are included as long as function-type casting, which is rare, is handled properly [25]. However, they tend to suffer from false positives—many unrelated functions are included as valid targets. This will cause significant inaccuracy in the following data-flow analysis. The problem becomes even more critical in CRIX because the construction of peer slices heavily relies on a precise call graph.

To the best of our knowledge, the hybrid approach proposed by Ge et al. [10] for finding indirect-call targets in OS kernels is the most precise one. It employs both taint analysis and type analysis to find the targets. Specifically, it first taint-tracks the propagations of function pointers to identify indirect-call targets. Moreover, for function pointers stored in `struct`-type objects, because the function pointers should typically be loaded from the objects of the same `struct` type, the approach uses the `struct` type to further restrict the indirect-call targets. To avoid false negatives, the approach has two assumptions: (1) the only allowed operation on a function pointer is assignment, and (2) there exists no data pointer to a function pointer. The approach uses static taint analysis to detect and report violations which will be fixed manually. In addition to the hybrid analysis, the approach also analyzes assembly code, which further restricts the indirect-call targets.

4.1.1 Two-Layer Type Analysis

To improve the existing type analysis–based approaches in finding indirect-call targets, we propose *two-layer type analysis*, which aims to dramatically refine the targets produced by previous type analyses. The first-layer type analysis uses function types to restrict indirect-call targets. The second-layer type analysis instead uses `struct` type to further restrict the targets, which is based on a similar observation as in the approach proposed by Ge et al. [10]. Specifically, in large systems such as OS kernels, the majority of taken addresses (e.g., 88% for the Linux kernel, according to our study in §6) of functions are first stored to a function-pointer field of a `struct`, and later, to dereference the addresses in indirect calls, they must be loaded from the `struct`. In LLVM IR, the type information of the `struct` in both store and load operations is present. Intuitively, in these cases, function addresses that are *never* stored in the specific `struct` will not be valid targets of the indirect calls that load the function addresses from the `struct`. This way, by further matching the `struct` types in the store and load operations, we can further refine the indirect-call targets. 12% of function addresses in the Linux kernel are not stored to `struct`. A common example in the Linux kernel is that a function address is stored to a function-pointer variable which is further used as an argument of another function. Indirect calls dereferencing these function pointers will not benefit from the second-layer struct-type matching.

Figure 5 shows an example, in which the addresses of functions `adp5589_reg` and `adp5585_reg` are stored in the `reg` field of a `struct` with type `adp_constants`, in line 10 and 16, respectively. Later on, the addresses are loaded from the field of the `struct` of the same type and dereferenced at line 4. Our two-layer type analysis finds exactly only *two* targets for the indirect call because there are no any other functions whose addresses are ever stored to the field of the `struct` type. In comparison, since the indirect call has only one argument of a basic type, traditional one-layer type analysis matches 20 functions as targets for the indirect call, 18 of which are false positives.

A struct may have multiple fields that hold function pointers. To further improve the analysis accuracy, our type analysis is field-sensitive. That is, it recognizes which field is holding the particular function pointer, by analyzing the offset of the field in the data struct. In some rare cases, when the offset is undecidable because the indices are non-constant, we roll back the analysis to be field-insensitive.

```
1  /* drivers/input/keyboard/adp5589-keys.c */
2  static int adp5589_gpio_add(...) {
3      /* Indirect call: "kpad->var" is of type "adp_constants" */
4      kpad->var->reg(ADP5589_GPIO_DIRECTION_A);
5  }
6
7  unsigned char adp5589_reg(unsigned char reg)
8  static const struct adp_constants const_adp5589 = {
9      // address of "adp5589_reg" assigned to the field "reg"
10   .reg      = adp5589_reg,
11 };
12
13 unsigned char adp5585_reg(unsigned char reg)
14 static const struct adp_constants const_adp5585 = {
15     // address of "adp5585_reg" assigned to the field "reg"
16   .reg      = adp5585_reg,
17 };
```

Figure 5: An example of how a function pointer is stored to and later loaded from a field of a `struct`.

4.1.2 Type-Escaping Analysis for False Negatives

Our two-layer type analysis is sound as long as the `struct` types holding function addresses do not escape—we cannot decide what function addresses a `struct` can hold. When a `struct`, say `structA`, has escaped, a function address stored to a different `struct`, say `structB`, can be loaded from the memory with `structA`; however, in this case, the function address will be missed by the type analysis because we cannot find that the function address is ever stored to `structA` but only `structB`. Such escaping cases exist when (1) the `struct` holding the function addresses is cast to or from a different type; (2) the function-pointer field of `struct` is stored to with a value of a different type (e.g., `unsigned long`). These cases may make the function addresses a `struct` can hold *undecidable*.

To handle this problem, we use conservative type analysis to find all store and casting operations and analyze the types in the sources and destinations based on the aforementioned criteria for deciding escaping cases. When an escaped type

is found, we conservatively discard the type in our two-layer type analysis. That is, if the function pointer of an indirect call is loaded from an escaped type, we use only one-layer type analysis for this indirect call. This way, we ensure that our two-layer type analysis does not introduce extra false negatives to existing one-layer type analysis.

Although CRIX shares the similar insight into further restricting indirect-call targets with the approach proposed by Ge et al. [10], CRIX differentiates itself from the approach. CRIX employs a two-layer design that allows the type analysis to be elastic. Whenever the second-layer type analysis fails, CRIX falls back to the first-layer type analysis. Second, the escaping analysis conservatively finds and discards invalid types to ensure the soundness.

4.2 Identifying Critical Variables

System software has a large number of variables. Conservatively checking all of them is not only unscalable but also generates an overwhelming number of false reports. Intuitively, important variables are often protected with security checks. We say that a variable is a (potential) critical variable if it is validated in a security check. By identifying security checks and their targets, we can identify critical variables. Note that a critical variable has different levels of *criticalness*. As will be shown §4.4, the criticalness is inferred based on check ratio of the occurrences of the critical variable. In this section, we first focus on identifying critical variables.

4.2.1 Identifying Security Checks for Critical Variables

Since we define validated variables in security checks as critical variables, CRIX first identifies security checks using a similar approach proposed in LRSan [43]. Specifically, checking failures typically require failure handling which has clear patterns: returning an error code or calling an error-handling function. We say that an `if` statement is a security check if its two branches satisfy the following two conditions: (1) one branch handles a checking failure, and (2) the other branch continues the normal execution. Note that an `if` statement whose two branches both handle checking failures is not a security check. Therefore, the key step to identify security checks is to determine whether the branches have the failure-handling patterns. Two typical failure-handling primitives are returning an error code and calling an error-handling function. Since LRSan supports only error-returning cases, we extend the idea by supporting error-handling functions.

System software such as the Linux kernel has a small number of basic error-handling functions. Such functions are often critical and implemented in assembly. For example, `BUG()`, `panic()`, and `dump_stack()` in Unix-like OS kernels are functions for handling unrecoverable errors. Moreover, functions such as `pr_err()` and `dev_err()` are used for reporting error messages, which have clear patterns. Specifically, such functions typically have a name or an argument with a severity level (e.g., `KERN_ERR`, `KERN_CRIT`, and `KERN_EMERG`). Moreover, such functions take a variable number of parameters. Detecting these patterns is straightforward for a static analysis tool. To ensure that our heuristic-based approach reports correct error-handling functions, we manually investigated the results and filter out false-positive cases. In total, we found 531 error-handling functions (available in the code repository). In comparison, while LRSan reports only 131K security checks, CRIX reports 308K security checks. Once we identify security checks, we extract the checked targets as critical variables.

4.2.2 Identifying Sources and Uses of Critical Variables

In the next step, CRIX collects the sources and uses of the critical variables (i.e., checked variables). It is important to identify sources (where a critical variable propagates from) and uses (where a critical variable is used) of critical variables for two reasons. First, criticalness of a variable can propagate. When a critical variable is moved to another variable, the destination variable also becomes critical. By identifying the sources and uses, we can identify families of critical variables that propagate from the same sources or propagate to the same uses. Second, by identifying a family of critical variables, we can analyze how frequently they are checked, which is used to infer the criticalness of a source or a use. We realize the identification of sources and uses through a standard *inter-procedural* data-flow analysis—backward analysis for identifying sources and forward analysis for identifying uses. The inter-procedural data-flow analysis uses the following definitions to identify sources and uses.

Definition of sources. If a value is never critical, we do not need to include it for further analysis. Therefore, we include only potentially-critical values as sources. The inter-procedural backward data-flow analysis collects the following variables as sources.

- Constants. Constants such as error codes are critical.
- Return values and parameters of certain functions. Input functions (e.g., `copy_from_user` and `get_user`) obtain inputs from the external entities, which are untrusted. In addition, functions implemented as handwritten assembly often perform critical operations. We include the corresponding parameters or return values of such functions as sources.
- Global variables. Global variables may contain critical values that may propagate to the whole program.
- Others. When CRIX cannot find a predecessor instruction, the current values are marked as sources.

Note that we do not include allocations as a source because they become critical only when critical values are written to the allocated memory.

Definition of uses. Further, the following operations are defined as potentially critical uses of critical variables.

- Pointer dereference. Any pointer dereferencing operation is a critical use of the pointer variable.
- Indexing in memory accesses. Using the offset variable in memory access is also a critical use.
- Binary operations. We conservatively treat binary operations such as arithmetic division as critical uses.
- Functions calls. When none of the above is found, we take the closest function call that takes the critical variable as a parameter as a critical use.
- None. If none of the above is found, we deem that this critical variable does not have any use. This is common for critical variables that are error codes.

Algorithm 1: Collect sources and uses of critical variables

```
collect_src_use_interprocedural(CVSet, FuncSet);
  Input: CVSet: Critical variables, i.e., identified checked variables;
         FuncSet: Input and assembly functions, collected in the
         pre-processing phase
  Output: SrcSet: Potentially critical sources of critical variables;
          UseSet: Potentially critical uses of critical variables

SrcSet ← UseSet ← ∅;
BackupSet ← CVSet;
// Collect sources
while Is_Not_Empty(CVSet) do
    CV ← pop top element from CVSet;
    if CV is Constant and CV is ErrorCode then
        SrcSet ← {CV} ⋃ SrcSet;
    else if CV is Global variable then
        SrcSet ← {CV} ⋃ SrcSet;
    else if CV is return value or param. of a function in FuncSet
      then
        SrcSet ← {CV} ⋃ SrcSet;
    else if CV has no parents (predecessors) then
        SrcSet ← {CV} ⋃ SrcSet;
    else
        Parent ← Predecessor of CV, via Backward Analysis;
        CVSet ← CVSet ⋃ Parent;
    end
end
// Collect uses
CVSet ← BackupSet;
while Is_Not_Empty(CVSet) do
    CV ← pop top element from CVSet;
    CVUseSet ← Forwardly collect immediate uses of CV;
    for Use ∈ CVUseSet do
        if Use is a pointer dereference or memory access then
            UseSet ← {Use} ⋃ UseSet;
        else if Use is a binary operation then
            UseSet ← {Use} ⋃ UseSet;
        else if Use is a parameter of function then
            UseSet ← {Use} ⋃ UseSet;
        else
            CVSet ← CVSet ⋃ {Use} ;
        end
    end
end
return SrcSet, UseSet;
```

Algorithm for identifying sources and uses. Based on the definition of sources and uses, the algorithm presented in Algorithm 1 collects all potentially critical sources and uses. The algorithm takes as input the set of critical variables (`CVSet`), and set of functions (`FuncSet`). `CVSet` are the checked variable extracted from a security check, and `FuncSet` is the set of pre-collected input functions (e.g., `copy_from_user`) and assembly functions. `FuncSet` is collected in the pre-processing phase of the CRIX, concurrently with security-check identification, as will be shown in §5. The algorithm then produces two sets as the output: `SrcSet` and `UseSet`, the source and use sets, respectively. `CVUseSet` contains the immediate and forward uses of the current `CV`, which are returned by LLVM's `value.users()` function. As shown in Algorithm 1, the analysis is recursive and inter-procedural. Note that the algorithm is used to collect potentially critical sources and uses, but not to infer criticalness. Criticalness is instead inferred by measuring how frequently a critical variable is checked before being used, as will be shown in §4.4.

4.3 Constructing Peer Slices

At this step, we have the sources and uses of critical variables. Seemingly, we can construct slices for the critical variables forwardly from their sources and backwardly from their uses, and cross-check the slices to find check deviations as potential missing-check cases. Such a naive approach will suffer from at least two problems. First, the slicing will easily lead to path explosion [15] given the complexity of OS kernels. Second, if slices do not share similar semantics and contexts, we cannot effectively detect missing-check bugs because missing a check in an unrelated slice does not necessarily indicate a potential bug. Consequently, such an approach will lead to significant false positives.

To solve these problems, for a source or a use, we must construct its peer slices. Such peer-slice construction should satisfy two requirements: (1) the construction should yield sufficient peer slices to enable cross-checking; (2) the peer slices should share similar semantics and contexts. Given a control-flow graph, we observed that `call` (both direct and indirect) and `return` instructions often generate peer paths. In particular, for sources, indirect calls and return instructions often have substantial targets. As the example shown in Figure 5, indirect call `pad->var->reg()` serves as a dispatcher that may target multiple semantically similar callee functions (e.g., `adp5589_reg` and `adp5585_reg`). Since the arguments in the callee functions all come from the same caller, they also share the similar contexts. For uses, when the used critical variable comes from an argument of the current function, direct calls to the function also generate substantial edges from the callers to the function (callee). Since the arguments passed from various callers to the same callee function, they are used as similar semantics in similar contexts.

Figure 6 illustrates how we find different classes of peer paths for sources and uses. For each critical-variable source, we perform forward data-flow analysis for it. When encoun-

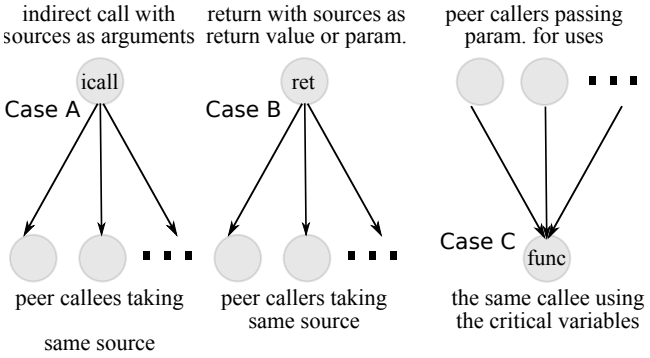


Figure 6: Different cases generating peer paths. `icall` is indirect call; `func` is the callee taking critical variables from peer callers.

tering an indirect call that takes the source as a parameter, we collect all the indirect-call callees as a set of peer paths. Similarly, when encountering a return instruction, we analyze whether the critical variable is returned or written into the memory pointed to by an argument (in this case, the critical variable may be further used in the callers through a pointer parameter). If so, we collect callers (starting from the next instruction following the call) as a set of peer paths. Our analysis is recursive. That is, the forward data-flow analysis continues to find more sets of peer paths until the end of the propagation of the critical variable or a critical use of the variable is found. For uses, we instead perform backward data-flow analysis from a use of a critical variable. If the critical variable comes from an argument of the current function, all callers of the function are collected as peer paths. The backward analysis is also recursive and ends until the source of the critical variable is found. Since a peer path may further contain multiple sub-paths, we use a simple `BFS` algorithm to flatten all sub-paths. Therefore, each peer path can be viewed as a single path. Finally, we construct peer slices by slicing the peer paths. The slicing ends at a conditional statement or the end of the path. Therefore, each slice has at most one conditional statement. Note that ending at the closest conditional statement would not cause false negatives because the slices sets are collected in a recursive manner, and our detection in §4.4.2 will cross-check each peer set.

For each critical-variable source and use, the peer-slice construction produces multiple sets of peer slices and categorizes them into four classes, each corresponds to a case in Figure 6.

- `Source-Ret` corresponds to `case B`. A critical variable is returned as the return value to multiple peer callers.
- `Source-Param` also corresponds to `case B`. However, in this case, a critical variable is "returned" an output parameter to multiple peer callers.
- `Source-Arg` corresponds to `case A`. A critical variable is passed to peer callees through an indirect call.
- `Use-Param` corresponds to `case C`. A critical variable used in a function is passed in from multiple peer callers.

4.4 Constructing and Cross-Checking Check Constraints

Until now, CRIX has produced multiple sets of peer slices of different classes for each critical-variable source and use. Each slice may or may not contain a conditional statement. The next step of CRIX is to cross-check the slices to detect deviations in the absence of security checks as potential missing-check cases. We choose to cross-check conditional statements instead of security checks (a subset of conditional statements) in this step for two reasons. First, the security-check identification part in CRIX have false negatives and may not identify all security checks; cross-checking security checks only may have significant false negatives because deviations can be normalized. Second, although cross-checking all conditional statements may introduce false positives, our fine-grained modeling for conditional statements can mitigate this issue.

A simple approach to cross-check slices for deviations is to treat conditional statements equally and quickly find deviating slices that do not have any conditional statement. Such coarse-grained analysis may have false negatives because conditional statements may have completely different semantics, and having a conditional statement does not mean the slice has checked the source or use. On the other hand, exactly comparing concrete values in conditional statements would be too restrictive, leading to false positives. For example, when two slices have `if (len < 8)` and `if (len < 16)`, respectively, treating them as different checks is too aggressive because both of them indeed enforce length checks. To avoid these problems, we must "qualitatively" understand the semantics of conditional statements in the slices. To this end, we propose to construct and model constraints from conditional statements. Note that the modeling focuses on the semantics of conditional statements, which does not consider their positions in the slices.

4.4.1 Modeling Conditional Statements as Constraints

As described in §4.3, a slice is flattened as a single code path using BFS, and a slice has at most one conditional statement. The goal of this step is to answer what classes of semantics a conditional statement has, with a proper granularity. We thus use two empirical rules to model the conditional statements based on the semantics of typical conditions and comparison operators. The modeled conditional statements will be cross-checked for missing-check bugs, as shown in §4.4.2.

1. If the conditional statement checks the return value of a function call, we identify the function's signature as the constraint. For example, if a variable is checked in a conditional statement, `if (IS_ERR(ret))`, we model the constraint as "`IS_ERR(int)`". This is, the slice uses `IS_ERR()` to check the source or use.
2. Otherwise, we model the conditional statement as "`<opcode_type, operand_type>`", where `opcode_type`

represents the type of the comparison, such as `eq`, `ne`, and `lt`; and `operand_type` represents the type of the condition operand. The type can be `var` (a variable), `zero`, `positive` constant, and `negative` constant. For example, if a condition statement is `if (len < 8)`, the constraint will be modeled as "`lt positive`".

4.4.2 Detecting Deviations as Potential Bugs

With the modeled constraints for all slices in a set, we cross-check them to find deviations. The idea of the detection is to calculate the `relative frequency (RF)` [18] for each constraint in the set. Since different constraints have different frequency distributions, we calculate the RF for each constraint in the set separately. More specifically, given a constraint, we define N_{nc} as the number of slices that do not have the constraint, and define N_t as the total number of slices in the set. With these two numbers, the RF is defined as in Equation 1.

$$RF = \frac{N_{nc}}{N_t} \tag{1}$$

The detection works as follows. Given a constraint in a peer-slice set, the detection counts how many slices do not have this *particular* constraint. Note that a slice that has a different constraint will also be counted. The count serves as N_{nc}. Since N_t is the total number of slices in the set, we can quickly obtain it and calculate the RF for the given constraint. If the RF is very small, i.e., most slices have the constraint, the detection reports slices that do not have the constraint as potential missing-check cases. A slice set may have multiple constraints, and the detection will go through the steps for each constraint in the slice set.

5 Implementation

We have implemented CRIX as multiple passes on top of LLVM, including a pass for constructing call graph and unrolling loops, a pass for finding security checks and critical variables, and a pass for detecting and reporting missing-check cases. CRIX's source code contains 4.5K lines of C++ code. The rest of the section describes some interesting implementation details in each phase.

5.1 Preprocessing Phase

Disabling inlining and IR pruning. To facilitate peer-slice construction, we aim to preserve callsites as much as possible. To this end, we chose to disable inlining by modifying `Clang`. A side effect of disabling inlining is that *inline* functions defined in header files will be copied to each module that uses them, leading to significant redundancy in LLVM IR. To prune the IR, we leverage debugging information to map the functions to its source code. This way, we can figure out multiple functions in IR share the same source code, and if so, we keep only one copy in the IR and discard all other copies. The pruning strategy reduces the original size of IR by approximately 30%.

Identifying indirect-call targets. To realize the two-layer type analysis, we first identify all store operations (either a store instruction in LLVM or a struct initializer) that assign a function address to a variable. We then analyze the type of the *memory* holding the variable. At this step, our analysis is conservative: the variable must be loaded from a pointer, and the pointer must be pointing to a field of a data structure. That is, the pointer must be a `GetElementPtrInst` in LLVM. With the type information in LLVM IR, we can then extract the base struct type from the pointer. Note that, we do not recursively find the struct type; if the pointer is not `GetElementPtrInst`, or the base type of the `GetElementPtrInst` is not a `struct` or is an aggregated type (e.g., `union`), we stop the analysis for the particular function address and roll back to the traditional one-layer type analysis. The filed-sensitive analysis is realized by analyzing the indices in the `GetElementPtrInst` which includes the index of the accessed field into the base type. The process of this step goes through all address-taken functions in all modules. The output of this step is a map from the hash of the type to the function addresses.

A challenge in implementing the two-layer type analysis to conservatively capture escaping types. As described in §4.1.2, we have a conservative policy to identify escaping types. To implement the type-escaping analysis, we analyze the operand types in cast and store operations (both instructions and global static initializers). If the operand types satisfy the policy, we identify them as escaping types.

After that, we match the second layer type for indirect calls. Similarly, we analyze the type of the *memory* holding the function pointer (address) in the same way—analyzing the corresponding `GetElementPtrInst`. By querying the map, we can find the matched functions for the indirect call. If we cannot find a match, we again roll back to the one-layer type analysis for the indirect call.

Unrolling loops. To avoid path explosion, we chose to unroll loops by treating `for` and `while` statements as `if` statements, which is a common strategy used in practice [45]. A loop has two special basic components: header block, latch block. A header block is the entrance node for a loop; a latch block contains an edge back to the header block. In order to unroll loops, we delete the back edge and add a new edge from the latch block and the successor block of the loop.

Pointer analysis. We perform points-to analysis for each pointer to a memory location within a function, relying on LLVM's AliasAnalysis infrastructure. The `MayAlias` results conservatively include pointers that may refer to the same object; two pointers referring to different fields of an object may also be included as aliases. To refine the results, we perform field-sensitive data-flow analysis for each pointer that is ever used in memory load/store or function calls as

parameters. Pointers that are validated to refer to different fields are excluded from the `MayAlias` results. A second issue with the points-to analysis is its significant runtime overhead, and we observed that this is mainly caused by a small number of objects that have a large number of pointers. We mitigate the problem by limiting the maximum number of pointers an object can alias simultaneously. By setting the number to 1000, our results showed that only 23 functions in the Linux kernel have aliased memory pointers with size greater than this limit. After applying the two improvements, the running time for points-to analysis is reduced from 103 minutes to only 24 minutes, and the average number of alias pointers of a object is reduced by 65%.

5.2 Analysis Phase

Modeling input functions and collecting assembly functions. In CRIX, specific return values and parameters of input functions and assembly functions are defined as sources (§4.2.2). As such, we need to collect a set of such functions. We define a function as input function if it may fetch data from outside. For example, `copy_from_user(dst, src, size)` copies the content from user-space memory `src` into the kernel-space memory `dst`. In total, we empirically collected 36 input functions (Table 3). We also specified which parameter or if the return value of these functions holds the inputs. Similarly, the kernel contains lots of assembly code as optimizations for performance reasons. Such functions are typically critical. Since LLVM does not support analysis of assembly code, we also model the assembly functions and treat them as sources. Identification of assembly functions is realized by scanning through LLVM IR files for `isa<InlineAsm>` instructions.

5.3 Postprocessing Phase

Selecting threshold for relative frequency. Missing checks within the Linux kernel are identified using various strategies described in §4. A case in a peer-slice set that has low relative frequency will be reported as a potential missing-check bug. The relative frequency is the ratio of occurrences of a constraint or "non-constraint" to the size of the peer-slice set. A uniform threshold for different categories might skew the results in favor of a particular type of bugs. To solve this challenge, we provided the relative frequency field as a tuning parameter and tested the results on various runs for various categories. We observed that the relative frequency works best between [0.1, 0.15] to detect sufficient missing-check bugs with reasonably low false reports, for all the categories.

Generating bug-fixing suggestions. Peer slices of the same critical variable can reveal many interesting details about the implementation. For each peer, we are able to reason about the constraints of the critical variable. Since we have constraints for each of the peer slices, one can suggest a possible security check in a possible location for missing-check cases. Statistically analyzing the "suggestions" returns us a reasonable bug fix. As such, CRIX always reports the most common suggestion to facilitate bug fixing. The report includes the most common constraint and which function the constraint should be applied to.

Bug Reporting. After collecting the constraints from the peer slices and using a user-defined relative frequency, we rank the missing-check output based on the relative frequency. we format the report to output the line contains the relative frequency, the Linux source code, the module containing the code, the number of times security check was checked, times missed among the peers, and most importantly, the bug-fixing suggestion. As expected, reported cases in the top of the ranking are more likely to be true bugs.

6 Evaluation

We extensively evaluate the scalability and effectiveness of CRIX using the Linux kernel. We also evaluate the effectiveness of our two-layer type analysis. The experiments were performed on Ubuntu 16.04 LTS with LLVM version 8.0 installed. The machine has a 64GB RAM and an Intel CPU (Xeon R CPU E5-1660 v4, 3.20GHz) with 8 cores. We tested the bug detection efficiency of CRIX, on the Linux kernel version 4.20.0-rc5 with the top git commit number `b72f711a4efa`, the latest patch as on Dec 6, 2018. Using the `allyessconfig`, we generated 17,343 LLVM IR bitcode files to cover as many modules as possible.

6.1 Precision in Finding Indirect-Call Targets

Results. In total, out of 57,299 indirect calls, 45,840 (80%) enjoyed our two-layer type analysis. 5,019 (8.8%) indirect calls suffer from type escaping thus disqualify the two-layer type analysis. Others indirect calls do not load function pointers from a `struct` thus do not trigger the two-layer type analysis. The high percentage confirms our observation that most function pointers are stored to and loaded from memory through data `struct`. We then calculate the average number of targets for an indirect call before and after applying our two-layer type analysis. The results show that the average number over all indirect calls for traditional type analysis is 134 while it is only 33 for our two-layer type analysis. We further calculate the average numbers over indirect calls that can benefit the two-layer type analysis. The results show that the average target number is 129 and 9 (i.e., 7%) before and after using our two-layer type analysis, respectively, which confirms that the analysis can dramatically refine the indirect-call targets.

Measuring the false positives of indirect-call targets is a challenging problem because of the complexity of pointer propagation and point-to relationships. Existing CFI tech-

niques use the average number of targets to represent the accuracy of target refinement. Given that CRIX reports only an average number of 9 for indirect calls that benefited from the two-layer type analysis, we expect the false-positive rate of our analysis to be low. In comparison, the hybrid approach proposed by Ge et al. [10] reports an average number of 6.64 for indirect calls in FreeBSD, and, the number is calculated over all indirect calls. We believe that the accuracy of the approach benefits from the combination of taint analysis and type analysis.

6.2 Analysis Performance and Numbers

CRIX completed the analyses of the kernel for missing-check cases in 64 minutes, of which pointer analysis required 24 minutes and the remaining analysis to identify and report missing-check cases required 28 minutes. By running CRIX over the whole kernel, with a threshold of `0.15`, the output contained 308K security checks from 1,028K conditional statements, and reported 804 cases.

6.3 Bug Findings

Table 2 presents the bug detection statistics of CRIX, running on the entire Linux kernel with a constant relative frequency of 0.15, across categories. We used a fixed number for relative frequency to avoid inconsistencies while comparing similar bugs across the various categories. CRIX reported 804 potential bugs and manual analysis confirmed 278 new bugs. To manually analyze all the bugs, it took three researchers, a total of 36 man-hours. We found that the cross-checking results over peer slices can significantly relieve the manual analysis by suggesting how and why peers enforce the security checks. The manual effort was mainly spent in checking if the critical variable is actually checked because the check may have been missed by CRIX due to issues such as aliasing. In most cases, the "suggested" source-check or check-use chains are across one or two functions, so the manual analysis overall is straightforward.

We submitted patches for all the bugs. Linux maintainers accepted 151 of the submitted patches to be applied to the latest Linux version or future releases. Maintainers confirmed 99 patches within a week of submission confirming the criticalness of fixing missing-check bugs. Figure 1 shows an example of the new bugs found by CRIX, which can cause multiple security issues such as NULL-pointer dereferencing. A detailed list of all the bugs is available in Table 4 and in Table 5, in the Appendix section. During our interaction with the maintainers, we not only fixed missing-check bugs determined by CRIX, but also fixed some other relevant bugs present in the error paths of security checks including but not limited to missing/incorrect error handling, missing resource releases, use-after-free, and dead code.

Further 76 bugs are included in more than one bug category. That is, these 76 bugs were detected twice, once each while generating the source and use constraints. However, these duplicates are within the chosen 804 cases, used to evaluate CRIX. Accounting for the duplicate bug reporting, the false-positive rate of CRIX is 65%. We believe this is an acceptable number for critical software such as OS kernels.

The distribution of bugs is heavily skewed towards driver code. The report showed 195 bugs in the driver modules and at least 27 driver modules had more than one missing-check. These bugs reinforce previous research studies that the driver code is indeed buggy as well as confirm the effectiveness of CRIX in detecting new missing checks. Second, we also computed the latent period of the detected bugs and the average time between the initial patch and detection is 1,675 days or approximately 4 years and 7 months. A significant observation is 27 out of these 278 bugs have a latent period of greater than 10 years and 6 patches' latent period is greater than 13 years.

The third interesting finding of our bugs involves the type of bugs. A total of 79 bugs involve memory allocation on the heap. Linux developers strictly maintain that every pointer returned by an `alloc`-like function be checked for emptiness. Interestingly, CRIX identified 11, 5, 11, and 12 calls of (`kzalloc, kmalloc, kcalloc` and `kmemdup`) respectively; all missing a check on the pointer to the allocated memory for emptiness. All these bugs can crash a system while dereferencing the NULL pointer, as well as provide an attack vector to launch a denial of service attack by unauthorized users. The numerous missing-check bugs confirm the effectiveness of CRIX in identifying security vulnerabilities.

One reason of concern in the output report is the duplication of bugs across various categories. CRIX performs backward data-flow analysis from use, and a forward data-flow analysis from source to identify missing-check bugs in various categories. With a constant relative frequency, a true missing-check bug will often be reported in both directions. To simplify our analysis, we ran CRIX performing both analyses at the same time and then eliminated the duplicate records. While this action does not impact the accuracy of the system, we observed a non-trivial difference in ranking order of the bug, when evaluating each category individually.

6.4 False Positives

As presented in §6.3, CRIX has false positives. We have investigated the causes of false positives. In this section, we present the main classes of causes.

Inaccurate points-to analysis. Pointer analysis [14] is a hard problem. CRIX's data-flow analysis engine generally relies on the Alias Analysis. However, the alias results provided by LLVM are often inaccurate. Al through we have refined the `MayAlias` results, there are still over 48% of false positives that are caused by the inaccuracy of pointer analysis. We will

Category	Example bug (related function)	Latent Period	R	A	C
Source-Ret	drivers/net/hyperv/netvsc_drv.c +1377 (kvmalloc_array)	4y 10m	449	300	156
Use-Param	net/ncsi/ncsi-netlink.c +253 (nla_nest_cancel)	2y 10m	247	150	115
Source-Param	drivers/gpu/drm/i810/i810_dma.c +307 (drm_legacy_ioremap)	4y 1m	83	42	4
Source-Arg	drivers/dma/ti/omap-dma.c +1056 (omap_dma_prep_dma_cyclic)	10y	25	8	3

Table 2: Bug detection statistics of CRIX on Linux kernel with relative frequency = 0.15. Columns R= bugs reported, A = Analyzed bugs, C = Confirmed bugs. The Latent Period is the average time differential for all confirmed bugs(C), within the category.

discuss potential improvements of pointer analysis in §7.

Inconsequential checks. While checks are necessary to guarantee the state of the kernel, programmers often ignore security checks in cases such as debugging code, failure-handling paths, driver-shutdown functions, resource cleanup functions, unlikely failures (e.g., `kmalloc` with `__GFP_NOFAIL`) or code that is already protected by synchronization primitives. Checks are redundant in these cases as an erroneous state has already existed or a valid state is guaranteed by the kernel. Such cases account for 25% of false positives.

Implicit checks. Programmers can reason about the state of the variable in an implicit way. For example, to test if an allocation of an object is successful, developers may use the object, without a security check, in a function, and use the return values of the function to test if the object was allocated successfully. In this case, although the object itself is never explicitly checked, the function call checks the object implicitly. Such cases contribute about 8% of the false positives. A potential solution to mitigating this problem is to maintain a list of "checker" functions.

Other causes. Besides these above-mentioned causes, false positives can also be caused by complex programmer logic, imprecise static analysis techniques, etc. All these account for the remaining 19% of the false positives.

6.5 False Negatives

CRIX provides a tuning parameter, the threshold of RF, while detecting missing-check cases. In other words, the threshold can influence the false-negative rate. In this section, we evaluate (1) the absolute false negatives that are missed when the RF threshold is set to 1; and (2) the relationship between the false-negative rate and the RF threshold.

In §2, we collected 119 missing-check bugs that have a clear security impact. To make the false-negative evaluation more robust, we collect 231 recently reported missing-check bugs in the Linux kernel based on its `Git` patch history. The patches containing the fixes of these 350 missing-check bugs are evaluated in our false-negative study. To reproduce these bugs, we revert the patches in the Linux kernel to the version used in our experiments.

By setting the RF threshold to 1, we determined 14 (4%) patches as absolute false negatives. Absolute false negatives are caused by two factors. First, the checked critical variables are not captured by CRIX because the error code or error-handling functions are not identified as part of a security check. Second, inaccurate pointer analysis identifies incorrect aliases for the critical variables. These aliases are mistakenly identified as valid security checks, bypassing the identification of actual missing-check bugs.

Second, we also evaluate the relationship between the threshold and false-negative rate, presented in Figure 7. We find that, as RF threshold increases, the false-negative rate decreases as prior false negatives are identified as missing-check bugs. We found that when the threshold is set to 0.13, the false-negative rate is 5% and reaches its elbow point. Further tuning the threshold has no impact on the false negative rate.

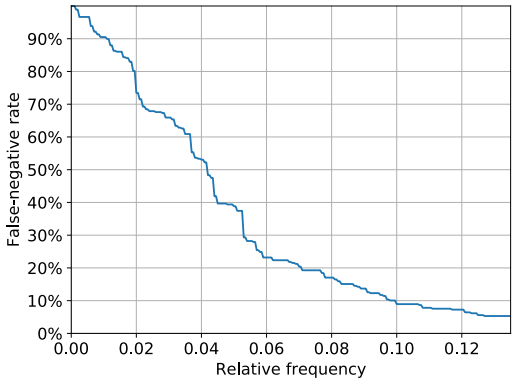


Figure 7: Relationship between the relative frequency (RF) threshold and the false-negative rate.

6.6 Portability

A program-specific component of CRIX is identifying security checks. Determining if a conditional statement is a security check, while scanning the kernel relies on the identification of error handling functions, and error codes. Identifying the error handling functions requires a limited amount of experience with the target code base. In Unix-like kernels, these functions share similar patterns, called "Single Unix Specification" [35], as presented in §4.2.1. All these kernels, also have a single global header file that defines the standard error codes.

Similarly, other kernels and programs like browsers also have corresponding header files containing the error codes.

Besides this step, the idea to generate an error control flow graph is generic to adapt to other systems. Once security checks, described in §4, are identified, the algorithm for identifying missing checks is easily adaptable to other software systems such as BSD kernels and C++ code base such as web browsers.

7 Discussion

Two-layer type analysis for more types. The current implementation of two-layer type analysis supports only `struct` type because it is the most commonly used type for memory holding function pointers. To further exploit the two-layer type analysis, we could extend it to support more types such as `array`, `global variable`, and `vector`. The type-escaping analysis in CRIX will ensure to eliminate false negatives when any type casting occurs or function pointers are moved across different types.

RF threshold. We discussed the false positives and false negatives of CRIX in §6. To balance false positives and false negatives, we suggest setting the RF threshold to a value between 0.1 and 0.15 for the Linux kernel. CRIX cross-checks peer slices to detect deviations. CRIX may have higher false-report rates in smaller target programs because they have small sets of peer slices.

Pointer analysis. Another major cause of false reports is the inaccuracy of alias analysis. To mitigate this problem, we intend to use Andersen pointer analysis [13] and Steensgaard pointer analysis [34] in the future. Given that, pointer analysis is used extensively in CRIX, we believe that this addition can significantly improve the overall accuracy.

Inconsequential checks. Besides alias analysis, the next major portion of false positives are due to programmer intended missing checks. Based on our interaction with Linux maintainers, we found that they are reluctant to fix missing-check cases in resource-release paths such as driver shutdown or state reset. Previous work by Saha et.al [31] proposed a pattern-based approach to find resource-release paths. As a potential solution, we may leverage the approach to filter out cases in resource-release paths and thus reduce the false positives in CRIX.

Determining exploitability and security impact of missing-check bugs. To automatically determine the exploitability of missing-check bugs, one can employ symbolic execution [29] and a theorem prover like Z3 [6] to generate inputs to trigger a missing-check bug. In addition, fuzzers can complement the limitations with symbolic execution. To automatically determine the security impact, one can analyze the uses of the checked variable to understand the potential security impact. For example, if a checked variable is used as the size variable in `memcpy()`, the potential impact can be memory corruption or information leak. For the identified new bugs, we found that more than half of them will cause Denial-of-Service, and quite a few of them will cause out-of-bound access, as shown in Table 4 and Table 5.

In general, automatically determining exploitability and security impact of a bug is a challenging research problem. A number of recent works [47] have investigated into this problem. If we can automatically decide the exploitability and security impact of a potential miss-check case, we can automatically confirm a missing-check bug/vulnerability and thus automatically eliminate false positives. We will leave such an analysis for future work.

8 Related Work

Missing-check detection. The most closely related works to CRIX are about missing-check detection. LRSan [43] detects lacking-recheck bugs, a subclass of missing-check bugs. CRIX detects general missing-check bugs that include lacking-recheck bugs. Juxta [23] detects semantic bugs using cross-checking between semantically equivalent implementations of file systems. Most bugs found by Juxta are missing-check bugs. CRIX can detect missing-check bugs in all subsystems in the OS kernels and do not require multiple implementations of a subsystem. Other works utilizing complementary implementation techniques to detect missing-check include Vanguard [32], Chucky [46], AutoISES [38], Rolecast [33], and MACE [24]. To the best of our knowledge, none of the tools are scalable to a system as large as the OS kernel nor have an equivalent technique to reason about the semantics and contexts of a critical variable.

Error-code propagation and handling. To detect missing-check bugs, CRIX relies on error-handling primitives to find critical variables. Techniques in error-code propagation and handling, within the Linux kernel, include EIO [12], Hector [31], and by Rubio-González et al.[30]. Similarly, APEx [17], ErrDoc [39], and EPEx [16] reason about the error-code propagation in open source SSL implementations, either automatically or via user definitions. Unlike CRIX, all the above systems target a limited range of error returning code specifications and thus have significant false negatives. Further, these techniques do not consider error-handling cases that do not return any error code. According to our study, such error-handling cases are common.

OS-kernel analysis. Given the complexity, analysis targeting the entire OS kernels is challenging. Recent advances on kernel analysis can be mainly categorized into kernel source-code analysis and static IR analysis. Smatch [5] and Coccinelle [27] find bugs in the Linux kernel. While Smatch [5] relies on syntax tree–based intra-procedural analysis to find simple bugs such as NULL-pointer dereferences. Coccinelle [27] performs code-pattern matching to find specified bugs. In comparison, CRIX leverages flow-sensitive, context-sensitive, and field-sensitive inter-procedural analyses to identify missing check bugs.

To benefit from rich analysis passes in LLVM, recently,

many tools analyze OS kernels on LLVM IR. K-Miner [11] improves the efficiency of data-flow analysis by partitioning the kernel code along separate execution paths stating from system-call entry points. Dr. Checker [21] is also a static data-flow analysis tool that identifies bugs in the drivers. While K-Miner and Dr. Checker serve as general bug detection tools, there are also some detection tools specialized for detecting a specific class of bugs in OS kernels. KINT [44] detects integer overflows using taint analysis; UniSan [20] detects information leaks caused by uninitialized data reads, also using taint analysis.

9 Conclusion

Missing-check bugs are a common cause of critical security vulnerabilities. In this paper, we have presented CRIX, a scalable and effective system for detecting missing-check bugs in OS kernels. CRIX's detection is semantic- and context-aware with an inter-procedural and context-, flow- and field-sensitive data-flow analysis engine. We realized the detection by proposing multiple new and general techniques. In particular, the two-layer type analysis can dramatically improve the precision in finding direct-call targets. The automated critical-variable inference narrows down the analysis to a very small scope, thus scaling expensive analyses to OS kernels. The peer-slice construction and constraint modeling for conditional statements enable semantic- and context-aware analysis. With these techniques, CRIX has reasonably low false-report rates and outstanding analysis performance. By applying CRIX to the Linux kernel, we found 278 new bugs and maintainers accepted 151 of our submitted patches. The evaluation results show that CRIX is scalable and effective in finding missing-check bugs in OS kernels.

10 Acknowledgment

We would like to thank our shepherd, Trent Jaeger, and the anonymous reviewers for their helpful suggestions and comments. We are also grateful to Stephen McCamant for providing valuable comments and to Linux maintainers for providing prompt feedback on patching bugs. This research was supported in part by the NSF award CNS-1815621. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Nov. 2005.

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with wit. In *Proceedings of the 29th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2008.

[3] T. Bletsch, X. Jiang, and V. Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2011.

[4] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1):16, 2017.

[5] D. Carpenter. Smatch - the source matcher, 2009. http://smatch.sourceforge.

[6] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, 2008.

[7] I. Dillig, T. Dillig, and A. Aiken. Static error detection using semantic inconsistency inference. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2007.

[8] X. Fan, Y. Sui, X. Liao, and J. Xue. Boosting the precision of virtual call integrity protection with partial pointer analysis for c++. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 329–340. ACM, 2017.

[9] R. M. Farkhani, S. Jafari, S. Arshad, W. Robertson, E. Kirda, and H. Okhravi. On the effectiveness of type-based control flow integrity. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 28–39. ACM, 2018.

[10] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-grained control-flow integrity for kernel software. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 179–194, 2016.

[11] D. Gens, S. Schmitt, L. Davi, and A.-R. Sadeghi. K-miner: Uncovering memory corruption in linux. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[12] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit. Eio: Error handling is occasionally correct. In *FAST*, volume 8, pages 1–16, 2008.

[13] B. Hardekopf and C. Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2007.

[14] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 54–61, New York, NY, USA, 2001. ACM. ISBN 1-58113-413-4.

[15] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa. Path-sensitive backward slicing. In *International Static Analysis Symposium*, pages 231–247. Springer, 2012.

[16] S. Jana, Y. J. Kang, S. Roth, and B. Ray. Automatically detecting error handling bugs using error specifications. In *USENIX Security Symposium*, pages 345–362, 2016.

[17] Y. Kang, B. Ray, and S. Jana. Apex: Automated inference of error specifications for c apis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 472–482. ACM, 2016.

[18] J. F. Kenney and E. S. Keeping. Mathematics of statistics-part one. 1954.

[19] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: Inferring the specification within. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, 2006.

[20] K. Lu, C. Song, T. Kim, and W. Lee. UniSan: Proactive Kernel Memory Initialization to Eliminate Data Leakages. In *Proceedings of the 23rd*

ACM Conference on Computer and Communications Security (CCS), Vienna, Austria, Oct. 2016.

[21] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna. DR. CHECKER: A soundy analysis for linux kernel drivers. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017.

[22] A. Milanova, A. Rountev, and B. G. Ryder. Precise call graphs for c programs with function pointers. *Automated Software Engg.*, 11(1): 7–26, Jan. 2004. ISSN 0928-8910.

[23] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[24] M. Monshizadeh, P. Naldurg, and V. Venkatakrishnan. Mace: Detecting privilege escalation vulnerabilities in web applications. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, Scottsdale, Arizona, Nov. 2014.

[25] B. Niu and G. Tan. Modular control-flow integrity. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Edinburgh, UK, June 2014.

[26] NVD. National vulnerability database, 2019. https://nvd.nist.gov.

[27] Y. Padioleau, J. L. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in linux device drivers. In *EuroSys*, 2008.

[28] A. Quach, A. Prakash, and L. K. Yan. Debloating software through piece-wise compilation and loading. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018.

[29] D. A. Ramos and D. Engler. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[30] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau. Error propagation analysis for file systems. In *ACM Sigplan Notices*, volume 44, pages 270–280. ACM, 2009.

[31] S. Saha, J.-P. Lozi, G. Thomas, J. L. Lawall, and G. Muller. Hector: Detecting resource-release omission faults in error-handling code for systems software. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2013.

[32] L. Situ, L. Wang, Y. Liu, B. Mao, and X. Li. Vanguard: Detecting missing checks for prognosing potential vulnerabilities. In *Proceedings of the Tenth Asia-Pacific Symposium on Internetware*, page 5. ACM, 2018.

[33] S. Son, K. S. McKinley, and V. Shmatikov. Rolecast: finding missing security checks when you do not know what checks are. In *ACM SIGPLAN Notices*, volume 46, pages 1069–1084. ACM, 2011.

[34] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, New York, NY, USA, 1996. ACM. ISBN 0-89791-769-3.

[35] W. R. Stevens and S. A. Rago. *Advanced programming in the UNIX environment*. Addison-Wesley, 2008.

[36] Y. Sui and J. Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 265–266. ACM, 2016.

[37] Y. Sui and J. Xue. Value-flow-based demand-driven pointer analysis for c and c++. *IEEE Transactions on Software Engineering*, 2018.

[38] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou. Autoises: Automatically inferring security specification and detecting violations. In *USENIX Security Symposium*, pages 379–394, 2008.

[39] Y. Tian and B. Ray. Automatically diagnosing and repairing error handling bugs in c. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 752–762. ACM, 2017.

[40] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *USENIX Security Symposium*, pages 941–955, 2014.

[41] O. Tripp, S. Guarnieri, M. Pistoia, and A. Aravkin. Aletheia: Improving the usability of static security analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 762–774. ACM, 2014.

[42] V. van der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 934–953. IEEE, 2016.

[43] W. Wang, K. Lu, and P. Yew. Check It Again: Detecting Lacking-Recheck Bugs in OS Kernels. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018.

[44] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Improving Integer Security for Systems with KINT. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.

[45] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 359–368. IEEE, 2009.

[46] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 499–510. ACM, 2013.

[47] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2139–2154. ACM, 2017.

[48] S. Özkan. Common vulnerabilities and exposures details, 2019. https://www.cvedetails.com.

A Appendix

Data fetch functions	
copy_from_user	_copy_from_user
__copy_from_user	raw_copy_from_user
strncpy_from_user	_strncpy_from_user
__strncpy_from_user	strndup_user
__copy_from_user_inatomic	memdup_user
__copy_from_user_inatomic_nocache	copyin
__constant_copy_from_user	memdup_user_nul
rds_message_copy_from_user	__get_user
snd_trident_synth_copy_from_user	vmemdup_user
ivtv_buf_copy_from_user	copyin_str
iov_iter_copy_from_user_atomic	fusword
__generic_copy_from_user	copyin_nofault
__copy_from_user_eva	fuword
__arch_copy_from_user	fubyte
__copy_from_user_flushcache	fuswintr
__asm_copy_from_user	get_user
copy_from_user_toio	copy_from_user_page
copy_from_user_nmi	copy_from_user_proc

Table 3: List of input functions collected based on heuristics.

Subsystem	Filename	Line#	Impact	Category	Status	LP
net	gvf.c	511	reliability	P	C	2
x86	ghv_init.c	107	DoS	S	A	1
x86	gtlb_uv.c	2013	DoS	S	S	8
char	hpet.c	978	reliability	S	S	4
firmware	gdriver.c	711	DoS	S	C	1
gpio	gpio-exar.c	150	reliability	U	A	2
gpu	gkfd_crat.c	404	DoS	U	S	1
gpu	gi915_gpu_error.c	230	reliability	S	S	2
gpu	gradeon_display.c	679	reliability	S	C	2
gpu	gvkms_crtc.c	227	reliability	U	A	<1
hid	hid-logitech-hidpp.c	1954	reliability	S	A	3
iio	gmax9611.c	531	DoS	U	S	2
iio	gmxs-lradc-adc.c	466	DoS	U	A	2
iio	ghmc5843_i2c.c	62	reliability	S	A	4
iio	ghmc5843_spi.c	62	reliability	S	A	4
infiniband	gcm.c	1921	DoS	S	C	6
infiniband	gi40iw_cm.c	3257	DoS	S	A	2
infiniband	gi40iw_cm.c	3260	DoS	S	A	2
input	gpm8xxx-vibrator.c	198	DoS	P	S	2
isdn	ghfcpci.c	2034	reliability	S	A	10
isdn	ghfcsusb.c	265	DoS	S	A	10
isdn	gmISDNinfineon.c	716	DoS	S	A	9
leds	leds-pca9532.c	531	crash /DoS	S	A	2
media	gstv090x.c	1449	reliability	S	S	10
media	gstv090x.c	1452	reliability	S	S	10
media	gstv090x.c	1456	reliability	S	S	10
media	gstv090x.c	2229	reliability	S	S	5
media	gstv090x.c	2607	reliability	S	S	10
media	gstv090x.c	2913	reliability	S	S	10
media	gstv090x.c	2957	reliability	S	S	10
media	gstv090x.c	2975	reliability	S	S	10
media	gvpss.c	520	DoS	S	A	6
media	grcar-core.c	267	DoS	S	S	1
media	grenesas-ceu.c	1684	DoS	S	S	1
media	grga.c	894	memory leak	S	A	1
media	grga.c	896	memory leak	S	A	1
media	grga.c	910	reliability	S	A	1
media	grga.c	875	reliability	S	A	2
media	grga.c	915	use-after-free	S	A	1
media	gvideo-mux.c	400	DoS	U	A	1
media	gvideo-mux.c	402	DoS	S	A	1
media	gusbvision-core.c	2301	reliability	S	S	9
memstick	gms_block.c	2141	DoS	U	C	5
mfd	sm501.c	1145	DoS	S	A	1
mmc	gmmc_spi.c	821	concurrency	U	A	9
net	gmcp251x.c	963	reliability	S	S	3
net	glan9303-core.c	1081	system crash	S	S	1
net	glan9303-core.c	1074	system crash	S	S	<1
net	gpcnet_cs.c	1424	DoS	S	A	8
net	gpcnet_cs.c	290	DoS	S	A	8
net	glio_main.c	1194	DoS	S	A	2
net	glio_vf_main.c	1961	DoS	S	S	2
net	glio_vf_main.c	612	DoS	S	S	2
net	glio_core.c	1213	DoS	S	A	1
net	glio_core.c	1685	DoS	S	A	3
net	gnicvf_main.c	2264	DoS	S	A	1
net	gfmvj18x_cs.c	549	DoS	S	A	8
net	gfm10k_main.c	42	reliability	A	A	2
net	gen_rx.c	721	DoS	U	S	<1
net	gocelot_board.c	256	DoS	U	C	1
net	gqla3xxx.c	3888	system crash	U	A	12
net	gqlge_main.c	4682	system crash	S	A	2
net	gsh_eth.c	3133	reliability	U	A	5
net	gravb_main.c	1996	reliability	U	A	3
net	grocker_main.c	2799	DoS	S	A	1
net	gdwmac-dwc-qos-eth.c	487	DoS	S	A	2
net	gdwmac-sun8i.c	1150	system crash	S	A	2
net	gfjes_main.c	1254	concurrency	U	S	3
net	gfjes_main.c	1255	concurrency	S	S	3
net	gnetvsc_drv.c	1377	DoS	S	A	<1
net	gadf7242.c	1269	DoS	S	A	<1
net	gcore.c	645	reliability	S	A	2
net	gcore.c	646	reliability	S	A	2
net	gcore.c	653	reliability	S	A	2
net	gcore.c	662	reliability	S	A	1
net	gcore.c	689	reliability	S	A	2
net	gcore.c	714	reliability	S	A	2
net	gcfg80211.c	5368	DoS	S	A	6
net	gcfg80211.c	5384	DoS	S	A	6
net	g3945-mac.c	3405	reliability	U	S	7
net	g4965-mac.c	6241	reliability	U	S	7
net	gcmdevt.c	342	DoS	U	A	7
net	gray_cs.c	395	system crash	U	S	8
net	gray_cs.c	409	system crash	S	S	8
net	gray_cs.c	423	system crash	S	S	8
net	gbase.c	471	system crash	S	S	4
net	gfw_common.c	648	DoS	S	A	8
net	gfw.c	600	DoS	U	A	8
net	gfw_common.c	623	DoS	U	A	8
net	gfw.c	744	DoS	U	A	8
net	gfw.c	448	DoS	U	A	8
net	gfw.c	562	DoS	S	A	8
net	gfw.c	1623	DoS	U	A	8
net	gfw.c	1759	DoS	U	A	8
staging	gfw.c	745	DoS	U	A	8
net	gcmdevt.c	342	DoS	U	A	8
net	gqlcnic_ethtool.c	1050	DoS	U	A	8
net	grsi_91x_mac80211.c	199	DoS	S	A	5
net	grsi_91x_mac80211.c	208	DoS	S	A	5
net	gmain.c	347	DoS	S	A	6
nfc	gse.c	345	DoS	S	S	4
nvdimm	btt_devs.c	200	DoS	S	C	3
nvdimm	btt_devs.c	217	system crash	S	C	3
nvdimm	namespace_devs.c	2250	DoS	S	A	2
pci	gpci-tegra.c	1552	buffer overflow	S	S	1
pci	gpcie-rcar.c	931	buffer overflow	S	A	5
pci	gpcie-xilinx.c	343	buffer overflow	S	C	4
pci	gpci-epf-test.c	571	DoS	U	A	2
pinctrl	gpinctrl-baytrail.c	1711	DoS	U	A	3
pinctrl	pinctrl-axp209.c	366	DoS	S	A	<1
power	gcharger-manager.c	2006	DoS	U	A	7
rapidio	rio_cm.c	2147	DoS	S	A	2
scsi	gcxgb4i.c	619	DoS	S	S	8
scsi	gql4_os.c	3206	DoS	S	A	7
scsi	gufs-hisi.c	546	DoS	U	A	<1
spi	spi-s3c64xx.c	294	DoS	U	S	5
spi	spi-topcliff-pch.c	1304	DoS	S	A	8
spi	spi-topcliff-pch.c	1307	DoS	S	A	8
staging	gaudio_manager.c	47	system crash	P	A	3
staging	grtw_xmit.c	1514	DoS	S	A	4
staging	grtl_phydm.c	182	system crash	S	A	1
thunderbolt	property.c	177	DoS	S	A	1
thunderbolt	property.c	550	DoS	S	A	1
tty	gmain.c	115	DoS	S	A	8
tty	gmain.c	135	DoS	U	A	8
tty	g8250_lpss.c	175	DoS	U	C	2
tty	gatmel_serial.c	1285	DoS	U	A	5
tty	gmxs-auart.c	1688	DoS	S	S	8
usb	gu132-hcd.c	3203	DoS	U	C	11
usb	galauda.c	438	DoS	U	S	13
usb	galauda.c	439	DoS	U	S	13
video	ghgafb.c	287	DoS	S	A	14
video	gimsttfb.c	1517	DoS	S	A	13
video	gomapdss-boot-init.c	113	DoS	U	A	3
affs	file.c	940	DoS	S	C	14
btrfs	extent-tree.c	7042	reliability	S	A	2
ipv6	gip6t_srh.c	212	DoS	S	A	1
ipv6	gip6t_srh.c	225	DoS	S	A	1
ipv6	gip6t_srh.c	235	DoS	S	A	1
openvswitch	datapath.c	449	DoS	U	A	7
smc	smc_ism.c	290	system crash	S	S	<1
strparser	strparser.c	552	DoS	S	A	2

Table 4: List of new bugs (1-142) detected with CRIX. LP = Latent Period of bugs in years. Column `Category` specifies the category of peer-slice set used to identify the bugs. A, P, S, and U indicate categories `Source-Arg`, `Source-Param`, `Source-Ret`, and `Use-Param` respectively. The `S`,`C`,`A` in the `Status` field represent patch status, `Submitted`, `Confirmed`, `Applied`, respectively.

Subsystem	Filename	Line#	Impact	Category	Status	LP
security	inode.c	339	reliability	S	A	5
ceph	osdmap.c	1900	DoS	S	S	7
isa	gsb8.c	113	reliability	U	A	14
pci	gechoaudio.c	1956	DoS	U	A	12
soc	gcs43130.c	2324	DoS	S	A	1
soc	grt5645.c	3452	system crash	U	A	<1
soc	soc-pcm.c	1236	system crash	S	S	4
md	raid10.c	3958	system crash	S	A	7
md	raid5.c	7399	system crash	S	A	7
usb	gusb_stream.c	106	DoS	S	A	10
usb	gusb_stream.c	107	DoS	S	A	10
ata	sata_dwc_460ex.c	1055	DoS	U	S	4
block	nbd.c	2117	DoS	U	S	2
net	gbcmmii.c	217	DoS	U	S	<1
slimbus	qcom-ngd-ctrl.c	1351	reliability	U	A	<1
ncsi	ncsi-netlink.c	253	reliability	U	A	1
ncsi	ncsi-netlink.c	257	DoS	U	A	1
openvswitch	conntrack.c	2146	DoS	U	S	1
openvswitch	datapath.c	466	DoS	U	A	4
openvswitch	datapath.c	475	DoS	U	A	4
openvswitch	datapath.c	477	reliability	U	A	4
tipc	group.c	942	DoS	U	A	<1
tipc	group.c	946	system crash	U	A	<1
tipc	socket.c	3226	DoS	U	A	4
tipc	socket.c	3231	reliability	U	A	4
extcon	extcon-axp288.c	145	reliability	S	A	4
thunderbolt	switch.c	1325	DoS	S	S	2
thunderbolt	xdomain.c	540	DoS	S	A	1
usb	gusb251xb.c	600	DoS	U	A	2
tty	gmax310x.c	1421	DoS	U	A	5
tty	gmvebu-uart.c	791	DoS	S	S	1
mtd	gvf610_nfc.c	856	DoS	S	A	3
mfd	mc13xxx-i2c.c	82	DoS	U	S	6
pinctrl	gberlin-bg4ct.c	453	DoS	U	S	3
pinctrl	gpinctrl-as370.c	334	DoS	U	S	<1
mfd	mc13xxx-spi.c	160	DoS	S	S	6
firmware	gdriver.c	801	DoS	S	A	2
net	gtls.c	227	DoS	U	A	<1
mmc	gdw_mmc-exynos.c	556	DoS	U	S	6
mmc	gdw_mmc-k3.c	461	DoS	S	S	5
mmc	gdw_mmc-pltfm.c	84	DoS	S	S	5
pci	gpci-host-generic.c	85	DoS	U	S	3
scsi	gtc-dwc-g210-pltfrm.c	63	DoS	U	S	3
soc	gsirf-audio-codec.c	466	system crash	S	A	5
slimbus	qcom-ngd-ctrl.c	1333	DoS	S	S	<1
x86	ghpet.c	79	DoS	U	A	11
udf	super.c	575	system crash	S	S	1
nfc	llcp_sock.c	726	DoS	S	A	7
scsi	gufshcd.c	1759	DoS	S	S	<1
thunderbolt	xdomain.c	771	DoS	S	A	1
scsi	gufshcd.c	1786	DoS	S	S	1
thunderbolt	icm.c	475	DoS	U	A	1
fmc	fmc-fakedev.c	283	DoS	S	S	5
usb	gsierra_ms.c	197	system crash	S	A	2
staging	gvchiq_2835_arm.c	212	DoS	S	C	4
thunderbolt	property.c	581	DoS	S	A	3
thunderbolt	property.c	582	buffer overflow	U	A	1
x86	gtlb_uv.c	2144	DoS	S	A	2
x86	gtlb_uv.c	2147	DoS	S	A	4
nfc	gse.c	329	DoS	U	S	1
gpio	gpio-aspeed.c	1227	DoS	S	A	2
soc	grt5663.c	3472	buffer overflow	S	C	2
soc	grt5663.c	3513	DoS	U	C	1
gpu	gv3d_drv.c	103	system crash	S	A	3
net	gmcr20a.c	534	system crash	S	A	5
net	gmcr20a.c	541	reliability	S	S	3
net	gmcr20a.c	546	reliability	S	S	3
media	gtda18250.c	705	reliability	S	C	2
soc	gcs35l34.c	263	reliability	S	S	7
dma	gomap-dma.c	1056	system crash	A	S	6
firmware	edd.c	279	system crash	A	S	4
net	gcfg80211.c	2302	system crash	A	S	4
rtc	rtc-ds1374.c	449	reliability	S	S	2
rtc	rtc-rx8010.c	193	system crash	S	S	2
mfd	tps65010.c	431	DoS	U	S	2
net	grx.c	732	DoS	U	C	<1
net	grx.c	733	DoS	U	S	3
usb	grealtek_cr.c	815	reliability	S	S	5
net	glag_conf.c	307	DoS	U	S	6
net	gmesh.c	799	system crash	S	S	2
net	gmesh.c	800	system crash	S	S	2
net	lag_conf.c	307	DoS	U	S	<1
net	p2p.c	1527	concurrency	S	S	6
message	mptctl.c	406	concurrency	S	S	10
message	mptscsih.c	1617	concurrency	S	S	10
message	mptsas.c	4803	concurrency	S	S	10
misc	tifm_7xx1.c	280	concurrency	S	S	4
pci	pcie-designware-host.c	309	DoS	U	S	1
gpu	virtgpu_kms.c	62	DoS	U	S	4
gpu	virtgpu_vq.c	48	DoS	U	S	6
input	usbtouchscreen.c	1076	DoS	U	S	8
usb	iuu_phoenix.c	369	DoS	U	S	12
usb	iuu_phoenix.c	177	DoS	U	S	12
usb	iuu_phoenix.c	729	DoS	U	S	12
usb	iuu_phoenix.c	389	DoS	U	S	12
usb	iuu_phoenix.c	253	DoS	U	S	12
usb	kobil_sct.c	248	DoS	U	S	4
usb	kobil_sct.c	339	DoS	U	S	4
usb	kobil_sct.c	354	DoS	U	S	4
usb	kobil_sct.c	284	DoS	U	S	3
ncsi	ncsi-netlink.c	250	DoS	U	S	3
openvswitch	conntrack.c	2131	reliability	S	S	<1
media	cx231xx-input.c	91	DoS	U	S	4
net	testmode.c	242	DoS	U	S	8
dma	fsl-edma-common.c	540	reliability	S	S	<1
dma	coh901318_lli.c	41	reliability	S	S	10
mtd	generic.c	69	reliability	S	S	3
net	e1000_hw.c	1046	buffer overflow	S	S	5
mfd	vx855.c	104	reliability	S	S	8
mfd	ab3100-core.c	926	reliability	S	S	8
crypto	cryptd.c	745	reliability	S	S	1
hwmon	ad7418.c	90	buffer overflow	S	S	3
hwmon	lm92.c	135	buffer overflow	S	S	3
scsi	gdth.c	5203	buffer overflow	S	S	4
staging	mmal-vchiq.c	1847	DoS	U	S	1
fsi	fsi-core.c	1250	DoS	U	S	2
net	cxgb3_offload.c	1268	DoS	U	S	7
iio	mxs-lradc-adc.c	470	DoS	U	S	1
net	myri10ge.c	2287	reliability	S	S	11
gpu	si.c	3614	reliability	S	S	6
slimbus	qcom-ngd-ctrl.c	1343	DoS	U	S	<1
net	e1000_hw.c	141	reliability	S	S	10
net	e1000_hw.c	1043	reliability	S	S	10
mtd	bcm63xxpart.c	65	buffer overflow	S	S	3
gpu	vc4_plane.c	1011	reliability	S	S	1
ext4	super.c	5866	reliability	S	S	8
net	event.c	105	buffer overflow	S	S	3
net	pch_gbe_main.c	1476	DoS	U	S	8
net	isl_ioctl.c	190	reliability	S	S	13
gpu	ast_mode.c	1201	reliability	S	S	7
hid	wacom_sys.c	2351	reliability	S	S	5
media	ov9650.c	609	buffer overflow	S	S	<1
soc	sti_uniperif.c	292	reliability	S	S	2
media	em28xx-cards.c	3987	reliability	S	S	2
usb	xhci-pci.c	269	reliability	S	S	1
net	nic_main.c	1229	crash	S	S	3

Table 5: Continued list of new bugs (143-278) detected with CRIX. LP = Latent Period of bugs in years. Column `Category` specifies the category of peer-slice set used to identify the bugs. A, P, S, and U indicate categories `Source-Arg`, `Source-Param`, `Source-Ret`, and `Use-Param` respectively. The `S,C,A` in the `Status` field represent patch status, `Submitted`, `Confirmed`, `Applied`, respectively.

DEEPVSA: Facilitating Value-set Analysis with Deep Learning for Postmortem Program Analysis

†*Wenbo Guo*[*], †*Dongliang Mu*[*], †*Xinyu Xing*, ‡*Min Du*, ‡*Dawn Song*
†*College of IST, Pennsylvania State University*
‡*Department of EECS, University of California, Berkeley*

Abstract

Value set analysis (VSA) is one of the most powerful binary analysis tools, which has been broadly adopted in many use cases, ranging from verifying software properties (*e.g.,* variable range analysis) to identifying software vulnerabilities (*e.g.,* buffer overflow detection). Using it to facilitate data flow analysis in the context of postmortem program analysis, it however exhibits an insufficient capability in handling memory alias identification. Technically speaking, this is due to the fact that VSA needs to infer memory reference based on the context of a control flow, but accidental termination of a running program left behind incomplete control flow information, making memory alias analysis clueless.

To address this issue, we propose a new technical approach. At the high level, this approach first employs a layer of instruction embedding along with a bi-directional sequence-to-sequence neural network to learn the machine code pattern pertaining to memory region accesses. Then, it utilizes the network to infer the memory region that VSA fails to recognize. Since the memory references to different regions naturally indicate the non-alias relationship, the proposed neural architecture can facilitate the ability of VSA to perform better alias analysis. Different from previous research that utilizes deep learning for other binary analysis tasks, the neural network proposed in this work is fundamentally novel. Instead of simply using off-the-shelf neural networks, we introduce a new neural network architecture which could capture the data dependency between and within instructions.

In this work, we implement our deep neural architecture as `DEEPVSA`, a neural network assisted alias analysis tool. To demonstrate the utility of this tool, we use it to analyze software crashes corresponding to 40 memory corruption vulnerabilities archived in Offensive Security Exploit Database. We show that, `DEEPVSA` can significantly improve VSA with respect to its capability in analyzing memory alias and thus escalate the ability of security analysts to pinpoint the root cause of software crashes. In addition, we demonstrate that our proposed neural network outperforms state-of-the-art neural architectures broadly adopted in other binary analysis tasks. Last but not least, we show that `DEEPVSA` exhibits nearly no false positives when performing alias analysis.

[*]Equal Contribution.

1 Introduction

Despite the best efforts of developers, software inevitably contains flaws that may be leveraged as security vulnerabilities. Modern operating systems integrate various security mechanisms to prevent software faults from being exploited [18, 36, 51, 53]. To bypass these defenses and hijack program execution, an attacker therefore needs to constantly mutate an exploit and make many attempts. While in their attempts, the exploit triggers a security vulnerability and makes the running process terminate abnormally.

To analyze the unexpected termination (*i.e.,* program crash) and thus pinpoint the root cause, software developers or security analysts need to perform backward taint analysis [17, 20, 39], track down how a bad value is passed to the crashing site and thus pinpoint the statements that led to the crash. Technically speaking, this process can be significantly facilitated – and even automated – if the control and data flows pertaining to the crash are available upon its termination.

Recently, a large amount of research has demonstrated that program execution can be recorded through hardware tracing (*e.g.,* [30, 55]) in a least intrusive manner. As a result, a software developer can easily restore the control flow pertaining to a program crash. However, the recovery of data flow from the execution trace alone is still challenging, especially when source code is not available. As it has been discussed in recent research [55], this is primarily because data flow construction is highly dependent upon the capability of memory alias analysis [4, 5].

Of all the memory alias analysis techniques proposed in past research, value-set analysis (VSA) is the most *effective* and *efficient* technique and has been broadly adopted to facilitate the ability of identifying memory alias at the binary

level [6]. Applied in the context of postmortem program analysis, it however exhibits an insufficient capability in handling memory alias identification. Technically speaking, this is mainly because VSA needs to infer memory references based on the context of a control flow. However, accidental termination of a running program only leaves behind incomplete control flow information, making memory alias analysis clueless.

To address this technical issue, we introduce a deep neural network to enhance the capability of VSA in memory alias analysis, especially in the context of software failure diagnosis. More specifically, we use this neural network to learn the memory regions that each memory access refers to. The rationale behind this approach is as follows. VSA divides the address space of a process into several non-overlapping regions (*i.e.,* stack, heap, and global) and deem pairs of memory references to different regions as non-alias. With incomplete control flow information pertaining to a software crash, VSA loses the execution context of a crashing program and typically exhibits bad performance in assigning memory references to different memory regions. Using deep learning, we can learn complex execution patterns pertaining to memory region accesses, restore the memory regions that VSA fails to infer through incomplete control flow, and finally enhance the capability of alias analysis for postmortem program analysis.

Different from previous research that utilizes deep learning to tackle other binary analysis problems (*e.g.,* [15, 48, 49, 56]), the deep neural network used in this work is novel. Instead of simply applying an off-the-shelf neural architecture to our problem domain, we propose a new neural network architecture. To be specific, our proposed solution first utilizes an instruction embedding network to capture the semantic of each instruction. Then, it employs a bi-directional sequence-to-sequence neural architecture to learn the dependency between the instructions and predict the memory access for each individual instruction. With this new design practice, we could capture the dependency relationship within and between instructions and thus accurately predict the memory regions that each instruction attempts to access. As we will discuss and demonstrate in Section 3 and 4, this perfectly reflects the characteristic of binary code analysis and significantly benefits alias analysis in the context of software failure diagnosis.

We implemented our proposed technique as DEEPVSA [1], a neural network-assisted alias analysis tool for postmortem program analysis. To the best of our knowledge, DEEPVSA is the first tool that takes advantage of deep learning to improve alias analysis in the context of postmortem program analysis. We manually analyzed program crashes corresponding to 40 memory corruption vulnerabilities gathered from the Offensive Security Exploit Database Archive [47] and compared our manual analysis with the analysis conducted by DEEPVSA.

[1]The code, data and models of DEEPVSA are available at https://github.com/Henrygwb/deepvsa/.

```
1     sub     esp, 0x14
2     call    malloc
      ......
3     ret
4     mov     [eax], test
5     mov     [esp+0x8], eax
-------------------------------------------------------
6     push    eax
7     call    child
8     push    ebp
9     mov     ebp, esp
10    mov     [0xC8], 0x0
11    mov     eax, [ebp+0x8]
12    mov     [eax], 0x1
13    mov     [eax+0x4], 0x2
14    mov     eax, 0
15    pop     ebp
16    ret
17    mov     eax, [esp+0xC]
18    call    [eax]          <--- crash site
```

Figure 1: An example instruction trace prior to a program crash.

We observed that DEEPVSA can accurately resolve approximately 35% of unknown memory relationships that VSA fails to identify when performing analysis on a crashing execution. In addition, we discovered that the escalation in alias analysis significantly improves the capability in tracking down the root cause of software crashes. For about 75% failure cases, DEEPVSA is capable of assisting backward taint analysis in identifying the root causes of their crashes. Compared with the broadly adopted neural networks in other binary analysis tasks, we also demonstrate that our new neural network architecture introduces no false positives in memory alias identification.

In summary, this paper makes the following contributions:

- We discover that deep neural networks are a viable approach towards addressing alias analysis issues in the context of software failure diagnosis.
- We propose a new neural network architecture which could be used to improve alias analysis for VSA and thus escalate the ability to diagnose the root cause of software crashes.
- We implement our deep learning technique as DEEPVSA– a tool for alias analysis facilitation – and demonstrate its effectiveness by using 40 distinct software crashes covering approximately 1.6 million lines of execution trace in total.

The rest of the paper is organized as follows. Section 2 provides an overview of value-set analysis and its limitations in postmortem program analysis. Section 3 presents the deep neural network we propose to improve alias analysis. Section 4 describes our implementation and evaluation, demonstrating the utility of DEEPVSA. Section 5 surveys related work. Finally, we conclude this work in Section 6.

	[eax]@4	...	[0xC8]@10	[ebp+0x8]@11	[eax]@12	[eax+4]@13	[esp+0xC]@17	[eax]@18
[eax]@4	-	...	0	0	1	0	0	1
...	...	...	...	...	...	...	...	...
[0xC8]@10	NA	...	-	...	0	0	0	0
[ebp+0x8]@11	NA	...	0	-	0	0	0	0
[eax]@12	NA	...	?	?	-	0	0	1
[eax+0x4]@13	NA	...	?	?	?	-	0	0
[esp+0xC]@17	NA	...	0	0	?	?	-	0
[eax]@18	NA	...	?	?	?	?	?	-

(a) Alias matrix identified by VSA. '0', '1' and '?' represent non-alias, alias and may-alias relationships respectively.

Line #	Complete Trace		Incomplete Trace without DL		Incomplete Trace with DL	
	A-loc	Value-set	*A-loc*	Value-set	*A-loc*	Value-set
1	esp	(⊥, [-0x14, -0x14], ⊥)	NA	NA	NA	NA
4	[eax] (⊥, ⊥, [0, 0])	(test, ⊥, ⊥)	NA	NA	NA	NA
5	[esp+0x8] (⊥, [-0xC, -0xC], ⊥)	(⊥, ⊥, [0, 0])	NA	NA	NA	NA
6	esp	(⊥, [-0x18, -0x18], ⊥)	esp	(⊥, [-0x4, -0x4], ⊥)	esp	(⊥, [-0x4, -0x4], ⊥)
	[esp] (⊥, [-0x18, -0x18], ⊥)	(⊥, ⊥, [0, 0])	[esp] (⊥, [-0x4, -0x4], ⊥)	(⊤, ⊤, ⊤)	[esp] (⊥, [-0x4, -0x4], ⊥)	(⊥, ⊥, [X, X])
7	esp	(⊥, [-0x1C, -0x1C], ⊥)	esp	(⊥, [-0x8, -0x8], ⊥)	esp	(⊥, [-0x8, -0x8], ⊥)
	[esp] (⊥, [-0x1C, -0x1C], ⊥)	([L17, L17], ⊥, ⊥)	[esp] (⊥, [-0x8, -0x8], ⊥)	([L17, L17], ⊥, ⊥)	[esp] (⊥, [-0x8, -0x8], ⊥)	([L17, L17], ⊥, ⊥)
8	esp	(⊥, [-0x20, -0x20], ⊥)	esp	(⊥, [-0xC, -0xC], ⊥)	esp	(⊥, [-0xC, -0xC], ⊥)
	[esp] (⊥, [-0x20, -0x20], ⊥)	(⊤, ⊤, ⊤)	[esp] (⊥, [-0xC, -0xC], ⊥)	(⊤, ⊤, ⊤)	[esp] (⊥, [-0xC, -0xC], ⊥)	(⊤, ⊤, ⊤)
9	ebp	(⊥, [-0x20, -0x20], ⊥)	ebp	(⊥, [-0xC, -0xC], ⊥)	ebp	(⊥, [-0xC, -0xC], ⊥)
10	[0xC8] ([0xC8, 0xC8], ⊥, ⊥)	([0x0, 0x0], ⊥, ⊥)	[0xC8] ([0xC8, 0xC8], ⊥, ⊥)	([0x0, 0x0], ⊥, ⊥)	[0xC8] ([0xC8, 0xC8], ⊥, ⊥)	([0x0, 0x0], ⊥, ⊥)
11	[ebp+0x8] (⊥, [-0x18, -0x18], ⊥)	(⊥, ⊥, [0, 0])	[ebp+0x8] (⊥, [-0x4, -0x4], ⊥)	(⊤, ⊤, ⊤)	[ebp+0x8] (⊥, [-0x4, -0x4], ⊥)	(⊥, ⊥, [X, X])
	eax	(⊥, ⊥, [0, 0])	eax	(⊤, ⊤, ⊤)	eax	(⊥, ⊥, [X, X])
12	[eax] (⊥, ⊥, [0, 0])	([0x1, 0x1], ⊥, ⊥)	[eax] (⊤, ⊤, ⊤)	([0x1, 0x1], ⊥, ⊥)	[eax] (⊥, ⊥, [X, X])	([0x1, 0x1], ⊥, ⊥)
13	[eax+4] (⊥, ⊥, [4, 4])	([0x2, 0x2], ⊥, ⊥)	[eax+4] (⊤, ⊤, ⊤)	([0x2, 0x2], ⊥, ⊥)	[eax+4] (⊥, ⊥, [X+0x4, X+0x4])	([0x2, 0x2], ⊥, ⊥)
14	eax	([0x0, 0x0], ⊥, ⊥)	eax	([0x0, 0x0], ⊥, ⊥)	eax	([0x0, 0x0], ⊥, ⊥)
15	ebp	(⊤, ⊤, ⊤)	ebp	(⊤, ⊤, ⊤)	ebp	(⊤, ⊤, ⊤)
	esp	(⊥, [-0x1C, -0x1C],⊥)	esp	(⊥,[-0x8,-0x8],⊥)	esp	(⊥,[-0x8,-0x8],⊥)
16	esp	(⊥, [-0x18, -0x18], ⊥)	esp	(⊥, [-0x4, -0x4], ⊥)	esp	(⊥, [-0x4, -0x4], ⊥)
17	[esp+0xC] (⊥, [-0xC, -0xC], ⊥)	(⊥, ⊥, [0, 0])	[esp+0xC] (⊥, [0x8, 0x8], ⊥)	(⊤, ⊤, ⊤)	[esp+0xC] (⊥, [0x8, 0x8], ⊥)	(⊥, ⊥, [X, X])
	eax	(⊥, ⊥, [0, 0])	eax	(⊤, ⊤, ⊤)	eax	(⊥, ⊥, [X, X])
18	[eax] (⊥, ⊥, [0, 0])	([0x1, 0x1], ⊥, ⊥)	[eax] (⊤, ⊤, ⊤)	(⊤, ⊤, ⊤)	[eax] (⊥, ⊥, [X, X])	([0x1, 0x1], ⊥, ⊥)

(b) *A-locs* and value-sets corresponding to complete and incomplete traces with and without the facilitation of deep learning (DL).

Table 1: The results of value-set analysis against the instruction trace shown in Figure 1.

2 Background and Problem Scope

As is described and discussed in many recent research works (e.g. [19, 55]), new hardware components could trace program execution in a least intrusive fashion. With this capability, security analysts could easily obtain the control flow pertaining to a software crash. Using the execution trace, it is however still challenging to pinpoint the root cause of the crash (i.e., the instructions truly attributive to the crash). On the one hand, this is because a security analyst barely has the access to the source code of the crashing program. On the other hand, this is because a security analyst needs to analyze the data flow of the crashing trace which involves memory alias analysis at the binary level. To tackle this challenge, value-set analysis (VSA) can be adopted. In this section, we first introduce how software instrumentation and hardware tracing are used to record program execution. Second, we briefly describe how to perform value-set analysis on a recorded execution trace. Third, we specify how to use the derived value set to perform alias analysis and thus diagnose the root cause of a software crash. Finally, we provide a more in-depth discussion about why VSA behaves poorly in many real-world applications.

2.1 Program Tracing for Software Debugging

Software instrumentation techniques have long been used to fully record program execution and thus facilitate the root cause diagnosis for a crashing program (e.g., [38, 37]). However, such an approach imposes significant overhead to a software normal operation. In order to minimize additional overhead, some lightweight instrumentation techniques have been proposed (e.g., [41, 40]). While they are less intrusive and informative for assisting software debugging, such a lightweight approach cannot be used to fully restore the

control flow pertaining to a software crash.

Recently, the advance in hardware-assisted processor tracing significantly ameliorates this situation. With the emergence of brand new hardware components, such as Intel PT [27] and ARM ETM [2], software developers and security analysts can trace instructions executed with nearly no overhead and save them in a circular buffer. At the time of a program crash, an operating system includes the trace into a crash dump. Since this post-crash artifact contains both the state of crashing memory and the execution history (*i.e.,* the last *N* instructions executed prior to the crash), software developers not only can inspect the program state at the time of the crash, but also fully reconstruct the control flow that led to the crash.

In this work, we focus on using an enhanced value-set analysis technique to analyze such an aforementioned post-crash artifact and thus facilitate the root cause diagnosis of a crashing program. It should be noted that the aforementioned lightweight software instrumentation approach is out of the scope of this research because they cannot provide a complete instruction trace for value-set analysis to identify memory alias and thus pinpoint the root cause of the crash.

2.2 Value-set Analysis

Value-set analysis is an algorithm designed for analyzing assembly code or an instruction trace in a static fashion. Based on the observation that memory layout generally follows, VSA partitions memory into 3 disjoint memory regions – global[2], stack and heap – and assigns instructions to the regions, accordingly. For some instructions, VSA achieves region assignment by examining the semantics of the instructions. For example, from a binary code perspective, accesses to global and stack variables appear as `[absolute-address]` and `[esp-offset]`. Thus, VSA can easily link the global and stack regions to the instructions `mov edx,[0x8050684]` and `lea eax,[esp+4]`, respectively. For other instructions, VSA performs a simple forward data flow analysis to determine the regions tied to instructions in a conservative fashion[3]. Take for example the instruction trace shown in Figure 1. The instruction at `line 4` indicates a write to the target memory `[eax]`. Through a forward data flow analysis, VSA could easily pinpoint that the value of `eax` was passed through `line 3` because the library function `malloc` places its return value in the register `eax`. Given that the semantics of `malloc` is to allocate a memory region on the heap and then return its reference to the caller function, VSA could easily assign the heap region to the instruction at `line 4`.

[2]Note that the global region consists of initialized and uninitialized data segments.

[3]By 'conservative fashion', we refer to the fact that VSA does not actively infer the value held in a memory cell if the data flow propagation is blocked by an unknown memory reference.

In addition to assigning instructions to memory regions in the ways above, VSA tracks down variable-like entities referred to as *a-locs*. By convention, an *a-loc* could be a register, a memory cell on the stack, on the heap, or in the global region. Take the instruction trace shown in Figure 1 as an example. The register *a-locs* contain all the registers `esp`, `eax` and `ebp`. The global *a-locs* contain `[0xC8]`. The stack *a-locs* include `[esp]`, `[esp+0x8]`, `[esp+0xC]` and `[ebp+0x8]`. The heap *a-locs* consist of `[eax]` and `[eax+0x4]`. It should be noticed that, as is illustrated in Table 1b, VSA represents a non-register *a-loc* as a combination of the value held by a memory cell and the value set indicating the address of that memory cell. For example, the instruction `mov [esp+0x8],eax` accesses the stack memory, and VSA specifies its corresponding stack *a-loc* as `[esp+0x8]` (⊥, `[-0xC, -0xC]`, ⊥). Here, `[esp+0x8]` indicates the name of the stack memory cell, and (⊥, `[-0xC, -0xC]`, ⊥) is the value set of the memory address or, in other words, the values that `esp+0x8` could potentially equal to at the site of that instruction.

For each *a-loc* identified, VSA computes a value set, indicating the set of values that each *a-loc* could potentially equal to. By convention, VSA represents such a value set as a *3-tuple* pertaining to the three memory regions partitioned. For each element in the tuple, VSA specifies a range of offsets which indicates the values that the *a-loc* could equal to with respect to the corresponding memory region.

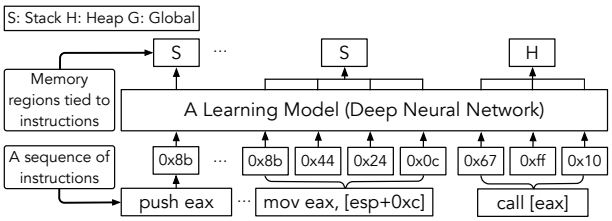
To illustrate this, we take the register *a-loc* `esp` as an example. As depicted in the first row of Table 1b, VSA specifies its value set as a *3-tuple* (`global` ↦ ⊥, `stack` ↦ `[-0x14, -0x14]`, `heap` ↦ ⊥), for brevity (⊥, `[-0x14, -0x14]`, ⊥). In this set, ⊥ is a symbol denoting the empty set of offsets (*i.e.,* ∅). It reflects the fact that the register `esp` is the stack pointer in x86 architecture and cannot refer to any memory cells on the heap or global region. Since the semantics of the first instruction is to offset `esp` by `0x14` from the starting point of the stack, VSA assigns the value set {`-0x14`} to the register *a-loc* `esp`, and attaches this set to the stack. It should be noticed that for specification consistency we write the value sets {`-0x14`} tied to the stack as `[-0x14, -0x14]`.

2.3 Alias Analysis and Root Cause Diagnosis

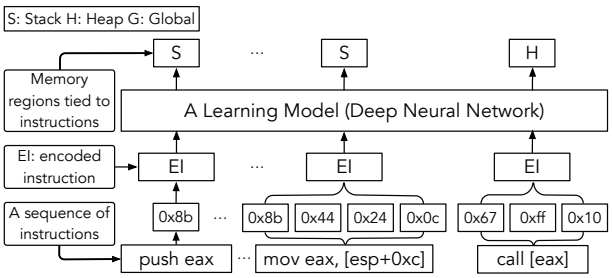
Alias Analysis. Given a control flow specified as a sequence of instructions executed prior to a program crash, VSA can track down *a-locs*, derive value sets, and perform memory alias analysis by examining the value set tied to each of the *a-locs*. To illustrate this, we again take the instruction trace depicted in Figure 1 as an example and assume they represent the entire execution trace prior to a program crash. Supposing that Table 1b indicates the value set tied to each of the *a-locs* identified from the instruction trace, we can easily observe that `[esp]` at `line 6` and `[ebp+0x8]` at `line 11` refer to the same memory region or in other words they are alias of each other. In addition, we can observe `[eax]`

at `line 4, 12 and 18` are also alias between each other. This is simply because the *a-locs* tied to these memory regions carry the overlapping value set corresponding to their addresses, *i.e.,* (⊥, `[-0x18, -0x18]`, ⊥) for `[esp]` and `[ebp+0x8]`; (⊥, ⊥, `[0,0]`) for `[eax]`. To better understand the effect of VSA on alias analysis, we derive all the alias and non-alias relationships from the value sets specified in Table 1b, and depict them in the upper triangular portion of the matrix shown in Table 1a.

Root Cause Diagnosis. With the alias analysis results and the value sets in hand, it is relatively easy to perform a backward taint analysis and thus track down the root cause of a program crash. To illustrate this process, we continue the example shown in Figure 1. Given that the program crashes at `line 18` when the program performs an indirect call, we can easily discover that the bad destination `[eax]` was passed through the instruction at `line 12` in which memory `[eax]` is assigned with a constant `0x1`. As is described above, `[eax]` at `line 12 and 18` are the alias of each other. Therefore, we can safely conclude the bad destination originally comes from the instruction `mov [eax],0x1` in `line 12`. Through this backward analysis, we could deem the instruction `mov [eax],0x1` as the root cause of the crash.



(a) A neural network taking machine code as its input.



(b) A neural network taking as input encoded instructions.

Figure 2: Two neural networks that identify memory region accesses pertaining to each instruction by taking as input a sequence of machine code and a sequence of encoded instructions respectively.

2.4 Problem Scope

As is described in the aforementioned example, VSA exhibits perfect performance in alias analysis and we could identify the root cause of the crash successfully. However, this does not imply that VSA could significantly resolve the memory alias issue and thus perfectly facilitate postmortem program analysis. To demonstrate this, we again take for example the instruction trace shown in Figure 1. However, different from the setup specified above, we assume the trace is available only starting from `line 6`. As is described in Section 2.1, hardware tracing components store a instruction trace in a circular buffer with limited size. As a result, it is commonplace that a security analyst cannot obtain a complete crashing trace but only a partial execution chronology prior to a program crash. By truncating the trace in our example, we emulate the scenario where there are only last *N* instructions recorded in a post-crash artifact.

In Table 1b, we also show the *a-locs* identified from this truncated trace. Compared with the value set derived from the full execution trace shown in the same figure, we can easily observe that nearly all the value sets tied to the *a-locs* are varied. This is because VSA performs an over-approximation in value-set construction and the missing context limits the capability of VSA with respect to reasoning memory regions or offsets within a region. Take the *a-loc* indicated by `[eax+0x4]` (⊤, ⊤, ⊤) as an example. Without the complete execution context of the crashing program, VSA conservatively assumes `eax` could equal to any value. Thus, memory `[eax+0x4]` could refer to any memory regions with an arbitrary offset indicated by the symbol ⊤. As is shown in the instruction in `line 13`, the value of `[eax+0x4]` is assigned by a value from a global region. Therefore, the value set tied to this *a-loc* can be represented as (`[0x2, 0x2]`, ⊥, ⊥). From the *a-locs* identified from the truncated trace along with their value set, we follow the aforementioned approach to examine value set intersection, and illustrate the alias and non-alias relationships in the lower triangular portion of the matrix shown in Table 1a. As we can easily observe, without the full execution trace, VSA over-approximates value sets tied to *a-locs*, and conservatively deems many memory pairs as may-alias relationships. Since may-alias represents uncertainty relationship, Table 1a illustrates them as the question symbol '*?*'. Using such results to derive the data flow for software crash diagnosis, it is not difficult to observe that a security analyst can barely yield any useful results or in other words pinpoint the root cause of the program crash for the simple reason that VSA has the limited capability in tracking down the memory alias.

3 Technical Approach

To address the problem above, we propose a technical approach driven by a deep neural network. In this section, we first discuss why deep learning could potentially facilitate VSA and thus improve software crash analysis. Second, we briefly describe neural network architectures commonly used in other binary analysis tasks. Third, we discuss the limita-

tion of these existing neural networks and then specify how to design a new neural architecture to better tackle our problem. Finally, we present the detail of our new neural architecture and specify how to integrate it into conventional VSA.

3.1 Overview

Recall that, when a crashing trace is incomplete, VSA exhibits an insufficient capability in alias analysis and thus fails root cause diagnosis. As is demonstrated above, this is because the missing context restricts the ability of VSA to determine the region of memory accesses for some instructions. To address this pitfall, we leverage a deep neural network to enhance VSA with the ability to infer memory region(s) for instructions. In the following, we describe the rationale behind this idea and illustrate why it could benefit the diagnosis of software crashes.

Rationale behind our idea. In many previous applications (*e.g.,* speech recognition [24] and API generation [25]), it has been demonstrated that some sequence-to-sequence neural network architectures can be used to learn patterns from a sequence of inputs, thus facilitating the determination of a label for each individual input. As a result, in order to augment conventional VSA with the ability to infer the memory region(s) that each instruction refers to, intuition suggests that we can view an execution trace as a sequence of machine code or instructions, partition memory into disjoint regions (*e.g.,* stack, heap and global), treat each region as an individual label tied to each instruction and eventually use a sequence-to-sequence deep neural network to predict that label for each instruction. For example, given the instruction `push 0x68732f2f` represented by machine code `[0x68, 0x2f, 0x2f, 0x73, 0x68]`, we could determine the stack region is tied to this instruction by using either of the two designs shown in Figure 2. As is depicted in the figure, the two designs take the input differently, one with machine code as the input directly to a deep learning model and the other with the encoded instructions as the input to a model. In Section 3.3, we compare these two designs and describe why we choose one over the other. In Section 4, we show their performance difference.

Effect upon root cause diagnosis. With the augmentation above, VSA could typically perform better alias analysis and thus benefit the diagnosis of a software crash. We illustrate this by again taking for example the instruction trace shown in Figure 1. Recall that, without the complete execution context, conventional VSA cannot determine the memory region that `eax` refers to. Therefore, it assumes `[eax]` and `[eax+0x4]` could represent any memory regions, assigns `eax` and `eax+0x4` with value-set (⊤, ⊤, ⊤) and eventually fails the root cause diagnosis of that crash.

Given the sequence of the instructions tied to the crashing trace, assume a deep neural network could correctly infer that, the register `eax` at `line 6` refers to a memory region at the heap. Then, VSA could assign `eax` with value-set (⊥, ⊥, `[X, X]`) where `[X, X]` denotes an unknown address on the heap. With this, VSA could further update the value sets for corresponding *a-locs*. We show the updated value sets in Table 1b under the column "*Incomplete Trace with DL*". As we can observe, the memory reference `[eax]` at `line 12 and 18` are aliased to each other because they both refer to the same memory address `[X, X]` on the heap. With this alias analysis result, VSA could quickly assist backward taint in tracking down the instruction at `line 12` – the root cause of the crash – even though this crashing trace is partial and incomplete.

3.2 Existing Neural Architectures

To perform binary analysis with deep learning, previous research typically utilized three types of recurrent neural networks (RNNs) – vanilla RNN [33], long short-term memory (LSTM) [22] and gated recurrent units (GRU) [13]. Here, we briefly describe them in turn.

3.2.1 Vanilla Recurrent Neural Network

A vanilla RNN (RNN for brevity) is specialized for processing a sequence of values $x^{(1)},\ldots,x^{(t)}$. When trained to perform a prediction from the past sequence of inputs, it typically maps the sequence to a fixed length vector $h^{(t)}$ through a function $g^{(t)}$:

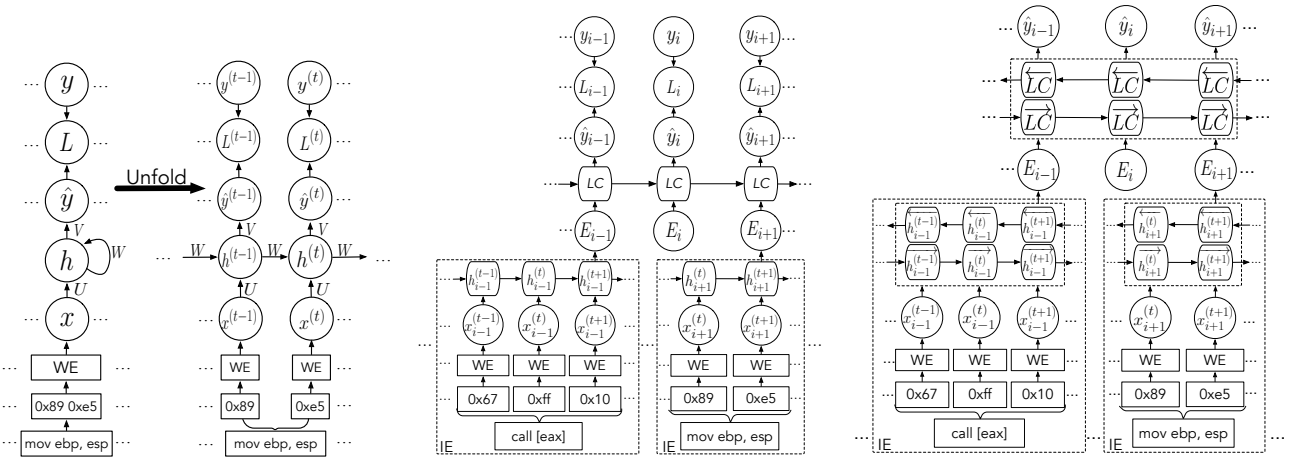
$$\begin{aligned} h^{(t)} &= g^{(t)}(x^{(t)},x^{(t-1)},x^{(t-2)},\ldots,x^{(2)},x^{(1)}),\\ &= f(h^{(t-1)},x^{(t)};\theta). \end{aligned}$$

As we can observe from this equation, the function $g^{(t)}$ takes the whole past sequence as input and produces a summary $h^{(t)}$ for that sequence. In an RNN, $h^{(t)}$ refers to a hidden state. As is illustrated in Figure 3a, an RNN can be unfolded as a chain structure where each hidden state is connected to the previous one [23]. As such, $g^{(t)}$ can be factorized into the repeated application of a function f, which controls the transition from the previous hidden state to the next one (*i.e.,* the recurrent neuron). For example, assuming the length of the chain to be 3 – indicating a finite number of hidden states – we can then obtain

$$\begin{aligned} h^{(3)} &= f(h^{(2)};\theta),\\ &= f(f(h^{(1)};\theta);\theta). \end{aligned}$$

To make predictions using the chain structure depicted in Figure 3a, an RNN follows a forward propagation in which it begins with an initial state $h^{(0)}$ and then utilizes the update equations below to compute the prediction $\hat{y}^{(t)}$ accordingly.

$$\begin{aligned} a^{(t)} &= \mathbf{W}h^{(t-1)} + \mathbf{U}x^{(t)} + b,\\ h^{(t)} &= \tanh(a^{(t)}),\\ o^{(t)} &= \mathbf{V}h^{(t)} + c,\\ \hat{y}^{(t)} &= \mathrm{softmax}(o^{(t)}). \end{aligned}$$



(a) The Vanilla RNN. (b) The hierarchical LSTM. (c) The bi-directional hierarchical LSTM.

Figure 3: Recurrent neural networks with various architectures serving for different purposes. Note that "LC" indicates LSTM cell, "IE" stands for the embedding for each instruction and "WE" refers to the word embedding.

Here, bias vectors b and c are parameters. $\tanh(\mathbf{W}h^{(t-1)} + \mathbf{U}x^{(t)} + b)$ is the detailed form of the recurrent neuron f in which tanh is an activation function [54]. Softmax refers to the softmax classifier [10]. Along with the weight matrices U, V and W, pertaining to input-to-hidden, hidden-to-output, and hidden-to-hidden connections respectively, the bias vectors can be learned by minimizing the loss function described below

$$\begin{aligned} L^{(t)} &= L(x^{(1)}, x^{(2)}, ..., x^{(t)}, y^{(1)}, y^{(2)}, ..., y^{(t)}) \\ &= \sum_t L^{(t)} \\ &= -\sum_t \log p_{model}(y^{(t)} | x^{(1)}, x^{(2)}, ..., x^{(t)}), \end{aligned}$$

where $p_{model}(y^{(t)}|x^{(1)}, \ldots, x^{(t)})$ is the probability from the prediction vector $\hat{y}^{(t)}$ corresponding to the entry for the true label vector $y^{(t)}$. Similar to other neural networks commonly used (*e.g.,* multi-layer perceptron [44] and convolution neural networks [32]), the minimization of the aforementioned loss function can be achieved by using different kinds of optimization algorithms (*e.g.,* stochastic gradient descent [11], ADAM [31], RMSprop [52]) with respect to the bias parameters and weight matrices. The details of these optimization algorithms can be found in [45].

3.2.2 Long Short-Term Memory

In the cybersecurity community, recent works have demonstrated that a vanilla RNN has already demonstrated great performance when performing binary analysis (*e.g.,* [15, 48]). However, it has been noted that, as is used in other applications such as speech recognition and machine translation, such an ordinary recurrent architecture is not sufficient in processing a long sequence of inputs. This is because a vanilla RNN naturally struggles to remember information for long periods of time or, in other words, suffers from derivative vanishing and explosion problems [26]. To address this issue, other works have used a long short-term memory (LSTM) model to carry out binary analysis.

Similar to a vanilla RNN depicted in Figure 3a, LSTM also has a chain structure. However, it replaces the aforementioned hidden states with LSTM cells, and each cell carries a set of parameters and a system of gating units that controls the flow of information. In an LSTM network, each cell has a state unit $s^{(t)}$ as well as three gating units – a forget gate unit $f^{(t)}$, an external input gate unit $g^{(t)}$, and an output gate $q^{(t)}$ – which together control the output $h^{(t)}$ of the LSTM cell via the following equation

$$\begin{aligned} s^{(t)} &= f^{(t)} \odot s^{(t-1)} + g^{(t)} \odot \sigma(\mathbf{W}h^{(t-1)} + \mathbf{U}x^{(t)} + b), \\ h^{(t)} &= q^{(t)} \odot \tanh(s^{(t)}). \end{aligned}$$

Here, $\sigma(\cdot)$ denotes a sigmoid function [29] which sets a value between 0 and 1, and $\odot$ represents the element-wise multiplication. b, U and W respectively indicate the biases, input weights, and recurrent weights into an LSTM cell. To compute the gate units, one could follow the equations below

$$\begin{aligned} g^{(t)} &= \sigma(\mathbf{W}^g h^{(t-1)} + \mathbf{U}^g x^{(t)} + b^g), \\ f^{(t)} &= \sigma(\mathbf{W}^f h^{(t-1)} + \mathbf{U}^f x^{(t)} + b^f), \\ q^{(t)} &= \sigma(\mathbf{W}^q h^{(t-1)} + \mathbf{U}^q x^{(t)} + b^q), \end{aligned}$$

where $\{b^f, b^g, b^q\}$, $\{\mathbf{U}^f, \mathbf{U}^g, \mathbf{U}^q\}$ and $\{\mathbf{W}^f, \mathbf{W}^g, \mathbf{W}^q\}$ are respectively: biases, input weights, and recurrent weights for the forget, external input, and output gates. Similar to b, $\mathbf{U}$ and $\mathbf{W}$, they are also the parameters that can be learned via the optimization algorithms mentioned above. Again, more details of parameter computation can be found at [23].

3.2.3 Gated Recurrent Units

As is described in previous research [15], gated recurrent units (GRU) can also be used for some of binary analysis tasks. GRU is an alternative LSTM which can also capture long term dependency. The main difference between GRU and LSTM is that GRU replaces the forget gate f and output gate q in LSTM with one update gate. More specifically, it integrates both forget and output gates into a single gating unit $u^{(t)}$. As a result, it reduces the parameters that a network has to learn and thus poses a lower computational cost. The following equations indicate how to compute the output $h^{(t)}$ of a GRU cell:

$$r^{(t)} = \sigma(\mathbf{W}^r h^{(t-1)} + \mathbf{U}^r x^{(t)} + b^r),$$
$$u^{(t)} = \sigma(\mathbf{W}^u h^{(t-1)} + \mathbf{U}^u x^{(t)} + b^u),$$
$$h^{(t)} = u^{(t)} \odot h^{(t-1)} + (1-u^{(t)}) \odot \tanh(\mathbf{W}(r^{(t)} \odot h^{(t-1)}) + \mathbf{U}x^{(t)} + b).$$

Here, $r^{(t)}$ stands for a reset gate which controls the influence of the past sequences of inputs upon the current one. $\{b^r, \mathbf{U}^r, \mathbf{W}^r\}$ and $\{b^u, \mathbf{U}^u, \mathbf{W}^u\}$ are gate weights. Along with the bias b and weights $\mathbf{U}$, $\mathbf{W}$, they need to be learned through the aforementioned optimization algorithms.

3.3 Our Neural Network Architecture

As we described in Section 3.1, we could utilize two different design mechanisms to predict the memory region that each instruction refers to. For the design shown in Figure 2a, we could simply leverage any of the aforementioned recurrent neural networks to take as input the sequence of machine code, learn the pattern hidden behind the machine code sequence and predict the memory region for each instruction. As they have already demonstrated in other binary analysis tasks (*e.g.,* [48, 15]), we could expect this design could perform reasonably well in memory region identification. However, following the intuition described below, we do not utilize this design. Rather, we develop our technique by using the alternative design shown in Figure 2b.

Take for example the instruction sequence `push ebp; mov ebp, esp` indicated by the byte sequence `[0x55, 0x89, 0xe5]`. An existing neural network model could take this machine code sequence as input and make predictions for their corresponding memory accesses based on the dependency between the bytes. It is not too difficult to observe that this simple approach neglects the semantics and contexts of these instructions. As is described in Section 2, in binary analysis, the semantics and contexts of instructions could be used as indicators to infer the memory accesses tied to instructions. Therefore, intuition suggests that it could be potentially beneficial for memory region identification if we could build a neural network with the ability to capture not only the dependency between the bytes but also that between instructions.

Inspired by this, we choose the design depicted in Figure 2b and build a hierarchical LSTM architecture. We depict the structure of this learning model in Figure 3b. As we can observe, similar to existing neural networks used for other binary analysis tasks, it first maps each byte into a vector by using a word embedding mechanism [9]. Then, it groups the bytes per each instruction and utilizes an embedding network to convert each group of bytes into an instruction embedding (*i.e.,* an encoded vector). Taking the instruction embedding as the input, our neural architecture further employs a sequence-to-sequence network [50] to predict the memory region tied to each instruction.

In comparison with the aforementioned off-the-shelf recurrent architectures largely adopted by other binary analysis tasks, the proposed hierarchical LSTM architecture is composed of two networks. The embedding network models the correlation of bytes in one instruction and the sequence-to-sequence network captures the dependency between instructions. By designing the model structure in this fashion, our neural network model is able to perform memory access predictions at the instruction level and learn the dependency between and within instructions at the same time.

However, it is not difficult to note that this new recurrent architecture cannot represent a backward analysis procedure, where the memory region(s) tied to an instruction is determined by the consecutive instructions. Yet we note that this backward analysis is feasible. To illustrate this, we take the following execution trace as an example.

```
00015670 <malloc>:
 53              push   ebx
 ...
 89 44 24 04     mov    DWORD PTR [esp+0x4],eax
 e8 6d b1 fe ff  call   800 <_libc_memalign@plt>
 83 c4 18        add    esp,0x18
 5b              pop    ebx
 c3              ret
```

As is illustrated above, the trace indicates the instructions and corresponding machine code executed while invoking the `malloc` function. Here, the highlighted instruction and machine code indicate the last definition of `[eax]` prior to the return of the function call. Given that the call to `malloc` places the return value in the register `eax`, indicating an address on the heap, we can reversely perform inference and conclude that the memory access tied to the highlighted instruction is within a heap region.

To enable our design with the capability of inferring memory regions in both forward and backward ways, we further upgrade our hierarchical LSTM model to a bi-directional chain structure [46]. As is shown in Figure 3c, our bi-directional chain structure is applied to both the embedding network and the sequence-to-sequence network. With respect to the embedding network, our neural architecture combines a network that moves forward, beginning from the start of the corresponding byte sequence, with another network that moves backward, starting from the end of the corresponding byte

sequence. Regarding the sequence-to-sequence network, our architecture concatenates the output of a forward embedding network with the output of a backward embedding network. Then, it takes the concatenation as input and performs memory access prediction for each individual instruction based on the sequence of instructions executed before and after that instruction.

3.4 Detail of Our Neural Architecture

Here, we describe more details of our proposed neural network architecture. More specifically, we specify how we process a crashing trace, perform corresponding computation, train the neural network and eventually utilize it to facilitate VSA.

Padding and word embedding. As is described above, our neural network utilizes a bi-directional embedding to encode each instruction prior to making predictions for their memory accesses. Before passing machine code to that embedding network, we process them as follows.

Assume we have a crashing trace containing n instructions $I_{1:n}$. For each instruction I_i, it could be represented as m bytes of machine code $b_i^{(1:m)}$. For an x86 machine, instructions do not share the same length. To design the same structure of embedding networks for instructions, we therefore pad instructions to a fixed length. To do this, we first convert each individual byte into an integer based on its value (e.g., encoding machine code `55` to its integer form `85`). Then, we pad that instruction with integer 256. In this way, we could ensure our padding does not introduce ambiguity to a target instruction. After the padding, we also utilize a word embedding to further process the padded crashing trace. In our work, our word embedding converts each byte into a one-hot vector with a dimensionality of 257. Then, the vector is multiplied with a matrix projecting the byte into a new vector (*i.e.,* $x_i^{(1:m)}$) typically with lower dimensionality.

Instruction embedding. For each instruction, we use a bi-directional LSTM model to further encode its word embedding and then generate an individual instruction embedding. Technically speaking, we achieve this by integrating the outputs of the forward and backward networks. More specifically, we utilize the following equations to compute the output of the forward network.

$$\overrightarrow{h_i^{(t)}} = \text{LSTM}(\overrightarrow{h_i^{(t-1)}}, x_i^{(t)}),$$
$$\overrightarrow{E_i} = \overrightarrow{h_i^{(m)}}.$$

Similarly, we compute the output for the backward network as follows.

$$\overleftarrow{h_i^{(t)}} = \text{LSTM}(\overleftarrow{h_i^{(t+1)}}, x_i^{(t)}),$$
$$\overleftarrow{E_i} = \overleftarrow{h_i^{(1)}}.$$

Here, $\overrightarrow{E_i}$ and $\overleftarrow{E_i}$ are the forward and backward embeddings of the instruction I_i, respectively. LSTM denotes an LSTM cell introduced above. As we can observe from the two sets of equations above, the hidden representation of the first and last bytes of the instruction, $h_i^{(m)}$ and $h_i^{(1)}$, contain the information that flows from the previous and consecutive bytes.

To combine the outputs of both forward and backward networks, we concatenate both representations in the form of $E_i = [\overrightarrow{E_i}, \overleftarrow{E_i}]$). Technically, it should be noted that we can use one single embedding network for all instructions, or employ different embedding networks for instructions. With the consideration of lowering computational overhead, our neural network architecture follows the first approach.

Sequence-to-sequence network. Given a sequence of instruction embeddings pertaining to the instructions in a crashing trace, we then use a sequence-to-sequence model mentioned above to predict the label (*i.e.,* , memory access region(s)) for each instruction. To be specific, the model takes as input the instruction embeddings $E_{1:n}$ and utilizes a bi-directional LSTM as the hidden layer of our neural architecture [4]. At the output layer of our neural network, it uses a softmax classifier to assign a corresponding label for each hidden state (*i.e.,* the hidden representation of each instruction). Different from previous deep neural network used in other binary analysis technique, which assigns a label to each byte, our new architecture gives us the ability to attach an individual prediction to each instruction.

Training strategy. Similar to the recurrent neural networks summarized in Section 3.2, we also need to leverage aforementioned optimization algorithms to estimate the parameters for our neural network. In binary analysis tasks, the training dataset is often significantly large, *e.g.,* one execution trace carries millions of lines of instructions. Using conventional gradient descent algorithms – like stochastic gradient descent – against a large data set, parameter estimation would experience significant computation overhead. To address this issue, we take advantage of mini-batch gradient descent, a variation of the gradient descent algorithm [28]. Technically speaking, this approach splits the training dataset into small batches, uses them to calculate model error through loss function and updates model parameters accordingly. Compared with other approaches, particularly stochastic gradient descent, mini-batch provides a computationally efficient process and enables parallel computations.

In addition to mini-batch gradient descent, we adopt `RMSprop` [34] to accelerate the optimization process needed for gradient descent computation. To be specific, we adjust the learning rate by dividing it by an exponentially decaying average of squared gradients. For more details, the reader could refer to an unpublished article available at Geoff Hinton's class [34]. Last but not least, we also pad the remaining sequences in the last batch with vectors where each element equals 256. In this way, we can represent each batch as a

[4] Note that we can also use GRU as an alternative to LSTM for the encoding network and the sequence to sequence network.

matrix with fixed size, making it capable of being efficiently processed at the same time.

Integration into VSA. Without the facilitation of a deep learning model and the clue of which memory region an instruction accesses, VSA initializes *a-locs* and value-set with (⊤, ⊤, ⊤), indicating the memory access in that instruction could refer to any memory regions. Using our deep neural architecture introduced above, we could have the neural network output the memory region that instruction accesses (*i.e.,* global, stack or heap). As is illustrated in Section 3.1, with this capability, we could initialize *a-locs* and value-set with ([X,Y], ⊥, ⊥), (⊥, [X,Y], ⊥) or (⊥, ⊥, [X,Y]), denoting a memory access in that instruction could refer to a particular memory area ranging from X to Y at a global, stack or heap region. Then, starting from the first instruction in the crashing trace, VSA could regularly perform forward analysis and update the value for X and Y. For example, as we have shown in Table 1b, when analyzing the instruction at line 6, our deep learning model initializes eax with value-set (⊥, ⊥, [X,Y]) and VSA updates X and Y with X=Y indicating the register eax refers to the memory address X at the heap region.

4 Evaluation

In this section, we describe our implementation, the dataset we utilized, set up our experiment, and summarize our experimental results. Through this evaluation, we seek to answer the following questions. ❶ Does our problem require a deep learning model or could it be resolved with conventional machine learning techniques? ❷ Can our proposed technique correctly link memory regions to instructions or, more precisely, identify the memory regions that instructions dereference? ❸ Compared with commonly adopted recurrent neural architectures that take as input the raw machine code, does the proposed neural network architecture (taking encoded instructions as the input to a neural network) exhibit better performance in terms of memory region identification? ❹ Can the memory regions identified improve the ability of VSA with respect to memory alias analysis and thus bring the positive impact upon the capability in software crash diagnosis?

4.1 Implementation

To answer the questions above, we must first train many deep neural network architectures. This requires a large training data set containing various instruction traces as well as the memory reference tied to each instruction. To facilitate the collection of the instruction traces as well as the corresponding memory accesses, we first implemented a tracing system which provides us with the ability to not only record the instructions that a target program executes but also the memory region each instruction refers to. While both Intel PT and ARM ETM could trace program execution, in this work, we utilize Intel Pin [35] to complete the implementation of our tracing system. This is because, in order to train a neural network, we have to obtain the ground truth of which memory regions instructions access but both hardware components do not provide us with such a capability (*i.e.,* recording memory regions referred by instructions).

In addition to the tracing system, we customized a VSA system which implemented an instruction parser using libdisasm and 84 distinct instruction handlers to perform value-set calculation. Going beyond alias analysis, the implementation of our customized VSA system also contains a backward taint component which takes the results of alias analysis and performs the root cause diagnosis for a crashing program. In total, our VSA implementation contains about 9,500 lines of C code. It should be noticed that the value-set calculation for instructions with similar semantics (*e.g.,* ja, jb, jc) were taken care of by a unique handler.

Recall that our ultimate goal is to use a deep neural network to facilitate VSA with respect to alias analysis and thus improve the effectiveness of software crash diagnosis. Last but not least, we therefore prototyped a neural network assisted VSA system and named it after DEEPVSA. In our implementation, DEEPVSA first utilizes a pre-trained deep neural network to predict memory accesses for each instruction. Then, it determines non-aliasing relationships based on the prediction by following the approach introduced in Section 3. Combining the results of the conventional value-set analysis with this non-aliasing analysis, our DEEPVSA finally performs backward taint analysis and thus pinpoints the root cause of a program crash. In this work, we ran all the aforementioned systems on a 32-bit Linux system with Linux kernel 4.4.0 running on an Intel i7-6600 quad-core processor with 16 GB RAM. We trained all the deep neural networks in this work on 2 Nvidia Tesla K40 GPUs and 4 Nvidia GTX 1080Ti GPUs using the Keras package [14] and with Tensorflow [1] as backend, amounting to about 2,000 lines of Python code. Upon the acceptance of this submission, we will release all of our systems along with our data set described below.

4.2 Data Set

As is mentioned above, we need to train many deep neural networks with various execution traces along with their corresponding memory accesses. In this work, we construct our training data set by using 78 unique programs in a package of GNU software – coreutils, ineutils and binutils. More specifically, we ran these programs by following their documentation and running examples. Using the aforementioned tracing system, we then gathered their execution traces along with their memory accesses. In total, these 78 programs generate a training data set with 96 distinct execution traces covering 49,193,919 lines of instructions.

To test our neural network and demonstrate the effective-

Index	Program	Non-alias (400)		Non-alias (800)		Non-alias (3200)		Non-alias (6400)		Non-alias (12800)		Root Cause	
		VSA	DEEPVSA	VSA	DEEPVSA	VSA	DEEPVSA	VSA	DEEPVSA	VSA	DEEPVSA	VSA	DEEPVSA
1	coreutils-8.4	66.58%	88.28%	66.58%	88.28%	66.58%	88.28%	66.58%	88.28%	66.58%	88.28%	✓	✓
2	coreutils-8.4	1.25%	33.09%	1.25%	33.09%	1.25%	33.09%	1.25%	33.09%	1.25%	33.09%	✓	✓
3	coreutils-8.4	62.77%	93.84%	62.77%	93.84%	62.77%	93.84%	62.77%	93.84%	62.77%	93.84%	✓	✓
4	nginx-1.4.0	68%	99%	68%	99%	68%	99%	68%	99%	68%	99%	✓	✓
5	nullhttpd-0.5.0	67.47%	72.30%	67.47%	72.30%	67.47%	72.30%	67.47%	72.30%	67.47%	72.30%	✓	✓
6	DXFScope-0.2	6.17%	42.39%	6.17%	42.39%	6.17%	42.39%	6.17%	42.39%	6.17%	42.39%	✓	✓
7	tiff-3.8.2	0.58%	30.50%	0.58%	30.51%	0.58%	30.51%	0.58%	30.51%	0.58%	30.51%	✓	✓
8	unrtf-0.19.3	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	✓	✓
9	gdb-6.6	23.03%	84.64%	23.34%	84.70%	41.13%	91.83%	41.13%	91.83%	41.13%	91.83%	✓	✓
10	openjpeg-2.1.1	11.93%	14.38%	11.93%	14.38%	11.93%	14.38%	92.67%	95.13%	92.67%	95.13%	✓	✓
11	python-2.7	65.00%	91.26%	71.14%	97.49%	71.14%	97.49%	71.14%	97.49%	71.14%	97.49%	✓	✓
12	poppler-0.8.4	0.00%	39.43%	0.00%	39.43%	0.00%	39.43%	60.60%	98.56%	60.60%	98.56%	✓	✓
13	htmldoc-1.8.27	0.077%	30.48%	0.077%	30.48%	0.077%	30.48%	0.077%	30.48%	0.077%	30.48%	✓	✓
14	unalz-0.52	0.05%	39.31%	0.05%	39.31%	0.05%	39.31%	0.05%	39.31%	0.05%	39.31%	✓	✓
15	psutils-p17	0.17%	45.20%	0.17%	45.20%	48.84%	90.44%	48.84%	90.44%	48.84%	90.44%	✗	✓
16	libpng-1.2.5	29.72%	80.76%	29.72%	80.76%	29.72%	80.76%	29.72%	80.76%	29.72%	80.76%	✓	✓
17	gas-2.12	0.02%	49.41%	0.02%	49.41%	0.02%	49.41%	0.02%	49.41%	0.02%	49.41%	✓	✓
18	SQLite-3.8.6	48.83%	96.82%	48.83%	96.82%	48.83%	96.82%	48.83%	96.82%	48.83%	96.82%	✗	✓
19	pcal-4.7.1	22.74%	85.42%	22.74%	85.42%	22.75%	85.43%	22.75%	85.43%	22.75%	85.43%	✗	✓
20	LaTeX2rtf-1.9	9.93%	10.55%	9.93%	10.55%	19.68%	30.16%	19.68%	30.16%	19.68%	30.16%	✗	✓
21	gif2png-2.5.2	43.56%	95.64%	43.56%	95.64%	43.56%	95.64%	43.56%	95.64%	43.56%	95.64%	✗	✓
22	abc2mtex-1.6.1	22.38%	71.54%	22.38%	71.54%	22.38%	71.54%	22.38%	71.54%	22.38%	71.54%	✓	✓
23	O3read-0.0.3	28.13%	75.47%	28.13%	75.47%	28.13%	75.47%	28.13%	75.47%	28.13%	75.47%	✗	✓
24	gdb-7.5.1	0.02%	55.30%	0.02%	55.30%	42.09%	93.56%	42.09%	93.56%	42.09%	93.56%	✗	✓
25	podofo-0.9.4	2.00%	22.15%	2.00%	22.15%	2.00%	22.15%	2.00%	22.15%	2.00%	22.15%	✓	✓
26	nasm-0.98.38	0.35%	44.78%	0.35%	44.78%	0.35%	44.78%	57.34%	99.24%	57.34%	99.24%	✓	✓
27	corehttp-0.5.3a	0.00%	40.98%	0.00%	40.98%	0.00%	40.98%	58.48%	94.40%	58.48%	94.40%	✓	✓
28	corehttp-0.5.3.1	0.00%	41.21%	0.00%	41.21%	0.00%	41.21%	58.08%	95.22%	58.08%	95.22%	✓	✓
29	unrar-3.9.3	21.29%	82.41%	21.29%	82.41%	21.29%	82.41%	21.29%	82.41%	21.29%	82.41%	✗	✓
30	prozilla-1.3.6	4.98%	56.53%	4.98%	56.53%	4.98%	56.53%	32.06%	77.97%	32.06%	77.97%	✗	✓
31	python-2.7.5	1.00%	3.01%	1.00%	3.01%	1.00%	3.01%	1.00%	3.01%	1.00%	3.01%	✓	✓
32	html2hdml-1.0.3	1.92%	34.55%	1.92%	34.55%	1.92%	34.55%	1.92%	34.55%	1.92%	34.55%	✓	✓
33	mcrypt-2.5.8	14.95%	53.02%	22.83%	59.84%	63.36%	100%	63.36%	100%	63.36%	100%	✗	✓
34	putty-0.66	5.06%	24.67%	5.06%	24.67%	5.06%	24.67%	18.58%	54.09%	18.58%	54.09%	✓	✓
35	mp3info-0.8.5a	1.9%	55.58%	1.9%	55.58%	3.82%	55.92%	3.82%	55.92%	3.82%	55.92%	✓	✓
36	LibSMI-0.4.8	70.44%	94.53%	70.44%	94.53%	70.44%	94.53%	70.44%	94.53%	70.44%	94.53%	✓	✓
37	JPegToAvi-1.5	0.00%	55.20%	0.00%	55.20%	0.00%	55.20%	0.00%	55.20%	13.81%	67.81%	✓	✓
38	aireplay-ng-1.2	4.46%	50.80%	4.96%	51.30%	49.17%	88.57%	49.17%	88.57%	49.17%	88.57%	✗	✗
39	ClamAV-0.93.3	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	✗	✗
40	0verkill-0.16	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	✗	✗
Total	-	-	-	-	-	-	-	-	-	-	-	27(✓)	37(✓)
Average	-	21.23%	57.49%	21.62%	57.84%	27.01%	62.79%	36.37%	72.07%	36.73%	72.40%	-	-

Table 2: The list of program crashes corresponding to memory corruption vulnerabilities. "Root cause" specifies whether the result of alias analysis successfully facilitate the root cause identification of software crashes. The percentages under VSA and DEEPVSA represent the amount of non-alias memory pairs identified. The number shown along with "non-alias" indicates the length of the trace prior to the site of the root cause instruction.

ness of DEEPVSA in alias analysis and root cause diagnosis, we exhaustively searched the Exploit Database Archive [47] and randomly selected 40 distinct vulnerability reports corresponding to 38 unique versions of software running on Linux. Following the description of each report, we compiled vulnerable programs[5], configured the underlying systems and ran the PoC programs tied to corresponding vulnerabilities. In this way, we triggered software failures, recorded their crashing traces and treated these traces as our testing data set. Using these crashing traces, we benchmarked DEEPVSA and examined the effectiveness of our proposed technique. Recall that the execution trace is stored in a circular buffer with a limited size (4KB) and that buffer is shared by multiple running processes. Since different lengths of an instruction trace stored in that shared buffer might influence memory alias identification, we retained different lengths of instructions for each of our test cases. This gives us the ability to identify the optimal memory size needed for a running process.

In Table 2, we present all the crashing programs selected[6]. From the table, we have the following observations. First of all, we can observe that the programs listed in the table has less overlaps with the programs in our training data set. This implies the dissimilarity between our training and testing data sets and thus avoids the possibility of using the same or similar data for model training and testing. Considering programs could invoke functions in the same shared library (e.g., glibc), and too many of such invocations could potentially

[5] In other binary analysis research works using deep learning, the binary is typically compiled with various optimization options. In this work, we compiled programs mostly with O2 option because many vulnerabilities cannot be reproduced if compiled with other options. Note that this does not influence the generalization of our approach because O2 is the default compilation options for most software.

[6] Note that we present the corresponding CVE/EDB-IDs as well as the length of each crashing trace in Appendix.

		Global	Heap	Stack	Other
Precision	HMM	67.99%	53.99%	74.47%	86.56%
	CRF	15.93%	12.31%	62.10%	71.82%
	Bi-RNN	98.63%	72.74%	95.30%	97.39%
	Bi-GRU	90.71%	78.44%	95.11%	98.40%
	Bi-LSTM	89.75%	78.47%	94.98%	97.92%
	Our Model	**96.98%**	**94.62%**	**98.92%**	**99.32%**
Recall	HMM	77.52%	39.31%	81.40%	85.28%
	CRF	10.23%	17.85%	51.95%	88.71%
	Bi-RNN	84.17%	83.72%	95.96%	95.43%
	Bi-GRU	88.04%	87.46%	97.59%	95.84%
	Bi-LSTM	91.71%	86.53%	97.16%	95.37%
	Our Model	**86.67%**	**95.99%**	**98.59%**	**99.45%**
F1 Score	HMM	72.44%	45.50%	77.78%	85.92%
	CRF	12.46%	14.57%	56.57%	79.38%
	Bi-RNN	90.83%	77.85%	95.63%	96.40%
	Bi-GRU	89.35%	82.71%	96.33%	97.10%
	Bi-LSTM	90.72%	82.30%	96.06%	96.63%
	Our Model	**91.54%**	**95.30%**	**98.75%**	**99.39%**

Table 3: The overall performance of different machine learning models.

introduce the risk of using the same data for training and testing, we further examine the instruction traces in the testing data set with those in the training. We discover that there are 14.02% of overlapping functions, appearing both in our test cases and the cases in our training set. In order to ensure our training and testing data sets do not share instructions, we eliminate the commonly shared instruction sequences from the training data set. This further avoids the situation where we perform alias analysis against a target crashing trace by using the model trained with itself.

Second, we can observe, the programs in the table cover a wide spectrum, ranging from sophisticated software like `gdb-7.5.1` with over `1.6M` lines of code to lightweight software such as `o3read-0.0.3` and `corehttp-0.5.3.1` with less than `1K` lines of code. To some extent, this diversity of our test cases imposes different levels of difficulty upon alias analysis and root cause diagnosis. Last but not least, we manually examine the memory access behaviors and observe that our test corpus encloses a variety of memory access behaviors, manifested as different amounts of memory dereferences across four disjoint memory regions (see Table 4 in Appendix). It should be noted that apart from the three memory regions that conventional VSA typically separates, we introduce '*other*' which represents the memory region pertaining to the text and global sections tied to dynamic libraries. This is an useful addition because the involvement of this region could allow us to extend conventional VSA to consider the following two memory access practices. ❶ An instruction dereferences a memory cell which held a piece of read-only data in the text section. ❷ A running process and dynamic library do not share the same global section and an instruction of the process accesses a memory cell indicating the global section of the dynamic library.

4.3 Experimental Setup

Using the systems mentioned in Section 4.1 as well as the data sets described in Section 4.2, we set up a series of experiments to evaluate our proposed technique and thus answer the four questions presented above.

To answer the first three questions (❶, ❷ and ❸) mentioned at the beginning of this section, we first trained 6 different machine learning models by using the training data set mentioned above. As is specified in Table 3, two of them are conventional machine learning models – Hidden Markov Model (HMM) as well as Conditional Random Field (CRF). While there are other machine learning approaches, such as decision tree or logistic regression, which might also work for our task, we select HMM and CRF as our baseline approaches and compare them with our proposed deep learning technique. This is because, by design, the approaches of our choice could take a sequence of input and yield a sequence of predictions, whereas other traditional machine learning approaches need to involve sophisticated feature engineering efforts in order to process a sequence of data input. In addition to conventional learning models, Table 3 depicts our proposed neural network architecture that takes instruction embedding as the input to a neural network as well as three aforementioned neural architectures that take as input the raw machine code. In this work, we compare the performance of these different neural architectures and examine whether the design of feeding instructions to a neural network outperforms that of taking raw machine code[7].

To obtain the performance measure of each machine learning models mentioned above, we applied the learning models to the aforementioned testing data set, used them to predict the memory region each instruction refers to and compare their prediction with the true labels (i.e., the memory regions a corresponding instruction truly refers to). For each memory access in the execution traces of the testing data set, we define a prediction as a correct identification if and only if the predicted memory regions aligns the true memory regions that the corresponding instruction refers to. With this definition, we further computed the precision, recall and F1 score for each machine learning model. To be more specific, we use the equations $\frac{P_M \bigcap T_M}{P_M}$, $\frac{P_M \bigcap T_M}{T_M}$ and $2 \cdot \frac{precision \cdot recall}{precision + recall}$ to compute precision, recall and F1 score, respectively. Here, P_M represents the set of memory accesses predicted to refer to memory region M where $M \in \{stack, heap, global, other\}$. T_M denotes the set of memory accesses truly referencing memory region M.

To explore the answer to our last question (❹), we further set up our experiment as follows. For each trace in our testing data set, we first applied our proposed neural network model to predict the memory regions tied to corresponding

[7] It should be noted that all the neural networks shown in the table are bi-directional. This is because previous research [48] indicates the bi-directional structure outperforms those designed with a single-directional chain particularly when using deep learning to performing binary analysis.

instructions. With these prediction results, we then utilized DEEPVSA. As is mentioned above, DEEPVSA is an extension of VSA. It is built with the additional ability to take the region prediction and determine non-alias relationships that the conventional VSA originally fails to identify. In addition, it leverages the results of alias analysis to perform backward taint analysis and thus pinpoint the root cause of the corresponding crash. Using these capabilities, our experiment compares the non-alias pairs that DEEPVSA and conventional VSA identified. Then, using the alias analysis results that DEEPVSA and conventional VSA derive, our experiment further examines their corresponding capability in facilitating the root cause diagnosis. When conducting our experiments, we also investigate the impact of the instruction trace length upon the non-alias identification. To be specific, we preserve different lengths of instructions prior to the root cause site (*i.e.,* 200, 400, 800, 1600, 3200, 6400, 12800 and 19600) and measure how different lengths impact alias identification. It should be noted we utilize 4KB of execution trace for our study if hardware cannot enclose the root cause site in its circular buffer.

4.4 Experimental Results

Performance of machine learning models. Table 3 shows the precision, recall and F1 score of various machine learning models, which demonstrate their capability of assigning correct memory regions to instructions. As we can easily observe, all deep neural network models significantly outperform traditional machine learning models. This is because a crashing trace is relatively long and deep learning approaches naturally have stronger capability than HMM and CRF in learning the patterns hidden in a long sequence. Of all the neural network models, we can also observe that our proposed neural network model (specified as 'our model') exhibits the highest classification performance (*i.e.,* with the highest F1 score). This indicates that, in comparison with the model taking as input the raw machine code, a learning model that takes instruction embedding as the input to a neural network could better capture the dependency hidden between instructions.

From Table 3, we also find that, in comparison with other deep learning models, our model typically demonstrates the performance improvement with only about $1\% \sim 12\%$. However, this does not imply that the utility of our model is only slightly better than those of other neural network models. In our binary analysis task, the crashing traces are relatively long. Using a neural network with even only 0.1% of improvement in precision, for example, we could reduce the amount of false positives or negatives by thousands. Given a long crashing trace containing hundreds of thousands of instructions, our performance improvement indicates a significant reduction in the memory regions mistakenly assigned by neural networks.

Performance of memory alias analysis. In addition to showing the superior performance of our model when conducting memory region identification, we demonstrate the performance of our model in terms of its ability to facilitate VSA with respect to memory alias analysis. In Table 2, we specify the percentage of non-alias pairs that VSA and DEEPVSA track down when given different lengths of crashing traces (400, 800, 3200, 6400, 12800). As we can observe, on average, conventional VSA tracks down about $21.23\% \sim 36.73\%$ non-alias pairs compared with $57.49\% \sim 72.40\%$ of non-alias pairs identified by DEEPVSA. This is more than a 35% increase in non-alias memory reference determination. These results perfectly reflect how conventional VSA generally fails to accurately identify memory regions when execution traces are incomplete. With the assistance of a deep neural network, VSA's ability to perform memory region identification can be enhanced resulting in a significant benefit for memory alias analysis.

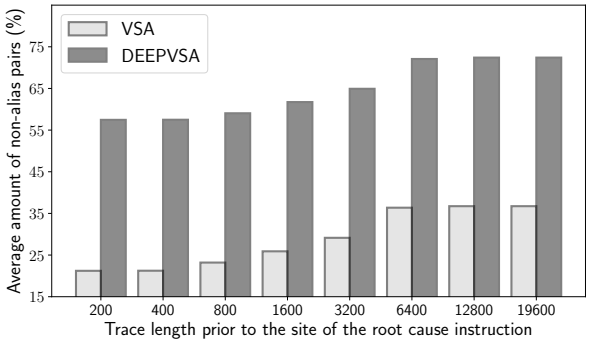


Figure 4: The average amount of non-alias memory pairs vs. the length of instructions retained.

In Table 2 and Figure 4, we further specify the impact of the execution trace length upon the ability to perform alias analysis. We observe that, for some crashing programs (*e.g.,* poppler-0.8.4 and JPegToAvi-1.5), the length of the execution trace stored in the circular buffer influences the capability of VSA and DEEPVSA upon determining memory alias relationships. With the increase in the length of an execution trace, we discover that both VSA and DEEPVSA demonstrate the improvement in their ability to analyze memory alias. This is because both techniques rely upon an execution context to perform alias analysis and a longer execution trace provides them with more abundant contexts. In addition, we observe that the capability of performing alias analysis converges when the length of the instructions (prior to the root cause instruction site) exceeds 12,800. This indicates that, even though DEEPVSA significantly improves VSA's capabilities for alias analysis, it does not completely address alias identification issues for a crashing trace. We believe there is still a room for future exploration in this space, particularly because VSA utilizes both memory regions and offsets to perform alias analysis while DEEPVSA only simply extends VSA with the consideration of coarse-grained memory region differences.

Recall that our DEEPVSA performs alias analysis by using a deep learning approach which cannot predict a memory region access with 100% of accuracy. As a result, along with the influence of a trace length upon alias analysis, we also investigate if inaccurate prediction actually causes DEEPVSA to incorrectly – or mistakenly – track down a non-memory alias pair and thus fail root cause diagnosis. We discover that, similar to conventional VSA, DEEPVSA exhibits zero error rate across all test cases shown in the table. This implies that DEEPVSA does not introduce unsoundness to alias analysis while our proposed deep neural network might mistakenly assign an incorrect region to an instruction. We believe the reason behind this surprising observation is as follows. Given a crashing trace, there is only a tiny portion of memory references that are truly aliased to each other. Even though our deep learning model mistakenly predicts regions for instructions, and DEEPVSA takes that inaccurate prediction as a strong indicator for determining non-alias relationships, the possibility of propagating that error to alias analysis is still extremely low.

Performance of root cause diagnosis. Going beyond specifying the facilitation of alias analysis, Table 2 also illustrates how the analysis of memory alias benefits backward taint analysis and thus the root cause identification. As we can observe from the table, compared with VSA – with which backward taint could successfully pinpoint the root cause of the crash for 27 test cases – DEEPVSA demonstrates superior performance in facilitating root cause diagnosis. We can observe that, with only 3 test cases, DEEPVSA fails to help backward taint to track down the root cause of a software crash. To understand the reasons behind the failure, we look closely into the instructions tainted. With respect to `0verkill-0.16` and `ClamAV-0.93.3`, we note that the failure results from the nature of the hardware which has only 4KB memory storage to record all execution traces. Even if we allocate this entire storage to the crashing process, the hardware is still not able to enclose the instructions pertaining to the root cause of the crash. Regarding the test case `aireplay-ng-1.2beta3`, we discover the crashing program invoked the system call `sys_read` which writes a data chunk to a certain memory region. Since both the size of the data chunk and the address of the memory are specified in registers, which value-set analysis fails to restore, `sys_read` intervenes the propagation of data flow, making the output of DEEPVSA less informative to failure diagnosis.

5 Related Work

This research work mainly focuses on analyzing memory alias in the binary level. Regarding the techniques we employed and the problems we addressed, the lines of works most closely related to our own include machine learning in binary analysis and memory alias analysis for assembly. In this section, we summarize previous studies and discuss their limitation in turn.

Memory alias analysis for assembly. There is a long history of research about analyzing memory alias in binary code. As pioneering research works, Debray *et al.* [21] and Cifuentes *et al.* [16] both propose the same type of technical approaches that compute the values a set of registers can hold at each program point and then use the values held in the registers to determine alias. Considering such techniques determine only the possible values held in each register, but not reason about values across memory operations, Brumley *et al.* propose a logic-based approach which derives all possible alias relationships by finding an over-approximation of the set of values that each memory location and register can hold at each program point [12]. At the high level, this logic-based approach is similar to value set analysis [7, 6, 42] because they both perform value reasoning across memory operations. However, different from the work proposed in [12], value set analysis neither assumes that all memory cells and register locations must be of a single fixed width, nor assumes reads and writes have to be no overlapping. As such, value set analysis is more practical for real-world applications, whereas the logic-based approach [12] has been tested only against simple toy examples.

In a recent research work [19], Cui *et al.* propose a practical debugging system REPT. Technically, it first ignores memory alias in data flow analysis and then utilizes an error correction mechanism to rectify the mistakes caused by memory alias. This approach has demonstrated its effectiveness and efficiency in dealing with some real world crashes. However, as is stated in [19], it inevitably introduces inaccurate analysis results. This is because the proposed correction mechanism does not always catch the occurrence of memory alias, which could sometimes result in incorrectness in root cause diagnosis for a crashing program. In addition, similar to value set analysis, incomplete execution trace imposes the difficulty for REPT in performing alias analysis. In this work, we proposed new deep-learning-based approach which not only inherits the capability of VSA in providing high-fidelity analysis results but more importantly enhances its ability to analyze memory alias.

Machine learning in binary analysis. There is an extensive body of work leveraging machine learning to perform binary analysis. Technically speaking, they can be categorized into two types – conventional machine learning based approaches as well as deep learning based ones.

With respect to the works using conventional machine learning techniques, their research focus is mainly on identifying the function boundary in the binary level. For example, Rosenblum *et al.* utilize conditional random fields to formulate function boundary identification [43] and demonstrate decent performance in terms of pinpointing function entry points. In a recent research work, Bao *et al.* propose ByteWeight [8] which significantly improves the performance for function boundary identification by using weighted

prefix trees.

Regarding the research works adopting deep learning techniques, their research focus includes identifying function boundary [48], pinpointing function type signature [15], tracking down similar binary code [56] and performing memory forensics [49]. Using a bi-directional recurrent neural network, Shin *et al.* improve function boundary identification and achieve a nearly perfect performance with respect to function boundary recognition [48]. Going beyond simply identifying function boundary, Chua *et al.* explore recurrent neural networks with respect to its ability to track down the arguments and types of functions in binary [15]. In recent work, deep learning techniques have also been utilized for binary code similarity detection, in which Xu *et al.* employ Multi-Layer Perception (`MLP`) to encode a control flow graph and then use the encoding to pinpoint vulnerable code fragments [56]. Last but not least, Song *et al.* use a graph based deep learning approach to derive abstract representations for kernel objects so that one could recognize those objects from raw memory dumps efficiently [49].

In this work, we also use machine learning for binary analysis. Different from the aforementioned research, we however focus on leveraging deep learning to improve memory alias identification. Technically speaking, our work is also unique. Unlike the works above, which mostly use an off-the-shelf deep neural architecture, our work introduces a new recurrent neural architecture, which takes the consideration of the data dependency residing in binary code. As is shown in Section 4, our proposed neural network significantly outperforms neural networks largely adopted in other binary analysis tasks.

6 Conclusion

In this paper, we introduce a new deep neural network architecture to facilitate value-set analysis for alias analysis and thus improve the capability in software crash analysis. We show that this new neural architecture can significantly improve value-set analysis with respect to its capability in handling memory alias analysis and benefit data flow analysis in the context of postmortem program analysis. Since the design of our proposed neural network architecture takes into consideration not only the semantics of instructions but also their contexts, it can better capture the dependency within and between the instructions in a sequence of machine codes, making alias identification more effective.

We implemented our proposed technique as `DEEPVSA`– a deep neural network assisted tool for alias analysis and crash diagnosis – and demonstrated its utility using real-world software crashes covering about 1.6 million lines of instructions. We showed that `DEEPVSA` can facilitate the determination of non-alias relationships with no false positives and benefit the diagnosis of program crashes. In addition, we demonstrated that our newly designed neural network outperforms off-the-shelf neural architectures. Following these findings, we safely conclude deep learning can be used for the facilitation of memory alias analysis and root cause diagnosis at the binary level. We expect this work can inspire further advancements in alias analysis and postmortem program analysis through deep neural networks.

Acknowledgement

We would like to thank our shepherd Konrad Rieck and the anonymous reviewers for their helpful feedback. This project was supported in part by NSF grants CNS-1718459, TWC-1409915. In addition, this work was partially supported by the CLTC (Center for Long-Term Cybersecurity), and FORCES (Foundations of Resilient CybErPhysical Systems) which is supported by NSF under the grants CNS-1238959, CNS-1238962, CSN-1239054 and CSN-1239166.

References

[1] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., ET AL. Tensorflow: a system for large-scale machine learning. In *Proceedings of the 11st USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2016).

[2] ARM. Embedded trace macrocell architecture specification. `http://www2.lauterbach.com/pdf/trace_arm_etm.pdf`, 2018.

[3] AUTHORS, A. The deepvsa project website. Anonymous link, 2019.

[4] BALAKRISHNAN, G., GRUIAN, R., REPS, T., AND TEITELBAUM, T. Codesurfer/x86a platform for analyzing x86 executables. In *Proceedings of the 14th International Conference on Compiler Construction (CC)* (2005).

[5] BALAKRISHNAN, G., AND REPS, T. Analyzing memory accesses in x86 executables. In *Proceedings of the 13rd International Conference on Compiler Construction (CC)* (2004).

[6] BALAKRISHNAN, G., AND REPS, T. Wysinwyx: What you see is not what you execute. *ACM Transactions on Programming Languages and Systems* (2010).

[7] BALAKRISHNAN, G., AND REPS, T. W. Analyzing memory accesses in x86 executables. In *Proceedings of the 13th International Conference on Compiler Construction (CC)* (2004).

[8] BAO, T., BURKET, J., WOO, M., TURNER, R., AND BRUMLEY, D. Byteweight: Learning to recognize functions in binary code. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)* (2014).

[9] BENGIO, S., AND HEIGOLD, G. Word embeddings for speech recognition. In *Proceedings of the 15th Annual Conference of the International Speech Communication Association (ISCA)* (2014).

[10] BISHOP, C. M. *Pattern Recognition and Machine Learning*. Springer, 2006.

[11] BOTTOU, L. Large-scale machine learning with stochastic gradient descent. In *Proceedings of the 15th International Conference on Computational Statistics (COMPSTAT)* (2010).

[12] BRUMLEY, D., AND NEWSOME, J. Alias analysis for assembly. In *CMU-CS-06-180* (2006).

[13] CHO, K., VAN MERRIËNBOER, B., GULCEHRE, C., BAHDANAU, D., BOUGARES, F., SCHWENK, H., AND BENGIO, Y. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).

[14] CHOLLET, F., ET AL. Keras. `https://keras.io/`, 2015.

[15] CHUA, Z. L., SHEN, S., SAXENA, P., AND LIANG, Z. Neural nets can learn function type signatures from binaries. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)* (2017).

[16] CIFUENTES, C., AND FRABOULET, A. Intraprocedural static slicing of binary executables. In *Proceedings of 13rd the International Conference on Software Maintenance (ICSM)* (1997).

[17] CLAUSE, J., LI, W., AND ORSO, A. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA)* (2007).

[18] COWAN, C., PU, C., MAIER, D., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., ZHANG, Q., AND HINTON, H. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium (USENIX Security)* (1998).

[19] CUI, W., GE, X., KASIKCI, B., NIU, B., SHARMA, U., WANG, R., AND YUN, I. REPT: Reverse debugging of failures in deployed software. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2018).

[20] CUI, W., PEINADO, M., CHA, S. K., FRATANTONIO, Y., AND KEMERLIS, V. P. Retracer: Triaging crashes by reverse execution from partial memory dumps. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)* (2016).

[21] DEBRAY, S., MUTH, R., AND WEIPPERT, M. Alias analysis of executable code. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (1998).

[22] GERS, F. A., SCHRAUDOLPH, N. N., AND SCHMIDHUBER, J. Learning precise timing with lstm recurrent networks. *Journal of machine learning research* (2002).

[23] GOODFELLOW, I., BENGIO, Y., COURVILLE, A., AND BENGIO, Y. *Deep learning*. MIT press Cambridge, 2016.

[24] GRAVES, A., MOHAMED, A.-R., AND HINTON, G. Speech recognition with deep recurrent neural networks. In *Proceedings of the 38th IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (2013).

[25] GU, X., ZHANG, H., ZHANG, D., AND KIM, S. Deep api learning. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)* (2016).

[26] HOCHREITER, S. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* (1998).

[27] INTEL. Intel processor trace tools. `https://software.intel.com/en-us/node/721535`, 2013.

[28] IOFFE, S., AND SZEGEDY, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the International conference on machine learning (ICML)* (2015).

[29] ITO, Y. Representation of functions by superpositions of a step or sigmoid function and their applications to neural network theory. *Neural Networks* (1991).

[30] KASIKCI, B., CUI, W., GE, X., AND NIU, B. Lazy diagnosis of in-production concurrency bugs. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)* (2017).

[31] KINGMA, D. P., AND BA, J. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representation (ICLR)* (2015).

[32] KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 36th Annual Conference on Neural Information Processing Systems (NeurIPS)* (2012).

[33] LECUN, Y., BENGIO, Y., AND HINTON, G. Deep learning. *nature* (2015).

[34] LI, M., ZHANG, T., CHEN, Y., AND SMOLA, A. J. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)* (2014).

[35] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., ET AL. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 26th ACM SIGPLAN Conference on Programming language design and implementation (PLDI)* (2005).

[36] MICROSOFT. /safeseh (safe exception handlers). `http://msdn2.microsoft.com/en-us/library/9a89h429.aspx`, 2003.

[37] MICROSOFT. Time travel debugging - record a trace. `https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/time-travel-debugging-record`, 2017.

[38] MOZILLA. rr: lightweight recording & deterministic debugging. `https://rr-project.org/`, 2019.

[39] NEWSOME, J., AND SONG, D. X. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *Proceedings of the 11st Network and Distributed System Security Symposium (NDSS)* (2005).

[40] OHMANN, P., BROOKS, A., D'ANTONI, L., AND LIBLIT, B. Control-flow recovery from partial failure reports. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2017).

[41] OHMANN, P., AND LIBLIT, B. Lightweight control-flow instrumentation and postmortem analysis in support of debugging. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2013).

[42] REPS, T. W., AND BALAKRISHNAN, G. Improved memory-access analysis for x86 executables. In *Proceedings of the 17th International Conference on Compiler Construction (CC)* (2008).

[43] ROSENBLUM, N. E., ZHU, X., MILLER, B. P., AND HUNT, K. Learning to analyze binary computer code. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI)* (2008).

[44] RUCK, D. W., ROGERS, S. K., KABRISKY, M., OXLEY, M. E., AND SUTER, B. W. The multilayer perceptron as an approximation to a bayes optimal discriminant function. *IEEE Transactions on Neural Networks* (1990).

[45] RUDER, S. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747* (2016).

[46] SCHUSTER, M., AND PALIWAL, K. K. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing* (1997).

[47] SECURITY, O. Offensive security exploit database archive. `https://www.exploit-db.com/`, 2009.

[48] SHIN, E. C. R., SONG, D., AND MOAZZEZI, R. Recognizing functions in binaries with neural networks. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security)* (2015).

[49] SONG, W., YIN, H., LIU, C., AND SONG, D. Deepmem: Learning graph neural network models for fast and robust memory forensic analysis. In *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2018).

[50] SUTSKEVER, I., VINYALS, O., AND LE, Q. V. Sequence to sequence learning with neural networks. In *Proceedings of the 38th Annual Conference on Neural Information Processing Systems (NeurIPS)* (2014).

[51] TEAM, P. Address space layout randomization (aslr). `http://pax.grsecurity.net/docs/aslr.txt`, 2003.

[52] TIELEMAN, T., AND HINTON, G. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning* (2012).

[53] VAN DE VEN, A., AND MOLNAR, I. Exec shield. `http://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf`, 2004.

[54] WIKIPEDIA CONTRIBUTORS. Hyperbolic function — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Hyperbolic_function&oldid=866654186`, 2018.

[55] XU, J., MU, D., XING, X., LIU, P., CHEN, P., AND MAO, B. Postmortem program analysis with hardware-enhanced post-crash artifacts. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)* (2017).

[56] XU, X., LIU, C., FENG, Q., YIN, H., SONG, L., AND SONG, D. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2017).

Index	CVE/EDB	Trace Len.	Statistics			
			Global	Heap	Stack	Other
1	2013-0221	228	6	14	106	4613
2	2013-0222	285	155	468	10895	844
3	2013-0223	341	27	74	5103	4301
4	2013-2028	348	24	2318	5268	715
5	2002-1496	136	141	3071	0	8723
6	2004-1271	3391	64	0	7031	3884
7	2009-2285	28387	77	23596	4287	1786
8	2004-1297	110	341	1641	6086	2326
9	NA-30142	419	357	2379	6523	1584
10	2016-7445	236	20	195	5544	3829
11	NA-38616	680	62	11332	639	11535
12	2008-2950	672	5	1632	7196	2195
13	2009-3050	704	151	1114	6572	1924
14	2005-3862	54203	15	4612	15543	7372
15	NA-890	2966	32	88	6369	4184
16	2004-0597	4107	18	365	8368	3818
17	2005-4807	15953	180	8584	4973	2514
18	2015-5895	1446	27	1642	6776	1840
19	2004-1289	13264	807	6503	7736	3233
20	2004-2167	1720	150	1492	1729	311
21	2009-5018	76603	75	175	26354	18975
22	2004-1257	56018	862	10192	31420	3333
23	2004-1288	69184	1189	0	24430	25256
24	NA-23523	1544	95	2833	7107	343
25	2017-5854	571	90	1382	10698	753
26	2004-1287	1212	660	5593	5948	332
27	2007-4060	4124	55	596	7438	2072
28	2009-3586	8612	80	1317	9854	2375
29	NA-17611	3384	2724	0	3407	367
30	2004-1120	2011	23	5060	7345	1655
31	NA-33251	16672	1	33168	474	25
32	2004-1275	41275	25	12039	10383	683
33	2012-4409	321	29	216	4769	2116
34	2016-2563	3586	1035	2349	8450	869
35	2006-2465	31806	10	4564	16304	5227
36	2010-2891	7611	0	715	14626	1764
37	2004-1279	29371	9	266	11924	10989
38	2014-8322	329	84	138	6908	2738
39	2008-5314	200000	0	0	118271	85077
40	2006-2971	200000	37208	0	4652	158140
Total	-	883830	-	-	-	-

Table 4: The detail of crashing programs. The CVE/EDB column specifies the vulnerability identifiers. In this column, NA indicates those vulnerabilities with an EDB identifier but not a CVE Identifier). "Trace Len" describes the number of instructions from the root cause site to the crashing site. The numbers under "statistics" indicate the amount of memory dereferences across 4 disjoint memory regions.

		Global	Heap	Stack	Other
Precision	HMM	65.69%	47.26%	71.38%	86.28%
	CRF	15.87%	33.57%	63.30%	73.09%
	Bi-RNN	95.53%	72.88%	94.18%	97.67%
	Bi-GRU	91.97%	74.21%	94.62%	97.44%
	Bi-LSTM	94.23%	77.12%	94.84%	98.57%
	Our Model	**98.30%**	**94.91%**	**98.83%**	**99.40%**
Recall	HMM	63.25%	29.82%	83.08%	83.72%
	CRF	8.17%	13.02%	55.56%	88.20%
	Bi-RNN	79.75%	83.56%	96.39%	95.07%
	Bi-GRU	89.89%	81.28%	96.93%	95.22%
	Bi-LSTM	88.79%	88.20%	97.56%	95.58%
	Our Model	**88.64%**	**95.71%**	**98.65%**	**99.53%**
F1 Score	HMM	64.49%	36.57%	76.79%	84.98%
	CRF	10.79%	18.76%	59.18%	79.94%
	Bi-RNN	86.93%	77.86%	95.27%	96.35%
	Bi-GRU	90.92%	77.59%	95.76%	96.31%
	Bi-LSTM	91.43%	82.29%	96.18%	97.06%
	Our Model	**93.22%**	**95.31%**	**98.74%**	**99.46%**

Table 5: The overall performance of different machine learning models trained with the execution traces without the elimination of common instruction sequences.

Appendix

Detail of crashing programs and their crashing trace. As is described in Section 4, for our evaluation, we select 40 crashing traces corresponding to 38 distinct versions of vulnerable software. Table 4 describes the detail of these selected programs, including the CVE/EDB identifiers tied to these programs as well as the length of their crashing traces. In addition, the table shows the memory access behaviors of each program. They are retrieved from the execution trace which combines the trace from the root cause site to the crashing site and its 19,200 prefix instructions. We have already made all of the selected programs publicly available. They can be downloaded from our project website [3]. It should be noted that Table 2 and 4 share the same index.

Learning model performance without the elimination of commonly-shared data. As is specified in Section 4, in order to avoid the risk of using the same data to train and test a learning model, we eliminate – from the training data set – the instruction sequences commonly shared by both our training and testing sets, and show the performance of the learning models trained on non-overlapping data set. As a comparison, we also conduct an experiment in which we do not eliminate the 14.02% of shared data from the training set, and train all the learning models over the overlapping data set. In Table 5, we depict the model performance under this setting. As we can observe from the table, the model performance is actually comparable regardless whether we trim off the commonly shared instruction sequences. This implies that the shared data has nearly no impact upon model classification and thus memory alias analysis.

CONFIRM: Evaluating Compatibility and Relevance of Control-flow Integrity Protections for Modern Software

Xiaoyang Xu
University of Texas at Dallas

Masoud Ghaffarinia*
University of Texas at Dallas

Wenhao Wang*
University of Texas at Dallas

Kevin W. Hamlen
University of Texas at Dallas

Zhiqiang Lin
Ohio State University

Abstract

CONFIRM (CONtrol-Flow Integrity Relevance Metrics) is a new evaluation methodology and microbenchmarking suite for assessing compatibility, applicability, and relevance of *control-flow integrity* (CFI) protections for preserving the intended semantics of software while protecting it from abuse. Although CFI has become a mainstay of protecting certain classes of software from code-reuse attacks, and continues to be improved by ongoing research, its ability to preserve intended program functionalities (*semantic transparency*) of diverse, mainstream software products has been under-studied in the literature. This is in part because although CFI solutions are evaluated in terms of performance and security, there remains no standard regimen for assessing compatibility. Researchers must often therefore resort to anecdotal assessments, consisting of tests on homogeneous software collections with limited variety (e.g., GNU Coreutils), or on CPU benchmarks (e.g., SPEC) whose limited code features are not representative of large, mainstream software products.

Reevaluation of CFI solutions using CONFIRM reveals that there remain significant unsolved challenges in securing many large classes of software products with CFI, including software for market-dominant OSes (e.g., Windows) and code employing certain ubiquitous coding idioms (e.g., event-driven callbacks and exceptions). An estimated 47% of CFI-relevant code features with high compatibility impact remain incompletely supported by existing CFI algorithms, or receive weakened controls that leave prevalent threats unaddressed (e.g., return-oriented programming attacks). Discussion of these open problems highlights issues that future research must address to bridge these important gaps between CFI theory and practice.

1 Introduction

Control-flow integrity (CFI) [1] (supported by vtable protection [29] and/or software fault isolation [73]), has emerged as one of the strongest known defenses against modern control-flow hijacking attacks, including return-oriented programming (ROP) [60] and other code-reuse attacks. These attacks trigger dataflow vulnerabilities (e.g., buffer overflows) to manipulate control data (e.g., return addresses) to hijack victim software. By restricting program execution to a set of legitimate control-flow targets at runtime, CFI can mitigate many of these threats.

Inspired by the initial CFI work [1], there has been prolific new research on CFI in recent years, mainly aimed at improving performance, enforcing richer policies, obtaining higher assurance of policy-compliance, and protecting against more subtle and sophisticated attacks. For example, between 2015–2018 over 25 new CFI algorithms appeared in the top four applied security conferences alone. These new frameworks are generally evaluated and compared in terms of performance and security. Performance overhead is commonly evaluated in terms of the CPU benchmark suites (e.g., SPEC), and security is often assessed using the RIPE test suite [80] or with manually crafted proof-of-concept attacks (e.g., COOP [62]). For example, a recent survey systematically compared various CFI mechanisms against these metrics for precision, security, and performance [13].

While this attention to performance and security has stimulated rapid gains in the ability of CFI solutions to efficiently enforce powerful, precise security policies, less attention has been devoted to systematically examining which general classes of software can receive CFI protection without suffering compatibility problems. Historically, CFI research has struggled to bridge the gap between theory and practice (cf., [84]) because code hardening transformations inevitably run at least some risk of corrupting desired, policy-permitted program functionalities. For example, introspective programs that read their own code bytes at runtime (e.g., many VMs, JIT compilers, hot-patchers, and dynamic linkers) can break after their code bytes have been modified or relocated by CFI.

Compatibility issues of this sort have dangerous security ramifications if they prevent protection of software needed in mission-critical contexts, or if the protections must be weak-

*These authors contributed equally to this work.

ened in order to achieve compatibility. For example, due in part to potential incompatibilities related to *return address introspection* (wherein some callees read return addresses as arguments) the three most widely deployed compiler-based CFI solutions (LLVM-CFI [69], GCC-VTV [69], and Microsoft Visual Studio MCFG [66]) all presently leave return addresses unprotected, potentially leaving code vulnerable to ROP attacks—the most prevalent form of code-reuse.

Understanding these compatibility limitations, including their impacts on real-world software performance and security, requires a new suite of CFI functional tests with substantially different characteristics than benchmarks typically used to assess compiler or hardware performance. In particular, CFI relevance and effectiveness is typically constrained by the nature and complexity of the target program's *control-flow paths* and *control data dependencies*. Such complexities are not well represented by SPEC benchmarks, which are designed to exercise CPU computational units using only simple control-flow graphs, or by utility suites (e.g., GNU Coreutils) that were all written in a fairly homogeneous programming style for a limited set of compilers, and that use a very limited set of standard libraries chosen for exceptionally high cross-compatibility.

To better understand the compatibility and applicability limitations of modern CFI solutions on diverse, modern software products, and to identify the coding idioms and features that constitute the greatest barriers to more widespread CFI adoption, we present ConFIRM (CONtrol-Flow Integrity Relevance Metrics), a new suite of CFI tests designed to exhibit code features most relevant to CFI evaluation.[1] Each test is designed to exhibit one or more control-flow features that CFI solutions must guard in order to enforce integrity, that are found in a large number of commodity software products, but that pose potential problems for CFI implementations.

It is infeasible to capture in a single test set the full diversity of modern software, which embodies myriad coding styles, build processes (e.g., languages, compilers, optimizers, obfuscators, etc.), and quality levels. We therefore submit ConFIRM as an extensible baseline for testing CFI compatibility, consisting of code features drawn from experiences building and evaluating CFI and randomization systems for several architectures, including Linux, Windows, Intel x86/x64, and ARM32 in academia and industry [7, 33, 45, 47, 75, 77–79].

Our work is envisioned as having the following qualitative impacts: (1) CFI designers (e.g., compiler developers) can use ConFIRM to detect compatibility flaws in their designs that are currently hard to anticipate prior to full scale productization. This can lower the currently steep barrier between prototype and distributable product. (2) Defenders (e.g., developers of secure software) can use ConFIRM to better evaluate code-reuse defenses, in order to avoid false senses of security. (3) The research community can use ConFIRM to identify and prioritize missing protections as important open problems worthy of future investigation.

We used ConFIRM to reevaluate 12 publicly available CFI implementations published in the open literature. The results show that about 47% of solution-test pairs exhibit incompatible or insecure operation for code features needed to support mainstream software products, and a *cross-thread stack-smashing* attack defeats all tested CFI defenses. Microbenchmarking additionally reveals some performance/compatibility trade-offs not revealed by purely CPU-based benchmarking.

In summary, our contributions include the following:

- We present ConFIRM, the first testing suite designed specifically to test compatibility characteristics relevant to control-flow security hardening evaluation.
- A set of 20 code features and coding idioms are identified, that are widely found in deployed, commodity software products, and that pose compatibility, performance, or security challenges for modern CFI solutions.
- Evaluation of 12 CFI implementations using ConFIRM reveals that existing CFI implementations are compatible with only about half of code features and coding idioms needed for broad compatibility, and that microbenchmarking using ConFIRM reveals performance trade-offs not exhibited by SPEC benchmarks.
- Discussion and analysis of these results highlights significant unsolved obstacles to realizing CFI protections for widely deployed, mainstream, commodity products.

Section 2 begins with a summary of technical CFI attack and defense details important for understanding the evaluation approach. Section 3 next presents ConFIRM's evaluation metrics in detail, including a rationale behind why each metric was chosen, and how it impacts potential defense solutions; and Section 4 describes implementation of the resulting tests. Section 5 reports our evaluation of CFI solutions using ConFIRM and discusses significant findings. Finally, Section 6 describes related work and Section 7 concludes.

2 Background

CFI defenses first emerged from an arms race against early code-injection attacks, which exploit memory corruptions to inject and execute malicious code. To thwart these malicious code-injections, hardware and OS developers introduced Data Execution Prevention (DEP), which blocks execution of injected code. Adversaries proceeded to bypass DEP with "return-to-libc" attacks, which redirect control to existing, abusable code fragments (often in the C standard libraries) without introducing attacker-supplied code. In response, defenders introduced Address Space Layout Randomization (ASLR), which randomizes code layout to frustrate its abuse. DEP and ASLR motivated adversaries to craft even

[1] https://github.com/SoftwareLanguagesSecurityLab/ConFIRM

more elaborate attacks, including ROP and Jump-Oriented Programming (JOP) [11], which locate, chain, and execute short instruction sequences (gadgets) of benign code to implement malicious payloads.

CFI emerged as a more comprehensive and principled defense against this malicious code-reuse. Most realizations consist of two main phases: (1) A program-specific *control-flow policy* is first formalized as a (possibly dynamic) control-flow graph (CFG) that whitelists the code's permissible control-flow transfers. (2) To constrain all control flows to the CFG, the program code is instrumented with guard code at all computed (e.g., indirect) control-flow transfer sites. The guard code decides at runtime whether each impending transfer satisfies the policy, and blocks it if not. The guards are designed to be uncircumventable by confronting attackers with a chicken-and-egg problem: To circumvent a guard, an attack must first hijack a control transfer; but since all control transfers are guarded, hijacking a control transfer requires first circumventing a guard.

Both CFI phases can be source-aware (implemented as a source-to-source transformation, or introduced during compilation), or source-free (implemented as a binary-to-binary transformation). Source-aware solutions typically benefit from source-level information to derive more precise policies, and can often perform more optimization to achieve better performance. Examples include WIT [5], NaCl [81], CFL [11], MIP [48], MCFI [49], RockJIT [50], Forward CFI [69], CCFI [42], πCFI [51], MCFG [66] CFIXX [14] and μCFI [35]. In contrast, source-free solutions are potentially applicable to a wider domain of software products (e.g., closed-source), and have a more flexible deployment model (e.g., consumer-side enforcement without developer assistance). These include XFI [26], Reins [78], STIR [77], CCFIR [84], bin-CFI [87], BinCC [74], Lockdown [54], TypeArmor [72], OCFI [45], OFI [75] and τCFI [47].

The advent of CFI is a significant step forward for defenders, but was not the end of the arms race. In particular, each CFI phase introduces potential loopholes for attackers to exploit. First, it is not always clear which policy should be enforced to fully protect the code. Production software often includes complex control-flow structures, such as those introduced by object-oriented programming (OOP) idioms, from which it is difficult (even undecidable) to derive a CFG that precisely captures the policy desired by human developers and users. Second, the instrumentation phase must take care not to introduce guard code whose decision procedures constitute unacceptably slow runtime computations [34]. This often results in an enforcement that imprecisely approximates the policy. Attackers have taken advantage of these loopholes with ever more sophisticated attacks, including Counterfeit Object Oriented Programming (COOP) [62], Control Jujutsu [28], and Control-Flow Bending [15].

These weaknesses and threats have inspired an array of new and improved CFI algorithms and supporting technologies in recent years. For example, to address loopholes associated with OOP, *vtable protections* prevent or detect virtual method table corruption at or before control-flow transfers that depend on method pointers. Source-aware vtable protections include GNU VTV [68], CPI [40], SAFEDISPATCH [37], Readactor++ [19], and VTrust [82]; whereas source-free instantiations include T-VIP [29], VTint [83], and VfGuard [58].

However, while the security and performance trade-offs of various CFI solutions have remained actively tracked and studied by defenders throughout the arms race, attackers are increasingly taking advantage of CFI compatibility limitations to exploit unprotected software, thereby avoiding CFI defenses entirely. For example, 88% of CFI defenses cited herein have only been realized for Linux software, but over 95% of desktops worldwide are non-Linux.[2] These include many mission-critical systems, including over 75% of control systems in the U.S. [39], and storage repositories for top secret military data [53]. None of the top 10 vulnerabilities exploited by cybercriminals in 2017 target Linux software [25].

While there is a hope that small-scale prototyping will result in principles and approaches that eventually scale to more architectures and larger software products, follow-on works that attempt to bridge this gap routinely face significant unforeseen roadblocks. We believe many of these obstacles remain unforeseen because of the difficulty of isolating and studying many of the problematic software features lurking within large, commodity products, which are not well represented in open-source codes commonly available for study by researchers during prototyping.

The goal of this research is therefore to describe and analyze a significant collection of code features that are routinely found in large software products, but that pose challenges to effective CFI enforcement; and to make available a suite of CFI test programs that exhibit each of these features on a small scale amenable to prototype development. The next section discusses this feature set in detail.

3 Compatibility Metrics

To measure compatibility of CFI mechanisms, we propose a set of metrics that each includes one or more code features from either C/C++ source code or compiled assembly code. We derived this feature set by attempting to apply many CFI solutions to large software products, then manually testing the functionalities of the resulting hardened software for correctness, and finally debugging each broken functionality step-wise at the assembly level to determine what caused the hardened code to fail. Since many failures manifest as subtle forms of register or memory corruption that only cause the program to crash or malfunction long after the failed operation completes, this debugging constitutes many hundreds

[2] http://gs.statcounter.com/os-market-share/desktop/worldwide

Table 1: CONFIRM compatibility metrics

Compatibility metric	Real-world software examples
Function Pointers	7-Zip, Adobe Reader, Apache, Calculator, Chrome, Dropbox, Firefox, JVM, Notepad, MS Paint, MS PowerPoint, PotPlayer, PowerShell, PuTTY, Skype, TeXstudio, UPX, Visual Studio, Windows Defender, WinSCP
Callbacks	7-Zip, Adobe Reader, Apache, Calculator, Chrome, Dropbox, Firefox, JVM, Notepad, MS Paint, MS PowerPoint, PotPlayer, PowerShell, PuTTY, TeXstudio, Visual Studio, Windows Defender, WinSCP
Dynamic Linking	7-Zip, Adobe Reader, Apache, Calculator, Chrome, Dropbox, Firefox, JVM, Notepad, MS Paint, MS PowerPoint, PotPlayer, PowerShell, PuTTY, Skype, TeXstudio, UPX, Visual Studio, Windows Defender, WinSCP
Delay-Loading	Adobe Reader, Calculator, Chrome, Firefox, JVM, MS Paint, MS PowerPoint, PotPlayer, Visual Studio, WinSCP
Exporting/Importing Data	7-Zip, Apache, Calculator, Chrome, Dropbox, Firefox, MS Paint, MS PowerPoint, PowerShell, TeXstudio, UPX, Visual Studio
Virtual Functions	7-Zip, Adobe Reader, Calculator, Chrome, Dropbox, Firefox, JVM, Notepad, MS Paint, MS PowerPoint, PotPlayer, PowerShell, PuTTY, TeXstudio, Visual Studio, Windows Defender, WinSCP
CODE-COOP Attack	Programs built on GTK+ or Microsoft COM can pass objects to trusted modules as arguments.
Tail Calls	Mainstream compilers provide options for tail call optimization. e.g. `/O2` in MSVC, `-O2` in GCC, and `-O2` in LLVM.
Switch-Case Statements	7-Zip, Adobe Reader, Apache, Calculator, Chrome, Dropbox, Firefox, JVM, MS Paint, MS PowerPoint, PotPlayer, PuTTY, TeXstudio, Visual Studio, WinSCP
Returns	Every benign program has returns.
Unmatched Call/Return Pairs	Adobe Reader, Apache, Chrome, Firefox, JVM, MS PowerPoint, Visual Studio
Exceptions	7-Zip, Adobe Reader, Apache, Calculator, Chrome, Dropbox, Firefox, JVM, MS Paint, MS PowerPoint, PotPlayer, PowerShell, PuTTY, Skype, TeXstudio, Visual Studio, Windows Defender, WinSCP
Calling Conventions	Every program adopts one or more calling convention.
Multithreading	7-Zip, Adobe Reader, Apache, Calculator, Chrome, Dropbox, Firefox, JVM, Notepad, MS Paint, MS PowerPoint, PotPlayer, PowerShell, PuTTY, Skype, TeXstudio, UPX, Visual Studio, Windows Defender, WinSCP
TLS Callbacks	Adobe Reader, Chrome, Firefox, MS Paint, TeXstudio, UPX
Position-Independent Code	7-Zip, Adobe Reader, Apache, Calculator, Chrome, Dropbox, Firefox, JVM, Notepad, MS Paint, MS PowerPoint, PotPlayer, PowerShell, PuTTY, Skype, TeXstudio, UPX, Visual Studio, Windows Defender, WinSCP
Memory Protection	7-Zip, Adobe Reader, Apache, Chrome, Dropbox, Firefox, MS PowerPoint, PotPlayer, TeXstudio, Visual Studio, Windows Defender, WinSCP
JIT Compiler	Adobe Flash, Chrome, Dropbox, Firefox, JVM, MS PowerPoint, PotPlayer, PowerShell, Skype, Visual Studio, WinSCP
Self-Unpacking	Programs decompressed by self-extractors (e.g., UPX, NSIS).
Windows API Hooking	Microsoft Office family software, including MS Excel, MS PowerPoint, MS PowerPoint, etc.

Table 2: Source code compiled to indirect call

Source code	Assembly code
1 `void foo() { return; }`	
2 `void bar() { return; }`	
3 `void main() {`	
4 `  void (*fptr)();`	1 `...`
5 `  int n = input();`	2 `call _input`
6 `  if (n)`	3 `test eax, eax`
7 `  fptr = foo;`	4 `mov edx, offset_foo`
8 `  else`	5 `mov ecx, offset_bar`
9 `  fptr = bar;`	6 `cmovnz ecx, edx`
10 `  fptr();`	7 `call ecx`
11 `}`	8 `...`

of person-hours amassed over several years of development experience involving CFI-protected software.

Table 1 presents the resulting list of code features organized into one row for each root cause of failure. Column two additionally lists some widely available, commodity software products where each of these features can be observed in non-malicious software in the wild. This demonstrates that each feature is representative of real-world software functionalities that must be preserved by CFI implementations in order for their protections to be usable and relevant in contexts that deploy these and similar products.

3.1 Indirect Branches

We first discuss compatibility metrics related to the code feature of greatest relevance to most CFI works: indirect branches. Indirect branches are control-flow transfers whose destination addresses are computed at runtime—via pointer arithmetic and/or memory-reads. Such transfers tend to be of high interest to attackers, since computed destinations are more prone to manipulation. CFI defenses therefore guard indirect branches to ensure that they target permissible destinations at runtime. Indirect branches are commonly categorized into three classes: indirect calls, indirect jumps, and returns.

Table 2 shows a simple example of source code being compiled to an indirect call. The function called at source line 5 depends on user input. This prevents the compiler from generating a direct branch that targets a fixed memory address at compile time. Instead, the compiler generates a register-indirect call (assembly line 7) whose target is computed at runtime. While this is one common example of how indirect branches arise, in practice they are a result of many different programming idioms, discussed below.

Function Pointers. Calls through function pointers typically compile to indirect calls. For example, using gcc with the `-O2` option generates register-indirect calls for function pointers, and MSVC does so by default.

Callbacks. Event-driven programs frequently pass function pointers to external modules or the OS, which the receiving code later dereferences and calls in response to an event. These callback pointers are generally implemented by using function pointers in C, or as method references in C++. Callbacks can pose special problems for CFI, since the call site is not within the module that generated the pointer. If the call site is within a module that cannot easily be modified (e.g., the OS kernel), it must be protected in some other way, such as by sanitizing and securing the pointer before it is passed.

Dynamic Linking. Dynamically linked shared libraries reduce program size and improve locality. But dynamic linking has been a challenge for CFI compatibility because CFG edges that span modules may be unavailable statically.

In Windows, *dynamically linked libraries* (DLLs) can be loaded into memory at load time or runtime. In load-time dynamic linking, a function call from a module to an exported DLL function is usually compiled to a memory-indirect call targeting an address stored in the module's *import address table* (IAT). But if this function is called more than once, the compiler first moves the target address to a register, and then generates register-indirect calls to improve execution performance. In run-time dynamic linking, a module calls APIs, such as `LoadLibrary()`, to load the DLL at runtime. When loaded into memory, the module calls the `GetProcAddress()` API to retrieve the address of the exported function, and then calls the exported function using the function pointer returned by `GetProcAddress()`.

Additionally, MSVC (since version 6.0) provides linker support for delay-loaded DLLs using the `/DELAYLOAD` linker option. These DLLs are not loaded into memory until one of their exported functions is invoked.

In Linux, a module calls functions exported by a shared library by calling a stub in its *procedure linkage table* (PLT). Each stub contains a memory-indirect jump whose target depends on the writable, lazy-bound *global offset table* (GOT). As in Windows, an application can also load a module at runtime using function `dlopen()`, and retrieve an exported symbol using function `dlsym()`.

Supporting dynamic and delay-load linkage is further complicated by the fact that shared libraries can also export data pointers within their export tables in both Linux and Windows. CFI solutions that modify export tables must usually treat code and data pointers differently, and must therefore somehow distinguish the two types to avoid data corruptions.

Virtual Functions. Polymorphism is a key feature of OOP languages, such as C++. Virtual functions are used to support runtime polymorphism, and are implemented by C++ compilers using a form of late binding embodied as *virtual tables* (vtables). The tables are populated by code pointers to virtual function bodies. When an object calls a virtual function, it indexes its vtable by a function-specific constant, and flows control to the memory address read from the table. At the assembly level, this manifests as a memory-indirect call. The ubiquity and complexity of this process has made vtable hijacking a favorite exploit strategy of attackers.

Some CFI and vtable protections address vtable hijacking threats by guarding call sites that read vtables, thereby detecting potential vtable corruption at time-of-use. Others seek to protect vtable integrity directly by guarding writes to them. However, both strategies are potentially susceptible to COOP [62] and CODE-COOP [75] attacks, which replace one vtable with another that is legal but is not the one the original code intended to call. The defense problem is further complicated by the fact that many large classes of software (e.g., GTK+ and Microsoft COM) rely upon dynamically generated vtables. CFI solutions that write-protect vtables or whose guards check against a static list of permitted vtables are incompatible with such software.

Tail Calls. Modern C/C++ compilers can optimize tail-calls by replacing them with jumps. Row 8 of Table 1 lists relevant compiler options. With these options, callees can return directly to ancestors of their callers in the call graph, rather than to their callers. These mismatched call/return pairs affect precision of some CFG recovery algorithms.

Switch-case Statements. Many C/C++ compilers optimize switch-case statements via a static dispatch table populated with pointers to case-blocks. When the switch is executed, it calculates a dispatch table index, fetches the indexed code pointer, and jumps to the correct case-block. This introduces memory-indirect jumps that refer to code pointers not contained in any vtable, and that do not point to function boundaries. CFI solutions that compare code pointers to a whitelist of function boundaries can therefore cause the switch-case code to malfunction. Solutions that permit unrestricted indirect jumps within each local function risk unsafety, since large functions can contain abusable gadgets.

Returns. Nearly every benign program has returns. Unlike indirect branches whose target addresses are stored in registers or non-writable data sections, return instructions read their destination addresses from the stack. Since stacks are typically writable, this makes return addresses prime targets for malicious corruption.

On Intel-based CISC architectures, return instructions have one of the shortest encodings (1 byte), complicating the efforts of source-free solutions to replace them in-line with secured equivalent instruction sequences. Additionally, many hardware architectures heavily optimize the behavior of returns (e.g., via speculative execution powered by shadow stacks for call/return matching). Source-aware CFI solutions that replace returns with some other instruction sequence can therefore face stiff performance penalties by losing these optimization advantages.

Unmatched call/return Pairs. Control-flow transfer mechanisms, including exceptions and setjmp/longjmp, can yield flows in which the relation between executed call instructions

and executed return instructions is not one-to-one. For example, exception-handling implementations often pop stack frames from multiple calls, followed by a single return to the parent of the popped call chain. Shadow stack defenses that are implemented based on traditional call/return matching may be incompatible with such mechanisms.

3.2 Other Metrics

While indirect branches tend to be the primary code feature of interest to CFI attacks and defenses, there are many other code features that can also pose control-flow security problems, or that can become inadvertently corrupted by CFI code transformation algorithms, and that therefore pose compatibility limitations. Some important examples are discussed below.

Multithreading. With the rise of multicore hardware, multithreading has become a centerpiece of software efficiency. Unfortunately, concurrent code execution poses some serious safety problems for many CFI algorithms.

For example, in order to take advantage of hardware call-return optimization (see §3.1), most CFI algorithms produce code containing guarded return instructions. The guards check the return address before executing the return. However, on parallelized architectures with flat memory spaces, this is unsafe because any thread can potentially write to any other (concurrently executing) thread's return address at any time. This introduces a TOCTOU vulnerability in which an attacker-manipulated thread corrupts a victim thread's return address after the victim thread's guard code has checked it but before the guarded return executes. We term this a cross-thread stack-smashing attack. Since nearly all modern architectures combine concurrency, flat memory spaces, and returns, this leaves almost all CFI solutions either inapplicable, unsafe, or unacceptably inefficient for a large percentage of modern production software.

Position-Independent Code. *Position-independent code* (PIC) is designed to be relocatable after it is statically generated, and is a standard practice in the creation of shared libraries. Unfortunately, the mechanisms that implement PIC often prove brittle to code transformations commonly employed for source-free CFI enforcement. For example, PIC often achieves its position independence by dynamically computing its own virtual memory address (e.g., by performing a call to itself and reading the pushed return address from the stack), and then performing pointer arithmetic to locate other code or data at fixed offsets relative to itself. This procedure assumes that the relative positions of PIC code and data are invariant even if the base address of the PIC block changes.

However, CFI transforms typically violate this assumption by introducing guard code that changes the sizes of code blocks, and therefore their relative positions. To solve this, PIC-compatible CFI solutions must detect the introspection and pointer arithmetic operations that implement PIC and adjust them to compute corrected pointer values. Since there are typically an unlimited number of ways to perform these computations at both the source and native code levels, CFI detection of these computations is inevitably heuristic, allowing some PIC instantiations to malfunction.

Exceptions. Exception raising and handling is a mainstay of modern software design, but introduces control-flow patterns that can be problematic for CFI policy inference and enforcement. Object-oriented languages, such as C++, boast first-class exception machinery, whereas standard C programs typically realize exceptional control-flows with gotos, longjumps, and signals. In Linux, compilers (e.g., gcc) implement C++ exception handling in a table-driven approach. The compiler statically generates read-only tables that hold exception-handling information. For instance, gcc produces a `gcc_except_table` comprised of *language-specific data areas* (LSDAs). Each LSDA contains various exception-related information, including pointers to exception handlers.

In Windows, *structured exception handling* (SEH) extends the standard C language with first-class support for both hardware and software exceptions. SEH uses stack-based exception nodes, wherein exception handlers form a linked list on the stack, and the list head is stored in the *thread information block* (TIB). Whenever an exception occurs, the OS fetches the list head and walks through the SEH list to find a suitable handler for the thrown exception. Without proper protection, these exception handlers on the stack can potentially be overwritten by an attacker. By triggering an exception, the attacker can then redirect the control-flow to arbitrary code. CFI protection against these SEH attacks is complicated by the fact that code outside the vulnerable module (e.g., in the OS and/or system libraries) uses pointer arithmetic to fetch, decode, and call these pointers during exception handling. Thus, suitable protections must typically span multiple modules, and perhaps the OS kernel.

From Windows XP onward, applications have additionally leveraged *vectored exception handling* (VEH). Unlike SEH, VEH is not stack-based; applications register a global handler chain for VEH exceptions with the OS, and these handlers are invoked by the OS by interrupting the application's current execution, no matter where the exception occurs within a frame.

There are at least two features of VEH that are potentially exploitable by attackers. First, to register a vectored exception handler, the application calls an API `AddVecoredExceptionHandler()` that accepts a callback function pointer parameter that points to the handler code. Securing this pointer requires some form of inter-module callback protection.

Second, the VEH handler-chain data structure is stored in the application's writable heap memory, making the handler chain data directly susceptible to data corruption attacks. Windows protects the handlers somewhat by obfuscating them using the `EncodePointer()` API. However, `EncodePointer()` does not implement a cryptographically secure function (since doing so would impose high overhead);

it typically returns the XOR of the input pointer with a process-specific secret. This secret is not protected against memory disclosure attacks; it is potentially derivable from disclosure of any encoded pointer with value known to the attacker (since XOR is invertible), and it is stored in the *process environment block* (PEB), which is readable by the process and therefore by an attacker armed with an information disclosure exploit. With this secret, the attacker can overwrite the heap with a properly obfuscated malicious pointer, and thereby take control of the application.

From a compatibility perspective, CFI protections that do not include first-class support for these various exception-handling mechanisms often conservatively block unusual control-flows associated with exceptions. This can break important application functionalities, making the protections unusable for large classes of software that use exceptions.

Calling Conventions. CFI guard code typically instruments call and return sites in the target program. In order to preserve the original program's functionality, this guard code must therefore respect the various calling conventions that might be implemented by calls and returns. Unfortunately, many solutions to this problem make simplifying assumptions about the potential diversity of calling conventions in order to achieve acceptable performance. For example, a CFI solution whose guard code uses `EDX` as a scratch register might suddenly fail when applied to code whose calling convention passes arguments in `EDX`. Adapting the solution to save and restore `EDX` to support the new calling convention can lead to tens of additional instructions per call, including additional memory accesses, and therefore much higher overhead.

The C standard calling convention (`cdecl`) is caller-pop, pushes arguments right-to-left onto the stack, and returns primitive values in an architecture-specific register (`EAX` on Intel). Each architecture also specifies a set of caller-save and callee-save registers. Caller-popped calling conventions are important for implementing variadic functions, since callees can remain unaware of argument list lengths.

Callee-popped conventions include `stdcall`, which is the standard convention of the Win32 API, and `fastcall`, which passes the first two arguments via registers rather than the stack to improve execution speed. In OOP languages, every nonstatic member function has a hidden *this pointer* argument that points to the current object. The `thiscall` convention passes the *this pointer* in a register (`ECX` on Intel).

Calling conventions on 64-bit architectures implement several refinements of the 32-bit conventions. Linux and Windows pass up to 14 and 4 parameters, respectively, in registers rather than on the stack. To allow callees to optionally spill these parameters, the caller additionally reserves a *red zone* (Linux) or 32-byte *shadow space* (Windows) for callee temporary storage.

Highly optimized programs also occasionally adopt non-standard, undocumented calling conventions, or even blur function boundaries entirely (e.g., by performing various forms of function in-lining). For example, some C compilers support language extensions (e.g., MSVC's `naked` declaration) that yield binary functions with no prologue or epilogue code, and therefore no standard calling convention. Such code can have subtle dependencies on non-register processor elements, such as requiring that certain Intel status flags be preserved across calls. Many CFI solutions break such code by in-lining call site guards that violate these undocumented conventions.

TLS Callbacks. Multithreaded programs require efficient means to manipulate thread-local data without expensive locking. Using *thread local storage* (TLS), applications export one or more TLS callback functions that are invoked by the OS for thread initialization or termination. These functions form a null-terminated table whose base is stored in the PE header. For compiler-based CFI solutions, the TLS callback functions do not usually need extra protection, since both the PE header and the TLS callback table are in unwritable memory. But source-free solutions must ensure that TLS callbacks constitute policy-permitted control-flows at runtime.

Memory Protection. Modern OSes provide APIs for memory page allocation (e.g., `VirtualAlloc` and `mmap`) and permission changes (e.g., `VirtualProtect` and `mprotect`). However, memory pages changed from writable to executable, or to simultaneously writable and executable, can potentially be abused by attackers to bypass DEP defenses and execute attacker-injected code. Many software applications nevertheless rely upon these APIs for legitimate purposes (see Table 1), so conservatively disallowing access to them introduces many compatibility problems. Relevant CFI mechanisms must therefore carefully enforce memory access policies that permit virtual memory management but block code-injection attacks.

Runtime Code Generation. Most CFI algorithms achieve acceptable overheads by performing code generation strictly statically. The statically generated code includes fixed runtime guards that perform small, optimized computations to validate dynamic control-flows. However, this strategy breaks down when target programs generate new code dynamically and attempt to execute it, since the generated code might not include CFI guards. *Runtime code generation* (RCG) is therefore conservatively disallowed by most CFI solutions, with the expectation that RCG is only common in a few, specialized application domains, which can receive specialized protections.

Unfortunately, our analysis of commodity software products indicates that RCG is becoming more prevalent than is commonly recognized. In general, we encountered RCG compatibility limitations in at least three main forms across a variety of COTS products:

1. Although typically associated with web browsers, *just-in-time* (JIT) compilation has become increasingly relevant as an optimization strategy for many languages,

including Python, Java, the Microsoft .NET family of languages (e.g., C#), and Ruby. Software containing any component or module written in any JIT-compiled language frequently cannot be protected with CFI.

2. Mobile code is increasingly space-optimized for quick transport across networks. *Self-unpacking executables* are therefore a widespread source of RCG. At runtime, self-unpacking executables first decompress archived data sections to code, and then map the code into writable and executable memory. This entails a dynamic creation of fresh code bytes. Large, component-driven programs sometimes store rarely used components as self-unpacking code that decompresses into memory whenever needed, and is deallocated after use. For example, NSIS installers pack separate modules supporting different install configurations, and unpack them at runtime as-needed for reduced size. Antivirus defenses hence struggle to distinguish benign NSIS installers from malicious ones [21].

3. Component-driven software also often performs a variety of obscure *API hooking* initializations during component loading and clean-up, which are implemented using RCG. As an example, Microsoft Office software dynamically redirects all calls to certain system API functions within its address space to dynamically generated wrapper functions. This allows it to modify the behaviors of late-loaded components without having to recompile them all each time the main application is updated.

   To hook a function f within an imported system DLL (e.g., `ntdll.dll`), it first allocates a fresh memory page f' and sets it both writable and executable. It next copies the first five code bytes from f to f', and writes an instruction at $f'+5$ that jumps to $f+5$. Finally, it changes f to be writable and executable, and overwrites the first five code bytes of f with an instruction that jumps to f'. All subsequent calls to f are thereby redirected to f', where new functionality can later be added dynamically before f' jumps to the preserved portion of f.

   Such hooking introduces many dangers that are difficult for CFI protections to secure without breaking the application or its components. Memory pages that are simultaneously writable and executable are susceptible to code-injection attacks, as described previously. The RCG that implements the hooks includes unprotected jumps, which must be secured by CFI guard code. However, the guard code itself must be designed to be rewritable by more hooking, including placing instruction boundaries at addresses expected by the hooking code ($f+5$ in the above example). No known CFI algorithm can presently handle these complexities.

3.3 Compositional Defense Evaluation

Some CFI solutions compose CFI controls with other defense layers, such as randomization-based defenses (e.g., [8, 9, 18, 45, 52, 77]). Randomization defenses can be susceptible to other forms of attack, such as memory disclosure attacks (e.g., [27, 63–65]). ConFIRM does not test such attacks, since their implementations are usually specific to each defense and not easy to generalize.

Evaluation of composed defenses should therefore be conducted by composing other attacks with ConFIRM tests. For example, to test a CFI defense composed with stack canaries, one should first simulate attacks that attempt to steal the canary secret, and then modify any stack-smashing ConFIRM tests to use the stolen secret. Incompatibilities of the evaluated defense generally consist of the union of the incompatibilities of the composed defenses.

4 Implementation

To facilitate easier evaluation of the compatibility considerations outlined in Section 3 along with their impact on security and performance, we developed the ConFIRM suite of CFI tests. ConFIRM consists of 24 programs written in C++ totalling about 2,300 lines of code. Each test isolates one of the compatibility metrics of Section 3 (or in some cases a few closely related metrics) by emulating behaviors of COTS software products. Source-aware solutions can be evaluated by applying CFI code transforms to the source codes, whereas source-free solutions can be applied to native code after compilation with a compatible compiler (e.g., gcc, LLVM, or MSVC). Loop iteration counts are configurable, allowing some tests to be used as microbenchmarks. The tests are described as follows:

fptr. This tests whether function calls through function pointers are suitably guarded or can be hijacked. Overhead is measured by calling a function through a function pointer in an intensive loop.

callback. As discussed in Section 3, call sites of callback functions can be either guarded by a CFI mechanism directly, or located in immutable kernel modules that require some form of indirect control-flow protections. We therefore test whether a CFI mechanism can secure callback function calls in both cases. Overhead is measured by calling a function that takes a callback pointer parameter in an intensive loop.

load_time_dynlnk. Load-time dynamic linking tests determine whether function calls to symbols that are exported by a dynamically linked library are suitably protected. Overhead is measured by calling a function that is exported by a dynamically linked library in an intensive loop.

run_time_dynlnk. This tests whether a CFI mechanism supports runtime dynamic linking, whether it supports retrieving symbols from the dynamically linked library at runtime,

and whether it guards function calls to the retrieved symbol. Overhead is measured by loading a dynamically linked library at runtime, calling a function exported by the library, and unloading the library in an intensive loop.

delay_load *(Windows only)***.** CFI compatibility with delay-loaded DLLs is tested, including whether function calls to symbols that are exported by the delay-loaded DLLs are protected. Overhead is measured by calling a function that is exported by a delay-loaded DLL in an intensive loop.

data_symbl. Data and function symbol imports and exports are tested, to determine whether any controls preserve their accessibility and operation.

vtbl_call. Virtual function calls are exercised, whose call sites can be directly instrumented. Overhead is measured by calling virtual functions in an intensive loop.

code_coop. This tests whether a CFI mechanism is robust against CODE-COOP attacks. For the object-oriented interfaces required to launch a CODE-COOP attack, we choose Microsoft COM API functions in Windows, and gtkmm API calls that are part of the C++ interface for GTK+ in Linux.

tail_call. Tail call optimizations of indirect jumps are tested. Overhead is measured by tail-calling a function in a loop.

switch. Indirect jumps associated with switch-case control-flow structures are tested, including their supporting data structures. Overhead is measured by executing a switch-case statement in an intensive loop.

ret. Validation of return addresses (e.g., dynamically via shadow stack implementation, or statically by labeling call sites and callees with equivalence classes) is tested. Overhead is measured by calling a function that does nothing but return in an intensive loop.

unmatched_pair. Unmatched call/return pairs resulting from exceptions and setjmp/longjmp are tested.

signal. This test uses signal-handling in C to implement error-handling and exceptional control-flows.

cppeh. C++ exception handling structures and control-flows are exercised.

seh *(Windows only)***.** SEH-style exception handling is tested for both hardware and software exceptions. This test also checks whether the CFI mechanism protects the exception handlers stored on the stack.

veh *(Windows only)***.** VEH-style exception handling is tested for both hardware and software exceptions. This test also checks whether the CFI mechanism protects callback function pointers passed to `AddVecoredExceptionHandler()`.

convention. Several different calling conventions are tested, including conventions widely used in C/C++ languages on 32-bit and 64-bit x86 processors.

multithreading. Safety of concurrent thread executions is tested. Specifically, one thread simulates a memory corruption exploit that attempts to smash another thread's stack and break out of the CFI-enforced sandbox.

tls_callback *(Windows source-free only)***.** This tests whether static TLS callback table corruption is detected and blocked by the protection mechanism.

pic. Semantic preservation of position-independent code is tested.

mem. This test performs memory management API calls for legitimate and malicious purposes, and tests whether security controls permit the former but block the latter.

jit. This test generates JIT code by first allocating writable memory pages, writing JIT code into those pages, making the pages executable, and then running the JIT code. To emulate behaviors of real-world JIT compilers, the JIT code performs different types of control-flow transfers, including calling back to the code of JIT compiler and calling functions located in other modules.

api_hook *(Windows only)***.** Dynamic API hooking is performed in the style described in Section 3.

unpacking *(source-free only)***.** Self-unpacking executable code is implemented using RCG.

5 Evaluation

5.1 Evaluation of CFI Solutions

To examine ConFIRM's effect on real CFI defenses, we used it to reevaluate 12 major CFI implementations for Linux and Windows that are either publicly available or were obtainable in a self-contained, operational form from their authors at the time of writing. Our purpose in performing this evaluation is not to judge which compatibility features solutions should be expected to support, but merely to accurately document which features are currently supported and to what degree, and to demonstrate that ConFIRM can be used to conduct such evaluations.

Table 3 reports the evaluation results. Columns 2–6 report results for Windows CFI approaches, and columns 7–14 report those for Linux CFI. All Windows experiments were performed on an Intel Xeon E5645 workstation with 24 GB of RAM running 64-bit Windows 10. Linux experiments were conducted on different versions of Ubuntu VM machines corresponding to the version tested by each CFI framework's original developers. All the VM machines had 16GB of RAM with 6 Intel Xeon CPU cores. The overheads for source-free approaches were evaluated using test binaries compiled with most recent version of gcc available for each test platform. All source-aware approaches were applied before or during compilation with the most recent version of LLVM for each test platform (since LLVM provides greatest compatibility between the tested source-aware solutions).

Table 3: Tested results for CFI solutions on CONFIRM

	LLVM (Windows)					LLVM (Linux)							
Test	**CFI**	**ShadowStack**	**MCFG**	**OFI**	**Reins**	**GCC-VTV**	**CFI**	**ShadowStack**	**MCFI**	**πCFI**	**πCFI (nto)**	**PathArmor**	**Lockdown**
fptr	6.35%	⚠	20.13%	4.35%	4.08%	⚠	6.97%	⚠	✗	−14.00%	−13.79%	⚠	174.92%
callback	⚠	⚠	⚠	128.39%	114.84%	⚠	⚠	⚠	✗	✗	✗	⚠	✗
load_time_dynlnk	2.74%	⚠	8.83%	3.36%	2.66%	⚠	1.33%	⚠	30.83%	31.52%	34.05%	74.54%	1.45%
run_time_dynlnk	⚠	⚠	17.63%	12.57%	11.48%	⚠	4.44%	⚠	✗	✗	✗	1,221.48%	✗
delay_load⊞	N/A	N/A	8.16%	3.61%	✗	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
data_symbl	✓	⚠	✓	✓	✗	✓	✓	⚠	✓	✓	✓	✓	✓
vtbl_call	5.62%	⚠	27.71%	35.94%	31.17%	33.56%	5.94%	⚠	✗	−8.19%	−9.31%	⚠	227.82%
code_coop	⚠	⚠	⚠	✓	✗	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠
tail_call	6.17%	⚠	9.51%	0.05%	0.05%	⚠	6.82%	⚠	✗	−17.69%	−17.37%	⚠	178.06%
switch	−5.80%	⚠	3.51%	22.82%	17.69%	⚠	−6.93%	⚠	−29.01%	−27.19%	−28.46%	⚠	85.85%
ret	⚠	18.04%	⚠	49.34%	48.49%	⚠	⚠	20.88%	70.72%	72.40%	71.52%	⚠	106.71%
unmatched_pair	⚠	⚠	⚠	✓	✓	⚠	⚠	⚠	✓	✓	✓	⚠	⚠
signal	✓	⚠	✓	✗	✗	✓	✓	⚠	✓	✓	✓	✗	✓
cppeh	✓	⚠	✓	✓	✗	✓	✓	⚠	✓	✓	✓	✗	✓
seh⊞	✓	⚠	✓	✓	✗	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
veh⊞	⚠	⚠	⚠	✓	✗	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
convention	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓
multithreading	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠
tls_callback⊞,$	N/A	N/A	N/A	✓	✗	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
pic	✓	✓	✓	⚠	⚠	✓	✓	✓	✓	✓	✓	✓	✓
mem	⚠	⚠	⚠	⚠	⚠	⚠	⚠	⚠	✗	✗	✗	✓	✗
jit	⚠	⚠	⚠	✗	✗	⚠	⚠	⚠	✗	✗	✗	⚠	✗
unpacking$	N/A	N/A	N/A	✗	✗	N/A	N/A	N/A	N/A	N/A	N/A	✗	✗
api_hook⊞	⚠	⚠	⚠	✗	✗	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

(nto) stands for *no tail-call optimization*

%: CFI defense passes compatibility and security test, and microbenchmark yields indicated performance overhead

✓: same as %, but this test provides no performance number

⚠: CFI defense passes compatibility but not security check

✗: test does not compile (compilation error), or crashes at runtime

N/A: test is not applicable to the CFI mechanism being tested

⊞: test is Windows-only

$: test is only for source-free defenses

Two forms of compatibility are assessed in the evaluation: A CFI solution is categorized as *permissively compatible* with a test if it produces an output program that does not crash and exhibits the original test program's non-malicious functionality. It is *effectively compatible* if it is permissively compatible and any malicious functionalities are blocked. Effective compatibility therefore indicates secure and transparent support for the code features exhibited by the test.

In Table 3, Columns 2–3 begin with an evaluation of LLVM CFI and LLVM ShadowCallStack on Windows. With both CFI and ShadowCallStack enabled, LLVM on Windows enforces policies that constrain impending control-flow transfers at every call site, except calls to functions that are exported by runtime-loaded DLLs. Additionally, LLVM on Windows does not secure callback pointers passed to external modules not compiled with LLVM, leaving it vulnerable CODE-COOP attacks. Although ShadowCallStack protects against return address overwrites, its shadow stack is incompatible with unmatched call/return pairs.

Column 4 of Table 3 reports evaluation of Microsoft's MCFG, which is integrated into the MSVC compiler. MCFG provides security checks for function pointer calls, vtable calls, tail calls, and switch-case statements. It also passes all tests related to dynamic linking, including load_time_dynlnk, run_time_dynlnk, delay_load, and data_symbl. As a part of MSVC, MCFG provides transparency for generating position-independent code and handling various calling conventions. With respect to exception handling, MCFG is permissively compatible with all relevant features, but does not protect vectored exception handlers. MCFG's most significant shortcoming is its weak protection of return addresses. In addition, it generates call site guard code at compile-time only. Therefore, code that links to immutable modules or modules compiled with a different protection scheme remains potentially insecure. This results in failures against callback corruption and CODE-COOP attacks.

Columns 5–6 of Table 3 report compatibility testing results for Reins and OFI, which are source-free solutions for Windows. Reins validates control-flow transfer targets for function pointer calls, vtable calls, tail calls, switch-case statements, and returns. It supports dynamic linking at load time and runtime, and is one of the only solutions we tested that secures callback functions whose call sites cannot be directly instrumented (with a high overhead of 114.84%). Like MCFG, Reins fails against CODE-COOP attacks. However, OFI extends Reins with additional protections that succeed against CODE-COOP. OFI also exhibits improved compatibility with delay-loaded DLLs, data exports, all three styles of exception handling, all tested calling conventions, and TLS callbacks. Both Reins and OFI nevertheless proved vulnerable against attacks that abuse position-independent code and memory management API functions.

The GNU C-compiler does not yet have built-in CFI support, but includes *virtual table verification* (VTV). VTV is first introduced in gcc 4.9.0. It checks at virtual call sites whether the vtable pointer is valid based on the object type. This blocks many important OOP vtable corruption attacks, although type-aware COOP attacks can still succeed by calling a different virtual function of the same type (e.g., supertype). As shown in column 7 of Table 3, VTV does not protect other types of control-flow transfers, including function pointers, callbacks, dynamic linking for both load-time and run-time, tail calls, switch-case jumps, return addresses, error handling control-flows, or JIT code. However, it is permissively compatible with all the applicable tests, and can compile any feature functionality we considered.

As reported in Columns 8–9, LLVM on Linux shows similar evaluation results as LLVM on Windows. It has better effective compatibility by providing proper security checks for calls to functions that are exported by runtime loaded DLLs. LLVM on Linux overheads range from -6.93% (for switch control structures) to 20.88% (for protecting returns).

MCFI and πCFI are source-aware control-flow techniques. We tested them on x64 Ubuntu 14.04.5 with LLVM 3.5. The results are shown in columns 10–12 of Table 3. ΠCFI comes with an option to turn off tail call optimization, which increases the precision at the price of a small overhead increase. We therefore tested both configurations, observing no compatibility differences between πCFI with and without tail call optimizations. Incompatibilities were observed in both MCFI and πCFI related to callbacks and runtime dynamic linking. MCFI additionally suffered incompatibilities with the function pointer and virtual table call tests. For callbacks, both solutions incorrectly terminate the process reporting a CFI violation. In terms of effective compatibility, MCFI and πCFI both securely support dynamic linking, switch jumps, return addresses, and unmatched call/return pairs, but are susceptible to CODE-COOP attacks. In our performance analysis, we did not measure any considerable overheads for πCFI's tail call option (only 0.3%). This option decreases the performance for dynamic linking but increases the performance of vtable calls, switch-case, and return tests. Overall, πCFI scores more compatible and more secure relative to MCFI, but with slightly higher performance overhead.

PathArmor offers improved power and precision over the other tested solutions in the form of contextual CFI policy support. Contextual CFI protects dangerous system API calls by tracking and consulting the control-flow history that precedes each call. Efficient context-checking is implemented as an OS kernel module that consults the last branch record (LBR) CPU registers (which are only readable at ring 0) to check the last 16 branches before the impending protected branch. As reported in column 13, our evaluation demonstrated high permissive compatibility, only observing crashes on tests for C++ exception handling and signal handlers. However, our tests were able to violate CFI policies using function pointers, callbacks, virtual table pointers, tail-calls, switch-cases, return addresses, and unmatched call/return pairs, resulting

Table 4: Overall compatibility of CFI solutions

Tests	LLVM (Windows)*	MCFG	OFI	Reins	GCC-VTV	LLVM (Linux)*	MCFI	πCFI	πCFI (nto)	Path-Armor	Lock-down
applicable	21	22	24	24	18	18	18	18	18	19	19
permissively compatible	21	22	20	12	18	18	11	14	14	16	14
effectively compatible	12	13	17	9	6	12	9	12	12	6	11
Permissive compatibility	100.00%	100.00%	83.33%	50.00%	100.00%	100.00%	61.11%	77.78%	77.78%	84.21%	73.68%
Effective compatibility	57.14%	59.09%	70.83%	37.50%	33.33%	66.67%	50.00%	66.67%	66.67%	31.58%	57.89%

*Compatibility of LLVM is measured with both CFI and ShadowCallStack enabled.

in a lower effective compatibility score. Its careful guarding of system calls also comes with high overhead for those calls (1221.48%). This affects feasibility of dynamic loading, whose associated system calls all receive a high performance penalty per call. Similarly, load-time dynamic linking exhibits a relatively high 74.54% overhead.

Lockdown enforces a dynamic control-flow integrity policy for binaries with the help of symbol tables of shared libraries and executables. Although Lockdown is a binary approach, it requires symbol tables not available for stripped binaries without sources, so we evaluated it using test programs specially compiled with symbol information added. Its loader leverages the additional symbol information to more precisely sandbox interactions between interoperating binary modules. Lockdown is permissively compatible with most tests except callbacks and runtime dynamic linking, for which it crashes. In terms of security, it robustly secures function pointers, virtual calls, switch tables, and return addresses. These security advantages incur somewhat higher performance overheads of 85.85–227.82% (but with only 1.45% load-time dynamic loading overhead). Like most of the other tested solutions, Lockdown remains vulnerable to CODE-COOP and multi-threading attacks. Additionally, Lockdown implements a shadow stack to protect return addresses, and thus is incompatible with unmatched call/return pairs.

5.2 Evaluation Trends

ConFIRM evaluation of these CFI solutions reveals some notable gaps in the current state-of-the-art. For example, all tested solutions fail to protect software from our cross-thread stack-smashing attack, in which one thread corrupts another thread's return address. We hypothesize that no CFI solution yet evaluated in the literature can block this attack except by eliminating all return instructions from hardened programs, which probably incurs prohibitive overheads. By repeatedly exploiting a data corruption vulnerability in a loop, our test program can reliably break all tested CFI defenses within seconds using this approach.

Since concurrency, flat memory spaces, returns, and writable stacks are all ubiquitous in almost all mainstream architectures, such attacks should be considered a significant open problem. Intel Control-flow Enforcement Technology (CET) [36] has been proposed as a potential hardware-based solution to this; but since it is not yet available for testing, it is unclear whether its hardware shadow stack will be compatible with software idioms that exhibit unmatched call-return pairs.

Memory management abuse is another major root of CFI incompatibilities and insecurities uncovered by our experiments. Real-world programs need access to the system memory management API in order to function properly, making CFI approaches that prohibit it impractical. However, memory API arguments are high value targets for attackers, since they potentially unlock a plethora of follow-on attack stages, including code injections. CFI solutions that fail to guard these APIs are therefore insecure. Of the tested solutions, only PathArmor manages to strike an acceptable balance between these two extremes, but only at the cost of high overheads.

A third outstanding open challenge concerns RCG in the form of JIT-compiled code, dynamic code unpacking, and runtime API hooking. RockJIT [50] is the only language-based CFI algorithm proposed in the literature that yet supports any form of RCG, and its approach entails compiler-specific modifications to source code, making it difficult to apply on large scales to the many diverse forms of RCG that appear in the wild. New, more general approaches are needed to lend CFI support to the increasing array of software products built atop JIT-compiled languages or linked using RCG-based mechanisms—including many of the top applications targeted by cybercriminals (e.g., Microsoft Office).

Table 4 measures the overall compatibility of all the tested CFI solutions. Permissive and effective compatibility are measured as the ratio of applicable tests to permissively and effectively compatible ones, respectively. All CFI techniques embedded in compilers (*viz.* LLVM on Linux and Windows, MCFG, and GCC-VTV), are 100% permissively compatible, avoiding all crashes. LLVM on Linux, LLVM on Windows, and MCFG secure at least 57% of applicable tests, while GCC-VTV only secures 33%.

OFI scores high overall compatibility, achieving 83% permissive compatibility and 71% effective compatibility on 24 applicable tests. Reins has the lowest permissive compatibility score of only 50%. PathArmor and Lockdown are permissively compatible with 84% and 74% of 19 applicable tests. However PathArmor can only secure 32% of the tests, giving it the lowest effective compatibility score.

Table 5: Correlation between SPEC CPU and ConFIRM performance

SPEC CPU Benchmark	CFI Solution									Benchmark Correlation
	MCFG	Reins	GCC-VTV	LLVM-CFI	MCFI	πCFI	πCFI (nto)	PathArmor	Lockdown	
perlbench				2.4	5.0	5.0	5.3	15.0	150.0	**0.09**
bzip2	−0.3	9.2		−0.7	1.0	1.0	0.8	0.0	8.0	**−0.12**
gcc					4.5	4.5	10.5	9.0	50.0	**0.02**
mcf	0.5	9.1		3.6	4.5	4.5	1.8	1.0	2.0	**−0.39**
gobmk	−0.2			0.2	7.0	7.5	11.8	0.0	43.0	**−0.09**
hmmer	0.7			0.1	0.0	0.0	−0.1	1.0	3.0	**0.33**
sjeng	3.4			1.6	5.0	5.0	11.9	0.0	80.0	**−0.03**
h264ref	5.4			5.3	6.0	6.0	8.3	1.0	43.0	**−0.09**
libquantum				−6.9	0.0	−0.3	−1.0	3.0	5.0	**0.51**
omnetpp	3.8		5.8		5.0	5.0	18.8			**−0.52**
astar	0.1		3.6	0.9	3.5	4.0	2.9		17.0	**0.92**
xalancbmk	5.5		24.0	7.2	7.0	7.0	17.6		118.0	**0.94**
milc	2.0			0.2	2.0	2.0	1.4	4.0	8.0	**0.40**
namd	0.1		−0.1	0.1	−0.5	−0.5	−0.5	3.0		**0.98**
dealII	−0.1		0.7	7.9	4.5	4.5	4.4			**−0.36**
soplex	2.3		0.5	−0.3	−4.0	−4.0	0.9	12.0		**0.89**
povray	10.8		−0.6	8.9	10.0	10.5	17.4		90.0	**0.88**
lbm	4.2			−0.2	1.0	1.0	−0.5	0.0	2.0	**−0.22**
sphinx3	−0.1			−0.8	1.5	1.5	2.4	3.0	8.0	**0.31**
ConFIRM median	9.51	4.59	33.56	5.19	30.83	−11.10	−11.60	648.01	140.82	**0.36**

5.3 Performance Evaluation Correlation

Prior performance evaluations of CFI solutions primarily rely upon SPEC CPU benchmarks as a standard of comparison. This is based on a widely held expectation that CFI overheads on SPEC benchmarks are indicative of their overheads on real-world, security-sensitive software to which they might be applied in practical deployments. However no prior work has attempted to quantify a correlation between SPEC benchmark scores and overheads observed for the addition of CFI controls to large, production software products. If, for example, CFI introduces high overheads for code features not well represented in SPEC benchmarks (e.g., because they are not performance bottlenecks for CFI-free software and were therefore not prioritized by SPEC), but that become real-world bottlenecks once their overheads are inflated by CFI controls, then SPEC benchmarks might not be good predictors of real-world CFI overheads. Recent work has argued that prior CFI research has unjustifiably drawn conclusions about real-world software overheads from microbenchmarking results [70], making this an important open question.

To better understand the relationship between CFI-specific operation overheads and SPEC benchmark scores, we therefore computed the correlation between median performance of CFI solutions on ConFIRM benchmarks with their performances reported on SPEC benchmarks (as reported in the prior literature). Although ConFIRM benchmarks are not real-world software, they can serve as microbenchmarks of features particularly relevant to CFI. High correlations therefore indicate to what degree SPEC benchmarks exercise code features whose performance are affected by CFI controls.

Table 5 reports the results, in which correlations between each SPEC CPU benchmark and ConFIRM median values are computed as Pearson correlation coefficients:

$$\rho_{x,y} = \frac{(\sum_{i=1}^{n} x_i \times y_i) - (n \times \bar{x} \times \bar{y})}{(n-1) \times \sigma_x \times \sigma_y} \quad (1)$$

where x_i and y_i are the CPU SPEC overhead and ConFIRM median overhead scores for solution i, $\bar{x}$ and $\bar{y}$ are the means, and σ_x and σ_y are the sample standard deviations of x and y, respectively. High linear correlations are indicated by $|\rho|$ values near to 1, and direct and inverse relationships are indicated by positive and negative ρ, respectively.

The results show that although a few SPEC benchmarks have strong correlations (namd, xalancbmk, astar, soplex, and povray being the highest), in general SPEC CPU benchmarks exhibit a poor correlation of only 0.36 on average with tests that exercise CFI-relevant code features. Almost half the SPEC benchmarks even have negative correlations. This indicates that SPEC benchmarks consist largely of code features unrelated to CFI overheads. While this does not resolve the question of whether SPEC overheads are predictive of real-world overheads for CFI, it reinforces the need for additional research connecting CFI overheads on SPEC benchmarks to those on large, production software.

6 Related Work

6.1 Prior CFI Evaluations

We surveyed 54 CFI algorithms and implementations published between 2005–2019 to prepare ConFIRM, over half

of which were published within 2015–2019. Of these, 66% evaluate performance overheads by applying SPEC CPU benchmarking programs. Examples of such performance evaluations include those of PittSFIeld [43], NaCl [81], CPI [40], REINS [78], bin-CFI [87], control flow locking [10], MIP [48], CCFIR [84], ROPecker [16], T-VIP [29], GCC-VTV [69], MCFI [49], VTint [83], Lockdown [54], O-CFI [45], CCFI [42], PathArmor [71], BinCC [74], πCFI [51], VTI [12], VTrust [82], VTPin [61], TypeArmor [72], PITTYPAT [24], RAGuard [85], GRIFFIN [30], OFI [75], PT-CFI [33], HCIC [86], μCFI [35], CFIXX [14], and τCFI [47].

The remaining 34% of CFI technologies that are not evaluated on SPEC benchmarks primarily concern specialized application scenarios, including JIT compiler hardening [50], hypervisor security [41,76], iOS mobile code security [22,55], embedded systems security [2–4], and operating system kernel security [20,31,38]. These therefore adopt analogous test suites and tools specific to those domains [17,23,56,57,67].

Several of the more recently published works additionally evaluate their solutions on one or more large, real-world applications, including browsers, web servers, FTP servers, and email servers. For example, VTable protections primarily choose browsers as their enforcement targets, and therefore leverage browser benchmarks to evaluate performance. The main browser benchmarks used for this purpose are Microsoft's Lite-Brite [44] Google's Octane [32], Mozilla's Kraken [46], Apple's Sunspider [6], and RightWare's BrowserMark [59].

Since compatibility problems frequently raise difficult challenges for evaluations of larger software products, these larger-scale evaluations tend to have smaller sample sizes. Overall, 88% of surveyed works report evaluations on 3 or fewer large, independent applications, with TypeArmor [72] having the most comprehensive evaluation we studied, consisting of three FTP servers, two web servers, an SSH server, an email server, two SQL servers, a JavaScript runtime, and a general-purpose distributed memory caching system.

To demonstrate security, prior CFI mechanisms are typically tested against proof-of-concept attacks or CVE exploits. The most widely tested attack class in recent years is COOP. Examples of security evaluations against COOP attacks include those reported for μCFI [35], τCFI [47], CFIXX [14], OFI [75], PITTYPAT [24], VTrust [82], PathArmor [71], and πCFI [51].

The RIPE test suite [80] is also widely used by many researchers to measure CFI security and precision. RIPE consists of 850 buffer overflow attack forms. It aims to provide a standard way to quantify the security coverage of general defense mechanisms. In contrast, CONFIRM focuses on a larger variety of code features that are needed by many applications to implement non-malicious functionalities, but that pose particular problems for CFI defenses. These include a combination of benign behaviors and attacks.

6.2 CFI Surveys

There has been one prior survey of CFI performance, precision, and security, published in 2016 [13]. It surveys 30 previously published CFI frameworks, with qualitative and quantitative comparisons of their technical approaches and overheads as reported in each original publication. Five of the approaches are additionally reevaluated on SPEC CPU benchmarks.

In contrast, CONFIRM establishes a foundation for evaluating *compatibility* and *relevance* of various CFI algorithms to modern software products, and highlights important security and performance impacts that arise from incompatibility limitations facing the state-of-the-art solutions.

7 Conclusion

CONFIRM is the first evaluation methodology and microbenchmarking suite that is designed to measure applicability, compatibility, and performance characteristics relevant to control-flow security hardening evaluation. The CONFIRM suite provides 24 tests of various CFI-relevant code features and coding idioms, which are widely found in deployed COTS software products.

Twelve publicly available CFI mechanisms are reevaluated using CONFIRM. The evaluation results reveal that state-of-the-art CFI solutions are compatible with only about 53% of the CFI-relevant code features and coding idioms needed to protect large, production software systems that are frequently targeted by cybercriminals. Compatibility and security limitations related to multithreading, custom memory management, and various forms of runtime code generation are identified as presenting some of the greatest barriers to adoption.

In addition, using CONFIRM for microbenchmarking reveals performance characteristics not captured by metrics widely used to evaluate CFI overheads. In particular, SPEC CPU benchmarks designed to assess CPU computational overhead exhibit an only 0.36 correlation with benchmarks that exercise code features relevant to CFI. This suggests a need for more CFI-specific benchmarking to identify important sources of performance bottlenecks, and their ramifications for CFI security and practicality.

Acknowledgments

The authors thank Tyler Bletsch, Dimitar Bounov, Mihai Budiu, Yueqiang Cheng, Xuhua Ding, Hong Hu, Jay Ligatti, Ben Niu, Mathias Payer, Michalis Polychronakis, R. Sekar, Zhi Wang, and Qingchuan Zhao for their provision of CFI solution implementations and installation assistance for evaluations. The research reported herein was supported in part by ONR award N00014-17-2995, DARPA award FA8750-19-C-0006, NSF awards #1513704 and #1834215, and an NSF IUCRC award from Lockheed Martin.

References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proc. 12th ACM Conf. Computer and Communications Security (CCS)*, pages 340–353, 2005.

[2] A. Abbasi, T. Holz, E. Zambon, and S. Etalle. ECFI: Asynchronous control flow integrity for programmable logic controllers. In *Proc. 33rd Annual Computer Security Applications Conf. (ACSAC)*, pages 437–448, 2017.

[3] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik. C-FLAT: Control-flow attestation for embedded systems software. In *Proc. 23rd ACM Conf. Computer and Communications Security (CCS)*, pages 743–754, 2016.

[4] S. Adepu, F. Brasser, L. Garcia, M. Rodler, L. Davi, A.-R. Sadeghi, and S. Zonouz. Control behavior integrity for distributed cyber-physical systems. *CoRR*, abs/1812.08310, 2018.

[5] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *Proc. 29th IEEE Sym. Security and Privacy (S&P)*, pages 263–277, 2008.

[6] Apple. Sunspider 1.0 JavaScript benchmark suite. https://webkit.org/perf/sunspider/sunspider.html, 2013.

[7] E. Bauman, Z. Lin, and K. W. Hamlen. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *Proc. 25th Network and Distributed Systems Security Sym. (NDSS)*, 2018.

[8] E. D. Berger and B. G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *Proc. 27th ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, pages 158–168, 2006.

[9] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proc. 12th USENIX Security Sym.*, 2003.

[10] T. Bletsch, X. Jiang, and V. Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proc. 27th Annual Computer Security Applications Conf. (ACSAC)*, pages 353–362, 2011.

[11] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: A new class of code-reuse attacks. In *Proc. 6th ACM Sym. Information, Computer and Communications Security (AsiaCCS)*, pages 30–40, 2011.

[12] D. Bounov, R. G. Kici, and S. Lerner. Protecting C++ dynamic dispatch through vtable interleaving. In *Proc. 23rd Network and Distributed System Security Sym. (NDSS)*, 2016.

[13] N. Burow, S. A. Carr, S. Brunthaler, M. Payer, J. Nash, P. Larsen, and M. Franz. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys*, 50(1):16:1–16:33, 2017.

[14] N. Burow, D. McKee, S. A. Carr, and M. Payer. CFIXX: Object type integrity for C++. In *Proc. 25th Network and Distributed System Security Symposium (NDSS)*, 2018.

[15] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-Flow Bending: On the effectiveness of control-flow integrity. In *Proc. 24th USENIX Conf. Security (USENIX)*, pages 161–176, 2015.

[16] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and H. R. Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *Proc. 21st Network and Distributed System Security Sym. (NDSS)*, 2014.

[17] R. Coker. Disk performance benchmark tool – Bonnie. https://www.coker.com.au/bonnie++, 2016.

[18] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conf.*, pages 63–77, 1998.

[19] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. D. Sutter, and M. Franz. It's a TRaP: Table randomization and protection against function-reuse attacks. In *Proc. 22nd ACM Conf. Computer and Communications and Security (CCS)*, pages 243–255, 2015.

[20] J. Criswell, N. Dautenhahn, and V. Adve. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *Proc. 35th IEEE Sym. Security and Privacy (S&P)*, pages 292–307, 2014.

[21] C. Crofford and D. McKee. Ransomeware families use NSIS installers to avoid detection, analysis. *McAfee Labs*, March 2017.

[22] L. Davi, R. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Proc. 19th Network and Distributed System Security Sym. (NDSS)*, 2012.

[23] A. C. de Melo. Performance counters on Linux. In *Linux Plumbers Conf.*, 2009.

[24] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee. Efficient protection of path-sensitive control security. In *Proc. 26th USENIX Security Sym.*, pages 131–148, 2017.

[25] S. Donnelly. Soft target: The top 10 vulnerabilities used by cybercriminals. Technical Report CTA-2018-0327, Recorded Future, 2018.

[26] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Proc. 7th USENIX Sym. Operating Systems Design and Implementation (OSDI)*, pages 75–88, 2006.

[27] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglu-Douskos, M. Rinard, and H. Okhravi. Missing the point(er): On the effectiveness of code pointer integrity. In *Proc. 36th IEEE Sym. Security & Privacy (S&P)*, pages 781–796, 2015.

[28] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control Jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proc. 22nd ACM Conf. Computer and Communications Security (CCS)*, pages 901–913, 2015.

[29] R. Gawlik and T. Holz. Towards automated integrity protection of C++ virtual function tables in binary programs. In *Proc. 30th Annual Computer Security Applications Conf. (ACSAC)*, pages 396–405, 2014.

[30] X. Ge, W. Cui, and T. Jaeger. GRIFFIN: Guarding control flows using Intel processor trace. In *Proc. 22nd ACM Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 585–598, 2017.

[31] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-grained control-flow integrity for kernel software. In *Proc. 1st IEEE European Sym. Security and Privacy (EuroS&P)*, pages 179–194, 2016.

[32] Google. Octane JavaScript benchmark suite. https://developers.google.com/octane, 2013.

[33] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin. PT-CFI: Transparent backward-edge control flow violation detection using Intel processor trace. In *Proc. 7th ACM Conf. Data and Application Security and Privacy (CODASPY)*, pages 173–184, 2017.

[34] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. Programming Languages and Systems (TOPLAS)*, 28(1):175–205, 2006.

[35] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee. Enforcing unique code target property for control-flow integrity. In *Proc. 25th ACM Conf. Computer and Communications Security (CCS)*, pages 1470–1486, 2018.

[36] Intel. Control-flow enforcement technology preview, revision 2.0. Technical Report 334525-002, Intel Corporation, June 2017.

[37] D. Jang, Z. Tatlock, and S. Lerner. SafeDispatch: Securing C++ virtual calls from memory corruption attacks. In *Proc. 21st Network and Distributed System Security Sym. (NDSS)*, 2014.

[38] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis. kGuard: Lightweight kernel protection against return-to-user attacks. In *Proc. 21st USENIX Security Sym.*, pages 459–474, 2012.

[39] F. Konkel. The Pentagon's bug bounty program should be expanded to bases, DOD official says. *Defense One*, 2017.

[40] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *Proc. USENIX Sym. Operating Systems Design and Implementation (OSDI)*, pages 147–163, 2014.

[41] D. Kwon, J. Seo, S. Baek, G. Kim, S. Ahn, and Y. Paek. VM-CFI: Control-flow integrity for virtual machine kernel using Intel PT. In *Proc. 18th Int. Conf. Computational Science and Its Applications (ICCSA)*, pages 127–137, 2018.

[42] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. CCFI: Cryptographically enforced control flow integrity. In *Proc. 22nd ACM Conf. Computer and Communications Security (CCS)*, pages 941–951, 2015.

[43] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *Proc. 15st USENIX Security Sym.*, 2006.

[44] Microsoft. Lite-Brite Benchmark. https://testdrive-archive.azurewebsites.net/Performance/LiteBrite, 2013.

[45] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz. Opaque control-flow integrity. In *Proc. 22nd Network and Distributed System Security Symposium (NDSS)*, 2015.

[46] Mozilla. Kraken 1.1 JavaScript benchmark suite. http://krakenbenchmark.mozilla.org, 2013.

[47] P. Muntean, M. Fischer, G. Tan, Z. Lin, J. Grossklags, and C. Eckert. τCFI: Type-assisted control flow integrity for x86-64 binaries. In *Proc. 21st Int. Sym. Research in Attacks, Intrusions, and Defenses (RAID)*, pages 423–444, 2018.

[48] B. Niu and G. Tan. Monitor integrity protection with space efficiency and separate compilation. In *Proc. 21st ACM Conf. Computer and Communications Security (CCS)*, pages 199–210, 2013.

[49] B. Niu and G. Tan. Modular control-flow integrity. In *Proc. 35th ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, pages 577–587, 2014.

[50] B. Niu and G. Tan. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *Proc. 23rd ACM Conf. Computer and Communications Security (CCS)*, pages 1317–1328, 2014.

[51] B. Niu and G. Tan. Per-input control-flow integrity. In *Proc. 22nd ACM Conf. Computer and Communications Security (CCS)*, pages 914–926, 2015.

[52] G. Novark and E. D. Berger. DieHarder: Securing the heap. In *Proc. 17th ACM Conf. Computing and Communications Security (CCS)*, 2010.

[53] Office of Inspector General. Evaluation of DHS' information security program for FY 2017. Technical Report OIG-18-56, Department of Homeland Security (DHS), 2018.

[54] M. Payer, A. Barresi, and T. R. Gross. Fine-grained control-flow integrity through binary hardening. In *Proc. 12th Int. Conf. Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 144–164, 2015.

[55] J. Pewny and T. Holz. Control-flow restrictor: Compiler-based CFI for iOS. In *Proc. 29th Annual Computer Security Applications Conf. (ACSAC)*, pages 309–318, 2013.

[56] Postmark. Email delivery for web apps. https://postmarkapp.com, 2013.

[57] R. Pozo and B. Miller. SciMark 2. http://math.nist.gov/scimark2, 2016.

[58] A. Prakash, X. Hu, and H. Yin. vfGuard: Strict protection for virtual function calls in COTS C++ binaries. In *Proc. 22nd Network and Distributed System Security Sym. (NDSS)*, 2015.

[59] RightWare. Basemark web 3.0. https://web.basemark.com, 2019.

[60] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Information and System Security (TISSEC)*, 15(1), 2012.

[61] P. Sarbinowski, V. P. Kemerlis, C. Giuffrida, and E. Athanasopoulos. VTPin: Practical vtable hijacking protection for binaries. In *Proc. 32nd Annual Computer Security Applications Conf. (ACSAC)*, pages 448–459, 2016.

[62] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming. In *Proc. 36th IEEE Sym. Security and Privacy (S&P)*, pages 745–762, 2015.

[63] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proc. 11th ACM Conf. Computer and Communications Security (CCS)*, pages 298–307, 2004.

[64] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proc. 34th IEEE Sym. Security & Privacy (S&P)*, pages 574–588, 2013.

[65] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *Proc. 2nd European Work. System Security (EUROSEC)*, pages 1–8, 2009.

[66] J. Tang. Exploring Control Flow Guard in Windows 10. Technical report, Trend Micro Threat Solution Team, 2015.

[67] The Wine Committee. Wine. http://www.winehq.org.

[68] C. Tice. Improving function pointer security for virtual method dispatches. In *GNU Cauldron Work.*, 2012.

[69] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *Proc. 23rd USENIX Security Sym.*, pages 941–955, 2014.

[70] E. van der Kouwe, G. Heiser, D. Andriesse, H. Bos, and C. Giuffrida. SoK: Benchmarking flaws in systems security. In *Proc. 4th IEEE Eurpean Sym. Security and Privacy (EuroS&P)*, 2019.

[71] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical context-sensitive CFI. In *Proc. 22nd ACM Conf. Computer and Communications Security (CCS)*, pages 927–940, 2015.

[72] V. van der Veen, E. Göktas, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *Proc. 37th IEEE Sym. Security and Privacy (S&P)*, pages 934–953, 2016.

[73] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proc. 14th ACM Sym. Operating Systems Principles (SOSP)*, pages 203–216, 1993.

[74] M. Wang, H. Yin, A. V. Bhaskar, P. Su, and D. Feng. Binary code continent: Finer-grained control flow integrity for stripped binaries. In *Proc. 31st Annual Computer Security Applications Conf. (ACSAC)*, pages 331–340, 2015.

[75] W. Wang, X. Xu, and K. W. Hamlen. Object flow integrity. In *Proc. 24th ACM Conf. Computer and Communications Security (CCS)*, pages 1909–1924, 2017.

[76] Z. Wang and X. Jiang. HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proc. 31st IEEE Sym. Security and Privacy (S&P)*, pages 380–395, 2010.

[77] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proc. 19th ACM Conf. Computer and Communications Security (CCS)*, pages 157–168, 2012.

[78] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Securing untrusted code via compiler-agnostic binary rewriting. In *Proc. 28th Annual Computer Security Applications Conf. (ACSAC)*, pages 299–308, 2012.

[79] R. Wartell, Y. Zhou, K. W. Hamlen, and M. Kantarcioglu. Shingled graph disassembly: Finding the undecidable path. In *Proc. 18th Pacific-Asia Conf. Knowledge Discovery and Data Mining (PAKDD)*, pages 273–285, 2014.

[80] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen. RIPE: Runtime intrusion prevention evaluator. In *Proc. 27th Annual Computer Security Applications Conf. (ACSAC)*, pages 41–50, 2011.

[81] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *Proc. 30th IEEE Sym. Security and Privacy (S&P)*, pages 79–93, 2009.

[82] C. Zhang, S. A. Carr, T. Li, Y. Ding, C. Song, M. Payer, and D. Song. VTrust: Regaining trust on virtual calls. In *Proc. 23rd Network and Distributed System Security Sym. (NDSS)*, 2016.

[83] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song. VTint: Protecting virtual function tables' integrity. In *Proc. 22nd Network and Distributed System Security Sym. (NDSS)*, 2015.

[84] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zo. Practical control flow integrity and randomization for binary executables. In *Proc. 34th IEEE Sym. Security and Privacy (S&P)*, pages 559–573, 2013.

[85] J. Zhang, R. Hou, J. Fan, K. Liu, L. Zhang, and S. A. McKee. RAGuard: A hardware based mechanism for backward-edge control-flow integrity. In *Proc. ACM Int. Conf. Computing Frontiers (CF)*, pages 27–34, 2017.

[86] J. Zhang, B. Qi, Z. Qin, and G. Qu. HCIC: Hardware-assisted control-flow integrity checking. *IEEE Internet of Things J.*, 6(1):458–471, 2019.

[87] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Proc. 22nd USENIX Conf. Security (USENIX)*, pages 337–352, 2013.

Point Break: A Study of Bandwidth Denial-of-Service Attacks against Tor

Rob Jansen
U.S. Naval Research Laboratory
rob.g.jansen@nrl.navy.mil

Tavish Vaidya
Georgetown University
tavish@cs.georgetown.edu

Micah Sherr
Georgetown University
msherr@cs.georgetown.edu

Abstract

As the Tor network has grown in popularity and importance as a tool for privacy-preserving online communication, it has increasingly become a target for disruption, censorship, and attack. A large body of existing work examines Tor's susceptibility to attacks that attempt to block Tor users' access to information (e.g., via traffic filtering), identify Tor users' communication content (e.g., via traffic fingerprinting), and de-anonymize Tor users (e.g., via traffic correlation). This paper focuses on the relatively understudied threat of denial-of-service (DoS) attacks against Tor, and specifically, DoS attacks that intelligently utilize bandwidth as a means to significantly degrade Tor network performance and reliability.

We demonstrate the feasibility of several bandwidth DoS attacks through live-network experimentation and high-fidelity simulation while quantifying the cost of each attack and its effect on Tor performance. First, we explore an attack against Tor's most commonly used default bridges (for censorship circumvention) and estimate that flooding those that are operational would cost $17K/mo. and could reduce client throughput by 44% while more than doubling bridge maintenance costs. Second, we explore attacks against the TorFlow bandwidth measurement system and estimate that a constant attack against all TorFlow scanners would cost $2.8K/mo. and reduce the median client download rate by 80%. Third, we explore how an adversary could use Tor to congest itself and estimate that such a congestion attack against all Tor relays would cost $1.6K/mo. and increase the median client download time by 47%. Finally, we analyze the effects of Sybil DoS and deanonymization attacks that have costs comparable to those of our attacks.

1 Introduction

Tor [28] is the most popular anonymous communication system ever deployed, with an estimated eight million daily active users [59]. These users depend on Tor to anonymize their connections to Internet services and distributed peers, and also to circumvent censorship by local authorities that control network infrastructure. Tor is used by ordinary citizens and businesses to protect their privacy online, by journalists and activists to more freely access and contribute digital content [7], and by criminals to perform illegal activities while avoiding identification [67].

As a result of its popularity, open-source codebase [9], and transparent development processes [10], Tor has gained significant attention from researchers who explore attacks that aim to deanonymize its users by gaining an advantageous view of network traffic [13]. Research directions for Tor attacks include website fingerprinting [18, 41, 42, 56, 69, 70, 77, 84, 85], routing [15, 16, 79, 81, 83], end-to-end correlation [54, 57, 58, 65, 66], congestion [31, 34, 51, 64], and side channels [43, 63]. Many of these attacks represent realistic threats for Tor users: some attacks are reported to have been launched by state sponsors against Tor in the wild [21, 22, 73, 76]. However, relatively understudied but arguably more viable is the threat of denial-of-service (DoS).

The Threat of Denial-of-Service: Bandwidth-based DoS against Tor is a relatively understudied but relevant threat. Previous work has explored the exhaustion of Tor relays' memory [51], CPU [14, 71], and socket descriptor resources [35], as well as selective service refusal [16]. While bandwidth-based DoS attacks against a *single target* have been considered [31], we are the first to study the feasibility, cost, and effects of launching such attacks against the *entire Tor network* of relays and other particularly vulnerable Tor components. Given Tor's limited resources and slow performance relative to the open web, further reducing performance through bandwidth DoS attacks also has the potential to reduce security by driving away users who may be unwilling to endure even slower load times [16, 25].

We argue that DoS attacks in general, and bandwidth DoS attacks in particular, are less complex and therefore more viable than many previous deanonymization attacks. Our DoS attacks either can be outsourced to third party "stresser" services that will flood a target with packets for an amortized cost of $0.74/hr. per Gbit/s of attack traffic (see §3.1), or utilize lightweight Tor clients running on dedicated servers at an amortized cost of $0.70/hr. per Gbit/s of attack traf-

fic (see §3.2). Nation-states are known to sponsor DoS attacks [60], and the ease of deployment and low cost of our attacks suggest that state actors could reasonably run them to disrupt Tor over both short and long timescales. We speculate that nation-states may, e.g., choose DoS as an alternative to traffic filtering as Tor continues to improve its ability to circumvent blocking and censorship [32]. Non-state actors could also reasonably deploy the attacks since they require only a few servers or can be completely outsourced.

Tor DoS attacks are not a hypothetical threat: existing evidence indicates that DoS attacks against the network have already been successfully deployed [23, 38, 40], requiring Tor to develop a subsystem to mitigate its effects [24, 39] (the subsystem does not mitigate our attacks). Although it may be challenging to detect and counter bandwidth-based DoS attacks, especially those that are designed to mimic realistic and plausible usage patterns, we believe that it is imperative to better understand such a threat as we develop defenses.

Our Contributions: This paper focuses on the costs and effects of DoS attacks that intelligently utilize bandwidth as a means to significantly degrade Tor performance and reliability. Following a discussion of the current pricing models for "stresser" (i.e., DoS-for-hire) services (§3.1) and dedicated servers (§3.2), we first explore the threat of a naïve flooding attack in which an adversary uses multiple "stresser" accounts to consume Tor relays' bandwidth by flooding them with packets (§4). We estimate that the cost to carry out such an attack against the entire Tor network is $7.2M/mo.

We then demonstrate the feasibility and effects of 3 major bandwidth DoS attacks against Tor in order of decreasing cost. First, we explore in §5 an attack that attempts to disrupt Tor's censorship circumvention system by flooding Tor's default bridges with packets, thereby causing bridge users to migrate to non-default bridges or lose access to Tor. We estimate that flooding Tor's 12 operational default bridges would cost $17K/mo. and would reduce bridge user throughput by 44% (if 25% of the users migrated to other bridges) while more than doubling meek bridge maintenance costs.

Second, we explore in §6 an attack that attempts to disrupt Tor's load balancing system by flooding TorFlow bandwidth scanners with packets, thereby causing inaccurate and inconsistent relay capacity measurement results. Through high-fidelity network simulation using Shadow [47], we find that such an attack reduces the median client download rate by 80%. We estimate that a constant flooding attack against Tor's 5 TorFlow scanners would cost $2.8K/mo.

Third, we explore in §7 an attack that uses the Tor protocol to consume relay bandwidth resources. In the attack, a Tor client builds thousands of 8-hop circuits and congests relays by downloading large files through the network. Using Shadow, we find that such an attack using 20k circuits increases the median client download time by 120% at an estimated cost of $6.3K/mo. and achieves a bandwidth amplification factor of 6.7. We also find that a stop reading strategy [51] reduces the estimated cost of a 20k circuit attack to $1.6K/mo., increases the median client download time by 47%, and achieves a bandwidth amplification factor of 26.

Finally, we analyze in §8 the effects of relay Sybil attacks that have costs comparable to those of our attacks.

Ethics and Responsible Disclosure: We emphasize that we do not carry out attacks against the live Tor network. We conduct some measurement experiments on Tor to better understand its composition and performance characteristics. However, we neither observe nor store any information about any Tor users (other than ourselves). We evaluate our attacks using high-fidelity Shadow simulations that are constructed to resemble the live Tor network. Additionally, we discussed our project with Tor developers, shared some of our results before submission of this paper, and sent a pre-print of our paper prior to its acceptance. We anticipate providing support as they develop any mitigations to our attacks.

2 Related Work

In this paper we focus specifically on attacks that target the Tor network, noting that attacks that target Internet protocols (e.g., TCP) or resources (e.g., web servers) have been rigorously studied in previous work.

Anonymity Attacks against Tor: There is a large body of work that examines attacks against Tor. The majority of these attacks aim to compromise anonymity—that is, to de-anonymize either targeted users or Tor users *en masse*. We highlight that research directions for anonymity attacks include website fingerprinting [18, 41, 42, 56, 69, 70, 77, 84, 85], routing [15, 16, 79, 81, 83], end-to-end correlation [54, 57, 58, 65, 66], congestion [31, 34, 51, 64], and side channels [43, 63]. (The reader may refer to previous work for a more complete taxonomy [13].) Although anonymity attacks are certainly problematic for Tor, the primary focus of this paper is on bandwidth-based DoS attacks that significantly degrade Tor network performance and reliability. Note that we compare our attacks to Sybil attacks that can be used for deanonymizing Tor users in §8.

Denial-of-Service Attacks against Tor: We are not the first to explore the network's susceptibility to DoS. In their seminal work, Evans et al. [31] exploit the lack of an upper bound on the length of Tor circuits in older Tor versions. They show that an attacker can perform a bandwidth amplification DoS attack by creating cyclic, arbitrary length circuits through high bandwidth Tor relays. The congestion created by the DoS attack affects the latency of legitimate circuits which can be used to determine the guard relay on a circuit. To mitigate this attack, Tor has since imposed a cap of eight relays in circuit creation [27, §5.6].

Similarly, Pappas et al. [71] propose an asymmetric, amplification *packet-spinning* DoS attack against legitimate Tor relays. The goal of the attack is to increase the chances of legitimate clients choosing the attacker's relays by keeping the legitimate relays busy with expensive cryptographic opera-

tions. The attacker uses a malicious relay to create a circular Tor circuit that starts and ends at the malicious relay. Their focus is on de-anonymization and they do not consider DoS attacks against the entire Tor network.

Borisov et al. [16] show that an attacker can de-anonymize a large fraction of Tor circuits by performing selective DoS on honest Tor relays to increase the probability that the attacker's relays will be chosen as guard and exit relays (and thus capable of performing traffic correlation [80]). Tor has since deployed a route manipulation (path bias) detection system to mitigate the effects of such an attack [26, §7].

Barbera et al. [14] propose an asymmetric DoS attack against Tor relays. The attack floods a targeted relay with CREATE cells that require public key operations to decrypt the cell. They show that by strategically targeting important relays, the attacker can slow down the entire Tor network due to overload on the remaining relays that are not under DoS attack. This is similar in aim to our work. However, our focus is less on protocol vulnerabilities and more on enumerating hotspots in Tor that, when attacked, could disrupt the network at large. We explore multiple avenues for causing network-wide performance and disruption.

Geddes et al. [35] demonstrate socket exhaustion attacks against various proposed replacements [12, 37] of Tor's transport protocol. They show that an attacker can disable arbitrary relays by exhausting their socket file descriptors and prevent legitimate connections from succeeding. However, the attack does not apply to the deployed Tor network, which does not employ the vulnerable transport protocols.

Jansen et al. [51] propose the *Sniper Attack*, a memory-based DoS attack that exploits Tor's end-to-end reliable data transport to consume memory by filling up the application level packet queues. Using simulation on Shadow [47], they show that an attacker can sequentially disable 20 exit relays in 29 minutes and make the Tor network unusable, while remaining undetected. The Tor Project has since rolled out defenses against the sniper attack [45]. We use some of the techniques from the sniper attack in our congestion attacks.

3 Threat Model and Attacker Costs

We consider an attacker who is determined to deny service to the Tor network. We make few assumptions about the capabilities or makeup of our adversary. In particular, our adversary need not control large regions of the Internet or be able to observe a large fraction of Tor traffic.

Instead, we model an adversary who has some bandwidth and computing capacity at its disposal. Certainly, a nation-state has such resources, but we imagine that such an adversary would likely prefer to avoid attribution and not conduct attacks from its own networks. More generally, our adversary can acquire (or rent) a distributed network of machines capable of sending traffic into the Tor network. We highlight two potential avenues for obtaining the resources necessary to carry out network-wide attacks against Tor: dedicated DoS "stresser" services (§3.1) and the use of more traditional (and legal) dedicated hosting services (§3.2).

We do not require that the adversary be able to position itself in arbitrary locations on the Internet, although we do assume that some portion of its traffic will reach its intended targets. For some attacks, we additionally require the adversary to operate a Tor relay, but as we describe below, it is advantageous for such attacks to run a relay that provides negligible bandwidth to the Tor network; that is, the relay could be cheaply instantiated on a shared cloud provider or other low-cost hosting service.

Attacker Goals: The goal of the attacker is to disrupt either (i) the Tor network in its entirety or (ii) a portion of it that affects an entire subpopulation of Tor users. The latter includes attacks against Tor's *bridge* infrastructure, the set of unpublished relays that permit the participation of users who are otherwise prevented from accessing the Tor network directly (e.g., due to censorship).

In general, we consider an attack successful if it entirely prevents users from accessing Tor or if it degrades performance to such an extent that the anonymity service becomes too burdensome to use. The latter is of course subjective, but informally, we set a high threshold for what we consider unusable performance. We also note that even in the current (non-attacked) Tor network, its slow performance is already perceived as an impediment to its more widespread use [13]. Degrading performance much beyond Tor's current levels may cause many users to abandon the network.

Attacker Costs: One of our goals is to estimate the monetary cost of performing various bandwidth DoS attacks against different elements of Tor infrastructure. To estimate such costs to the attacker, we build a cost model from publicly available information on pricing of various online stresser services and dedicated hosting services.

3.1 Stresser Services

There is an active online market for stresser (also called DoS-for-hire or booter) services. These provide the capability to launch DoS attacks against any target, using a web-based interface, at a relatively low monthly cost. Most commonly, the attacks use a distributed botnet of compromised hosts to target a single victim, flooding it with requests.

We summarize the stresser service landscape in Table 1, although this table does not likely capture all available stresser sites. (Since such services are illegal in most jurisdictions, they are not widely advertised and there is significant churn in the industry, making it difficult to obtain a comprehensive list.) We emphasize that Table 1 reports *advertised* attack strengths. Although others have empirically evaluated the achieved attack strengths of these services [74, 75], we elected not to repeat their experiments due to ethical concerns (specifically, the strong possibility of incurring collateral damage). Previous work has found them

Table 1: The estimated mean hourly cost to flood a single target with 1 Gbit/s using various online stresser services. The amortized cost is the hourly price per Gbit/s of traffic per target.

Stresser Service	Time (hrs)	Num Attks	Strength (Gbit/s)	$/mo. (USD)	$/target/hr. (USD)	Amort. (USD)
bootyou.net	3	3	45-50	$ 40	$ 4.44	$ 0.10
booter.xyz	1.67	1	150-200	$ 50	$ 30	$ 0.20
str3ssed.me	1	1	250	$ 55	$ 55	$ 0.22
cloudstress.com	1	1	750	$ 55	$ 55	$ 0.07
ragebooter.net	2	3	10+	$ 60	$ 10	$ 1.00
critical-boot.com	1	1	8-12	$ 40	$ 40	$ 5.00
fiberstresser.com	1	1	750	$ 55	$ 55	$ 0.07
netstress.org	0.67	1	320	$ 45	$ 68.25	$ 0.21
quantumbooter.net	1	2	50	$ 60	$ 30	$ 0.60
vbooter.org	1	3	48-64	$ 40	$ 13.33	$ 0.28
iddos.net	2	1	50	$ 50	$ 25	$ 0.50
downthem.org	0.22	2	200	$ 60	$ 135	$ 0.68
					Mean amortized cost ($/target/hour/Gbit/s):	**$ 0.74**

Table 2: The estimated mean hourly cost to flood a single target with 1 Gbit/s using various dedicated server providers. The amortized cost is the hourly price per Gbit/s of traffic. Prices include 4 CPU cores with minimum 16 GB RAM and 500 GB storage.

Service	Speed (Gbit/s)	Quota (TB)	$/mo. (USD)	Amort. (USD)
Liquid Web	1.00	5	$ 249.00	$ 0.35
InMotion	1.00	10	$ 166.59	$ 0.23
DreamHost	Unkn.	Unmet.	$ 249.00	–
GoDaddy	1.00	Unmet.	$ 239.99	$ 0.33
BlueHost	0.10	15	$ 249.99	$ 3.47
1&1	1.00	Unmet.	$ 130.00	$ 0.18
FatCow	Unkn.	15	$ 239.99	–
OVH	0.50	Unmet.	$ 119.99	$ 0.33
SiteGround	1.00	10	$ 269.00	$ 0.37
YesUpHost	1.00	100	$ 249.00	$ 0.35
			Mean amortized cost ($/hour/Gbit/s):	**$ 0.70**

capable of launching bandwidth DoS attacks with measured attack rates in hundreds of Gigabits per second [74, 75] .

In our analysis in subsequent sections, we consider the average amortized cost of the attacker to flood a single target (i.e., IP address) with 1 Gbit/s of attack traffic for an hour; we found this to be $0.74. Although at first blush, this may appear unrealistically inexpensive, we note that this is *more* costly than obtaining the equivalent bandwidth from legitimate dedicated hosting providers; see §3.2. In §4, we evaluate the cost of using stresser services to overwhelm the capacity of the Tor network en masse, and consider more efficient and targeted stresser attacks in §5 and §6.

3.2 Dedicated Server Costs

Online dedicated hosting services provide customers with remotely located and managed physical machines. Users have full access to the provisioned resources such as CPU, RAM, and disk storage, at a fixed cost. Typically, providers impose some limits on the amount of monthly traffic, and charge different rates for the provisioned network bandwidth. Table 2 reports the pricing schemes for several popular dedicated hosting services. The average amortized hourly cost for transferring data at 1 Gbit/s is $0.70.

Unlike stresser services, dedicated hosting services do not cater to network attackers. They are much more likely to police their traffic and terminate service for customers who are obviously attempting to perform flooding attacks. We thus do not consider their use for naïve flooding of Tor components (e.g., to overwhelm their capacity).

Dedicated servers are well-suited for an attacker who is looking to disrupt the Tor network by leveraging some aspects of Tor's design or protocols. The attacker can use the resources provided by dedicated hosting services to launch such application layer bandwidth DoS attacks on Tor. We explore how dedicated hosting services could serve as a platform for causing severe congestion of Tor relays in §7.

4 Naïve Flooding Attacks against Tor Relays

A straightforward method of attacking Tor is to flood relays with spurious traffic. In this section, we analyze the cost of using stresser services to disrupt the entire Tor network.

Saturating Links: To simplify our analysis, we assume a model of the Internet in which every node i has a finite bandwidth capacity $\mathscr{C}_i$, measured in bits per second (bit/s). We do not consider asymmetric bandwidth since Tor relays receive and send traffic in roughly equal proportions; if node i has asymmetric connectivity, we can consider $\mathscr{C}_i$ to be the minimum of its upstream and downstream capacities.

We assume that an adversary can effectively deny service to a targeted node v if (i) it can cause traffic to arrive at the target at a rate greater than $\mathscr{C}_v$ and (ii) such traffic cannot be filtered upstream. Importantly, the second criterion requires the attacker to initiate a distributed DoS attack from multiple sources (i.e., IP addresses) that cannot easily be enumerated or blocked. Additionally, the communication should resemble legitimate traffic (e.g., be directed at a relevant TCP port). Stresser services generally meet these requirements.

Our first assumption—i.e., a node v effectively becomes unusable if it receives attack traffic at a rate greater than $\mathscr{C}_v$—is admittedly an oversimplification. However, we speculate that saturating v's link would induce a high packet loss rate of 50% or more for legitimate clients, since such clients would have to compete for v's connectivity. TCP performs poorly at such high packet loss rates [61, 68].[1] Stresser services offer attack rates that *vastly* exceed the estimated bandwidth capacities of Tor relays.

Estimating Tor Relay Link Capacity: For a successful flooding attack, the rate at which the attack traffic arrives at

[1] Computing TCP's performance for a given packet loss rate is complex since there are a variety of TCP congestion control algorithms (e.g., Tahoe, Reno, etc.). However, we can derive the theoretical network limit based on the Mathis et al. [61] formula: assuming an average RTT of 40ms, an MSS of 1460B, and a 50% loss rate, the maximum possible throughput achievable by TCP is just $(\mathrm{MSS}/\mathrm{RTT}) \cdot (1/\sqrt{\mathrm{loss}}) = 0.41$ MiB/s.

the target should be equal to or greater than the target's network link capacity. Importantly, we distinguish between the link capacity $\mathscr{C}_v$ of a victim relay and its effective throughput, the latter of which depends on rate limiting, its selection probability, etc. The flooding attack instead depends on overwhelming the victim's actual connectivity, i.e., $\mathscr{C}_v$.

Unfortunately, Tor relays do not publish their link capacities. To estimate a given relay's link capacity, we consider its bandwidth history as recorded in the previous year (from `2017-11-01` to `2018-11-01`) by the Tor Metrics Portal [11]. For each day, we find the maximum observed bandwidth for the relay and map this bandwidth to the next highest value in a fixed set of *bandwidth offerings* that are commonly available: 1, 10, 100, 200, 500, 1,000 and 10,000 Mbit/s. For example, a relay with a maximum observed bandwidth of 1,200 Mbit/s will be considered to have 10 Gbit/s network link. We thus assume that an attacker must direct 10 Gbit/s of attack traffic to overwhelm the relay's capacity. We again emphasize that this is an estimate; the actual capacity at any given time may vary significantly if relay operators configure Tor bandwidth limitation options (operators can set instantaneous bandwidth rate limits and total monthly usage limits).

Attack Cost: We estimate that the total link capacity across the Tor network ranged from 429 to 575 Gbit/s over the year; for our analysis, we use the average of 512.73 Gbit/s. We require that at least one stresser account be used for each Tor relay (since stresser services usually restrict the number of targets to one). Additional stresser accounts are needed to saturate relays with high bandwidth capacities. Applying our cost model, an attacker can use stresser services to flood *all* relays in the Tor network at a cost of about $10K/hr. (or $7.2M/mo.). An adversary can roughly halve its costs by targeting only exit relays, which are required for traffic exiting the network. Overall, however, we find that disrupting Tor by renting stresser services is an expensive proposition, only potentially viable for a nation-state adversary.

Limitations: A limitation of our analysis is that it is not based on empirical evidence (since we were not willing to use such services) and relies on advertised attack rates. Although Santanna et al. have found such services to reasonably deliver high-bandwidth [74, 75], it is possible that they provide a much lower attack strength than advertised. We also rely on the assumption that packets are not filtered upstream, which may not always be valid (as discussed below).

Mitigation: It is possible that ISPs could render such attacks ineffective by filtering traffic. For example, ISPs could discover hosts belonging to the stresser services and filter traffic originating at those hosts. However, such rules may be difficult to maintain given the dynamic nature of the Internet.

Filtering all incoming requests to Tor relays that do not originate at other relays would be an ineffective strategy. In particular, entry relays must allow for clients anywhere on the Internet to initiate a circuit, and any relay may be chosen as an entry by clients implementing non-default path selection algorithms. Filtering attempts may also be complicated by the churn rate of Tor relays and would interfere with the process of bootstrapping new relays to the network. Finally, dropping packets on the relay is an ineffective defense since the dropped packets have already consumed bandwidth.

Traditional DoS defenses such as the use of CDNs are not compatible with Tor since aggregating relays onto a small number of CDN providers would diminish anonymity; it is also unclear how a relay could operate within a CDN. Relay operators may consider migrating to popular cloud services that offer DoS protection services [1, 2]. However, the security and privacy implications of migrating to such services is unknown and may risk exposure to traffic correlation attacks.

Perhaps the most tractable mitigation strategy is to increase the total relay bandwidth capacity of the network.

5 Congesting Tor Bridges

Tor provides anonymous communication to clients, but does not conceal the network locations of its relays, subjecting them to trivial blocking. To counter censors that block access to Tor relays, Tor logically separates *anonymity* (accessing the Internet without revealing network location) from *unblockability* (gaining access to the Tor network). The latter is achieved through the use of *bridge* relays that are not published in the Tor directories. Bridges serve as alternative ingress points into the Tor network for users who cannot directly connect to Tor entry guards.

In this section, we explore the effects of using stresser services (§3.1) to flood Tor bridge relays. We differentiate between three classes of bridges:

Default Bridges: The Tor Browser Bundle (TBB) includes a set of 38 hard-coded default bridges (as of version 8.0.3). Users who cannot directly access Tor relays can configure TBB to connect via one of these default bridges.

A special case of default bridges is *meek bridges* [4, 6] that reside on popular cloud providers and communicate with Tor clients via HTTPS. Censors cannot easily distinguish meek traffic from more typical HTTPS traffic entering the cloud. Disrupting meek thus entails entirely blocking access to the cloud provider, which is presumed to impose too high a collateral cost to the censor. Meek bridges, however, are expensive to operate (since cloud services are not free) and are susceptible to cloud providers disallowing their use [17, 33].

Unlisted Bridges: Users can also request an unlisted bridge either directly from TBB, via bridges.torproject.org, or through email. To prevent a censor from trivially enumerating the bridges, Tor limits the amount of bridges it disseminates to a single requesting IP or email address. However, such protections are obviously brittle and numerous techniques exist for discovering unlisted bridges [20, 30].

Private Bridges: Finally, private bridges are not disseminated by the Tor Project, either because their operators did not notify the Tor Project that they exist or because the Tor Project opted not to disseminate them.

5.1 The State of Tor's Bridges

We first examine the performance of the network's bridges. We focus on the 25 default bridges that use the obfs4 obfuscation protocol[2] since 90% of all bridge users use default bridges [62] and obfs4 is the bridge type recommended by Tor. To test their performance, we use a modified version of Tor to download a 6 MiB file through each bridge. Surprisingly, we find that only 48% (12/25) of the obfs4 default bridges included in TBB are operational.

Figure 1 plots the cumulative distribution (y-axis) of the throughput of the functioning obfs4 default bridges (blue line) when downloading a 6 MiB file on 2018-04-10. Each default obfs4 bridge downloaded the 6 MiB file three times; the CDF plots the average of these downloads. For consistency, we fixed the middle and exit relays, choosing relays with high selection probabilities (and thus high bandwidths). The median throughput of the default bridges is 368 KiB/s; there is a large variation over the default bridges however, ranging from 67 KiB/s to 1,190 KiB/s.

To compare against the performance of unlisted bridges, we requested 135 unlisted obfs4 bridges from the Tor Project's bridge authority via its web and email interfaces. Roughly 70% (95/135) of the acquired unlisted bridges were found to be functional. As shown in Figure 1 (orange line), the unlisted bridges generally outperformed the default bridges, which is expected given Matic et al.'s finding [62] that suggests approximately 90% of bridge traffic is conducted through default bridges. We suspect that the high demand on the few operational default bridges leads to worse performance than the less frequently used unlisted bridges.

As a point of comparison, historical data from the Tor Metrics Portal reveals that non-bridge circuits on Tor during the same time period yielded an average throughput of 786 KiB/s and experienced negligible failure rates [11].

In summary, Tor's bridges are generally far more brittle compared to the network's advertised relays, offering much greater failure rates for default (52%) and unlisted (30%) bridges (compared to 0% for Tor relays) and lower average throughput (545 KiB/s and 681 KiB/s for default and unlisted bridges, respectively, versus 786 KiB/s without bridges).

5.2 Attacking Default Bridges

Ninety percent of bridge users use default bridges [62], and only 12 working default obfs4 bridges are included in the TBB. We first estimate how costly it would be for an attacker to disrupt all of the default bridges. Then, for various migration models in which some percentage of affected bridge users switch to unlisted bridges, we estimate the performance and pricing effects of the migration.

Denying Access to the Default Bridges: Since bridge relays do not publish their bandwidth capacities, our analysis assumes that the distribution of link capacities for n default bridge relays is the same as the distribution of link capacities for the fastest n non-bridge Tor relays. Thus, saturating one default bridge's Internet connectivity requires an amount of bandwidth equal to the link capacity of the fastest Tor relay, and saturating 10 default bridges' requires bandwidth equal to the combined link capacity of the fastest 10 Tor relays. Following the link capacity estimates based on bandwidth offerings as described in §4, we estimate that the set of 12 operational default bridges consists of two 10 Gbit/s links and ten 1 Gbit/s links (a total of 30 Gbit/s) and that the full set of 38 default bridges consists of two 10 Gbit/s links and thirty-six 1 Gbit/s links (a total of 56 Gbit/s). Recall from the pricing model in §3.1 that a 1 Gbit/s stresser account costs $0.74/hr. Attacking the 12 operational obfs4 bridges thus requires 30 of such stresser accounts at a cost of $0.74·30=$22.20 for each hour of downtime (or roughly $22.20·24·31≈$17K per month). Repairing the remaining default bridges offers only a small improvement: denying service to 38 bridges requires 56 stresser accounts at a cost of $0.74·56=$41.44/hr. ($41.44·24·31≈$31K/mo.) which we posit is well within the budget of a nation-state adversary. We emphasize that these are estimates since bridges' true link capacities are unknown.

If the default bridges are successfully attacked, there are several potential consequences. In the worst case, the set of default bridges will not be updated and the users who had depended on them will abandon Tor altogether. The Tor Project could also update its list of default bridges (e.g., by pushing an update to TBB), but such a solution is only temporary since an attacker could simply retarget its DoS efforts.

Users who are dependent on bridges may switch to using either unlisted bridges (since they are more plentiful and more difficult to enumerate) or to meek bridges.

The Cost of Migrating to Unlisted Bridges: We base our analysis on (i) the distribution of throughput we measure from unlisted bridges (Figure 1), (ii) the simplifying assumption that how Tor is used by bridge users is independent of the particular type of bridge used to gain entry to the network, and (iii) Matic et al.'s observation that suggests approximately 90% of bridge traffic traverses through default bridges [62]. If all default bridge users switched to unlisted bridges, applying our simplifying assumption, we would therefore expect the load on the unlisted bridges to increase by a factor of nine (since they previously carried just 10% of bridge traffic). More generally, when a fraction f of default bridge users shift to using unlisted bridges, the unlisted bridges should expect to see a corresponding increase in traffic of a factor of $9 \cdot f$. This trend is plotted in Figure 2.

Given that most (90%) of bridge traffic that is handled by the default bridges, even a small migration of default bridge traffic to unlisted bridges has performance consequences. Even if a quarter of previously default bridge users switch to unlisted bridges, their performance will significantly suffer, decreasing from 762 KiB/s to 338 KiB/s in the median.

[2] obfs4 obfuscates Tor traffic to appear as a random sequence of bytes, making it hard for DPI systems to classify.

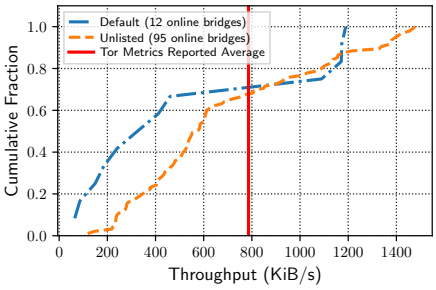


Figure 1: Cumulative distribution of bridge throughput when downloading 6 MiB files. The vertical line at 786 KiB/s shows the throughput for clients that directly connect to Tor to download a 5 MiB file.

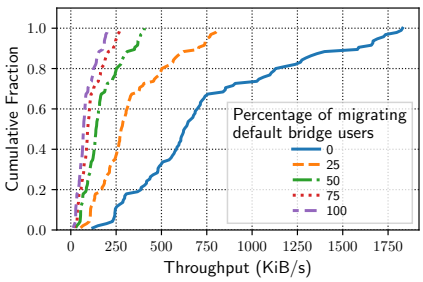


Figure 2: Throughput of Tor users who use unlisted bridges, as a function of the percentage of default bridge users who switch to using unlisted bridges.

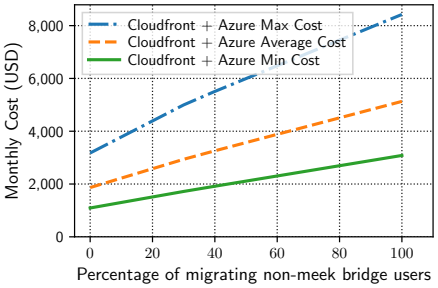


Figure 3: The minimum, maximum, and average cost of maintaining meek when some fraction of users switch to meek, based on meek usage data and CloudFront and Azure pricing models.

The Cost of Switching to Meek Bridges: If the non-meek default bridges become unavailable, we expect some fraction of users to switch to meek bridges. Censors cannot easily disrupt meek bridges without inflicting significant collateral damage since it cannot easily distinguish meek traffic from more typical HTTPS traffic that accesses the cloud provider. However, the increased use of meek incurs a cost (since cloud services are not free).

To estimate the resulting monthly cost of maintaining meek bridges as non-meek users switch over, we estimate the bandwidth consumption of migrating users by constructing a regression model. The bandwidth consumption estimate is required to calculate the cost of operating the meek frontend as cloud providers charge their users based on the bandwidth consumption. To construct the regression model, we first use the statistics from the Tor Metrics Portal to estimate (i) the number of meek users and (ii) the traffic transferred by the meek frontend as reported by meek frontend operators [5] over this same timespan. We use this regression model to estimate the consumed bandwidth as a function of increased traffic from users migrating to meek bridges. We then use the estimated bandwidth usage to derive the expected meek operational cost by applying the current pricing model of the cloud service providers. Unfortunately, the Tor Project stopped providing usage statistics for meek bridges, so we restrict our analysis to usage that occurred between March 2016 and March 2017 (where such data is available [5]). We also note that Amazon and Google stopped supporting the use of domain fronting on their respective cloud services [17, 33][3]. Given these limitations, our analysis represents a rough approximation.

Figure 3 shows the monthly cost of operating meek bridges as a function of the fraction of non-meek bridge users who switch to using meek bridges. We plot the ranges of estimated monthly costs, since cloud providers charge different amounts based on the locations of clients. We note that if half of non-meek users begin using meek bridges, then *in the best case*, the operational cost of maintaining meek bridges will double. If clients are disproportionately in locations in which providers charge higher rates, then the operational costs could be as high as six times the current amount.

Mitigation: Meek bridges offer the best protection against DoS. Unfortunately, supporting a large user base is expensive. One potential strategy for reducing costs is to require bridge users to watch ads or perform small tasks (akin to Mechanical Turk) to finance the cost of their bridge use. Additionally, the recent proliferation of encrypted Server Name Identification (SNI) parameters [36, 44] may enable new methods of domain fronting [32] that are compatible with lower-cost hosting providers.

[3]This highlights a particular brittle aspect of meek bridges—they are completely dependent upon the cloud service on which they reside.

6 Unbalancing Load

In this section, we seek to better understand the extent to which an adversary can disrupt Tor by using stresser services (§3.1) to launch bandwidth DoS attacks on TorFlow [72], a critical component in Tor's load balancing process.

6.1 Relay Performance and Path Selection

As of `2018-11-01`, the Tor network contains 6,436 voluntarily operated relays, the status of which is maintained by 9 Tor *directory authorities* in a signed *network consensus document*. When using Tor, clients download and verify a recent consensus, and use it to select paths of relays through which they build *circuits* and tunnel Internet connections.

Tor uses a load-balancing system in order to provide low-latency anonymous communication (i.e., suitable for browsing websites) due to high client resource demand and a large variance in the bandwidth capacities offered by relays (see §4). The load balancing system is composed of two primary components: a relay performance estimation mechanism and a performance-aware path selection algorithm.

Relay Performance Estimation: Although Tor initially estimated relay performance according to self-reported *advertised bandwidth* capacities, Bauer et al. showed how a low-resource adversary could attract significant traffic to mali-

cious relays (to improve end-to-end correlation attacks) by lying about their available bandwidth [15]. Perry subsequently designed and published the TorFlow relay measurement system to reduce the extent to which Tor trusts relays to honestly report their bandwidth capacity [72], and Tor has been using it to measure relays for nearly a decade (despite alternative designs [19, 55, 78]).

TorFlow is a measurement tool that scans Tor relays to measure their relative performance. To measure Tor relays, TorFlow (i) sorts the consensus list of relays by their previously-expected performance, (ii) partitions the sorted list into *slices* of 50 relays each, (iii) distributes the slices among 9 subprocesses that run in parallel, (iv) creates 2-hop circuits using pairs of relays that belong to the same slice (and so can provide similar performance), and (v) downloads one of a set of 13 fixed-sized files (2^i for $i \in [4, 16]$ KiB) from a known destination through each circuit. TorFlow repeats this process until it has attempted to download through each relay at least 5 times, after which it uses the mean of the measured download completion times to compute a weight for each relay that represents its performance relative to the other measured relays.

The output of the TorFlow measurement process is a *version 3 bandwidth* (V3BW) file specifying the weights and other information about each scanned relay. Currently, 5 of the 9 directory authorities also act as *bandwidth authorities* [3]: they obtain a V3BW file and participate in a voting protocol to determine an authoritative set of relay weights that will appear in the next network consensus. Note that a bandwidth authority operator may obtain a V3BW file by running TorFlow (potentially on a distinct machine from that which runs their directory authority) or by obtaining one from another trusted source that is running TorFlow.

Performance-Aware Path Selection: Tor's path selection algorithm biases relay selection to favor those providing more resources and better performance. Clients using the default selection algorithm will choose relays roughly proportional to the weights assigned to them (via the bandwidth authority voting procedure) and listed in the consensus. As a result, client traffic will be driven to the better performing relays. Previous work has shown that Tor's relay selection strategy does a reasonable job of balancing load [82].

6.2 Detecting TorFlow Scanners

The TorFlow relay scanners constitute attractive targets for DoS attacks: disrupting the scanners may result in significant variation in relays' weights which could degrade load-balancing and security. If the bandwidth authorities were taken offline, Tor would eventually fall back to an equal weighting (uniformly at random) strategy, which would have a detrimental effect on client performance [82]. Previous work observed the ability to detect TorFlow scanners due to their connection patterns and fixed-size file downloads [55], which we further explore.

TorFlow scanners stand out from normal Tor clients because: (i) they download one of a set of 13 fixed-size files, (ii) they choose new entry relays for each circuit (disabling the guard feature), and (iii) they use two-hop circuits.

To discover the network addresses of the TorFlow scanners, we first determined the range in the number of Tor cells required to download each of the fixed-size files. We then operated a low-bandwidth relay and patched it with a small program that looked for connecting clients (potential candidates for TorFlow scanners) that exhibited similar telltale fetches. To provide some ground truth, we also operated our own TorFlow scanner. Within 48 hours, we were able to identify six IP addresses that fetched files through our relay and fit the pattern of a TorFlow scanner. We operated our TorFlow scanner detection software for 5.5 days, during which it did not identify any additional potential scanners. Although we temporarily stored candidate scanner IP addresses in memory (in order to determine uniqueness), we did not write them to stdout or the filesystem (in order to avoid accidentally recording the IP of a human Tor user). We did, however, record that one of the six identified candidate TorFlow scanners was indeed our own; we posit that the other five correspond to the five scanners operated by the Tor Project.

6.3 Attacking TorFlow Scanners

Given that an adversary can identify TorFlow scanners by their IP address, they can use bandwidth DoS attacks to disrupt the relay scanning process and therefore degrade the accuracy of the relay weights produced by TorFlow. A bandwidth DoS attack will clog the TorFlow scanners' links, increasing latency and packet loss on those links and extending the time it takes the scanners to successfully complete file downloads through Tor relays. Therefore, the adversary may effectively manipulate the scanner into believing that relays provide worse performance than they can actually provide. Since TorFlow weights relays by their performance, the adversary can effectively reduce the accuracy of the relay weights which may disrupt the load balancing process.

6.3.1 Attack Strategies

We explore several strategies that an adversary may use to conduct bandwidth DoS attacks on TorFlow scanners with a goal of increasing the file download times measured by TorFlow and disrupting the load balancing process. Each strategy will come at a different cost due to the bandwidth required to conduct the attack and the length at which the attack must be sustained.

Constant: The most straightforward strategy is to simply flood each TorFlow scanner with bandwidth at a constant rate over time. This brute-force strategy is the easiest to set up and should require minimal monitoring and maintenance by the adversary throughout the duration of attack.

Periodic: Since TorFlow produces weights that represent relay performance *relative to other relays*, and because a constant attack strategy may similarly affect all relay measure-

ments, a constant strategy may be suboptimal. Therefore, we also consider a periodic strategy where the adversary floods the victim with bandwidth for a duration of time λ while periodically pausing the attack for a duration of time π. The reasoning behind this strategy is that the scanner will measure normal download times for some relays but significantly reduced download times for others, and the large difference will have a greater impact on the final set of relay weights.

Targeted: We also consider a targeted strategy where the adversary carefully selects periods of time during which to run the DoS attack and otherwise does not alter the victim scanner's network conditions. In particular, we observe that the greatest impact in performance will likely result from significantly depressing the relative weights of the best performing relays. Therefore, the adversary targets the scanner with a bandwidth DoS attack while it is measuring the fastest relays. We discuss below how to determine when the fastest relays are being measured.

6.3.2 Attack Strength and Other Assumptions

For any strategy used by the adversary, we assume that it can utilize a stresser service (see §3.1) to limit the victim's effective bandwidth to rate γ while increasing packet loss on the victim's link by ρ. We assume that the adversary can increase or decrease the attack strength to achieve these effects. (See §6.5 for a discussion of cost.) We also assume that the adversary can receive feedback on the attack by closely monitoring the consensus weights and checking how relays' weights are changing over time. It can monitor the Tor metrics website and data to observe changes in Tor performance. It can iteratively adjust the attack strength and strategy over time in an attempt to produce a greater effect. We also assume that the adversary is capable of setting up and running its own TorFlow scanner instance (the code is open-source), and use it to directly observe how an ongoing attack is affecting the TorFlow measurements and outputs.

The *Targeted* attack strategy depends on being able to target the slice containing the fastest relays. We speculate that the adversary would be able to detect when the fastest slice is being measured by running a fast relay itself and observing when its relay is first measured by a TorFlow scanner. Once detected, the adversary could enable the attack for the time required to measure the slice, which it could estimate empirically by running a TorFlow scanner itself and observing the times to measure the fastest slice over several scan periods. (We ran a TorFlow instance, analyzed its output, and computed the time to measure the fastest slice over 20 scans. We found that the median time to scan the fastest slice was 249 minutes, with an interquartile range of 73 minutes.) Note that these techniques would require additional time, bandwidth, and skill compared to a brute-force attack, and that scan times may be inconsistent over time and network location. See §6.6 for further discussion.

6.4 Evaluation

We evaluated the DoS attack strategies and effects in Shadow [47], a high-fidelity network simulation framework that directly executes Tor. We used Shadow to create a private Tor network that is completely contained inside of our lab environment in order to guarantee that our attacks do not harm the safety or privacy of real Tor users or the network. All of the experiments that we present in this section use Shadow `v1.13.0` and Tor `v0.3.0.10`.

Network Setup: We used standard Shadow and Tor network generation tools and methods [48] to generate a private Tor network with 100 Tor relays, 3,000 Tor clients, and 1,000 server, and to generate background traffic [53]. 2,619 of the clients are *web* clients that download a 320 KiB file, "think" by pausing for a time selected uniformly at random in the range [1,60] seconds, and then repeat. 81 of the clients are *bulk* clients that repeatedly download a 5 MiB file without pausing between successive downloads. We also run 300 *benchmark* clients that reproduce Tor's performance benchmarks by occasionally downloading 50 KiB, 1 MiB, and 5 MiB files using fresh circuits throughout each experiment. We use the most recently published Shadow network topology graph [53] to model inter-host latency.

We implemented a TorFlow plugin for Shadow by significantly refactoring and extending previous work [55]. We used the plugin to scan the relays in our network and produce V3BW files which were then added to the consensus and used by the clients to build paths. We first ran one longer experiment allowing TorFlow time to scan through all relays several times, and then we used the final V3BW file that TorFlow produced as the starting point for all other experiments.

Parameter Settings: We simulated a bandwidth attack by adjusting TorFlow's available bandwidth γ and added packet loss ρ. During each phase where the attack is active, we limit TorFlow's bandwidth to $\gamma = 500$ Kbit/s (62.5 KiB/s) and we add a $\rho = 2\%$ chance of packet loss occurring independently on all incoming and outgoing packets. We set our TorFlow instance to conduct 4 parallel *probes* (2-hop relay measurements), to partition the relays into 10 slices of 10 relays each, and to probe each relay at least 3 times per round before producing a new V3BW file.

We ran a baseline *No Attack* experiment and experiments with each attack strategy. When running the *Constant* attack strategy, the attack is active (the γ and ρ rates applied) for the duration of the experiment. In the *Periodic* attack strategy, the attack cycles through an active period lasting $\lambda = 60$ seconds and an inactive period lasting $\pi = 20$ seconds. In the *Targeted* attack strategy, the attack is active while relays in the slice containing the fastest guard relay in the network are being measured, and inactive otherwise.

TorFlow Scanner Performance: The performance of the TorFlow measurement probe downloads across our experiments is shown in Table 3. As shown in the table, our results indicate that the *Constant* attack is the most effective at caus-

Table 3: The failure rate of TorFlow probe downloads, and the mean (± standard deviation) download rate for each TorFlow probe download and time to complete a full network scan.

Strategy	Fail Rate	Download Rate	Scan Time
No Attack	6.0%	390±381 KiB/s	47±21 min.
Periodic	8.6%	256±292 KiB/s	59±20 min.
Targeted	14%	275±293 KiB/s	80±19 min.
Constant	22%	8.7±5.1 KiB/s	173* min.

*Only a single scan completed in our 300 minute simulation.

ing TorFlow download errors (which increased to 22% from 6% with *No Attack*). The *Constant* attack is also the most effective at limiting the probe download rate, achieving a reduction in mean download rate of about 381 KiB/s, and intuitively increasing the time to scan all relays in the network by about 126 minutes. We find that the *Periodic* and *Targeted* strategies are less effective than the *Constant* strategy, but still do have a measurable effect on the scanner.

Relay Performance: Figure 4(a) shows the relay performance in terms of the distribution of total relay goodput (summed across all relays in the network) over every second during the simulation. We notice a similar trend as with TorFlow performance: in the medians, total relay goodput drops by 86 MiB/s (56%) from 153 MiB/s with *No Attack* to 67 MiB/s with the *Constant* strategy, and relay utilization gets progressively lower with the *Periodic*, *Targeted*, and *Constant* strategies, respectively. Such significant drops in throughput indicates that the new weights produced by TorFlow during the attacks no longer do a good job of balancing client load across relays, and the network is less capable of utilizing its available bandwidth resources.

Client Performance: The effects of our attacks on the mean download rate (during active downloads) per client are shown in Figure 4(b). Every attack has a significant effect, with the mean download rate of the median client being reduced by 45 KiB/s from 56 KiB/s with *No Attack* to 11 KiB/s with the *Constant* attack. Client performance also suffers in terms of the download failure rate per client as shown in Figure 4(c): the failure rate for the median client increases by about 23% from about 3% with *No Attack* to about 26% with the *Constant* attack.

Overall, our results show the extent to which an adversary may disrupt Tor performance using straightforward DoS attacks on easy-to-detect TorFlow scanners, and that the simplest constant attack strategy was the most effective.

6.5 Attack Cost

We assume that our TorFlow DoS attacks could be launched using a stresser service. In §3.1 we describe that the amortized cost of a stresser service to provide 1 Gbit/s of attack traffic is $0.74/hr. The *Constant* attack strategy requires that we constantly run the DoS attack on each scanner. Tor runs 5 TorFlow scanners of unknown capacity. If we assume that they all run on 1 Gbit/s links, then the cost to run the DoS attack on all 5 scanners for one month would be $0.74·5·24·31≈$2.8K.

6.6 Discussion

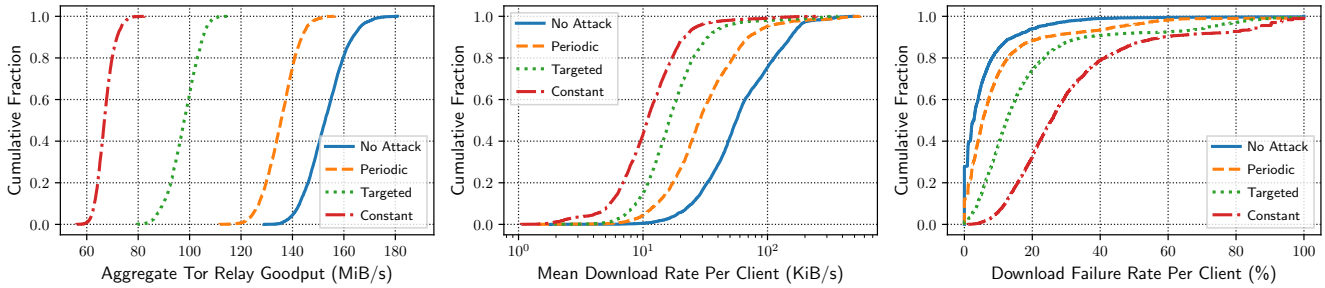
Limitations: A limitation of our study of the effects of bandwidth DoS on the TorFlow scanners is that we used a smaller-scale Tor network than that which is publicly accessible (100 relays compared to 6,436). We used a smaller network primarily due to resource limitations and because TorFlow takes a significant amount of time to scan all relays. Using a smaller network allowed us to (i) run longer experiments, (ii) scan the network faster because there are fewer relays to measure, and (iii) complete more scanning rounds.

Due to scale we configured a single TorFlow instance in our experiments measuring 10 relays per slice, using 4 parallel probe subprocesses, and collecting at least 3 probe measurements per relay; Tor runs 5 TorFlow instances measuring 50 relays per slice while using 9 parallel probe subprocesses and collecting at least 5 probe measurements per relay. The process of partitioning a larger set of relays among more slices and more parallel subprocesses could lead to different inconsistencies than those captured by our simulations. We believe that running additional parallel subprocesses would increase the average bandwidth rate of the TorFlow scanner, which may increase the effectiveness of a constant attack strategy. However, the adversary would require additional bandwidth to attack all or a majority of scanners in parallel.

The *Targeted* attack requires the ability to estimate when the fastest slice is being measured and the amount of time required to measure that slice. These estimates may be complicated by TorFlow's inconsistent relay partitioning and subprocess assignment functions and its parallel measurement processes. In the worst case, the *Targeted* attack would degrade to a *Constant* attack, which performed best in our experiments anyway and for which we estimated cost in §6.5.

The attacks depend on stresser services delivering a high rate of traffic to the target, which is not a stealthy operation. This could potentially trigger automated or manual DoS mitigation techniques, and we did not consider how such defenses would affect the attacks. We rely on the assumption that packets are not filtered upstream from the target scanner, which may not always be valid (as we will discuss below).

Attack Extension: Our focus in this paper is on relatively simple and straightforward bandwidth-based DoS attacks. However, it is possible to extend the attack if we consider a more powerful adversary. For example, a network-level adversary that can observe connections from a TorFlow scanner can selectively disrupt TorFlow as it is scanning a target set of relay IP addresses. This would allow an adversary to selectively increase the time to scan the target set of relays, causing the scanner to detect that those relays are "overloaded" and reduce their weights (and therefore the probability that those relays are used by clients) accordingly. Such an attack could be used to drive additional traffic to other mali-



(a) Relay Utilization **(b)** Client Performance **(c)** Client Error Rate

Figure 4: Performance metrics as measured when the network is not under attack (*No Attack*) and when the bandwidth authorities are attacked using different strategies (*Periodic*, *Targeted*, and *Constant*): (a) the distribution of aggregate Tor relay goodput per second (summed across all relays for every second); (b) the distribution of mean download rates per client; and (c) the distribution of download failure rates per client.

cious relays, improving the ability of an adversary to conduct attacks on anonymity, e.g., traffic correlation attacks [66].

Mitigation: Since the attacks rely on stresser services, the mitigation strategies discussed in §4 also apply here. Specifically, it is possible that ISPs could render attacks against the bandwidth authorities ineffective by filtering traffic en route. For example, ISPs could discover hosts belonging to the stresser services and filter traffic originating at those hosts, or ISPs could install custom rules to filter incoming traffic to the bandwidth authorities since they need only make outgoing connections (in case they are not run alongside Tor relays). However, filtering attempts may be complicated by packet spoofing. (Note that dropping packets on the bandwidth authority is an ineffective defense since the dropped packets already consumed bandwidth.)

Perhaps the best mitigation is to migrate to a decentralized bandwidth measurement system that does not share TorFlow's security problems. For example, TorFlow's centralized scanning approach can be easily detected because it uses fingerprintable traffic signatures from a small set of static IP addresses. While it may be difficult to obfuscate the source, destination, and traffic signature while still providing accurate results [55], a system that utilizes distributed trust may help thwart malicious behavior. For example, a peer-based measurement system run by existing relays would preclude the need for centralized measurement infrastructure that is vulnerable to DoS and could complicate scanner and measurement detection [55, 78].

7 Congesting Tor Relays

In §4, §5, and §6 we evaluated the effects of using stresser services (§3.1) to flood Tor relays, Tor bridges, and Tor bandwidth authorities, respectively. In this section, we move away from stresser services and explore the extent to which an adversary might degrade Tor network performance by using the Tor protocol itself to congest Tor relays. The attack strategies that we discuss in this section utilize dedicated servers (§3.2) and modified Tor clients.

7.1 Relay Usage

All non-bridge Tor relays are publicly known and distributed to Tor clients in network consensus documents in order to facilitate the Tor client path selection and circuit building processes. By default, clients select 3-hop Tor paths and use them to build circuits that are usable for 10 minutes. Clients must use an *exit relay* that allows exiting the Tor network on the desired TCP port as the last relay in their Tor circuits in order to communicate with Internet peers that are not Tor-aware. Additionally, by default, clients use *guard relays* as their entry into the Tor network, and guard or *middle relays* (i.e., non-guard, non-exit relays) in the middle position of their circuits. As discussed in §6, each relay is assigned a weight by the directory authorities corresponding to its performance relative to all other relays as measured by TorFlow [72]. Clients use these weights during path selection to bias their choice of relays toward those providing better performance.

7.2 Abusing Relay Bandwidth

The Tor protocol contains features that offer protocol flexibility, but also allow for abuse. Although exits are required to be used in the last position of circuits exiting Tor, the Tor protocol technically allows any relay to be used in any non-exit position. Additionally, the Tor protocol technically allows circuits containing up to 8 relays. To utilize these features, a client may select its own custom path of relays and build circuits through them either by modifying their Tor client code directly, or by interacting with Tor using the Tor control interface and protocol. Note that building custom circuits is already supported by existing Tor control clients like stem [8].

7.2.1 Attack Strategies

Given the above features, an adversary may conduct a concerted bandwidth consumption DoS attack by building custom circuits and downloading large files through them.

Long Paths: The most basic form of our Tor bandwidth DoS attack makes use of the ability to create 8-hop Tor circuits.

For every byte of data downloaded by a client through such a *long path*, relays in the Tor network will download and upload that byte 8 times in total. This amplification works significantly in favor of the adversary.

Tunneling: Tor previously allowed infinite-length circuits until it was shown that long paths (e.g., 24 hops) could be used to congest Tor relays and deanonymize clients [31]. Although the Tor protocol now restricts circuits to 8 relays in length [27, §5.6], paths of unrestricted length are still technically possible by using multiple Tor clients and tunneling each client's TCP onion connection to its entry relay through another client's circuit[4] [51].

Stop Reading: In order to decrease the cost of downloading large files, the adversary may use a *Stop Reading* strategy. Using this strategy, the adversary first creates a new TCP onion connection to the first-hop relay in its chosen long path, even if a connection to that relay already exists. The adversary then builds an attack circuit using its chosen path and waits for it to complete successfully, making sure to assign the circuit to the new TCP connection. Once the circuit is built, the adversary sends a request for a large data blob from some public server (e.g., the Internet Archive), measures the time to download the first 25 KiB, and then instructs Tor to stop reading from the circuit's TCP connection to the entry relay (immediately after receiving 25 KiB). Although the client stopped reading, it can still write: the adversary uses the measured time to download the first 25 KiB to estimate the frequency with which it should send Tor circuit and stream `SENDME` flow control cells[5] which instruct the exit relay to continue to send data toward the client. (Estimating the circuit throughput rate is important, because sending more `SENDME` cells than is expected by the exit will cause the exit to abort the circuit.)

A stop reading strategy was first described and used as part of the Sniper Attack [51], but we are the first to observe that each attack circuit should use a new and unique TCP connection in order to limit interference with other circuits built in parallel. This requires minor modifications to the Tor client code since Tor by default multiplexes circuits over existing TCP connections. We show in §7.3 that our bandwidth DoS attack scales to thousands of circuits using this approach.

7.2.2 Attack Targets

Single Relay: An adversary may use an 8-hop long path to conduct a congestion attack on a target victim relay by including the victim in the same circuit multiple times. Since an honest relay will not extend a circuit to the same relay that extended to it, the victim v must be placed in circuit positions such that two distinct honest relays h_1 and h_2 are placed in subsequent positions before repeating the victim (i.e., $v \leftrightsquigarrow h_1 \leftrightsquigarrow h_2 \leftrightsquigarrow v$, etc.). Therefore, a victim may appear in the same circuit a maximum of 3 times (either in positions 1, 4, and 7, or 2, 5, and 8).

Relay Subgroups: An adversary may also target specific subgroups of relays that represent particularly attractive targets. Such subgroups may include the group of all exit relays (since their resources are the most scarce), the group of all publicly known bridges (we explored the impact of such an attack in §5), hidden service directories, and the group of 9 directory authorities (which also serve as relays).

All Relays: An adversary may also attempt to congest the entire Tor network with the goal of degrading performance to the extent that Tor becomes unusable to a majority of its user base, which would significantly reduce Tor's security in addition to its performance [25]. In order to congest all relays, the adversary makes a weighted selection of relays following the weights published in the network consensus. In other words, the adversary uses the same path selection policy as honest clients do by default. This will ensure that the adversary will choose and congest relays with the same distribution that clients attempt to use them: the DoS attack will cause more congestion on relays that are chosen more often by clients, and should therefore impact more users.

7.2.3 Attack Strength

The strength of our attack can be described in terms of the number of long path circuits ϕ that the adversary builds in parallel. Each circuit should use at least 2 parallel streams (i.e., downloads) in order to fully utilize the circuit flow control mechanism (the circuit window is 1,000 cells, twice that of the stream window). Whenever circuits close or downloads finish (complete or time out), circuits are replaced with new ones in order to maintain the attack strength over time.

7.3 Evaluation

As in §6.4, we use Shadow [47] to safely measure the effects of our DoS attacks in a private Tor network. All experiments in this section use Shadow `v1.13.0` and Tor `v0.3.1.10`.

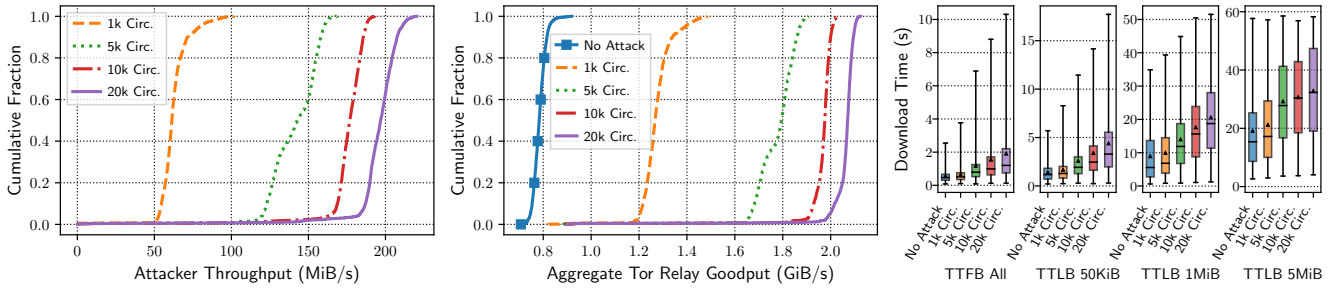
Network Setup: We use the same tools and methods as described in §6.4 to generate a Tor network containing 634 relays (10% of the size and capacity of the public network), 15,000 clients (14,259 *web* clients, 441 *bulk* clients, and 300 *benchmark* clients), and 2,000 servers. Node behaviors are also as described in §6.4.

We implemented our DoS attacks in a C program containing 3,265 lines of code (LoC) which we compiled as a Shadow plugin. Our attacks utilize a Tor `v0.3.1.10` client that we modified (409 LoC) to support creating new TCP connections for attack circuits as well as commands to stop reading and for sending `SENDME` cells.

Parameter Settings: Throughout our experiments, we explore the effects of the *Long Path* strategy across attack strengths (number of circuits ϕ) on performance. We parallelize our attack by running $\phi/1,000$ identical processes on new attack hosts (i.e., 1,000 circuits per host), each of

[4] Tor will tunnel TCP onion connections through a proxy (e.g., another Tor client) when using the `Socks4Proxy` or `Socks5Proxy` torrc options.

[5] A stream `SENDME` cell is sent for every 50 received cells (25 KiB) and a circuit `SENDME` for every 100 received cells (50 KiB).



(a) Attacker Throughput **(b)** Relay Goodput **(c)** Client Performance

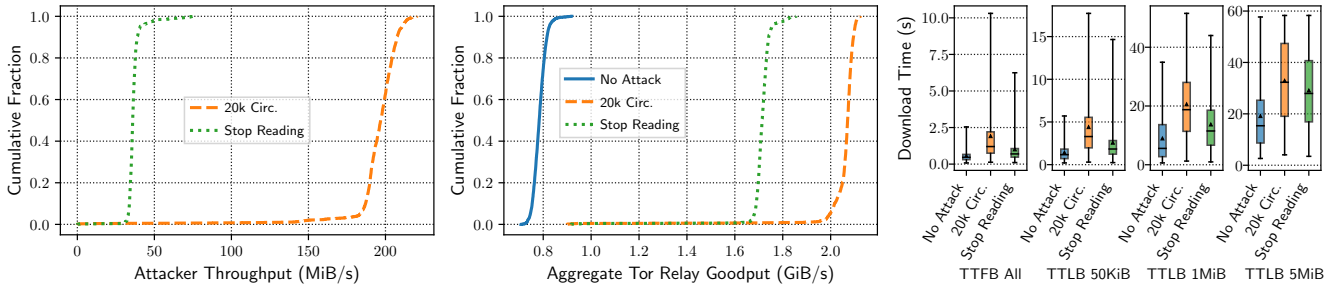
Figure 5: Performance metrics as measured throughout our network-wide DoS attack on Tor relays: (a) the distribution of attacker throughput rates; (b) the distribution of aggregate Tor relay goodput rates (summed over all relays for each second); and (c) the distribution of *benchmark* client download times to first byte (TTFB) and last byte (TTFB) for files of sizes 50 KiB, 1 MiB, and 5 MiB (the box shows the interquartile range, the ▲ shows the mean, and the lower and upper whiskers extend to the minimum and the 99th percentile values, respectively).

which is configured with a 1 Gbit/s network link to ensure that we could measure the full bandwidth cost of the attack. We attach 2 streams that each request 10 MiB of data to each 8-hop attack circuit whose path is chosen depending on the target (we explore a single relay and all relays as targets). We set each attack circuit to time out if any of its streams either have not completed within 5 minutes, or have not received a byte in the most recent 60 seconds. Finally, we run experiments with and without the *Stop Reading* strategy in order to understand the cost and impact of DoS with and without it. (We did not evaluate the *Tunneling* strategy or a strategy that targets *Relay Subgroups*.)

Single Relay Attack: We evaluated the *Long Path* strategy in a *Single Relay* attack against the most highly weighted middle relay in our network. We evaluated attack strengths of 100, 500, and 1,000 circuits. In all cases, the attacker required less than 2.6 MiB/s of throughput to conduct the attack. The victim relay's throughput increased from 978 KiB/s with no attack to 3.8 MiB/s with 100 attack circuits and 5 MiB/s with both 500 and 1,000 attack circuits (in the medians). Interestingly, 500 circuits was enough to consume all of the victim's 5 MiB/s capacity. In the medians, the time to download 1 MiB through the victim increased from 3.3 seconds with no attack to 28 seconds with 1,000 circuits, while the download failure rate increased from 0% with no attack to 63% with 1,000 attack circuits. Our results indicate that the attack has a clear effect on both relay throughput and client performance on a small scale.

All Relays Attack: We evaluated the *Long Path* strategy in an *All Relays* attack using ϕ = 1k, 5k, 10k, and 20k attack circuits. The performance measured during the attacks and compared to a no attack baseline experiment is shown in Figure 5. Figure 5(a) shows the cumulative distribution of the total attacker throughput used during each attack. Intuitively, the throughput used by the adversary increases with the number of attack circuits: from a median of 61 MiB/s in a ϕ=1k circuit attack to a median of 197 MiB/s in a ϕ=20k circuit attack. Note that the increase in attacker throughput is not linear in ϕ, indicating that we may be reaching relay bandwidth resource limits. Figure 5(b) supports this claim, showing the distribution over each second of Tor network goodput (summed over all relays). In the medians, aggregate relay goodput increases from 802 MiB/s with no attack to 1,297 MiB/s for ϕ=1k (an increase of 495 MiB/s) and 2,120 MiB/s for ϕ=20k (an increase of 1,318 MiB/s). Interestingly, we can see from these results that the effect of ϕ=10k circuits is much greater than the effect of doubling the attack strength to ϕ=20k circuits, suggesting diminishing returns. Finally, the effect of our attack on client performance is shown in Figure 5(c) across a range of download time metrics. Generally, the time to complete downloads of various sizes increases significantly with the attack strength (e.g., TTFB increases by 138% and TTLB increases by 120% in the medians across all downloads for ϕ=20k), and the variance in performance also increases as attack circuits are added to the network.

Figure 6 shows the effect of the *Stop Reading* strategy on performance. Compared to the regular *Long Path* strategy in the ϕ=20k attack, Figure 6(a) shows that the *Stop Reading* strategy consumed only 36 MiB/s of attacker bandwidth in the median while Figure 6(b) shows that the attack was still able to consume 1,755 MiB/s of total relay bandwidth in the median (an increase of 953 MiB/s over no attack). Figure 6(c) shows that the effect on client performance is also slightly less pronounced (TTFB increases by 48% and TTLB increases by 47% in the medians across all downloads), which is to be expected given that relays are less congested. We expect that we could further increase congestion by scaling up the *Stop Reading* attack at relatively little bandwidth cost to the attacker. A summary of our results in Table 4 shows that the *Stop Reading* strategy achieved the highest bandwidth amplification factor of 26 primarily due to the reduction in attacker bandwidth usage.



(a) Attacker Throughput **(b)** Relay Goodput **(c)** Client Performance

Figure 6: The effects of the *Stop Reading* attack on the performance metrics as measured throughout our network-wide 20,000 circuit DoS attacks on Tor relays: (a) the distribution of attacker throughput rates; (b) the distribution of aggregate Tor relay goodput rates (summed over all relays for each second); and (c) the distribution of *benchmark* client download times to first byte (TTFB) and last byte (TTFB) for files of sizes 50 KiB, 1 MiB, and 5 MiB (the box shows the interquartile range, the ▲ shows the mean, and the lower and upper whiskers extend to the minimum and the 99th percentile values, respectively).

Table 4: Summary of the total increase in relay bandwidth usage our attack achieved with the given attacker bandwidth usage (all in MiB/s) and the resulting bandwidth amplification factors.

	ϕ=1k	ϕ=5k	ϕ=10k	ϕ=20k	Stop*
Relay Bandwidth	495	1,035	1,221	1,318	953
Attacker Bandwidth	61	143	177	197	36
Amplification Factor	8.1	7.2	6.9	6.7	26

* **Stop** is a ϕ=20k attack using the *Stop Reading* strategy.

7.4 Attack Cost

We assume that our DoS attacks could be launched on a dedicated server. In §3.2 we describe that the amortized cost in the dedicated server model for 1 Gbit/s of traffic is $0.70/hr. (which includes the monthly hardware rental and any bandwidth costs). Our ϕ=20k attack with the *Stop Reading* strategy requires only 288 Mbit/s (36 MiB/s) in our scaled down private Tor network which contains about 10% of both the number of relays in and the bandwidth capacity of the public Tor network (due to the relay sampling approach of Jansen et al. [48]). If we assume that the attack scales linearly with Tor's bandwidth capacity, then our attack would require 10·288 Mbit/s≈3 Gbit/s. The cost to rent three dedicated servers supporting 1 Gbit/s of traffic for one month would then be $0.70·3·24·31≈$1.6K. (The regular version of the attack consumes ≈4 times as much bandwidth, requiring 12 dedicated servers and costing $0.70·12·24·31≈$6.3K/mo.)

Our attack additionally utilizes multiple client IP addresses, each of which is used to maintain an average of 1,000/634≈1.6 circuits per relay. If (i) the public Tor network contains 10·634=6,340 relays, (ii) we create 10·20,000=200,000 circuits, and (iii) we maintain the average 1.6 circuits per relay per client IP address rate, then we would require 200,000/6,340/1.6≈20 client IP addresses. If we assume that one IP address is provided for each of our three dedicated servers, and the cost is $5 per additional IP address, then the additional monthly cost for purchasing 17 more IP addresses is 17·$5=$85.

Thus, we estimate that the total cost to run a ϕ=200k circuit attack using the *Stop Reading* strategy against the public Tor network is still $1.6K/mo. due to rounding. (We estimate that the cost of the regular version of the attack is still $6.3K/mo. due to rounding.)

7.5 Discussion

Limitations: We used Shadow in order to ethically conduct full-network Tor simulations, and simulation inherently incurs some inaccuracy. However, while no simulator is perfect, Shadow has been shown to exhibit network behavior and performance that is very similar to Linux [50, 52]. Additionally, Shadow has been used to measure performance when Tor is generally overloaded (as in our evaluation) [47, 49, 52], and it has been used to measure the effects of specific DoS attacks against Tor [51].

Our experiments are limited in scale. We simulated a private Tor network with about 10% of both the number of relays in and the bandwidth capacity of the public Tor network. The cost to the adversary to conduct our attack may not scale linearly with the amount of Tor capacity as we assumed in §7.4, or there may be other issues that arise when scaling up our attack. We note that we are limited by the capabilities of our tools and resources and highlight that it would be unethical to conduct this work at scale on the public Tor network.

Attack Extensions: We did not evaluate the effects of onion connection tunneling on DoS (i) because Tor could prevent the attack by updating the default exit policy to prevent exiting to a Tor relay, and (ii) in order to provide a more conservative estimate of the bandwidth and monetary costs of performing our bandwidth DoS attack. However, we believe that the technique would be simple to deploy. Additionally, it would be interesting to explore the effects of our attacks on performance when targeting subgroups of relays.

Mitigations: It is extremely challenging to mitigate bandwidth DoS attacks on Tor because the circuits that we build in our attack download an amount of traffic that a reasonable client could realistically download. The *Long Path* part of the attack could be mitigated if Tor changes its protocol to further restrict the length of circuits, however, in this case an adversary could switch to using hidden service circuits which are 6 hops by default.

The *Stop Reading* part of our attack uses a separate TCP connection to the entry relay for each attack circuit, and builds many such connections and circuits in parallel. Tor implemented mitigations to this kind of DoS attack and merged them in early 2018 [24, 39] in response to reports of DoS against relays [23, 38, 40]. In the new subsystem, relays will refuse new TCP connections from any IP address that creates more than 3 concurrent connections, and they will refuse new circuits from the IP address if it *also* creates more than 3 circuits per second with an allowable burst of 90 circuits (these were the default settings on 2018-11-01). Our Tor experiments were configured to run with these DoS mitigations in place, and our attacks did not trigger the DoS defense on any relay. Our attacks were able to stay under the connection threshold because we utilize every relay in the network as an entry relay and we maintain only 1,000 circuits per client IP address (1.6 circuits per relay per client IP address on average). While we do believe that the implemented mitigations are effective against some attacks, we note that the proliferation of IPv6 addressing may further reduce their effectiveness.

A defense against the Sniper Attack [45, 51] was merged in Tor `v0.2.4.14-alpha` (released on 2013-06-13). The defense detects and kills the circuit with the longest waiting cell at the head of the queue if the relay is under memory (RAM) pressure. The defense was active but was not triggered in our experiments since we only download 20 MiB of data through each circuit before abandoning it (our goal is to consume bandwidth rather than a victim's RAM as in the Sniper Attack, so we do not require long queues).

In order to further limit the impact of the *Stop Reading* strategy, we recommend the implementation and deployment of the authenticated `SENDME` design as previously described [51] and specified [46]. With authenticated `SENDME`s, a client would need to continue reading data in order to continue producing authentic `SENDME` cells, and the exit would destroy circuits on which it received invalid `SENDME`s. This defense would limit a stop reading DoS strategy to 1,000 cells (500 KiB) per circuit, effectively mitigating it.

8 Sybil Attacks

We previously explored several bandwidth-based DoS attacks against Tor while estimating the cost to conduct each attack and their effects on Tor performance; we summarize our cost estimates in Table 5. In this section, we compare our DoS attacks with a Sybil attack in which an adversary instead uses its budget to run several high-bandwidth Tor relays in order to affect as much Tor user traffic as possible.

Table 5: A summary of the costs of our main attacks.

Attack (Section)	Service	Bandwidth	Cost
Bridge Congestion (§5)	stresser	30 Gbit/s	$17K/mo.
Load Unbalancing (§6)	stresser	5 Gbit/s	$2.8K/mo.
Relay Congestion (§7)	ded. server	3 Gbit/s	$1.6K/mo.

Table 6: The effective mean aggregate bandwidth resources of Tor relays from `2017-11-01` to `2018-11-01` (in Gbit/s), computed using positional bandwidth weights from `2018-11-01`.

Bandwidth	Entry	Middle	Exit	Total
Usage	42.4 (36.0%)	42.9 (36.4%)	32.5 (27.6%)	118
Capacity	86.7 (35.0%)	96.7 (39.1%)	64.0 (25.9%)	247

Relay Resources: To determine which type of relays would be most advantageous, we computed the effective positional bandwidth usage by and capacity of Tor relays over the year preceding `2018-11-01`. The *effective bandwidth* accounts for relay flags and position weights, both of which are used to determine in which position a relay will be selected. From the results shown in Table 6, we can see that exit bandwidth is the scarcest, with only 27.6% of the total bandwidth used and 25.9% of the total bandwidth capacity.

Sybil DoS Attack: An adversary could run Sybil relays and then arbitrarily degrade the performance of all traffic forwarded through its Sybils, or deny service by dropping circuits. Note that for such an attack to work, the adversary must (i) maintain a high selection probability by providing high performance during periods in which it is measured by Tor's bandwidth measurement system, and (ii) not trigger Tor's abusive relay detection systems (e.g., exit scanners) to avoid getting ejected from the network. We assume that these requirements can be met for the purposes of this analysis.

Due to exit bandwidth scarcity, an adversary can maximize its probability of appearing at least once in a circuit by running all exit relays. We assume that the aggregate bandwidth *usage* (i.e., network load) will remain constant as the adversary adds additional bandwidth *capacity* (i.e., Sybil relays), and that the probability that the adversary serves as the exit in a circuit is approximately equal to its fractional exit capacity (Table 6). Then, Sybil DoS attacks with bandwidth budgets of 30, 5, and 3 Gbit/s (Table 5) could arbitrarily degrade performance for $30/(30+64)\approx 32\%$, $5/(5+64)\approx 7.2\%$, and $3/(3+64)\approx 4.5\%$ of exit circuits, respectively. Comparatively, our attack in §5 affects *all* non-private bridge circuits, and our attacks in §6 and §7 affect *all* circuits.

Sybil Deanonymization Attack: If an adversary is able to observe both the entry and exit points in a circuit (its relays are chosen in the first and last circuit positions), then it is generally assumed that the circuit is *vulnerable* to compromise because traffic correlation can be performed to deanonymize the user with high probability [65, 66]. Note

Table 7: The fraction of circuits affected by Sybil attacks.

Bandwidth	Sybil DoS	Sybil Deanonymization
30 Gbit/s	32% degraded	21% entry · 5.3% exit ≈ 1.1% total
5 Gbit/s	7.2% degraded	4.5% entry · 1.2% exit ≈ 0.06% total
3 Gbit/s	4.5% degraded	2.8% entry · 0.8% exit ≈ 0.02% total

that a selective service refusal attack, where an adversary refuses to forward traffic on any circuit it is not in a position to compromise [16], could be mitigated by Tor's route manipulation (path bias) detection system [26, §7].

In order to observe both ends, an adversary must operate at least one entry guard and at least one exit relay. The entry position is more difficult to obtain since Tor clients use the same guard relay for months at a time [29]. Therefore, previous work has found that a 5:1 guard-to-exit relay bandwidth allocation maximizes the probability of observing both sides of a circuit at least once [54].

A Sybil deanonymization attack with a bandwidth budget of 3 Gbit/s (Table 5) and a 5:1 guard-to-exit relay bandwidth allocation would allow the adversary to observe the entry for $\frac{5}{6}\cdot 3/(\frac{5}{6}\cdot 3+86.7)\approx 2.8\%$ of Tor clients and observe the exit for $\frac{1}{6}\cdot 3/(\frac{1}{6}\cdot 3+64.0)\approx 0.8\%$ of circuits built by those clients. Thus, approximately 0.02% of circuits would be vulnerable. Table 7 shows results for other bandwidth budgets and summarizes our Sybil attack analysis.

Discussion: Note that Sybil attacks require fixed costs, because relays must generally be fast and reliable in order to be properly utilized by the network. Additionally, attacks that require guard relays can take months to observe a full set of *some* clients (due to guard rotation times), and much longer to observe a set containing *specific* clients. Conversely, our attacks are more flexible because they do not require fixed costs, can be run with clients rather than service providers (relays), and can be repeatedly started and stopped as necessary. Further, our attacks immediately affect all clients (rather than some sample), and our relay congestion attack (§7) benefits from the anonymity that Tor provides.

9 Conclusion

This paper performs a multifaceted examination of Tor's vulnerability to DoS, considering both the efficacy of DoS attacks as well as the adversary's cost of performing them. On the positive side, we find that Tor's growth has made it more resilient at least to simple attacks: disrupting the service by naïvely flooding Tor relays using stresser services is an expensive proposition and requires $7.2M/month.

Unfortunately, however, several aspects of Tor's design and rollout make it susceptible to more advanced attacks. We find that Tor's bridge infrastructure is heavily dependent on a small set of fixed default bridges, the operational of which can be disrupted at a cost of $17K/month. Additionally, Tor's mechanism for measuring load is too centralized and brittle, and even inexpensive techniques (e.g., costing $2.8K/month) can significantly perturb these processes and cause dramatic performance degradation across the network. Finally, attackers can saturate Tor's capacity by constructing long paths in the network, and exploit protocol vulnerabilities to decrease the costs of such attacks; for example, we find that an attacker can significantly degrade the performance of the network for as little as $1.6K/month. We also compare our attacks to Sybil attacks and highlight that our load balancing and relay congestion attacks are more effective and flexible than Sybil attacks with the same budget.

For each attack, we describe mitigation strategies that Tor could adopt to improve its resiliency. In particular, we recommend additional financing for meek bridges, moving away from load balancing approaches that rely on centralized scanning, and Tor protocol improvements (in particular, the use of authenticated `SENDME` cells).

Acknowledgments

We thank the anonymous reviewers for their valuable feedback that helped to improve this paper. We thank Nikita Borisov for shepherding our paper and David Goulet for discussions about DoS mitigation in Tor. This work has been partially supported by the Office of Naval Research, the National Science Foundation under grant number CNS-1527401, and the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-16-C-0056. The opinions, findings, and conclusions or recommendations expressed in this work are strictly those of the authors and do not necessarily reflect the official policy or position of any employer or funding agency.

References

[1] AWS Shield Managed DDoS protection. `https://aws.amazon.com/shield/`, November 2018.

[2] Azure DDoS Protection Standard overview. `https://docs.microsoft.com/en-us/azure/virtual-network/ddos-protection-overview`, November 2018.

[3] Consensus Health: Bandwidth Scanner Status. `https://consensus-health.torproject.org/#bwauthstatus`, November 2018.

[4] meek. `https://trac.torproject.org/projects/tor/wiki/doc/meek`, November 2018.

[5] meek Costs. `https://trac.torproject.org/projects/tor/wiki/doc/meek#Costs`, November 2018. Meek Pluggable Transport Frontend Costs.

[6] meek Overview. `https://trac.torproject.org/projects/tor/wiki/doc/meek#Overview`, November 2018. Meek Pluggable Transport Overview.

[7] Online Survival Kit. `https://rsf.org/en/online-survival-kit`, November 2018.

[8] Stem: a Python Controller Library for Tor. `https://stem.torproject.org`, November 2018.

[9] Tor Git Repository Browser. `https://gitweb.torproject.org`, November 2018.

[10] Tor Bug Tracker and Wiki. https://trac.torproject.org, November 2018.

[11] Tor Metrics Portal. https://metrics.torproject.org/, November 2018.

[12] M. AlSabah and I. Goldberg. PCTCP: Per-circuit TCP-over-IPsec Transport for Anonymous Communication Overlay Networks. In *Conference on Computer and Communications Security (CCS)*, 2013.

[13] M. Alsabah and I. Goldberg. Performance and Security Improvements for Tor: A Survey. *ACM Comput. Surv.*, 49(2):32:1–32:36, September 2016.

[14] M. V. Barbera, V. P. Kemerlis, V. Pappas, and A. D. Keromytis. CellFlood: Attacking Tor Onion Routers on the Cheap. In *European Symposium on Research in Computer Security (ESORICS)*, 2013.

[15] K. Bauer, D. McCoy, D. Grunwald, T. Kohno, and D. Sicker. Low-resource Routing Attacks Against Tor. In *Workshop on Privacy in the Electronic Society (WPES)*, 2007.

[16] N. Borisov, G. Danezis, P. Mittal, and P. Tabriz. Denial of Service or Denial of Security? In *Conference on Computer and Communications Security (CCS)*, 2007.

[17] R. Brandom. Amazon Web Services Starts Blocking Domain-fronting, Following Google's Lead. https://www.theverge.com/2018/4/30/17304782/amazon-domain-fronting-google-discontinued, April 2018. The Verge Online News Article.

[18] X. Cai, X. C. Zhang, B. Joshi, and R. Johnson. Touching from a Distance: Website Fingerprinting Attacks and Defenses. In *Conference on Computer and Communications Security (CCS)*, 2012.

[19] H. Darir, H. Sibai, N. Borisov, G. Dullerud, and S. Mitra. TightRope: Towards Optimal Load-balancing of Paths in Anonymous Networks. In *Workshop on Privacy in the Electronic Society (WPES)*, 2018.

[20] R. Dingledine. Research Problems: Ten Ways to Discover Tor Bridges. https://blog.torproject.org/research-problems-ten-ways-discover-tor-bridges, October 2011. Blog Post.

[21] R. Dingledine. Tor security advisory: "relay early" traffic confirmation attack. https://blog.torproject.org/tor-security-advisory-relay-early-traffic-confirmation-attack, July 2014. Blog Post.

[22] R. Dingledine. Did the FBI Pay a University to Attack Tor Users? https://blog.torproject.org/did-fbi-pay-university-attack-tor-users, November 2015. Blog Post.

[23] R. Dingledine. could Tor devs provide an update on DOS attacks? Tor-Relays Email 014175, December 2017. https://lists.torproject.org/pipermail/tor-relays/2018-January/014175.html.

[24] R. Dingledine. Experimental DoS mitigation is in tor master. Tor-Relays Email 014357, January 2018. https://lists.torproject.org/pipermail/tor-relays/2018-January/014357.html.

[25] R. Dingledine and N. Mathewson. Anonymity Loves Company: Usability and the Network Effect. In *Workshop on the Economics of Information Security (WEIS)*, 2006.

[26] R. Dingledine and N. Mathewson. Tor Path Specification. https://gitweb.torproject.org/torspec.git/tree/path-spec.txt, November 2018. Section 7.

[27] R. Dingledine and N. Mathewson. Tor Protocol Specification. https://gitweb.torproject.org/torspec.git/tree/tor-spec.txt, November 2018. Section 5.6.

[28] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium*, 2004.

[29] R. Dingledine, N. Hopper, G. Kadianakis, and N. Mathewson. One Fast Guard for Life (or 9 Months). In *Privacy Enhancing Technologies Symposium (PETS)*, 2014.

[30] Z. Durumeric, E. Wustrow, and J. A. Halderman. ZMap: Fast Internet-wide Scanning and its Security Applications. In *USENIX Security Symposium*, 2013.

[31] N. S. Evans, R. Dingledine, and C. Grothoff. A Practical Congestion Attack on Tor using Long Paths. In *USENIX Security Symposium*, 2009.

[32] D. Fifield, C. Lan, R. Hynes, P. Wegmann, and V. Paxson. Blocking-resistant Communication through Domain Fronting. In *Privacy Enhancing Technologies Symposium (PETS)*, 2015.

[33] S. Gallagher. Google Disables "Domain Fronting" Capability Used to Evade Censors. https://arstechnica.com/information-technology/2018/04/google-disables-domain-fronting-capability-used-to-evade-censors, April 2018. Ars Technica Online News Article.

[34] J. Geddes, R. Jansen, and N. Hopper. How Low Can You Go: Balancing Performance with Anonymity in Tor. In *Privacy Enhancing Technologies Symposium (PETS)*, 2013.

[35] J. Geddes, R. Jansen, and N. Hopper. IMUX: Managing Tor Connections from Two to Infinity, and Beyond. In *Workshop on Privacy in the Electronic Society (WPES)*, 2014.

[36] A. Ghedini. Encrypt it or Lose it: How Encrypted SNI Works. https://blog.cloudflare.com/encrypted-sni/, September 2018. Blog Post.

[37] D. Gopal and N. Heninger. Torchestra: Reducing Interactive Traffic Delays over Tor. In *Workshop on Privacy in the Electronic Society (WPES)*, 2012.

[38] D. Goulet. Ongoing DDoS on the Network. Tor-Project Email 001604, December 2017. https://lists.torproject.org/pipermail/tor-project/2017-December/001604.html.

[39] D. Goulet. Denial of Service mitigation subsystem. Tor Trac Ticket 24902, 2018. https://trac.torproject.org/projects/tor/ticket/24902.

[40] D. Goulet. Circuit cell queue can fill up memory. Tor Trac Ticket 25226, 2018. https://trac.torproject.org/projects/tor/ticket/25226.

[41] J. Hayes and G. Danezis. k-fingerprinting: a Robust Scalable Website Fingerprinting Technique. In *USENIX Security Symposium*, 2016.

[42] D. Herrmann, R. Wendolsky, and H. Federrath. Website Fingerprinting: Attacking Popular Privacy Enhancing Technologies with the Multinomial Naïve-Bayes Classifier. In *Workshop on Cloud Computing Security*, 2009.

[43] N. Hopper, E. Y. Vasserman, and E. Chan-Tin. How Much Anonymity Does Network Latency Leak? *ACM Transactions on Information and System Security (TISSEC)*, 13(2):13, 2010.

[44] C. Huitema and E. Rescorla. SNI Encryption in TLS Through Tunneling. Internet-Draft draft-ietf-tls-sni-encryption-02, Internet Engineering Task Force, 2018.

[45] R. Jansen. New Tor Denial of Service Attacks and Defenses. https://blog.torproject.org/new-tor-denial-service-attacks-and-defenses, January 2014. Blog Post.

[46] R. Jansen and R. Dingledine. Authenticating sendme cells to mitigate bandwidth attacks. Tor Proposal 289, 2016.

[47] R. Jansen and N. Hopper. Shadow: Running Tor in a Box for Accurate and Efficient Experimentation. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[48] R. Jansen, K. S. Bauer, N. Hopper, and R. Dingledine. Methodically Modeling the Tor Network. In *Workshop on Cyber Security Experimentation and Test (CSET)*, 2012.

[49] R. Jansen, P. Syverson, and N. Hopper. Throttling Tor Bandwidth Parasites. In *USENIX Security Symposium*, 2012.

[50] R. Jansen, J. Geddes, C. Wacek, M. Sherr, and P. Syverson. Never Been KIST: Tor's Congestion Management Blossoms with Kernel-Informed Socket Transport. In *USENIX Security Symposium*, 2014.

[51] R. Jansen, F. Tschorsch, A. Johnson, and B. Scheuermann. The Sniper Attack: Anonymously Deanonymizing and Disabling the Tor Network. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[52] R. Jansen, M. Traudt, J. Geddes, C. Wacek, M. Sherr, and P. Syverson. KIST: Kernel-Informed Socket Transport for Tor. *ACM Transactions on Privacy and Security (TOPS)*, 22(1):3:1–3:37, December 2018.

[53] R. Jansen, M. Traudt, and N. Hopper. Privacy-Preserving Dynamic Learning of Tor Network Traffic. In *Conference on Computer and Communications Security (CCS)*, 2018. See also `https://tmodel-ccs2018.github.io`.

[54] A. Johnson, C. Wacek, R. Jansen, M. Sherr, and P. Syverson. Users Get Routed: Traffic Correlation on Tor By Realistic Adversaries. In *Conference on Computer and Communications Security (CCS)*, 2013.

[55] A. Johnson, R. Jansen, N. Hopper, A. Segal, and P. Syverson. PeerFlow: Secure Load Balancing in Tor. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2017(2), April 2017.

[56] S. Li, H. Guo, and N. Hopper. Measuring Information Leakage in Website Fingerprinting Attacks and Defenses. In *Conference on Computer and Communications Security (CCS)*, 2018.

[57] Z. Ling, J. Luo, W. Yu, X. Fu, D. Xuan, and W. Jia. A New Cell Counter Based Attack Against Tor. In *Conference on Computer and Communications Security (CCS)*, 2009.

[58] Z. Ling, J. Luo, W. Yu, X. Fu, W. Jia, and W. Zhao. Protocol-Level Attacks against Tor. *Computer Networks*, 57(4):869–886, 2013.

[59] A. Mani, T. W. Brown, R. Jansen, A. Johnson, and M. Sherr. Understanding Tor Usage with Privacy-Preserving Measurement. In *Internet Measurement Conference (IMC)*, 2018.

[60] B. Marczak, N. Weaver, J. Dalek, R. Ensafi, D. Fifield, S. McKune, A. Rey, J. Scott-Railton, R. Deibert, and V. Paxson. An analysis of China's "Great Cannon". In *Workshop of Free and Open Communication on the Internet (FOCI)*, 2015. See also `https://citizenlab.ca/2015/04/chinas-great-cannon`.

[61] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *ACM Computer Communication Review*, 27(3):67–82, 1997.

[62] S. Matic, C. Troncoso, and J. Caballero. Dissecting Tor Bridges: a Security Evaluation of Their Private and Public Infrastructures. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[63] P. Mittal, A. Khurshid, J. Juen, M. Caesar, and N. Borisov. Stealthy Traffic Analysis of Low-Latency Anonymous Communication using Throughput Fingerprinting. In *Conference on Computer and Communications Security (CCS)*, 2011.

[64] S. J. Murdoch and G. Danezis. Low-Cost Traffic Analysis of Tor. In *Symposium on Security and Privacy (S&P)*, 2005.

[65] S. J. Murdoch and P. Zieliński. Sampled Traffic Analysis by Internet-Exchange-Level Adversaries. In *Workshop on Privacy Enhancing Technologies (PET)*, June 2007.

[66] M. Nasr, A. Bahramali, and A. Houmansadr. DeepCorr: Strong Flow Correlation Attacks on Tor Using Deep Learning. In *Conference on Computer and Communications Security (CCS)*, 2018.

[67] P. H. O'Neill. Tor's Ex-director: 'The Criminal Use of Tor has Become Overwhelming'. `https://www.cyberscoop.com/tor-dark-web-andrew-lewman-securedrop`, May 2017. Cyberscoop Online News Article.

[68] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP Throughput: A Simple Model and its Empirical Validation. *ACM Computer Communication Review*, 28(4):303–314, 1998.

[69] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel. Website Fingerprinting in Onion Routing Based Anonymization Networks. In *Workshop on Privacy in the Electronic Society (WPES)*, 2011.

[70] A. Panchenko, F. Lanze, A. Zinnen, M. Henze, J. Pennekamp, K. Wehrle, and T. Engel. Website Fingerprinting at Internet Scale. In *Network and Distributed System Security Symposium (NDSS)*, 2016.

[71] V. Pappas, E. Athanasopoulos, S. Ioannidis, and E. P. Markatos. Compromising Anonymity using Packet Spinning. In *International Conference on Information Security*, 2008.

[72] M. Perry. TorFlow: Tor Network Analysis. In *Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs)*, 2009.

[73] K. Poulsen. Feds Are Suspects in New Malware That Attacks Tor Anonymity. `https://www.wired.com/2013/08/freedom-hosting`, August 2013. Wired Online News Article.

[74] J. J. Santanna, R. van Rijswijk-Deij, R. Hofstede, A. Sperotto, M. Wierbosch, L. Z. Granville, and A. Pras. Booters—An analysis of DDoS-as-a-service attacks. In *Integrated Network Management (IM)*, 2015.

[75] J. J. Santanna, R. d. O. Schmidt, D. Tuncer, A. Sperotto, L. Z. Granville, and A. Pras. Quiet Dogs Can Bite: Which Booters Should We Go After, and What Are Our Mitigation Options? *IEEE Communications Magazine*, 55(7):50–56, 2017.

[76] B. Schneier. Attacking Tor: how the NSA targets users' online anonymity. `https://www.theguardian.com/world/2013/oct/04/tor-attacks-nsa-users-online-anonymity`, October 2013. The Guardian Online News Article.

[77] P. Sirinam, M. Imani, M. Juarez, and M. Wright. Deep Fingerprinting: Undermining Website Fingerprinting Defenses with Deep Learning. In *Conference on Computer and Communications Security (CCS)*, 2018.

[78] R. Snader and N. Borisov. EigenSpeed: Secure Peer-to-Peer Bandwidth Evaluation. In *International Workshop on Peer-to-Peer Systems (IPTPS)*, 2009.

[79] Y. Sun, A. Edmundson, L. Vanbever, O. Li, J. Rexford, M. Chiang, and P. Mittal. RAPTOR: Routing Attacks on Privacy in Tor. In *USENIX Security Symposium*, 2015.

[80] P. Syverson, G. Tsudik, M. Reed, and C. Landwehr. Towards an Analysis of Onion Routing Security. In *Designing Privacy Enhancing Technologies*, 2001.

[81] H. Tan, M. Sherr, and W. Zhou. Data-plane Defenses against Routing Attacks on Tor. *Proceedings on Privacy Enhancing Technologies (PoPETS)*, 2016(4), 2016.

[82] C. Wacek, H. Tan, K. Bauer, and M. Sherr. An Empirical Evaluation of Relay Selection in Tor. In *Network and Distributed System Security Symposium (NDSS)*, 2013.

[83] R. Wails, Y. Sun, A. Johnson, M. Chiang, and P. Mittal. Tempest: Temporal Dynamics in Anonymity Systems. *Proceedings on Privacy Enhancing Technologies (PoPETS)*, 2018(3), 2018.

[84] T. Wang and I. Goldberg. Improved Website Fingerprinting on Tor. In *Workshop on Privacy in the Electronic Society (WPES)*, 2013.

[85] T. Wang, X. Cai, R. Nithyanand, R. Johnson, and I. Goldberg. Effective Attacks and Provable Defenses for Website Fingerprinting. In *USENIX Security Symposium*, 2014.

No Right to Remain Silent: Isolating Malicious Mixes

Hemi Leibowitz
Bar-Ilan University, Israel

Ania M. Piotrowska
University College London, UK

George Danezis
University College London, UK

Amir Herzberg
University of Connecticut, US

Abstract

Mix networks are a key technology to achieve network anonymity and private messaging, voting and database lookups. However, simple mix network designs are vulnerable to malicious mixes, which may drop or delay packets to facilitate traffic analysis attacks. Mix networks with provable robustness address this drawback through complex and expensive proofs of correct shuffling but come at a great cost and make limiting or unrealistic systems assumptions. We present *Miranda*, an efficient mix-net design, which mitigates active attacks by malicious mixes. *Miranda* uses both the detection of corrupt mixes, as well as detection of faults related to a pair of mixes, without detection of the faulty one among the two. Each active attack – including dropping packets – leads to reduced connectivity for corrupt mixes and reduces their ability to attack, and, eventually, to detection of corrupt mixes. We show, through experiments, the effectiveness of *Miranda*, by demonstrating how malicious mixes are detected and that attacks are neutralized early.

1 Introduction

The increasing number of bombshell stories [27, 19, 10] regrading mass electronic surveillance and illicit harvesting of personal data against both ordinary citizens and high-ranking officials, resulted in a surge of anonymous and private communication tools. The increasing awareness of the fact that our daily online activities lack privacy, persuades many Internet users to turn to encryption and anonymity systems which protect the confidentiality and privacy of their communication. For example, services like WhatsApp and Signal, which offer protection of messages through end-to-end encryption, gained popularity over the past years. However, such encryption hides only the content but not the meta-data of the message, which carries a great deal of privacy-sensitive information. Such information can be exploited to infer who is communicating with whom, how often and at what times. In contrast, the circuit-based onion routing Tor network is an example of anonymity systems that protect the meta-data. Tor is currently the most popular system offering anonymity, attracting almost 2 million users daily. However, as research has shown [47, 40, 42, 41], Tor offers limited security guarantees against traffic analysis.

The need for strong anonymity systems resulted in renewed interest in *onion mixnets* [12]. In an onion mixnet, sender encrypts a message multiple times, using the public keys of the destination and of multiple mixes. Onion mixnets are an established method for providing provable protection against meta-data leakage in the presence of a powerful eavesdropper, with low computational overhead. Early mixnets suffered from poor scalability, prohibitive latency and/or low reliability, making them unsuitable for many practical applications. However, recently, researchers have made significant progress in designing mixnets for high and low latency communication with improved scalability and performance overhead [44, 48, 13]. This progress is also visible in the industrial sector, with the founding of companies whose goal is to commercialise such systems [1, 2].

Onion mixnets offer strong anonymity against passive adversaries: a single honest mix in a cascade is enough to ensure anonymity. However, known mixnet designs are not robust against *active* long-term traffic analysis attacks, involving *dropping* or *delaying* packets by malicious mixes. Such attacks have severe repercussions for privacy and efficiency of mix networks. For example, a disclosure attack in which a rogue mix strategically drops packets from a specific sender allows the attacker to infer with whom the sender is communicating, by observing which recipient received fewer packets than expected [4]. Similarly, Denial-of-Service (DoS) attacks can be used to enhance de-anonymization [9], and $(n-1)$ attacks allow to track packets over honest mixes [45].

It is challenging to identify and penalize malicious mixes while retaining strong anonymity and high efficiency. Trivial strategies for detecting malicious mixes are fragile and may become vectors for attacks. Rogue mixes can either hide their involvement or worse, make it seem like honest mixes are unreliable, which leads to their exclusion from the net-

work. Several approaches to the problem of active attacks and reliability were studied, however, they have significant shortcomings, which we discuss in Section 8.

In this work, we revisit the problem of making decryption mix networks robust to malicious mixes performing active attacks. We propose *Miranda*[1], an efficient reputation-based design, that detects and isolates active malicious mixes. We present security arguments that demonstrate the effectiveness of Miranda against active attacks. The architectural building blocks behind Miranda have been studied by previous research, but we combine them with a novel approach which takes advantage of detecting failure of inter-mix links, used to isolate and disconnect corrupt mixes, in addition to direct detection of corrupt mixes. This allows Miranda to mitigate corrupt mixes, without requiring expensive computations.

Miranda disconnects corrupt mixes by carefully gathering evidence of their misbehavior, resulting in the removal of links which are misused by the adversary. The design includes a set of secure and decentralized *mix directory authorities* that select and distribute mix cascades once every *epoch*, based on the gathered evidence of the faulty links between mixes. Repeated misbehaviors result in the complete exclusion of the misbehaving mixes from the system (see Figure 1).

We believe that Miranda is an important step toward a deployable, practical strong-anonymity system. However, Miranda design makes several significant simplifying assumptions. These include (1) a fixed set of mixes (no churn), (2) a majority of benign mixes (no Sybil), (3) reliable communication and efficient processing (even during DoS), and (4) synchronized clocks. Future work should investigate, and hopefully overcome, these challenges; see Section 9.

Contributions. Our paper makes the following contributions:

- We present Miranda, an efficient, low-cost and scalable novel design that detects and mitigates active attacks. To protect against such attacks, we leverage reputation and local reports of faults. The Miranda design can be integrated with other mix networks and anonymous communication designs.
- We propose an encoding for secure *loop messages*, that may be used to securely test the network for dropping attacks – extending traditional mix packet formats for verifiability.
- We show how Miranda can take advantage of techniques like community detection in a novel way, which further improves its effectiveness.
- We analyze the security properties of Miranda against a wide range of attacks.

[1] "Miranda warning" is the warning used by the US police, in order to notify people about their rights before questioning them. Since Miranda prevents adversaries from silently (but actively) attacking the mix network, we refer to it as *no right to remain silent*.

Overview. The rest of this paper is organized as follows. In Section 2, we discuss the motivation behind our work, and define the threat model and security goals. In Section 3, we present important concepts of Miranda. In Sections 4 and 5, we detail the core protocols of Miranda, which detect and penalize active attacks. In Section 6, we present improved, *community-based* detection of malicious mixes. In Section 7, we evaluate the security properties of Miranda against active attacks. In Section 8, we contrast our design to related work. Finally, we discuss future work in Section 9 and conclude in Section 10.

2 The Big Picture

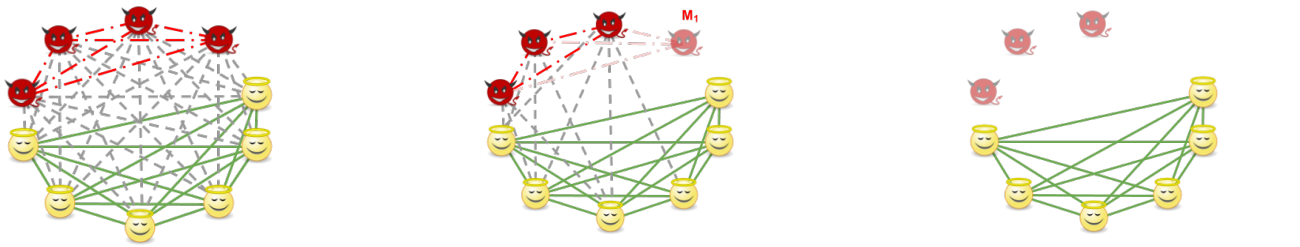
In this section, we outline the general model of the Miranda design, define the threat model, and motivate our work by quantifying how active attacks threaten anonymity in mix networks. Then, we summarize the security goals of Miranda.

2.1 General System Model

We consider an anonymous communication system consisting of a set of users communicating over the *decryption mix network* [12] operating in synchronous batches, denoted as *rounds*. Depending on the path constraints, the topology may be arranged in separate cascades or a Stratified network [21]. We denote by $\mathcal{M}$ the set of all mixes building the anonymous network. For simplicity, in this work we assume that the set of mixes $\mathcal{M}$ is fixed (no churn). See discussion in Section 9 of this and other practical challenges.

Messages are end-to-end *layer encrypted* into a cryptographic packet format by the sender, and the recipient performs the last stage of decryption. Mixes receive packets within a particular round, denoted by r. Each mix decodes a successive layer of encoding and shuffles all packets randomly. At the end of the round, each mix forwards all packets to their next hops. Changing the binary pattern of packets by removing a single layer of encryption prevents bit-wise correlation between incoming and outgoing packets. Moreover, mixing protects against an external observer, by obfuscating the link between incoming and outgoing packets.

Message packet format. In this paper, we use the Sphinx cryptographic packet format [15]. However, other packet formats can be used, as long as they fulfill certain properties. The messages encoded should be of *constant length* and *indistinguishable* from each other at any stage in the network. Moreover, the encryption should guarantee *duplicates detection*, and eliminate tampered messages (*tagging attacks*). The packet format should also allow senders to encode arbitrary routing information for mixes or recipients. We denote the result of encoding a message as $\mathsf{Pack}(\mathsf{path}, \mathsf{routingInfo}, \mathsf{rnd}, \mathsf{recipient}, \mathsf{message})$, where rnd



(a) Connectivity graph in the beginning. All mixes are willing to communicate with each other.

(b) Miranda detects active attacks and removes the links between the honest and dishonest nodes (Section 4.3).

(c) Miranda applies community detection (Section 6) to further detect dishonest nodes and disconnect them from the honest nodes.

Figure 1: High-level overview of the process of isolating malicious mixes in Miranda.

denotes a random string of bits used by the packet format.

2.2 Threat Model

We consider an adversary whose goal is to de-anonymize packets traveling through the mix network. Our adversary acts as a *global observer*, who can eavesdrop on all traffic exchanged among the entities in the network, and also, knows the rate of messages that Alice sends/receives [2]. Moreover, all malicious entities in the system collude with the adversary, giving access to their internal states and keys. The adversary may control many participating entities, but we assume a majority of honest mixes and *directory servers* (used for management, see Section 3). We allow arbitrary number of malicious clients but assume that there are (also) many honest clients - enough to ensure that any first-mix in a cascade, will receive a 'sufficient' number of messages in most rounds - say, 2ω, where ω is sufficient to ensure reasonable anonymity, for one or few rounds.

In addition, Miranda assumes reliable communication between any pair of honest participants, and ignores the time required for computations - hence, also any potential for Miranda-related DoS. In particular, we assume that the adversary cannot arbitrarily drop packets between honest parties nor delay them for longer than a maximal period. This restricted network adversary is weaker than the standard Dolev-Yao model, and in line with more contemporary works such as XFT [35] that assumes honest nodes can eventually communicate synchronously. It allows more efficient Byzantine fault tolerance schemes, such as the one we present. In practice, communication failures *will* occur; see discussion in Section 9 of this and other practical challenges.

We denote by n the total number of mixes in the network ($|\mathcal{M}| = n$), n_m of which are malicious and n_h are honest ($n = n_m + n_h$). We refer to cascades where all mixes are malicious as *fully malicious*. Similarly, as *fully honest* we refer to cascades where all nodes are honest, and *semi-honest* to the ones where only some of the mixes are honest. A link between an honest mix and a malicious mix is referred to as a *semi-honest* link.

[2] We emphasize that this is a non-trivial adversarial advantage. In reality, the adversary might not know Alice's rate, and therefore might be more limited regarding de-anonymization attacks.

2.3 What is the Impact of Active Attacks on Anonymity?

Active attacks, like dropping messages, can result in a catastrophic advantage gained by the adversary in linking the communicating parties. To quantify the advantage, we defined a security game, followed by a qualitative and composable measure of security against dropping attacks. To our knowledge, this is the first analysis of such attacks and we provide full details in [34]. Our results support the findings of previous works on statistical disclosure attacks [4] and DoS-based attacks [9], arguing that the traffic analysis advantage gained from dropping messages is significant. We found that the information leakage for realistic volumes of traffic (10–100 messages per round) is quite significant: *the adversary can improve de-anonymization by about 20%*. For larger traffic rates (more than 1000 messages per round) the leakage drops but expecting each client to receive over 1000 messages per round on average seems unrealistic, unless large volumes of synthetic cover traffic is used. The lesson drawn from our analysis and previous studies is clear: it is crucial to design a mechanism to detect malicious nodes and remove them from the system after no more than a few active attacks. The Miranda design achieves this goal.

2.4 Security Goals of Miranda

The main goal of a mix network is to hide the correspondence between senders and recipients of the messages in the network. More precisely, although the communication is over cascades that might contain malicious mixes, the Miranda design aims to provide protection which is *indistinguishable from the protection provided by an 'ideal mix'*, i.e., a single mix node which is known to be honest.

The key goals of Miranda relate to alleviating and discouraging active attacks on mix networks, as they have a significant impact on the anonymity through traffic analysis. This is achieved through the detection and exclusion of misbehaving mixes. The Miranda design offers the following protections against active attacks:

Detection of malicious nodes. Every active attack by a corrupt mix is detected with non-negligible probability, by at least one entity.

Separation of malicious nodes. Every active attack by a rogue mix results, with a non-negligible probability, in the removal of at least one link connected to the rogue mix - or even removal of the rogue mix itself.

Reducing attacks impact over multiple epochs. Repeated application of Miranda lowers the overall prevalence and impact of active attacks by corrupt mixes across epochs, limiting the ability of the adversary to drop or delay packets.

3 Rounds, Epochs and Directories

In Miranda, as in other synchronous mixnet designs, time is broken into *rounds*, and in each round, a mix 'handles' all messages received in the previous round. However, a more unique element of Miranda is that rounds are collected into *epochs*. Epochs are used to manage Miranda; the beginning of each epoch includes announcement of the set of cascades to be used in this epoch, after a selection process that involves avoidance of mixes detected as corrupt - and of links between two mixes, where one or both of the mixes reported a problem.

The process of selecting the set of cascades for each epoch, is called the *inter-epoch process*, and is performed by a set of d servers refered to as *directory authorities*, following [14], which maintain a list of available mixes and links between them. We assume that a number d_m of authorities can be malicious and collude with the adversary or deviate from the protocol, in order to break the security properties. By d_h we denote the number of honest authorities $(d = d_m + d_h)$, which follow the protocol truthfully.

During each epoch, there are multiple rounds where users communicate over the mix network. Both users and mixes report any misbehavior they encounter to the directory authorities. The *directory authorities* process these reports, and, before the beginning of a new epoch, they select a set of cascades available in that epoch. The newly generated cascades will reflect all reported misbehaviors. Namely, cascades exclude links that were reported, or mixes involved in too many reports, or detected via Miranda's *community-based attacker detection mechanisms*, described in Section 6.

We denote the number of reports which marks a mix as dishonest and causes its exclusion from the network as *thresh* and emphasize that *thresh* is cumulative over rounds and even epochs. In this paper, we simply use $thresh = n_m + 1$, which suffices to ensure that malicious mixes cannot cause Miranda to exclude honest mixes. However, we find it useful to maintain *thresh* as a separate value, to allow the use of larger value for *thresh* to account for a number of failures of honest mixes or links between honest mixes, when the Miranda design is adopted by a practical system.

Significant, although not prohibitive, processing and communication is involved in the inter-epoch process; this motivates the use of longer epochs. On the other hand, during an entire epoch, we use a fixed set of cascades, which may reduce due to failures; and clients may not be fully aware of links and mixes detected as faulty. This motivates the use of shorter epochs. These considerations would be balanced by the designers of an anonymous communication system, as they incorporate the Miranda design.

4 Intra-Epoch Process

In this section, we present the mechanisms that operate during an epoch to deter active attacks, including dropping attacks. We start by describing how active attacks are detected and how this deters malicious behavior. Next, we discuss nodes who refuse to cooperate.

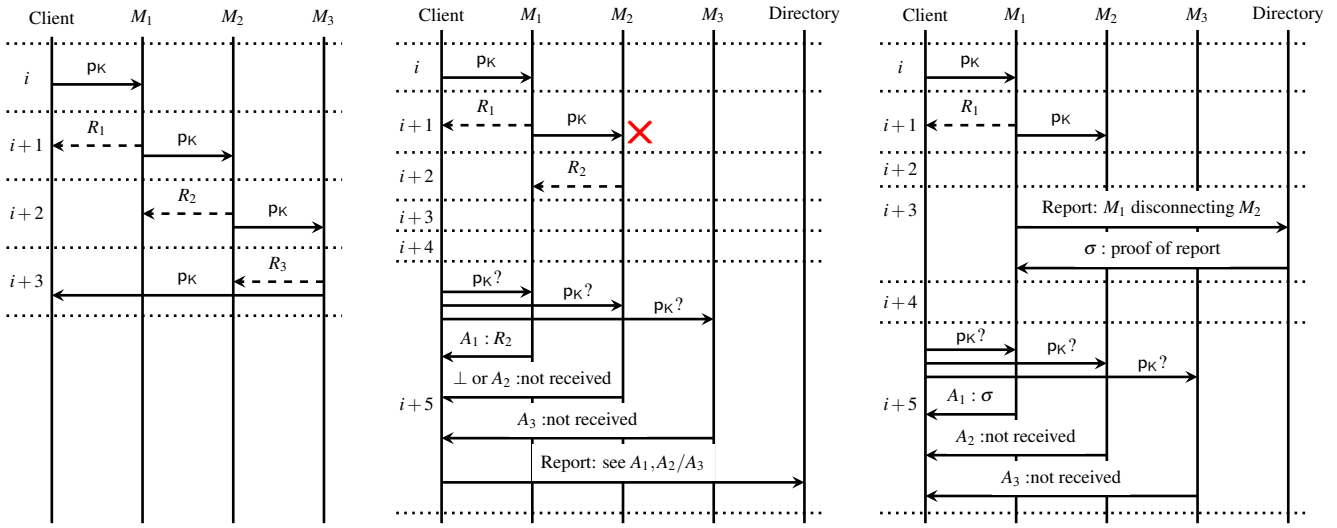
Note that in this section, as in the entire Miranda design, we assume reliable communication between any pair of honest participants. As we explain in Subsection 2.2, a practical system deploying Miranda should use a lower-layer protocol to deal with (even severe) packet losses, and we developed such efficient protocol - see [5].

4.1 Message Sending

At the beginning of each epoch, clients acquire the list of all currently available cascades from the directory authorities. When Alice wants to send a message, her client filters out all cascades containing mixes through which she does not wish to relay messages. We denote the set of cascades selected by Alice as C_A. Next, Alice picks a random cascade from C_A, which she uses throughout the whole epoch, and encapsulates the message into the packet format. For each mix in the cascade, we include in the routing information the exact round number during which the mix should receive the packet and during which it should forward it. Next, the client sends the encoded packet to the first mix on the cascade. In return, the mix sends back a *receipt*, acknowledging the received packet.

4.2 Processing of Received Packets

After receiving a packet, the mix decodes a successive layer of encoding and verifies the validity of the expected round r and well-formedness of the packet. At the end of the round, the mix forwards all valid packets to their next hops. Miranda requires mixes to acknowledge received packets by sending back receipts. A receipt is a digitally signed [31]



(a) Successful loop packet p_k sent during round i and received during round $i+3$. Each mix M_i sends back receipt R_i.

(b) Example of naive dropping of loop message p_k by M_2, which drops p_k yet sends back a receipt. Since p_k did not come back the client queries all mixes during round $i+5$ for the proof of forwarding. M_2 either claims that it did not receive p_k (A_2), thus providing the client a proof that conflicts with the receipt R_2, or M_2 does not cooperate ($\perp$). In both cases the directory authority verifies received A_i's and excludes malicious M_2.

(c) Loop packet fails to complete the loop due to non-responding mix. M_1 did not receive receipt from M_2 on round $i+2$ and issues a disconnection in round $i+3$. The client performs the query phase on round $i+5$ and receives the proof of disconnection. The result: M_2 failed to send a receipt to M_1, and thus lost the link to it.

Figure 2: A diagram illustrating loop packets and isolation process. We denote receipt from mix M_i as R_i, and the response as A_i. Note that both in (b) and (c) the entire query and report phases occur during round $i+5$, but it could also be spanned across several rounds, as long as it has a bounded time-frame. For example, if desired, answering the query for p_k could be done in round $i+6$ instead of limiting it to the same round.

statement confirming that a packet p was received by mix M_i. Receipts must be sent and received by the preceding mix within the same round in which packet p was sent.

Generating receipts. For simplicity, we denote a receipt for a single packet p as $\text{receipt} \leftarrow \text{Sign}(\text{p} \,||\, \text{receivedFlag} = 1)$, where $\text{Sign}(\cdot)$ is a secure digital signature algorithm, and $\text{Verify}(\cdot)$ is its matching verification function[3]. However, generating receipts for each packet individually incurs a high computational overhead due to costly public key signature and verification operations.

To reduce this overhead, mixes gather all the packets they received during round r in Merkle trees [37] and sign the root of the tree once. Clients' packets are grouped in a single Merkle tree T_C and packets from mix M_i are grouped in a Merkle tree T_{M_i}. Mixes then generate two types of receipts: (1) receipts for clients and (2) aggregated receipts for mixes. Each client receives a receipt for each message she sends. Client receipts are of the form: $\text{receipt} = (\sigma_\text{C}, \Gamma_\text{p}, \text{r})$, where: σ_C is the signed root of T_C, Γ_p is the appropriate information needed to verify that packet p appears in T_C, and r is the round number. Similarly, each mix, except the last one, receives a receipt in response to all the packets it forwarded in the last round. However, unlike client receipts, mixes expect back a *single* aggregated receipt for all the packets they sent to a specific mix. An aggregated receipt is in the form of: $\text{receipt} = (\sigma_i, \text{r})$, where: σ_i denotes the signed root of T_{M_i} and r is the round number. Since mixes know which packets they forwarded to a particular mix, they can recreate the Merkle tree and verify the correctness of the signed tree root using a single receipt. Once a mix sent an aggregated receipt, it expects back a signed confirmation on that aggregated receipt, attesting that it was delivered correctly. Mixes record the receipts and confirmations to prove later that they behaved honestly in the mixing operation.

Lack of a receipt. If a mix does not receive an aggregated receipt or does not receive a signed confirmation on an aggregated receipt it sent within the expected time slot[4], the mix disconnects from the misbehaving mix. The honest mix detaches from the faulty mix by informing the directory authorities about the disconnection through a *signed link disconnection receipt*. Note, that the directories cannot identify which of the disconnecting mixes is the faulty one merely based on this message, because the mix who sent the complaint might be the faulty one trying to discredit the hon-

[3] Although Sign and Verify use the relevant cryptographic keys, we abuse notations and for simplicity write them without the keys.

[4] Recall that we operate in a synchronous setting, where we can bound the delay of an acknowledgement.

est one. Therefore, the directory authorities only disconnect the ***link*** between the two mixes. The idea of disconnecting links was earlier investigated in various Byzantine agreement works [23], however, to our knowledge this approach was not yet applied to the problem of mix network reliability.

Anonymity loves company. Note, however, that this design may fail even against an attacker who does not control any mix, if a cascade receives less than the *minimal anonymity set size* ω. We could ignore this as a very unlikely event, however, Miranda ensures anonymity also in this case - when the first mix is honest. Namely, if the first mix receives less than ω messages in a round, it would not forward any of them and respond with a special 'under-ω receipt' explaining this failure. To prevent potential abuse of this mechanism by a corrupt first mix, which receives over ω messages yet responds with under-ω receipt, these receipts are shared with the directories, allowing them to detect such attacks.

4.3 Loop Messages: Detect Stealthy Attacks

In a *stealthy active attack*, a mix drops a message - yet sends a receipt as if it forwarded the message. To deter such attacks, clients periodically, yet randomly, *send loop messages to themselves*. In order to construct a *loop message*, the sender S, chooses a unique random bit-string K_S. Loop messages are encoded in the same manner as regular messages and sent through the same cascade C selected for the epoch, making them *indistinguishable* from other messages at any stage of their routing. The *loop message* is encapsulated into the packet format as follows:

$$\mathsf{p_K} \leftarrow \mathsf{Pack}(\mathsf{path}=\mathsf{C}, \mathsf{routingInfo}=\mathsf{routing}, \mathsf{rnd}=\mathsf{H}(\mathsf{K_S}) \\ \mathsf{recipient}=\mathsf{S}, \mathsf{message}=\text{“loop”})$$

The tuple $(\mathsf{S}, \mathsf{K_S}, \mathsf{C}, \mathsf{routing})$ acts as the *opening value*, which allows recomputing $\mathsf{p_K}$ as well as all its intermediate states $\mathsf{p_K^i}$ that mix M_i should receive and emit. Therefore, revealing the *opening value* convinces everyone that a particular packet was indeed a *loop message* and that its integrity was preserved throughout its processing by all mixes. Moreover, the construction of the *opening value* ensures that only the creator of the loop packet can provide a valid *opening value*, and no third party can forge one. Similarly, nobody can reproduce an opening value that is valid for a non-loop packet created by an honest sender.

If a loop message fails to complete the loop back, this means that one of the cascade's mixes misbehaved. The sender S queries all the mixes in the cascade for evidence whether they have received, processed and forwarded the loop packet. This allows S to isolate the cascade's problematic link or misbehaving mix which caused the packet to be dropped. S then reports the isolated link or mix to the directory authorities and receives a signed confirmation on her report. This confirmation states that the link will no longer be used to construct future cascades. We detail the querying and isolation process in Section 4.3.2.

4.3.1 When to send loop messages?

The sending of loop messages is determined according to α, which is the required *expected probability of detection* - a parameter to be decided by the system designers. Namely, for every message, there is a fraction α chance of it being a loop message. To achieve that, if Alice sends β messages in round r, then $\lceil \frac{\alpha \cdot \beta}{1-\alpha} \rceil$ additional loop messages are sent alongside the genuine messages.

This may seem to only ensure α in the context of the messages that Alice *sends* but not against an attack on messages *sent to* Alice. However, notice that if a corrupt mix M_i drops messages sent to Alice by an honest sender Bob, then M_i faces the same risk of detection - by Bob.

If Alice can sample and estimate an upper bound γ on the number of messages that she will *receive* in a particular round, then she can apply additional defense. Let x be the number of rounds that it takes for a loop message to come back, and let r denote the current round. Let's assume that Alice knows bound γ on the maximal number of messages from honest senders, that she will receive in round $\mathsf{r}+\mathsf{x}$. Then, to detect a mix dropping messages sent to her with probability α, it suffices for Alice to send $\lceil \frac{\alpha \cdot \gamma}{1-\alpha} \rceil$ loop messages in round r. More precisely, given that Alice sends β messages in round r, in order for the loop messages to protect both messages sent in that round and messages received in round $\mathsf{r}+\mathsf{x}$ she should send $\lceil \frac{\alpha \cdot \max(\beta,\gamma)}{1-\alpha} \rceil$ loop messages in round r.

Within-round timing. If the Miranda senders would send each message immediately after receiving the message from the application, this may allow a corrupt first mix to distinguish between a loop message and a 'regular' message. Namely, this would occur if the attacker knows the exact time at which the application calls the 'send' function of Miranda to send the message. To foil this threat, in Miranda, messages are always sent only during the round following their receipt from the application, and after being shuffled with all the other messages to be sent during this round.

4.3.2 Isolating corrupt mixes with loop messages

Since clients are both the generators and recipients of the attack-detecting *loop messages*, they know exactly during which round r the loop should arrive back. Therefore, if a *loop message* fails to complete the loop back to the sender as expected, the client initiates an *isolation* process, during which it detects and isolates the specific problematic node or link in the cascade. The isolation process starts with the client querying each of the mixes on the cascade to establish whether they received and correctly forwarded the loop packet. During the querying phase, the client first reveals to the respective mixes the packet's *opening value*, in order

to prove that it was indeed a loop packet. Next, the client queries the mixes for the receipts they received after they delivered that packet. When clients detect a problematic link or the misbehaving mix, they report it to the directory authorities, along with the necessary proofs that support its claim. This is in fact a broadcasting task in the context of the well-known *reliable broadcast problem* and can be solved accordingly [36]. Each directory authority that receives the report verifies its validity, and if it is correct, stores the information to be used in future cascade generation processes. Then, the client chooses another cascade from the set of available cascades and sends future packets and loop messages using the new route. For an illustration of loop packets and the isolation process, see Figure 2.

When a client asks an honest mix to prove that it received and correctly forwarded a packet, the mix presents the relevant receipt. However, if a mix did not receive this packet, it attests to that by returning an appropriate signed response to the client. If a loop message did not complete the loop because a malicious mix dropped it and did not send a receipt back, the honest preceding mix would have already disconnected from the misbehaving mix. Thus, the honest mix can present the appropriate *disconnection receipt* it received from the directory authorities as an explanation for why the message was not forwarded (see Figure 2c).

The malicious mix can attempt the following actions, in order to perform an active attack.

Naive dropping. A mix which simply drops a loop packet after sending a receipt to the previous mix can be detected as malicious beyond doubt. When the client that originated the dropped loop packet queries the preceding mix, it presents the receipt received from the malicious mix, proving that the packet was delivered correctly to the malicious node. However, the malicious mix is unable to produce a similar receipt, showing that the packet was received by the subsequent mix, *or* a receipt from the directories proving that it reported disconnection from the subsequent mix. The malicious mix may simply not respond at all to the query. However, the client will still report to the directories, along with the proofs from the previous and following mixes, allowing the directories to resolve the incident (contacting the suspected mix themselves to avoid any possible 'framing') (see Figure 2b).

Blaming the neighbors. Malicious mixes performing active dropping attacks would prefer to avoid complete exclusion. One option is to drop the packet, and not send a receipt to the previous mix. However, this causes the preceding mix to disconnect from the malicious one at the end.

Alternatively, the corrupt mix may drop the packet after it generates an appropriate receipt. To avoid the risk of its detection as a corrupt mix, which would happen if it was a loop message, the corrupt mix may disconnect from the subsequent mix - again losing a link. Therefore, a corrupt mix that drops a packet either loses a link, or risks being exposed (by loop message) and removed from the network.

Delaying packets. A malicious mix can also delay a packet instead of dropping it, so that the honest subsequent mix will drop that packet. However, the honest subsequent mix still sends a receipt back for that packet, which the malicious mix should acknowledge. If the malicious mix acknowledges the receipt, the malicious mix is exposed when the client performs the *isolation process*. The client can obtain a signed receipt proving that the malicious mix received the packet on time, and also the acknowledged receipt from the honest mix that dropped the delayed packet. The latter contains the round number when the packet was dropped, which proves the malicious mix delayed the packet and therefore should be excluded. Otherwise, if the malicious mix refuses to sign the receipt, the honest mix disconnects from the malicious one. Therefore, the delaying attack also causes the mix to either lose a link or to be expelled from the system.

The combination of packet receipts, link disconnection notices, the isolation process and loop messages, forces malicious mixes to immediately lose links when they perform active attacks. Failure to respond to the preceding mix or to record a disconnection notice about the subsequent mix in a timely manner creates potentially incriminating evidence, that would lead to a complete exclusion of the mix from the system. This prevents malicious mixes from silently attacking the system and blaming honest mixes when they are queried in the isolation mechanism. The mere threat of loop messages forces malicious mixes to drop a link with an honest mix for each message they wish to suppress, or risk exposure.

4.4 Handling missing receipts

Malicious mixes might attempt to circumvent the protocol by refusing to cooperate in the isolation procedure. Potentially, this could prevent clients from obtaining the necessary proofs about problematic links, thus preventing them from convincing directory authorities about problematic links. If malicious mixes refuse to cooperate, clients contact a directory authority and ask it to perform the isolation process on their behalf. Clients can prove to the directory authorities that the loop packet was indeed sent to the cascade using the receipt from the first mix. If all mixes cooperate with the directory authority, it is able to isolate and disconnect the problematic link. Otherwise, if malicious mixes do not cooperate with the directory authority, it excludes those mixes from the system.

We note that a malicious client may trick the directory authorities into performing the isolation process on its behalf repeatedly, against honest mixes. In that case, directory authorities conclude that the mix is honest, since the mix can provide either a receipt for the message forwarded or a disconnection notice. However, this is wasteful for both direc-

tory authorities and mixes. Since clients do not have to be anonymous vis-a-vis directory authorities, they may record false reports and eventually exclude abusive clients. Furthermore, the clients have to send proofs from the following mix of not having received the packet, which cannot be done if there was no mix failure.

Malicious entry mix. If a first mix does not send a receipt, the client could have simply chosen another cascade; however, this allows malicious mixes to divert traffic from cascades which are not fully malicious, without being penalized, increasing the probability that clients would select other fully malicious cascades instead. To avoid that, in Miranda, clients *force* the first mix to provide a receipt, by relaying the packet via a trusted ***witness***. A witness is just another mix that relays the packet to the misbehaving first mix. Now, the misbehaving node can no longer refuse to produce a receipt, because the packet arrives from a mix, which allows the isolation process to take place. Note that since a witness sends messages on behalf of clients, the witness *relays* messages without the ω constraint (as if it was a client).

If the witness itself is malicious, it may also refuse to produce a receipt (otherwise, it loses a link). In that case, the client can simply choose another witness; in fact, if desired, clients can even send via multiple witnesses concurrently to reduce this risk - the entry mix can easily detect the 'duplicate' and handle only one message. This prevents malicious mixes from excluding semi-honest cascades without losing a link. Moreover, although the refused clients cannot prove to others that they were rejected, they can learn about malicious mixes and can avoid all future cascades that contain them, including fully malicious cascades, which makes such attacks imprudent.

5 Inter-Epoch Process

In this section, we discuss the inter-epoch operations, taking place toward the end of an epoch; upon its termination, we move to a new epoch. The inter-epoch process selects a new random set of cascades to be used in the coming epoch, avoiding the links reported by the mixes, as well as any mixes detected as corrupt.

Until the inter-epoch terminates and the mixes move to the new epoch, the mixes continue with the intra-epoch process as before; the only difference is that newly detected failures, would be 'buffered' and handled only in the following run of the inter-epoch process, to avoid changing the inputs to the inter-epoch process after it has begun.

The inter-epoch process consists of the following steps.

5.1 Filtering Faulty Mixes

Directory authorities share amongst themselves the evidences they received and use them to agree on the set of faulty links and mixes. The evidences consist of the reports of faulty links from mixes, clients or authorities performing the *isolation process*. The directory authorities exchange all new evidences of faulty links and mixes, i.e., not yet considered in the previous inter-epoch computation process. Every directory can validate each evidence it received and broadcast it to all other directories. Since we assume majority of honest directories and synchronous operation, we can use known broadcast/consensus protocols, and after a small number of rounds, all honest directory authorities have exactly the same set of faulty links.

Note, that only links connected to (one or two) faulty mixes are ever disconnected. Hence, any mix which has more than $thresh$ links disconnected must be faulty (due to the assumption that $thresh > n_m$), and hence the directories exclude that mix completely and immediately. Since the directory authorities share exactly the same set of faulty links, it follows that they also agree on exactly the same set of faulty mixes. We call this exclusion process a *simple malicious mix filtering* step. In Section 6, we discuss more advanced filtering techniques, based on *community detection*.

Simple malicious mix filtering technique. To perform the simple malicious mix filtering, each directory authority can build a graph that represents the connectivity between mixes. Namely, consider an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where the vertices map to the mixes in the system ($\mathcal{V} = \mathcal{M}$), and an edge $(M_i, M_j) \in \mathcal{E}$ means that the link between mixes M_i and M_j was not dropped by either mix. Let $\overline{\mathcal{G}} = (\overline{\mathcal{V}}, \overline{\mathcal{E}})$ be the complement graph of $\mathcal{G}$ and let $\mathsf{Deg}_{\overline{\mathcal{G}}}(M_i)$ denote the degree of the vertex M_i in graph $\overline{\mathcal{G}}$. In the beginning, before any reports of faults have arrived at the directory authorities, $\mathcal{G}$ is a complete graph and $\overline{\mathcal{G}}$ is an empty graph. As time goes by, $\mathcal{G}$ becomes sparser as a result of the links being dropped, and proportionally, $\overline{\mathcal{G}}$ becomes more dense. The filtering mechanism removes all mixes that lost $thresh$ links or more, i.e., $\{M_i \mid \forall M_i \in \overline{\mathcal{G}} : \ \mathsf{Deg}_{\overline{\mathcal{G}}}(M_i) \geq thresh\}$, where $thresh = n_m + 1$. The filtering mechanism checks the degree $\mathsf{Deg}_{\overline{\mathcal{G}}}(M_i)$ in graph $\overline{\mathcal{G}}$, since the degree in $\overline{\mathcal{G}}$ represents how many links M_i lost. We emphasize that when such malicious mix is detected and removed, the number of malicious mixes in the system is decreased by one ($n_m = n_m - 1$) and proportionally so does $thresh$ ($thresh = thresh - 1$). As a result, whenever the mechanism removes a malicious mix it repeats the mechanism once again, to see whether new malicious mixes can be detected according to the new $thresh$ value. An illustration of this process is depicted in Figure 3.

5.2 Cascades Selection Protocol

After all directory authorities have the same view of the mixes and their links, they select and publish a (single) set of cascades, to be used by all clients during the coming epoch. To allow clients to easily confirm that they use the correct set of cascades, the directory authorities collectively sign the set

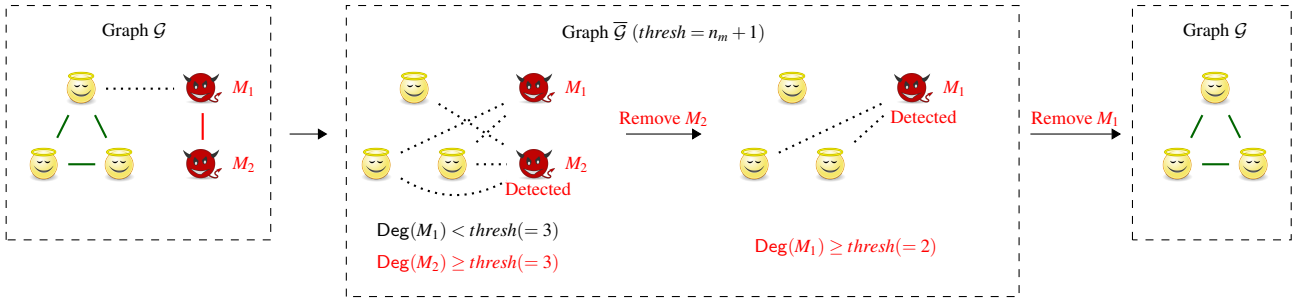


Figure 3: An illustration of the simple malicious mix filtering (without community detection).

that they determined for each epoch, using a threshold signature scheme [46, 25]. Hence, each client can simply retrieve the set from any directory authority and validate that it is the correct set (using a single signature-validation operation).

The *cascades selection protocol* allows all directory authorities to agree on a random set of cascades for the upcoming epoch. The input to this protocol, for each directory authority, includes the set of mixes $\mathcal{M}$, the desired number of cascades to be generated $\mathsf{n_c}$, the length of cascades ℓ and the set of faulty links $\mathcal{F}_\mathcal{L} \subset \mathcal{M} \times \mathcal{M}$. For simplicity, $\mathcal{M}$, $\mathsf{n_c}$ and ℓ are fixed throughout the execution.

The goal of all directory authorities is to select the same set of cascades $\mathcal{C} \subseteq \mathcal{M}^\ell$, where $\mathcal{C}$ is uniformly chosen from all sets of cascades of length ℓ, limited to those which satisfy the selected *legitimate cascade predicates*, which define a set of constraints for building a cascade. In [34], we describe several possible legitimate cascade predicates, and discuss their differences.

Given a specific legitimate cascade predicate, the protocol selects the same set of cascades for all directory authorities, chosen uniformly at random among all cascades satisfying this predicate. This is somewhat challenging, since sampling is normally a random process, which is unlikely to result in exactly the same results in all directory authorities. One way of ensuring correct sampling and the same output, is for the set of directories to compute the sampling process jointly, using a multi-party secure function evaluation process, e.g., [26]. However, this is a computationally-expensive process, and therefore, we present a much more efficient alternative. Specifically, all directories run exactly the same sampling algorithm and for each sampled cascade validate it using exactly the same legitimate cascade predicate. To ensure that the results obtained by all honest directory authorities are identical, it remains to ensure that they use the same random bits as the seed of the algorithm. To achieve this, while preventing the faulty directory authorities from biasing the choice of the seed bits, we can use a coin-tossing protocol, e.g., [7], among the directory authorities[5].

[5]Note, that we only need to generate a small number of bits (security parameter), from which we can generate as many bits as necessary using a pseudo-random generator.

6 Community-based Attacker Detection

So far, the discussion focused on the core behaviour of Miranda and presented what Miranda can do and how it is done. Interestingly, Miranda's mechanisms open a doorway for advanced techniques, which can significantly improve the detection of malicious mixes. In this section, we discuss several techniques that can leverage Miranda's faulty links identification into a powerful tool against malicious adversaries. Among others, we use community detection techniques. Community detection has been used in previous works to achieve Sybil detection based on social or introduction graphs [16, 17]. However, we assume that the problem of Sybil attacks is solved through other means, such as admission control or resource constraints. Encouragingly, many other techniques can be employed; yet, we hope that the following algorithms will be also useful in other applications where applicable, e.g., where community detection is needed.

We begin with the following observation.

Observation 1. *For every two mixes M_i, M_j that have an edge in $(M_i, M_j) \in \overline{\mathcal{E}}$, at least one of them is a malicious mix.*

Observation 1 stems directly from our assumption that honest mixes never fail. Therefore, a dropped link must be between either an honest mix and a malicious mix or between two malicious mixes. Following this observation, one possible strategy is *aggressive pair removal*, i.e., remove both mixes, if one or both of them report failure of the link connecting them. This strategy seems to provide some benefits - the adversary seems to 'lose more', however it comes at an excess cost of possible exclusion of honest nodes. Therefore, we focus on less aggressive techniques that exclude malicious mixes *without* excluding also honest ones.

Threshold Detection Algorithm. Since the *aggressive removal* of both mixes connected by the failed link from $\mathcal{G}$ is not efficient, we adopt the idea of *virtual removal* of the conflicting pair. By *virtually* we mean that virtually removed mixes are not classified as malicious and they are only removed from $\overline{\mathcal{G}}$ for the duration of the algorithm's execution, and not from $\mathcal{G}$ nor $\mathcal{M}$. We present the *Threshold Detec-*



(a) Graph $\overline{\mathcal{G}}$, the simple malicious mix filtering technique cannot detect M_2 because $\mathsf{Deg}_{\overline{\mathcal{G}}}(M_2) = 2 < 3 = thresh$.

(b) An execution of the *ThresholdDetection* on the same $\overline{\mathcal{G}}$ graph virtually removes M_1 and M_5 which decreases *thresh*, resulting in $\mathsf{Deg}_{\overline{\mathcal{G}}}(M_2) = 2 \geq 2 = thresh$ and detection of M_2 as a malicious mix.

Figure 4: An illustration of how virtually removing mixes from $\overline{\mathcal{G}}$ can expose malicious mixes. Algorithm 2 refers to the graph in 4b as $\overline{\mathcal{G}_1}$, since it is the same graph $\overline{\mathcal{G}}$ as in 4a but without M_1 and without M_1's neighbors.

tion technique in Algorithm 1. The algorithm takes as input graph $\overline{\mathcal{G}} = (\overline{\mathcal{V}}, \overline{\mathcal{E}})$, where an edge $(M_i, M_j) \in \overline{\mathcal{E}}$ represents the disconnected link between M_i and M_j. The algorithm starts by invoking the SIMPLEMALICIOUSFILTERING procedure (described in Section 5.1) on the graph $\overline{\mathcal{G}}$ (line 12). Next, the algorithm invokes the VIRTUALPAIRREMOVAL procedure on $\overline{\mathcal{G}}$ to *virtually* remove a pair of mixes from $\overline{\mathcal{G}}$ (line 14). Following observation 1, at least one malicious mix was *virtually* removed, thus the *virtual* threshold $thresh'$ value is decreased by 1 (line 15). We use the $thresh'$ variable to keep track of the virtually removed malicious mixes and the global *thresh* value is decreased only when a malicious mix was actually detected (line 4), and the rest only change the virtual threshold $thresh'$. After that, the algorithm invokes the procedure SIMPLEMALICIOUSFILTERING again on the *updated* $\overline{\mathcal{G}}$ graph, i.e., without the pair of mixes that were virtually removed by the VIRTUALPAIRREMOVAL procedure. The algorithm repeats lines 14-16 as long as there are edges in $\overline{\mathcal{G}}$. For an illustration why the *ThresholdDetection* algorithm is better than the original *simple malicious mix filtering* see Figure 4.

We next improve upon the detection of malicious mixes by the *ThresholdDetection* algorithm, while still never removing honest mixes. Our improvement is based on Observation 2 below; but before presenting it, we need some preliminaries.

We first define a simple notion which can be applied to any undirected graph. Specifically, let $G^0 = (V^0, E^0)$ be an arbitrary undirected graph. A sequence $\{G^j\}_{j=0}^{\mu}$ of subgraphs of G^0 is a *removal sequence* of length $\mu \geq 1$ of G^0, if for every $j : \mu \geq j \geq 1$, $G^j = G^{j-1} - v_j$. Namely, G^j is the same as G^{j-1}, except for removal of some node $v_j \in G^{j-1}$, and of all edges connected to v_j. A removal sequence is *legitimate* if every removed node v_j has at least one edge.

Let us define the graph $\overline{\mathcal{G}_i}$ to be the resulting graph after removing from $\overline{\mathcal{G}}$ the node M_i together with all its *neighbors*, denoted as $\mathsf{N}(M_i)$.

Algorithm 1 *ThresholdDetection*$(\overline{\mathcal{G}} = (\overline{\mathcal{V}}, \overline{\mathcal{E}}))$

```
1:  procedure SIMPLEMALICIOUSFILTERING(G, thresh')
2:      for every M_i ∈ G s.t. Deg_G(M_i) ≥ thresh' do
3:          M_i is malicious (remove from G, G, M).
4:          thresh ← thresh − 1
5:          thresh' ← thresh' − 1
6:
7:  procedure VIRTUALPAIRREMOVAL(G)
8:      Pick an edge (M_i, M_j) ∈ E.
9:      Remove mixes M_i, M_j from G.
10:
11: thresh' ← thresh.
12: Invoke SIMPLEMALICIOUSFILTERING(G).
13: while E ≠ ∅ do
14:     Invoke VIRTUALPAIRREMOVAL(G).
15:     thresh' ← thresh' − 1.
16:     Invoke SIMPLEMALICIOUSFILTERING(G).
```

Observation 2. *If $\overline{\mathcal{G}_i}$ has a legitimate removal sequence of length μ_i, then there are at least μ_i malicious nodes in $\overline{\mathcal{G}_i}$.*

We use Observation 2 to identify malicious mixes, using the following claim.

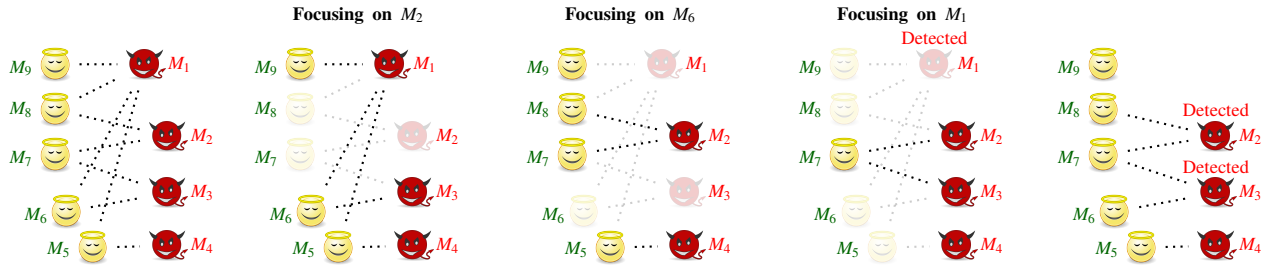
Claim 1. *Every node M_i that satisfies $\mathsf{Deg}_{\overline{\mathcal{G}}}(M_i) > n_m - \mu_i$ is a malicious node.*

Proof. Assume to the contrary, that there exists a mix M_i such that $\mathsf{Deg}_{\overline{\mathcal{G}}}(M_i) > n_m - \mu_i$ but M_i is an honest mix. Since there are n_m malicious mixes in $\mathcal{M}$, and μ_i of them are not neighbors of M_i, then the maximum number of malicious mixes that can be also neighbors of M_i is $n_m - \mu_i$, since M_i is honest. But if $\mathsf{Deg}_{\overline{\mathcal{G}}}(M_i) > n_m - \mu_i$, then at least one of the neighbors of M_i is also honest, which contradicts the assumption that honest links never fail. Therefore, if $\mathsf{Deg}_{\overline{\mathcal{G}}}(M_i) > n_m - \mu_i$ then M_i must be a malicious mix. □

For example, see Figure 4b which depicts the graph $\overline{\mathcal{G}_1}$. By observing $\overline{\mathcal{G}_1}$, we know that at least one of the mixes M_2, M_3 are malicious (since they share an edge), therefore, $\mu_i \geq 1$ since we successfully identified a malicious mix which is not in $\{M_1 \cup \mathsf{N}(M_1)\}$. Alternatively, the same argument can be made regarding M_2 and M_4 *instead* of the pair M_2 and M_3. Since after removing M_2, M_4 from $\overline{\mathcal{G}_1}$ there are no edges left in $\overline{\mathcal{G}_1}$, then $\mu_1 = 1$.

Algorithm 2 presents the *Community Detection* algorithm, which leverages Claim 1 to detect malicious mixes. An illustration of the operation of this algorithm is demonstrated in Figure 5.

Notice that the algorithm only examines nodes with a degree larger than 1 (line 3). The reason is that if $\mathsf{Deg}_{\overline{\mathcal{G}}}(M_i) = 0$ then M_i did not perform an active attack yet, thus it cannot be detected, and if $\mathsf{Deg}_{\overline{\mathcal{G}}}(M_i) = 1$ then M_i cannot be classified based on its neighbors. Therefore, an execution of the



(a) $\forall M_i : \mathrm{Deg}_{\overline{\mathcal{G}}}(M_i) < thresh$, simple malicious mix filtering does not detect malicious mixes.

(b) When we observe $\overline{\mathcal{G}_2}$, i.e., $\overline{\mathcal{G}}$ after the removal of M_2 and $N(M_2)$, two scenarios are possible. In the first scenario, $\mu_2 = 3$ (e.g., if M_1 and M_9 are removed first), thus $\mathrm{Deg}_{\overline{\mathcal{G}}}(M_2) = 2 > 1 = n_m - \mu_2$, and therefore M_2 is detected as malicious. In the second scenario, $\mu_2 = 2$ (e.g., if M_1 and M_6 are removed first), thus $\mathrm{Deg}_{\overline{\mathcal{G}}}(M_2) = 2 \leq 2 = n_m - \mu_2$, and therefore M_2 is *not* detected as malicious (yet). A similar situation occurs with M_3 when observing $\overline{\mathcal{G}_3}$.

(c) When we observe $\overline{\mathcal{G}_6}$, two malicious mixes can be identified, thus $\mu_6 = 2$. As a result, since $\mathrm{Deg}_{\overline{\mathcal{G}}}(M_6) = 2 \leq 2 = n_m - \mu_6$, M_6 is not classified as malicious (nor should it be). Note that even if M_3 was removed in (b), then $\mathrm{Deg}_{\overline{\mathcal{G}}}(M_6) = 1$ and therefore the algorithm cannot classify it based on its neighbors. The same explanations apply to the rest of the honest mixes.

(d) When we observe $\overline{\mathcal{G}_1}$, only one malicious mix can be identified, thus $\mu_1 = 1$. As a result, since $\mathrm{Deg}_{\overline{\mathcal{G}}}(M_1) = 4$ is larger than $n_m - \mu_1 = 3$, M_1 is detected as malicious.

(e) If M_2 and M_3 were not detected as malicious as explained in (b), then after the removal of M_1 in (c) they will be detected, because the removal of M_1 causes $n_m = 4 \rightarrow n_m = 3$. Since the algorithm runs in a loop, when the algorithm will re-check $\overline{\mathcal{G}_2}$, it will discover that $\mu_2 = 2$ and thus $\mathrm{Deg}_{\overline{\mathcal{G}}}(M_2) = 2 > 1 = n_m - \mu_1$, which results in removal of M_2. The same goes for M_3. After the removal of M_1, M_2 and M_3, the algorithm cannot classify M_4 as malicious based on its neighbors, since M_4 only dropped one link. However, the algorithm has the option to *aggressively* remove both M_4, M_5.

Figure 5: A demonstration how Miranda's community detection can significantly improve the detection of malicious mixes using an example graph $\overline{\mathcal{G}}$ and $thresh = n_m + 1$.

Algorithm 2 *CommunityDetection*$(\overline{\mathcal{G}} = (\overline{\mathcal{V}}, \overline{\mathcal{E}}))$

```
1:  n'_m ← n_m
2:  while Ē ≠ ∅ do
3:      for each M_i ∈ V̄ s.t. Deg_Ḡ(M_i) > 1 do
4:          Construct Ḡ_i = (V̄_i, Ē_i) from Ḡ.
5:          μ_i ← 0
6:          while Ē_i ≠ ∅ do
7:              Invoke VirtualPairRemoval(Ḡ_i).
8:              μ_i ← μ_i + 1
9:          if Deg_Ḡ(M_i) > n'_m − μ_i then
10:             M_i is malicious (remove from G, Ḡ, M).
11:             n_m ← n_m − 1, n'_m ← n'_m − 1
12:     if Ē ≠ ∅ then
13:         Invoke VirtualPairRemoval(Ḡ).
14:         n'_m ← n'_m − 1
```

CommunityDetection might not be able to detect all malicious mixes that exposed themselves, e.g., mixes with a degree that equals to 1. If desired, there is always the opportunity to execute the aggressive pair removal technique *after* the *CommunityDetection* algorithm to potentially remove more malicious mixes (with price of possible removal of an honest mix). Also, randomly picking a pair of mixes that share an edge in $\overline{\mathcal{G}}$ might not always be the optimal strategy. In small graphs, the algorithm can exhaust all possible removal variations, but this is a time-consuming option in large graphs. A more sophisticated picking strategy might yield better results; however, when we experimented with some possible strategies, we did not notice a significant improvement over the random picking strategy.

The techniques discussed in this section provide Miranda a significant advantage, since malicious mixes can be detected even if they do not pass *thresh*. Merely the threat of such techniques is significant in deterring active attacks. In Section 7.4 we analyze the security of the mechanisms discussed here and evaluate them empirically. In [34] we present alternative scheme for community detection based on random walks.

7 Analysis of Active Attacks

In this section, we analyze the impact of active attacks in the presence of Miranda. We first analyze Miranda against traditional and non-traditional active attacks, including attacks designed to abuse the protocol to increase the chances of clients choosing fully malicious cascades. We continue by examining the security of loop messages and conclude this section by evaluating how community detection strengthens Miranda.

7.1 Resisting Active Attacks

As discussed in Section 4, a malicious mix that drops a packet sent from a preceding mix or destined to a subsequent mix, loses at least one link; in some cases, the malicious mix gets completely excluded. Hence, the adversary quickly loses its attacking capabilities, before any significant impact is introduced. However, the adversary might try other approaches in order to link the communicating users or gain advantage in the network, as we now discuss.

A malicious first mix can refuse clients' packets; however, such attack is imprudent, since clients can migrate to other cascades. Furthermore, clients can force the malicious mix to relay their packets, using a witness. Similarly, it is ineffective for the last mix of a cascade to drop *all* packets it receives, since clients learn through isolation that the dropped loop packets successfully arrived at the last mix. Although clients cannot prove the mix maliciousness, they avoid future cascades containing the malicious mix, including fully malicious cascades.

Instead of directly dropping packets, adversaries can cause a packet to be dropped by delaying the packet. However, such attack is also detected.

Claim 2. *A malicious mix that delays a packet, is either expelled from the system or loses a link.*

Argument. When an honest mix receives a delayed packet, it drops it. However, the honest mix still sends a receipt back for that packet. If the malicious mix acknowledges the receipt, the malicious mix is exposed when the client performs the isolation process: the client can obtain a signed receipt proving that the malicious mix received the packet on time, and also the acknowledged receipt from the honest mix that dropped the delayed packet. The latter contains the round number when the packet was dropped, which proves the malicious mix delayed the packet and therefore should be excluded. Otherwise, if the malicious mix refuses to sign the receipt, the honest mix disconnects from the malicious mix. □

Injecting malformed packets. Notice how the honest mix that dropped the delayed message still sends back a receipt for it. The reason is that the dropping mix cannot be sure that the previous mix did delay the message. Instead, this can be the result of an adversary that crafts a packet with the same round number in two successive layers.

Claim 3. *An adversary cannot craft a loop message that causes a link loss between two honest mixes.*

Argument. Any loop message has to be well-formed in order for directory authorities to accept it. An adversary can craft a message with invalid round numbers in the packet's routing information, which would cause the honest mix to drop the packet. However, although the honest mix drops the packet, it still sends back a receipt for that packet. Otherwise, the preceding mix, which has no way of knowing that the next layer is intentionally malformed, would disconnect from the subsequent mix. While the adversary can obtain a proof showing that a loop message was dropped, it cannot prove that the loop message was well-formed. □

Aggressive active attacks. In order to de-anonymize the network users, the adversary can choose a more aggressive approach and drop a significant number of packets. For example, in the $(n-1)$ attack [45] applied to the full network, the adversary tracks a target packet from Alice by blocking other packets from arriving to an honest mix, and instead injecting their own packets. Another example is the intersection attack [8], where the adversary tries disconnecting target clients. If the adversary cannot directly disconnect a client with a targeted attack, it can disconnect a client by dropping an entire batch of packets where one of them belongs to the client (the adversary simply does not know which). However, it is important to note, that if an adversary can engineer a scenario where a single target packet is injected and mixed with only messages that the adversary controls, *any* mix-based system is vulnerable. Nevertheless, we argue that Miranda inflicts serious penalty on the adversary who attempts to perform an aggressive dropping of packets.

Claim 4. *Miranda deters aggressive active attacks.*

Argument. Aggressive active attacks require the ability to drop many packets. In Miranda, a malicious mix that drops any packet from another mix without sending back a receipt, loses a link (see Section 4 and Figure 2c). Alternatively, if the malicious mix drops packets but does send receipts for these dropped packets, clients can prove that the malicious mix received their (loop) packets and did not forward them, which results in the exclusion of the malicious mix (see Figure 2b). A malicious entry mix may drop packets from clients, since losing a link to a client is not a serious 'penalty'; but in Miranda, clients then use a *witness mix* (see Section 4.4) – forcing the mix to either relay their packets, or - lose a link to a mix or risk discovery, as discussed above.

Miranda enforces a minimum number of ω packets for mixing by the entry mix. This is designed to protect the *rare* cases where a client sends via an entry mix which is used only by few (or no) other clients, which could allow an eavesdropper attack; we now explain why this cannot be abused to facilitate an active attack (by the first mix).

Recall, that in this case, as in our entire analysis of corrupt-mix attacks, we assume that at least 2ω honest clients send packets to the (possibly corrupt) entry mix; and, as mentioned above, the mix cannot simply 'drop' these (since clients will use witness and then the corrupt mix will lose - at least - a link).

Instead, the corrupt mix could send to these clients, or most of them, the special 'under-ω receipt', claiming (incorrectly) that it didn't receive ω messages during this round.

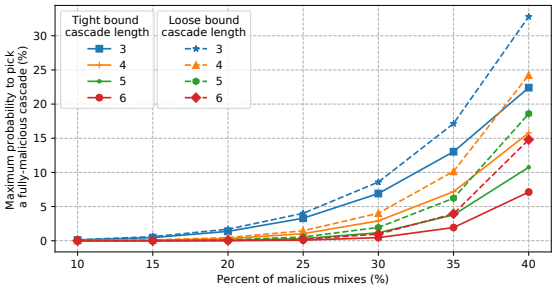


Figure 6: The maximum probability of picking a fully malicious cascade as a function of the cascade length and the power of the adversary.

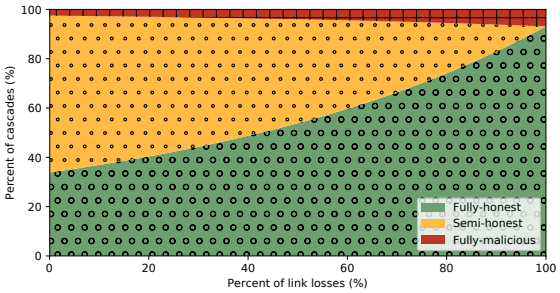


Figure 7: The probability of picking particular classes of cascades after each link loss. The parameters of the simulated mix network are $l = 3$, $n = 100$ and $n_m = 30$.

However, senders *report* these (rare) under-ω receipts to the directories, who would quickly detect that this mix is corrupt. □

7.2 Fully Malicious Cascades Attacks

If the packets are relayed via a *fully malicious cascade*, an adversary can trivially track them. Consequently, adversaries would like to divert as much traffic as possible to the fully malicious cascades. Attackers can try to maximize their chances by: (1) increasing the probability that fully malicious cascades are included in the set $\mathcal{C}$ produced by the directory authorities during the inter-epoch process, and/or (2) increasing the probability that clients pick a fully malicious cascade from $\mathcal{C}$ during an epoch.

Because cascades are chosen uniformly over all valid cascades, the only way the adversary can influence the cascades generation process is by excluding semi-honest cascades. However, they can only exclude cascades by dropping links they are a part of, therefore, the adversary cannot exclude any honest links or honest mixes[6], meaning they cannot exclude any fully honest cascades. However, adversaries are able to disconnect semi-honest cascades by disconnecting semi-honest links and thereby increase the probability of picking a fully malicious cascade. Interestingly, we found that such an attack only slightly increases the chance of selecting a fully malicious cascade – while significantly increasing the chance of selecting a fully honest cascade (see Claim 5). Further, this strategy makes it easier to detect and eliminate sets of connected adversarial domains (see section 6).

Claim 5. *Let C_{Adv} denote a set of fully malicious cascades. The maximum probability to pick a fully malicious cascade during cascades generation process, after the semi-honest cascades were excluded by the adversary is*

$$\Pr(c \in C_{Adv}) \leq \left(\frac{n_m}{n_h - l + 1}\right)^l.$$

Argument. See [34].

Figure 6 and Figure 7 present the probability of picking a fully malicious cascade depending on the number of mixes colluding with the adversary and the percentage of lost links.

Once n_c cascades are generated, the adversary could try to bias the probability of clients choosing a fully malicious cascade. To do so, the adversary can sabotage semi-honest cascades [9] through dropping messages, and in an extreme case, exclude them all. We illustrate in Figure 8 the attack cost, expressed as the number of links the adversary must affect in order to achieve a certain probability of success in shifting clients to a fully malicious cascade. Note, that the larger the number of cascades $\mathsf{n_c}$, the more expensive the attack, and the lower the probability of success.

7.3 Security of Loop Messages

Since loop messages are generated and processed in the same way as genuine messages, the binary pattern does not leak any information. However, adversaries can still seek ways to predict when loop messages are sent; for example, by observing the timing pattern and the rate of sent messages.

Detecting loop messages. Adversaries can try to guess whether a particular message is a loop message or not. A successful guess allows the adversary to drop non-loop messages without being detected, while still sending receipts for them to the previous mix. We formulate the following claim:

Claim 6. *Assume that an adversary that does not control the last mix in the cascade, drops a packet. The probability of this message being a loop message sent by a non-malicious client is at least α.*

Argument. It suffices to consider packets sent by non-malicious clients. When a non-last mix receives such pack-

[6] Even if all adversarial mixes disconnect from an honest mix, it is still not enough for exclusion, since $thresh > n_m$.

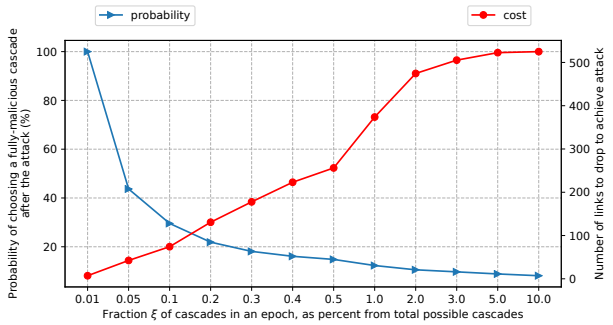


Figure 8: The costs (red, right axis) and success probability (blue, left axis) of performing DoS [9] attacks based on the fraction of cascades active in every epoch. Cost is measured in links the adversary must sacrifice; as Figure 9 shows, even the minimal 'cost' essentially implies detection of *all* active corrupt mixes. Furthermore, using just 1% of the possible cascades suffices to reduce success probability to about 10% or less.

ets, it does not know the destination. Furthermore, as described in section 4.3, loop packets are sent by non-malicious clients according to the rate defined by α of genuine traffic and are bitwise indistinguishable from genuine packets. Hence, even if the mix would know the identity of the sender, e.g., by being the first mix, the packet can still be a loop message with probability at least α. □

Note that a malicious non-last mix that drops a loop message, yet sends a receipt for it and remains connected to the next mix, would be proven malicious and excluded from the network. On the other hand, if such mix does not send a receipt, then it loses a link.

Malicious last mix. Claim 6 does not address the last mix. There are two reasons for that: first, in contrast to mixes, clients do not send receipts back to mixes. Therefore, a last mix cannot prove it actually delivered the packets. Secondly, the last mix may, in fact, identify non-loop messages in some situations. For example, if a client did not send packets in round r, then all the packets it is about to receive in round $\mathsf{r}+x$ (where x is the number of rounds it takes to complete a loop) are genuine traffic sent by other clients. Therefore, these messages can be dropped without detection.

However, dropping of messages by the last mix can also be done against the *ideal mix* (see Section 2.4), e.g., by a man-in-the-middle attacker. In fact, similar correlation attacks can be performed even without dropping packets, if clients have specific sending patterns. Therefore, mitigating this attack is beyond Miranda goals, and should be handled by the applications adopting Miranda [7].

[7] For example, [24, 44] use fixed sending rate (thus, foiling the attack). A concerned client can simply make sure to send additional loop packets in every round where no genuine traffic is relayed.

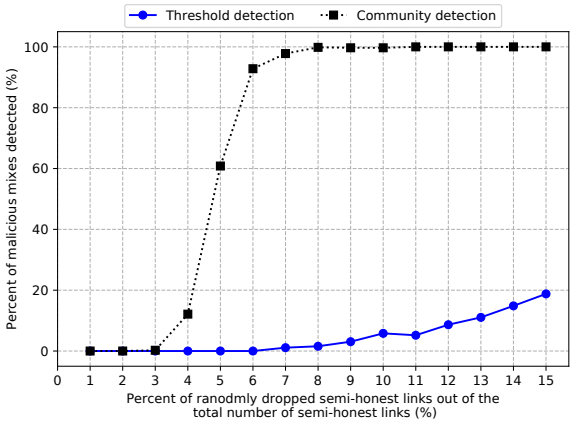


Figure 9: The effect of using community detection against malicious mixes.

7.4 Evaluation of Community Detection

The discussion in Section 6 presented several community detection techniques to leverage Miranda's reported links information into a detection tool that removes malicious mixes from the system. We now argue that the contribution of these mechanisms is both important and secure.

7.4.1 Empirical Results

We implemented the *Threshold Detection* and *Community Detection* algorithms described in Section 6, and evaluated them as follows. We generated a complete graph of $n = 100$ mixes where $n_m = 33$ of them are malicious. We modeled a random adversary by randomly dropping a fraction of the semi-honest links, making sure that any mix does not drop more than or equal to $thresh = n_m + 1$ links.

Figure 9 demonstrates the effectiveness of the algorithms. The *Threshold Detection* algorithm starts to become effective when roughly 10% of the semi-honest links are reported and improves as the number of reports increases. In comparison, the *Community Detection* algorithm presents significantly better results, starting when 4% of the semi-honest links are dropped and after 8% the algorithm is able to expose all possible malicious mixes. Considering that the *Community Detection* algorithm can only operate on malicious mixes that dropped more than one link, these results show that the algorithm effectively mitigates the non-strategic adversary. In [34], we discuss and compare another possible community detection algorithm, which potentially yields even better results.

7.4.2 Security Analysis

In essence, both the *Threshold Detection* algorithm and the *Community Detection* algorithm do the same thing: they both remove malicious mixes from the system. Therefore,

the only way for a strategic adversary to abuse these algorithms is to strategically drop links in a way that causes these algorithms to wrongfully remove honest mixes from the system, due to misclassification of honest mixes as malicious. We now argue that the *ThresholdDetection* and *CommunityDetection* algorithms are secured against such attack.

Claim 7. *An honest mix $M_i \in \overline{\mathcal{G}}$ never satisfies $\mathsf{Deg}_{\overline{\mathcal{G}}}(M_i) \geq thresh$.*

Proof. Assume to the contrary that there exists an honest mix $M_i \in \overline{\mathcal{G}}$ that satisfies $\mathsf{Deg}(M_i) \geq thresh$. However, if this is the case, then $\mathsf{Deg}_{\overline{\mathcal{G}}}(M_i) \geq thresh$, which implies $\mathsf{Deg}_{\mathcal{G}}(M_i) \leq n - thresh \leq n_h - 1$, which means that at least one honest mix disconnected from M_i, contradicting the assumption that honest links never fail. □

Claim 8. *The ThresholdDetection algorithm never removes honest mixes.*

Proof. According to the implementation of *Threshold Detection*, the algorithm only removes mix $M_i \in \overline{\mathcal{G}}$ that satisfies $\mathsf{Deg}(M_i) \geq thresh$. However, following Claim 7, this cannot happen for honest mixes. □

Claim 9. *The CommunityDetection algorithm never removes honest mixes.*

Proof. According to the implementation of *Community Detection*, the algorithm only removes mix $M_i \in \overline{\mathcal{G}}$ that satisfies $\mathsf{Deg}(M_i) > n_m - \mu_i$, which according to Claim 1 never happens for honest mixes. □

8 Related Work

In this section, we place our system in the context of existing approaches and compare Miranda with related works. First, we focus on works that present a similar design to Miranda. Next, we discuss how Miranda improves upon previous mix network designs. Finally, we briefly outline other techniques used to support reliable mixing.

Receipts. The idea of using digitally signed receipts to improve the reliability of the mix network was already used in many designs. In Chaum's original mix network design [12] each participant obtains a signed receipt for packets they submit to the entry mix. Each mix signs the output batch as a whole, therefore the absence of a single packet can be detected. The detection that a particular mix failed to correctly process a packet relies on the fact that the neighbouring mixes can compare their signed inputs and outputs. Additionally, [12] uses the untraceable return addresses to provide end-to-end receipts for the sender.

Receipts were also used in reputation-based proposals. In [20], receipts are used to verify a mix failure and rank their reputation in order to identify the reliable mixes and use them for building cascades. The proposed design uses a set of trusted global witnesses to prove the misbehavior of a mix. If a mix fails to provide a receipt for any packet, the previous mix enlists the witnesses, which try to send the packet and obtain a receipt. Witnesses are the key part of the design and have to be engaged in every verification of a failure claim, which leads to a trust and performance bottleneck. In comparison, Miranda does not depend on the witnesses, and a single one is just used to enhance the design. Moreover, in [20] a failure is attributed to a single mix in a cascade, which allows the adversary to easily obtain high reputation and misuse it to de-anonymize clients. Miranda rather than focusing on a single mix, looks at the link between the mixes.

In the extended reputation system proposed in [22] the reputation score is quantified by decrementing the reputation of all nodes in the failed cascade and incrementing of all nodes in the successful one. In order to detect misbehaviors of malicious nodes, the nodes send *test messages* and verify later via a snapshot from the last mix, whether it was successfully delivered. Since the test messages are indistinguishable, dishonest mixes risk being caught if they drop any message. However, the penalty for dropping is very strong – if a single mix drops any message, the whole cascade is failed. Therefore, because a single mix's behavior affects the reputation of all mixes in the cascade, the malicious nodes can intentionally fail a cascade to incriminate honest mixes. This design also proposed the *delivery receipts*, which the recipient returns to the last mix in the cascade in order to prove that the message exited the network correctly. If the last mix is not able to present the receipt, then the sender contacts a random node from the cascade, which then asks the last mix to pass the message and attempts to deliver the message.

Trap messages and detecting active attacks. The idea of using trap messages to test the reliability of the network was discussed in many works. The original DC-network paper [11] suggested using *trap* messages, which include a safety contestable bit, to detect message disruption. In contrast, the flash mixing [28] technique, which was later proved to be broken [38], introduces two dummy messages that are included in the input, and are later de-anonymized after all mixes have committed to their outputs. This allows the participants to verify whether the mix operation was performed correctly and detect tampering. However, both of those types of trap messages are limited to these particular designs.

The RGB-mix [18] mechanism uses heartbeat *loop* messages to detect the (n-1) attacks [45]. Each mix sends heartbeat messages back to itself, and if the (n-1) attack is detected the mix injects cover traffic to confuse the adversary. However, the key assumption of the proposed mechanism is limited only for anonymity among mix peers.

Mixmaster [39] and Mixminion [14] employed an infrastructure of *pingers* [43], special clients sending probe traffic

through the different paths in the mix network and recording publicly the observed reliability of delivery. The users of the network can use the obtained reliability statistics to choose which nodes to use.

Recent proposals for anonymous communication have also employed built-in reliability mechanisms. For example, the new Loopix [44] mix-network system uses *loop cover traffic* to detect (n-1) attacks, both for clients and mixes. However, this idea is limited to detecting only aggressive (n-1) attacks, but mix nodes systematically dropping single packets can operate undetected. Moreover, the authors do not also specify any after-steps or how to penalize misbehaving mixes.

The Atom [33] messaging system is an alternative design to a traditional mix networks and uses *trap* messages to detect misbehaving servers. The sender submits *trap* ciphertext with the ciphertext of a message, and later uses it to check whether the relaying server modified the message. However, the trap message does not detect which mix failed. Moreover, Atom does not describe any technique to exclude malicious servers, and a failed trap only protects against releasing the secret keys.

Other approaches. The literature on secure electronic elections has been preoccupied with reliable mixing to ensure the integrity of election results by using zero-knowledge proofs [3, 6, 29] of correct shuffling to verify that the mixing operation was performed correctly. However, those rely on computationally heavy primitives and require re-encryption mix networks, which significantly increase their performance cost and limits their applicability. On the other hand, the more 'efficient' proofs restrict the size of messages to a single group element that is too small for email or even instant messaging.

An alternative approach for verifying the correctness of the mixing operation were mix-nets with randomized partial checking (RPC) [30]. This cut-and-choose technique detects packet drops in both Chaumian and re-encryption mix-nets, however, it requires interactivity and considerable network bandwidth. Moreover, the mix nodes have to routinely disclose information about their input/output relations in order to provide evidence of correct operation, what was later proven to be flawed [32].

9 Limitations and Future Work

Challenges for making Miranda practical. The Miranda design includes several significant simplifying assumptions, mainly: (1) fixed set of mixes, (2) majority of benign mixes, (3) reliable communication and processing, and (4) synchronized clocks. Such assumptions are very limiting in terms of practical deployment; practical systems, e.g., Tor, cannot 'assume away' such issues. Future work should try to avoid these assumptions, while maintaining tight security analysis and properties as done in Miranda, or identify any inherent trade-offs.

Avoiding the clock synchronization assumption seems easy - simply adopt a secure clock synchronization protocol. However, avoiding the other three assumptions ((1) to (3)) seems much more challenging.

First, consider assumptions (1) and (2), i.e., assuming a *fixed set of mixes with majority of benign mixes*. These assumptions are central to Miranda design; since the goal of Miranda is to provide a way to penalize active attackers. If the adversary can simply retire penalized malicious nodes and replace them with new nodes that have an untarnished reputation, then there is no real gain in even trying to penalize or expose the adversary, and it becomes hard to argue why we can even assume most mixes are benign. However, a practical mixnet must allow a dynamic set of mixes, for both scalability and churn - mixes joining and leaving over time.

Next, consider the third assumption: *reliable communication and processing*. In practice, communication and processing failures will definitely happen - in particular, as a result of intentional DoS attacks. We believe that future work may deal with this significant challenge by both *minimizing failures*, by designing robust underlying mechanisms such as highly-resilient transport layer; and *refined assumptions and analysis*, e.g., considering incentives and game-theory analysis, to ensure that the system is robust to 'reasonable' levels of failures.

These issues are significant challenges for future research, essential towards the implementation of Miranda in practical systems. For example, such research must develop a reasonable model to allow nodes to join (or re-join), without allowing the adversary to gain majority by adding many mixes, as in Sybil attacks, and to retain the impact of removing corrupt mixes.

Extension to Continuous Mixnet. Miranda is designed for a synchronous mixnet. Recent research in mix networks showed that continuous-time mixes, especially pool mixes, may allow anonymity for low latency communication [44]. Future work may investigate how to integrate Miranda with continuous mixnets such as Loopix [44]. Such integration would raise challenges, such as, how would a mix know when it should receive the response from the next mix, esp. without leaking information to an attacker.

10 Conclusion

In this work, we revisited the problem of protecting mix networks against active attacks. The analysis performed showed that active attacks can significantly increase the adversary's chances to correctly de-anonymize users. Miranda achieves much better efficiency than previous designs, but at the same time quickly detects and mitigates active adversaries. Miranda employs previously studied techniques such as packet

receipts and loop traffic alongside novel techniques to ensure that each dropped packet penalizes the adversary. We take a new approach of focusing on problematic links between mixes, instead of mixes themselves. We also investigate how community detection enhances our mechanism effectively. The overall contribution of our work is an efficient and scalable detection and mitigation of active attacks. For additional details, including implementation details and efficiency, see [34].

Acknowledgments

We are grateful to our shepherd, Roger Dingledine, and to the anonymous reviewers, for their helpful and constructive feedback. This work was partially supported by the IRIS Grant Ref: EP/R006865/1 and by an endowment from the Comcast corporation. The opinions expressed in the paper are those of the researchers themselves and not of the universities or sources of support.

References

[1] Nym technologies, 2019. `https://nymtech.net/`.

[2] Panoramix project, 2019. `https://panoramix.me/`.

[3] Masayuki Abe. Mix-networks on permutation networks. In *International Conference on the Theory and Application of Cryptology and Information Security*, 1999.

[4] Dakshi Agrawal and Dogan Kesdogan. Measuring anonymity: The disclosure attack. *IEEE Security & Privacy*, 2003.

[5] Anonymous. QuicR: extending Quic for resiliency to extreme packet losses, 2019. Available from the authors.

[6] Stephanie Bayer and Jens Groth. Efficient zero-knowledge argument for correctness of a shuffle. In *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2012.

[7] Mihir Bellare, Juan A. Garay, and Tal Rabin. Distributed pseudo-random bit generators : A new way to speed-up shared coin tossing. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 1996.

[8] Oliver Berthold, Andreas Pfitzmann, and Ronny Standtke. The disadvantages of free mix routes and how to overcome them. In *Designing Privacy Enhancing Technologies*. Springer, 2001.

[9] Nikita Borisov, George Danezis, Prateek Mittal, and Parisa Tabriz. Denial of service or denial of security? In *Proceedings of the 14th ACM conference on Computer and communications security*, 2007.

[10] Carole Cadwalladr and Emma Graham-Harrison. Revealed: 50 million facebook profiles harvested for cambridge analytica in major data breach. *The Guardian*, 2018.

[11] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of cryptology, Springer*, 1988.

[12] David L Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 1981.

[13] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. 2015.

[14] George Danezis, Roger Dingledine, and Nick Mathewson. Mixminion: Design of a type iii anonymous remailer protocol. In *IEEE Symposium on Security and Privacy*, 2003.

[15] George Danezis and Ian Goldberg. Sphinx: A compact and provably secure mix format. In *30th IEEE Symposium on Security and Privacy (S&P)*, 2009.

[16] George Danezis, Chris Lesniewski-Laas, M. Frans Kaashoek, and Ross J. Anderson. Sybil-resistant DHT routing. In *10th European Symposium on Research in Computer Security ESORICS*, 2005.

[17] George Danezis and Prateek Mittal. Sybilinfer: Detecting sybil nodes using social networks. In *Proceedings of the Network and Distributed System Security Symposium, NDSS*, 2009.

[18] George Danezis and Len Sassaman. Heartbeat traffic to counter (n-1) attacks: red-green-black mixes. In *Proceedings of the 2003 ACM workshop on Privacy in the electronic society*, 2003.

[19] Harry Davies. Ted Cruz using firm that harvested data on millions of unwitting Facebook users. 2015.

[20] Roger Dingledine, Michael J Freedman, David Hopwood, and David Molnar. A reputation system to increase mix-net reliability. In *International Workshop on Information Hiding*, 2001.

[21] Roger Dingledine, Vitaly Shmatikov, and Paul Syverson. Synchronous batching: From cascades to free routes. In *International Workshop on Privacy Enhancing Technologies*, 2004.

[22] Roger Dingledine and Paul Syverson. Reliable MIX cascade networks through reputation. In *International Conference on Financial Cryptography*, 2002.

[23] Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 1983.

[24] Nethanel Gelernter, Amir Herzberg, and Hemi Leibowitz. Two cents for strong anonymity: the anonymous post-office protocol. 2018.

[25] Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Robust threshold DSS signatures. In *Advances in Cryptology—EUROCRYPT*, 1996.

[26] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, 1987.

[27] Glenn Greenwald and Ewen MacAskill. NSA Prism program taps in to user data of Apple, Google and others. 2013.

[28] Markus Jakobsson. Flash mixing. In *Proceedings of the 18th ACM symposium on Principles of distributed computing*, 1999.

[29] Markus Jakobsson and Ari Juels. Millimix: Mixing in small batches. Technical report, DIMACS Technical report, 1999.

[30] Markus Jakobsson, Ari Juels, and Ronald L Rivest. Making mix nets robust for electronic voting by randomized partial checking. In *USENIX Security Symposium*, 2002.

[31] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 2001.

[32] Shahram Khazaei and Douglas Wikström. Randomized partial checking revisited. In *RSA Conference*. Springer, 2013.

[33] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. Atom: Horizontally scaling strong anonymity. In *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[34] Hemi Leibowitz, Ania Piotrowska, George Danezis, and Amir Herzberg. No right to remain silent: Isolating malicious mixes - full version. `https://eprint.iacr.org/2017/1000`.

[35] Shengyun Liu, Christian Cachin, Vivien Quéma, and Marko Vukolic. Xft: practical fault tolerance beyond crashes. 2015.

[36] Nancy A Lynch. *Distributed algorithms*. Elsevier, 1996.

[37] Ralph Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology—CRYPTO*, 1987.

[38] Masashi Mitomo and Kaoru Kurosawa. Attack for flash mix. In *International Conference on the Theory and Application of Cryptology and Information Security*, 2000.

[39] Ulf Möller, Lance Cottrell, Peter Palfrader, and Len Sassaman. Mixmaster protocol – version 2. *IETF Draft*, 2004.

[40] Steven J Murdoch. Hot or not: Revealing hidden services by their clock skew. In *Proceedings of the 13th ACM conference on Computer and communications security*, 2006.

[41] Steven J Murdoch and George Danezis. Low-cost traffic analysis of Tor. In *IEEE Symposium on Security and Privacy*, 2005.

[42] Lasse Overlier and Paul Syverson. Locating hidden servers. In *IEEE Symposium on Security and Privacy*, 2006.

[43] Peter Palfrader. Echolot: a pinger for anonymous remailers, 2002.

[44] Ania M. Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The Loopix anonymity system. In *26th USENIX Security Symposium*, 2017.

[45] Andrei Serjantov, Roger Dingledine, and Paul Syverson. From a trickle to a flood: Active attacks on several mix types. In *International Workshop on Information Hiding*, 2002.

[46] Victor Shoup. Practical threshold signatures. In *International Conference on the Theory and Application of Cryptographic Techniques*, 2000.

[47] Paul Syverson, Gene Tsudik, Michael Reed, and Carl Landwehr. Towards an analysis of onion routing security. In *Designing Privacy Enhancing Technologies*. Springer, 2001.

[48] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP*. ACM, 2015.

On (The Lack Of) Location Privacy in Crowdsourcing Applications

Spyros Boukoros[1], Mathias Humbert[2], Stefan Katzenbeisser[1,3], and Carmela Troncoso[4]

[1]*Department of Computer Science, TU-Darmstadt, Germany*
[2]*Swiss Data Science Center, ETH Zurich and EPFL, Switzerland*
[3]*Department of Computer Science and Mathematics, University of Passau, Germany*
[4]*SPRING Lab, EPFL, Switzerland*

Abstract

Crowdsourcing enables application developers to benefit from large and diverse datasets at a low cost. Specifically, mobile crowdsourcing (MCS) leverages users' devices as sensors to perform geo-located data collection. The collection of geo-located data raises serious privacy concerns for users. Yet, despite the large research body on location privacy-preserving mechanisms (LPPMs), MCS developers implement little to no protection for data collection or publication. To understand this mismatch, we study the performance of existing LPPMs on publicly available data from two mobile crowdsourcing projects. Our results show that well-established defenses are either not applicable or offer little protection in the MCS setting. Additionally, they have a much stronger impact on applications' utility than foreseen in the literature. This is because existing LPPMs, designed with location-based services (LBSs) in mind, are optimized for utility functions based on users' locations, while MCS utility functions depend on the values (e.g., measurements) associated with those locations. We finally outline possible research avenues to facilitate the development of new location privacy solutions that fit the needs of MCS so that the increasing number of such applications do not jeopardize their users' privacy.

1 Introduction

Crowdsourcing is a participative online activity in which the undertaking of a task is outsourced to a group of individuals [29]. This new paradigm of distributing a fragmented task, is an efficient, scalable business model that allows the cheap (or often free) massive collection of data. Indicative of the growth of this data collection methods is the appearance of over 2,000 crowdsourcing platforms [1, 56] in the last years [83]. Furthermore, according to recent industrial reports [40], in the last decade, 85% of top global brands have already adopted crowdsourcing, and in 2018, 75% of the world's highest performing enterprises would use crowdsourcing. For instance, Google [2], Microsoft [3] and Mozilla [4] use crowdsourcing to build WiFi location databases.

A driving force of the crowdsourcing ecosystem growth is the widespread adoption of smart mobile devices, which enable users to collect geo-located data on their devices and share it with central servers to attain a particular objective. Mobile crowdsourcing applications (MCS) have millions of users around the world. For instance, OpenStreetMaps [5], a map generation project from contributed GPS points, reports 4.3 million users in 2018,[1] with 1 million active map editors contributing over 4 billion GPS points. Similarly, OpenSignal [6], a popular network-measuring application, reports over 20 million users.[2] Safecast [7], a citizen science project collecting environmental data, currently reports over 75 million measurements from approximately three thousand users. Many other applications are available [4, 6, 8–18].

MCS can bring great benefits for organizations and society. However, the collection and sharing of geo-located data raises serious privacy concerns, as demonstrated by scandals related to the publication of data by fitness applications [19, 20] or irresponsible data analysis by transportation companies [21]. Location data can be used to identify points of interest (POIs) [49, 52, 64], infer users preferences, or de-anonymize anonymous traces [89]. This risk increases when considering auxiliary publicly available information [30, 63, 70], and persists even when protections are put in place [77, 78].

Over the last decade, the research community has proposed a vast number of LPPMs to address these issues [76], some of which can provide strong differentially private guarantees [28, 34, 48] and even offer optimal utility [33, 73]. Even though it seems like the location privacy question is technically solved, the reality is that these LPPMs *solely focus on one use case*. They are generally geared towards LBSs in which users sporadically reveal their location in return for a service (e.g., to find nearby restaurants). In this context, utility is user-centric and hinges on the precision of the reported locations. In MCS applications, on the contrary, geo-located data is often shared continuously and over long periods and, while

[1] https://wiki.openstreetmap.org/wiki/Stats
[2] https://opensignal.com/methodology#over-20-million-users-of-our-app

the data utility is still correlated with the location precision, it is foremost tied to the values of the measurements reported at these locations (e.g., WiFi signal strength, or radiation level). Moreover, MCS utility cannot be captured with a user-centric approach as, by definition, MCS benefits from aggregating data collected by a large amount of users.

In this paper, we conduct the first in-depth evaluation of the effectiveness of LPPMs in the context of MCS. We use two representative applications, Safecast [7] and Radiocells [9], which make their contributors' data publicly available on their websites and which have very different utility functions. We propose two new privacy metrics based on statistical measures developed for binary classification and information retrieval to capture the privacy gain provided by the LPPMs with respect to the identification of areas and points of interest. We also consider new utility measures that, instead of relying on distance-based errors, quantify the accuracy of the aggregate values of data collectively generated.

The results of our experimental evaluation on real data contradict common beliefs regarding the privacy-utility trade-off offered by different LPPMs. First, location hiding methodologies, which in LBSs help concealing trajectories [60], do not bring any privacy benefits to MCS users. This is mainly because, in MCS, the volume of geo-located data is larger and contains points reported over long periods of time (more than a day). Second, differentially private mechanisms [28] offer good protection only for very strong parameters, and even when they are optimized for utility [33], they dramatically perturb the radiation measurements. For instance, in Safecast, we observed that it tremendously changed some areas' radiation levels, urging people to evacuate a place, and completely hindered the ability to localize radiation hotspots (location with elevated radiation). Finally, generalization techniques, usually dismissed in LBSs because of their poor utility, offer one of the best privacy-utility balance in MCS.

In summary, existing LPPMs are not well aligned with the needs of MCS applications. Therefore, new research is needed to approach the design of optimal LPPMs based on collective, value-based, utility metrics instead of user-centric, location-based utility.

Our contributions can be summarized as follows:

✓ We propose novel privacy and utility metrics suitable to evaluate the performance of LPPMs for MCS data publishing patterns.

✓ Using real data collected from two representative MCS applications, we show that existing LPPMs impose too high utility price and that many of them do not even provide good privacy guarantees in the context of MCS.

✓ We discuss technical and non-technical countermeasures to improve the privacy protection of MCS users.

2 Mobile Crowdsourcing Applications

In this section, we introduce the two crowdsourcing applications studied in detail in this paper.

2.1 Safecast

Safecast [7] is a volunteer-centered organization whose goal is to monitor the global radiation levels and detect abnormalities in near real time. Safecast crowdsources the collection of radiation data by providing users with devices that collect radiation measurements every five seconds.

Safecast dataset. This dataset contains 64.2 million measurements from 608 users, collected from 2011 to 2017. Radiation measurements contain the user's name, a unique user ID, the device's ID, latitude and longitude, a UTC timestamp, and the radiation value and units. No registration is required to access these data and Safecast's privacy policy[3] states that to enable flexibility "Anyone is free to use with no licensing restrictions". For our experiments we removed IDs corresponding to organizations, malformed entries, and converted all UTC times to local. After this process, the dataset has almost 56.7 million measurements from 540 users.

Safecast utility. The Safecast project uses the collected data to study different phenomena related to radiation. In this paper, we consider two of the main uses of the data.

First, we consider the interactive map to visualize radiation published on Safecast website. Safecast computes the visualized radiation levels from the crowdsourced measurements as follows. For a given region of interest, Safecast filters the measurements within the region and computes the average radiation at each location over the last 270 days. Second, they discretize the area to 2.25 million grid points (1500 discrete locations per axis). They create the displayed map using nearest-neighbor interpolation on the averaged radiation measurements associated to the points of the grid. The reported radiation is measured in counts per minute (cpm), expressing how many ionized particles are detected per minute by a monitoring instrument. This use case, which relies on averaging and interpolation, represents a setting in principle amenable to noise in the data.

Second, we consider the detection of *hotspots* – specific areas where radiation is above a pre-defined threshold. These hotspots indicate locations where radiation could be harmful for public safety. Once identified, Safecast might send experts to perform on-site examination to better understand the causes and consequences of such dangerous zones. Therefore, it is crucial that the localization of hotspots is accurate.

2.2 Radiocells

Radiocells [9] is a community project whose goal is to provide an open-source alternative to commercial, closed source, geo-

[3] https://blog.safecast.org/faq/licenses/

location databases for cell towers and wifi base stations. They also aim to provide raw telecommunication infrastructure data for use in diverse scientific studies. Radiocells crowdsources the collection of measurements via a mobile application called 'Radiobeacon'.[4] With this application, users continuously collect measurements as they perform daily activities. Users choose when to start and stop measuring, and when to upload the measurements to the Radiocells server. Furthermore, they can select a specific area where measurements will not be recorded, e.g., to protect their home locations. We do not study the impact of this defense in this paper, but previous work shows that it is rather fragile [57].

Radiocells dataset. The raw data uploaded to the server is publicly available for download. It is licensed under Creative Commons Attribution-ShareAlike 3.0 Unported and ODbL licences aimed at not restricting the use of the data.[5] Amongst other information, the measurements include: signal strength, cell (antenna) ID, location, timestamp, and smartphone model, software, OS version, and manufacturer. In an effort to preserve users' privacy, this dataset does not contain usernames. However, the combination of the smartphone characteristics, the location, and the network provider is likely to represent a quasi-identifier. We downloaded data for 2013 to 2017, obtaining 25 million measurements. To separate users' measurements, we grouped the measurements according to phone manufacturer, phone model, country and network operator. We obtained 998 potential unique users, of which we only kept those that had more than 100 measurements. We also removed users with spatial inconsistencies, i.e., we removed all users whose speed between two contiguous measurements was greater than 200 km/h. The dataset finally contains 568 users and about 4 million measurements.

Radiocells utility. Amongst other purposes, the Radiocells data can be used to geolocate antennas. Such information is useful to enable scientific studies about antennas distribution and signal quality in specific places. Contrary to Safecast, Radiocells does not provide documentation, nor provide code indicating how they produce their map of antennas. Thus, we use the location function described by OpenCellID [8], another crowdsourcing project with the same goal, which defines the location of an antenna as the average of the latitudes and longitudes of the measurements referring to this antenna.

3 Protecting Location Privacy in MCS

In this section, we describe the existing LPPMs we evaluate in our study. These LPPMs are designed for LBSs settings, which are different than MCS in two aspects. First, LBSs aim at fulfilling an individual need related to one user's location (e.g., find nearby restaurants), while MCS aims at fulfilling a common objective through collaborative measurements. Second, LBSs can often work with sparser geo-located data (just few points per geo-located query) than MCS, which requires continuous data collection and in a larger volume.

3.1 Defenses

We consider three type of LPPMs [65, 81]: (i) spatial obfuscation, (ii) hiding, and (iii) generalization. We do not consider the use of dummy locations or synthetic data [32, 35]. Both approaches focus on producing plausible artificial locations, but to the best of our knowledge there is no proposal that provides the means to generate measurements (or other values) to be associated to these locations. In fact, we argue that generating fake measurements, even using prior information, is bound to pollute the real-time measurements that these applications aim at collecting.

Spatial obfuscation. The state of the art in spatial obfuscation, which perturbs reported locations with noise, is *geo-indistinguishability (GeoInd)* [28]. This mechanism adapts differential privacy to location data, providing privacy guarantees independent from the adversary's prior information. This approach is widely used in the literature [22, 48, 62, 68, 75, 88]. Following the original definition in [28], we obfuscate locations by adding planar Laplacian noise. The magnitude of this noise is controlled by the parameter $\varepsilon = l/r$ which guarantees that the ratio between the probabilities of two points being the real location in an area of radius r is at most l.

Release-GeoInd. As with any differentially private mechanisms, in GeoInd the level of privacy decreases linearly with the number of reported locations. To address this limitation we implement a mechanism inspired by the predictive approach proposed in [34]. This defense reports a new noisy location if, and only if, the user has moved at least z meters away from his previous location. Otherwise, it repeats the last reported location. We call this approach "Release-GeoInd".

GeoInd-OR. Remapping[6] obfuscated locations to popular places according to prior knowledge on users' movements can offer optimal utility without reducing privacy [33, 73]. We complement GeoInd with the remapping approach in [33]. We refer to this approach as "GeoInd-OR".

Hiding. This defense achieves privacy by suppressing some of the users' locations [60, 61]. The released locations are *not* perturbed. We consider two hiding strategies: (i) a "Random" strategy in which users release a random subset of their points, and (ii) a "Release" strategy in which users only reveal a new point when they have traveled at least x meters away from the previously reported location.

Generalization. This defense reduces the precision with which locations are reported [31, 55]. We implement this approach by reducing the precision of the reported GPS coordinates [65]. We denote this defense as "Rounding".

[4] https://f-droid.org/packages/org.openbmap/

[5] https://radiocells.org/license

[6] A remapping g is a function $g : \mathbb{R}^2 \rightarrow \mathbb{R}^2$ that maps an output $z \in \mathbb{R}^2$ to another output $z' \in \mathbb{R}^2$ according to the probability density function $g(z'|z)$.

3.2 Measuring Privacy

Location privacy metrics in the literature are mostly based on a function of the distance between the real location of the user and the one inferred by the adversary [80, 81]. This function could measure the *correctness* of the adversary's inference (e.g. using, Hamming or Euclidean distances [81]), or the *uncertainty* of the adversary regarding the user's location (e.g., using entropy [73]). These metrics are very well suited for the case of LBSs, where users release one location per query, and the adversary tries to infer that location. However, they are hard to use in the MCS setting, where the adversary has access to locations released continuously over several days. In this case it is hard to establish between which points to compute a distance, or across which points to compute probability distributions for entropy-based metrics.

We also argue that the metrics above do not capture privacy in a manner understandable by users and developers of crowdsourcing applications. How much privacy is an error of 10 meters or 500 meters? It is clear that one is larger than the other, but not how much privacy they provide regarding the potential inference of sensitive information. Even more complicated is the case of entropy, whose units of measurement – bits, nats, or hartleys – are rarely known, let alone interpretable, by layman people.

Privacy gain. We propose to quantify privacy as the loss of adversarial inference power regarding two privacy dimensions understandable by users: geographical area and POIs. To quantify this loss, we use two well-established statistical measures: precision and recall. The former captures the increase in privacy when, after a defense, the adversary identifies many false candidate locations along with the user's real whereabouts. Here, the adversary has low *precision* ($\frac{TP}{TP+FP}$, where TP and FP refer to *true positives* and *false positives*, respectively). The latter captures the increase when, after the defense, the adversary cannot correctly identify the original locations visited by the user. Here, the adversary has low *recall* ($\frac{TP}{TP+FN}$, where FN refers to *false negatives*).

Spatial privacy gain. Spatial privacy considers the geographical *area* in which the adversary infers the user can be. We define the true positives (TP) as the intersection of the areas where the user can be before and after applying the defense (i.e., the area inferred by the adversary that corresponds to the user's real location). Similarly, we define the false positives (FP) to be the set difference of the area after the defense and the area before the defense (i.e., the area inferred by the adversary where the user was not present), and false negatives (FN) as the set difference of the area before the defense and the area inferred after the defense (i.e., the area where the user has been but that is missed by the adversary).

POI privacy gain. In reality though, the geographic area itself may not reflect users' privacy [80]: if there is only one point of interest in a large area, privacy should be low; and in small areas with many POIs (e.g., a block in a city), privacy should be large. We propose a complementary metric based on POIs. In this case, true positives (TP) are the POIs in the intersection of areas before and after the defense is applied. Similarly, false positives (FP) are POIs identified after the defense that were not present before, and false negatives (FN) are the POIs inferred initially that are missed after the defense.

Table 1: Safecast (top) and Radiocells (bottom) measurements per region. Vulnerable users are those with at least one cluster.

Region	Users	Measurements	Average per user	Standard deviation	Vulnerable users
Tokyo	30	2,701,367	90,046	203,576	24 (80%)
Fukushima	104	7,765,773	74,671	260,671	65 (62%)
World	540	56,655,768	105,504	70,954	349 (65%)
World	568	3,710,547	6,532	17,312	91 (16%)

3.3 Measuring Utility

Similarly to privacy, in LBSs, utility is measured as a function of distance between real and obfuscated locations of one user. This is unsuitable for MCS where location depends on the precision of the aggregate of multiple users' geolocated measurements. We now introduce the utility metrics used in our evaluation.

Distance-based. We call distance-based metrics those associated to LPPMs in the context of LBSs. In our experiments, we use the per-location haversine distance[7] between original and obfuscated locations.

Aggregate statistics. Most MCS providers are interested in aggregate statistics computed over individuals' contributions. This is the case for Safecast and Radiocells, where the radiation map, respectively the coordinates of the antennas, are derived from average measurements of MCS users. In our evaluation, we consider as MCS utility metrics the actual utility functions of the projects as described in Section 2.

4 Existing LPPMs Performance in MCS

4.1 Experimental setup

We experiment on all data available from Safecast and Radiocells. For Safecast, we additionally consider two regions in Japan with very different radiation profiles: Tokyo, where the radiation profile is quite uniform, and Fukushima, where the nuclear incident at the Daiichi power plant [23] in 2011 created areas with elevated radiation. Table 1 summarizes the statistics (number of users, total amount of measurements, and measurements per user) of the regions under study.

We evaluate the privacy gain and the utility loss of an LPPM as follows:

[7] Distance between two points on a sphere given their longitudes and latitudes.

Step 1. Adversary's inference. Inspired by previous works, we use clustering to implement inference on the regions and the points of interest for all users. [36, 42, 46, 49, 59, 64, 69, 86]. Concretely, we use the density-based clustering algorithm (DBSCAN) [46]. Contrary to other clustering algorithms (such as K-Means), DBSCAN is robust to noise and outliers and does not require to specify the number of clusters a priori (see Appendix A.1). We keep the five clusters with the highest number of points, and *we consider their total area as the geographical area input to the Spatial Gain metric*. Once clusters are identified, we use the OSM API[8] to find the POIs in the clusters of the targeted user. *We consider all points in the top five clusters as input for the POI Gain metric*. Table 1 reports the percentage of users *vulnerable* to our attacks *before* the defenses are applied, i.e., the percentage of users for which we find at least one cluster. For Safecast-Tokyo, we only report statistics for the 30 users considered when using GeoInd-OR (see Section 4.3).

Step 2. LPPM Application. We apply the LPPM to all users' data and repeat the actions in Step 1 to infer their regions and points of interest. Note that when Rounding to 2 or 3 decimals, obfuscated locations are separated by approximately 1,100 meters and 110 meters, respectively. Thus, our parametrization of DBSCAN is bound to not find any clusters. However, an adversary would know that given an obfuscated point, the actual location of the user is within a square of size 110, resp. 1,100 meters, centered in the reported location. Thus, for this case, instead of using DBSCAN clusters, we pick the squares of the respective sizes around the five most frequently reported obfuscated locations.

Step 3. Privacy gain. We compare the area (in square kilometers) of the clusters before and after the LPPM to compute the Spatial privacy gain, and the POIs inside the clusters for the POI privacy gain.

Step 4. Utility loss. In the case of aggregate statistics, the utility loss is application dependent. For Safecast, we consider the absolute difference in cpm per grid point between the radiation values on the application's interactive map (see Section 2.1), before and after the LPPM. In Radiocells, we consider as utility loss the distance between the location of the antennas before and after the LPPM.

4.2 Validating the Inference Strategy

We now validate the suitability of DBSCAN as strategy for inferring areas and points of interest in the context of MCS. Specifically, we test its suitability to identify workplaces on data from Safecast and OpenStreetMaps. As both projects' public data contain identifiable information about their users, we can validate the inferences against information available on other online platforms. We choose workplaces for ease of validation, but we note that it is just one of many inferences that could be done using location data [42]. Our results below confirm that DBSCAN is a suitable choice as basis to compute areas and POIs to input in our privacy metrics.

[8] https://wiki.openstreetmap.org/wiki/Overpass_API

Table 2: Safecast dataset statistics.

Measurements	Users	Avg measurements	Avg days
<10k	213	3,331	5
10k-100k	230	38,341	20
100k-1M	87	270,387	105
>1M	10	1,958,760	632

Ethical considerations. For these experiments we do not collect any personal data other than that made publicly available by the MCS projects. We have limited our inferences to the minimum to validate the suitability of DBSCAN. We only report aggregated or anonymized data such that no individual's data is exposed. We have notified the service providers about our findings, and we have shared our code with them so that they can make informed decisions regarding improvements of the privacy situation. Our code is open-source so that it can also be used by other crowdsourcing applications and improved by the research community [24]. This procedure has been approved by EPFL's Human Research Ethics Committee (HREC).

Safecast. To evaluate the effectiveness of DBSCAN in different situations, we split the users in the dataset into four groups according to their amount of measurements they report. For each group, Table 2 shows the number of users, their average amount of measurements, and the average number of days in which they took at least one measurement. From each group we select as targets for inference the 10 users with the most measurements that provide their real names. Since in the group with >1M there are only 4 users with real names, we end up with 34 target users in total. This allows us to manually validate our inferences in reasonable time.

Identifying workplaces. We run DBSCAN on every users' measurements during working hours (Monday to Friday from 9AM to 5PM). We configure DBSCAN to find clusters with at least 80 points separated by 60 meters and, if no clusters are found, we increase the distance by 30 meters (up to 120 meters maximum) and decrease the number of points by 15 (down to 35 points). These parameters have been chosen empirically to optimize the adversary's success, see Appendix A.5. To keep the manual analysis feasible, we only consider the five clusters with the highest number of points.

We expect that the users' workplace is one of the POIs within the inferred geographic area. In many cases, however, this area is large and contains many POIs. To ease manual validation, we use X-means clustering [74] to split these large clusters, and consider as POIs the centroids of the two largest subclusters. We end up with at most 10 POIs per user. We use the MapQuest API [25] to obtain these locations' addresses and, if existing, the names of the businesses at those coordinates. We recall that in our LPPM evaluation below,

we consider all points in the clusters as input for the POI Gain metric. This represents a resourceful adversary that can afford checking manually all the points and filter out those corresponding to businesses. We note that considering more points could cause more false positives, but the semantics of locations often makes it easy to filter these out, e.g., lakes or parks can be usually discarded as workplaces.

Once we have candidate workplaces, we validate them using social networks such as Twitter or LinkedIn, or the users' personal webpages. We note that 9 of our target users did not have a publicly available profile or had too common names to find their correct information, thus we could not validate their inferred workplaces. Overall, we recover the workplace of 35% of the target users. This result is consistent across the groups: 40% of the users with less than 10k measurements, 20% of the users with 10k-100k measurements, 50% of the users with 100k-1M measurements, and 25% of the users with more than 1M measurements. We conclude that DBSCAN performs well for POI identification irrespectively of the amount of data shared by the users. Surprisingly, this means that privacy seems not correlated to the volume of data made available to the adversary. On the contrary, it seems to be highly dependent on the collection patterns of the users. We observe that people fall in one of two categories: (i) Those who travel to different places with the goal of obtaining measurements, whose work addresses cannot be inferred; and (ii) those who measure radiation during their daily activities, whose work place we can find. The Safecast co-founders, who are the top contributors in terms of data points, fall in the first category, explaining the lower inference power for users with more than 1M measurements.

Our results confirm recent findings in the literature regarding personal information inferences from location data [42, 45]. Yet, we want to stress that the threat may be worse for MCS, due to the volume of data exposed by participants. For reference, Safecast's lowest contributing group has on average 3k measurements per user (see Table 2) while in the Twitter analysis performed by Drakonakis et al. [42] only the top contributing users (less than 0.06%) have more than 3k geolocated tweets. Thus, even if the number of MCS users is not as large as social networks' users, we expect a significant fraction of them to be vulnerable to privacy attacks.

Other POIs. A deeper analysis of the times and semantics of the POIs identified by DBSCAN revealed further information about Safecast users. Among others, we could infer two users' membership to specific organizations: one member of the Scientology church who reported many points from the Church of Scientology Celebrity Centre in a major city; and a Masonic lodge member who regularly visited the lodge headquarters. We could verify this information online for both users. We also identified two work-related activities: a US-based scientist working on a project about radiation around a lake in the Southern part of the US, and a photographer working in a Japanese city. We validated these inferences using Research Gate and the webpage of the artist, respectively. Finally, we could follow the education steps of a European PhD student. Her points of interest over time reveal the university where she obtained her master's degree, an exchange with another European university, and the university where she is completing her doctoral studies. We verified these facts on her CV available online.

OpenStreetMaps. Contrary to Safecast, OSM does not have an open API for accessing users' data. Yet, traces from users who have chosen to make their data available can be easily obtained from OSM's website.[9] To minimize the impact on OSM servers, and comply with their non-crawling policy, we manually downloaded data for 30 users with a large amount of contributions,[10] of which 17 used their actual names (or indicative nicknames). Although the majority of the points in the dataset were rather old (most of them at least 7-8 years old), we were able to verify previous workplaces for 3 of the 17 users (17%). We note that, for some users, we found out that they did not have a standard place of employment during data collection period (e.g., students). However, for *all* users, their POIs were within the area where they worked or lived. We used this fact to infer two of the users' short vacation trips which we manually verified with information publicly accessible from their social media accounts.

4.3 Privacy Gain

Defenses implementation. For the GeoInd defense, we set the privacy parameter $l = \ln(1.6)$, and use radius $r \in \{50, 150, 300\}$ meters which yields $\varepsilon \in \{0.01, 0.003, 0.001\}$. Remapping the locations for the LPPM GeoInd-OR requires computing the posterior probability for every candidate location. This operation is rather costly when the number of locations being considered grows. To keep a reasonable experimentation time, we only test GeoInd-OR for the Tokyo region in the Safecast dataset. We use 80% of the users to construct the prior probability distribution describing users' movements, and the remaining 20% to evaluate the effectiveness of the approach. We chose this 20% manually to keep a balanced testing set. It is composed of the top 10 users with many (more than 50k), moderate (between 10k and 50k), and few (less than 10k) measurements. Finally, for the Release-GeoInd mechanism, we use $l = \ln(1.6)$, $r = 50$ meters, and we select the distance between released locations to be $z \in \{30, 60, 90\}$ meters. We provide details about the implementation of these LPPMs in Appendices A.2 and A.3.

We implement the Random mechanism tossing a biased coin every time a location is about to be reported. The bias is set so that users release on average 40%, 60% or 80% of their measurements. For the Release mechanism, we sort all the locations reported by a user in chronogical order, and

[9] https://www.openstreetmap.org/traces
[10] http://resultmaps.neis-one.org/oooc

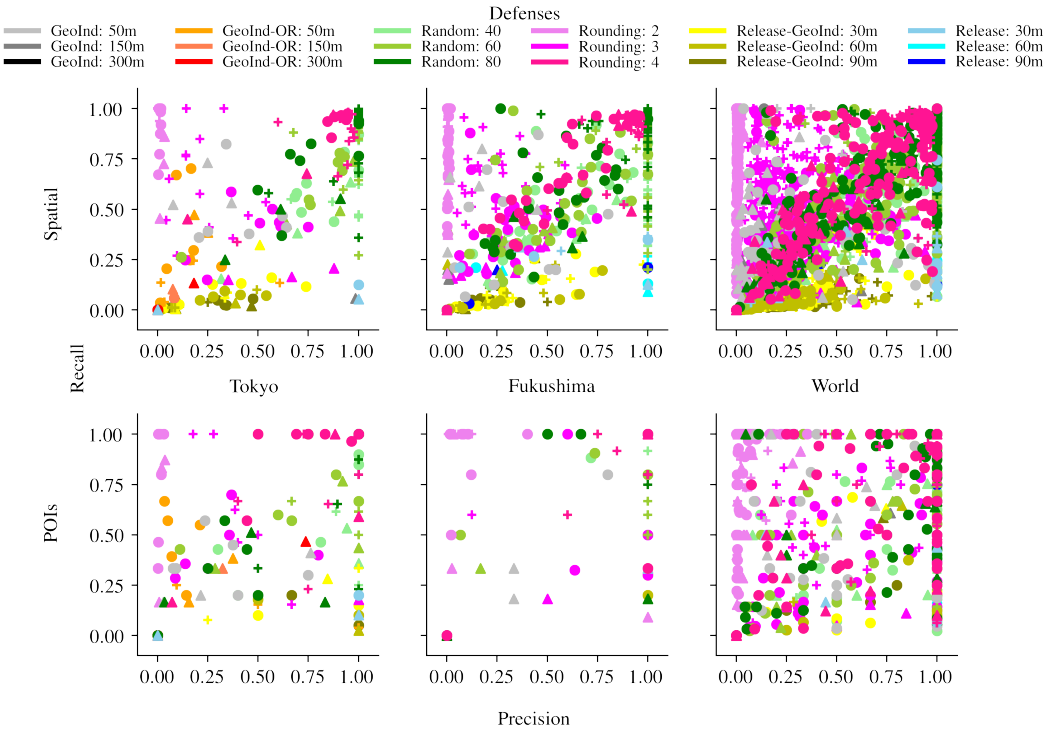


Figure 1: Safecast privacy gain: Spatial (top) and POIs (bottom). Amount of measurements per user + : <10k, ● : [10k,50k], ▲ : >50k. Each point on the graphs represents one user.

release a new location only if it is separated by (at least) $x \in \{30, 60, 90\}$ meters from the previously reported one. If two locations are less than x meters apart but in different days, we release them both.

Last, we implement Rounding by rounding to 2, 3, or 4 decimals the latitude and longitude of the users' locations. Effectively, this reduces the location accuracy to roughly 1,100 meters, 110 meters and 11 meters, respectively.

4.3.1 Safecast

We first evaluate the privacy gain of the LPPMs in the Safecast dataset. Figure 1 shows the Spatial (top) and POI (bottom) gain for Tokyo, Fukushima, and the whole world. (Figure 12 in the appendix shows the results for each of the defenses separately for the whole world.) The x-axis represents precision, and the y-axis recall. Each point in the graph represents a user, and the markers' shape indicate the amount of measurements she contributes. The colors represent the LPPMs. To compute these graphs, we configure DBSCAN to find clusters with at least 75 points separated by at most 30 meters (roughly the size of a small building). As in [42], we also require that, for each cluster, users either stay more than 30 minutes, or visit it for at least two days. A first reason to fix these parameters is to evaluate the gain for all users under the same conditions. A second reason is that the loose parameters used in Section 4.2 can yield very sparse clusters with few points that are hard to break by removing or perturbing locations. Thus, the defenses would perform equally bad and we would gain little information about their properties. Tightening the parameters reduces the work inference success to 21% (some clusters are not found), which still represents a significant risk.

Defenses that provide large gains result in points close to the figure axes. Points near the y-axis indicate low precision, i.e., cases in which the adversary correctly identifies some (or even all) of the true locations but also inferred many other wrong locations. Points near the x-axis indicate low recall, i.e., cases in which the adversary correctly identifies some real locations, but misses many others. Unsurprisingly, we observe a high variance in the defenses' performance since it is highly dependent on the user behavior. However, it is possible to identify some trends.

We first discuss the Spatial privacy gain (Figure 1, top). For the least privacy-preserving parameter ($r = 50$m), GeoInd significantly decreases the number of vulnerable users (grey points in the figure) from the values reported in Table 1. The reduction is 50% for Tokyo (from 24 vulnerable users to 12), 45% for Fukushima, and 45% for the whole world. When the mechanism is strengthened ($r = 300$m), GeoInd adds so much noise (see Figure 10 in Appendix A.4 for reference) that no users are vulnerable after the defense. In summary, GeoInd seems to provide fairly good privacy gain in Tokyo

and Fukushima. Yet, when we look at the whole dataset, it becomes clear that the protection provided by GeoInd is highly dependent on the users' movement patterns.

The Release-GeoInd (yellow) mechanism works generally better than GeoInd. Even though more users are vulnerable (only between 4% and 13% of the users become not-vulnerable) and the adversary obtains reasonable precision, it yields very low recall. This is because in this method users keep reporting the same obfuscated location until they move. This repetition results in clusters being found on fake locations that often do not overlap with the original ones. This reduction becomes more significant as the defense is configured to provide more privacy (larger z).

GeoInd-OR performs slightly better than vanilla GeoInd. This is because the remapping results in points being repeatedly mapped to popular places causing the generation of clusters around those not-real locations.

Similar to vanilla GeoInd, the Release mechanism (blue) significantly reduces the number of vulnerable users – by more than 50% even for the least conservative parameter. However, when precision is very high, i.e., when a cluster is found, it corresponds to a real location. The reason is that even though the user hides many points, if a location is visited regularly, the user will eventually report enough points around this location to make the cluster identifiable by the adversary.

The Random hiding mechanism (green) does not perform well. First, it reduces the number of vulnerable users less than other defenses (10% decrease in Tokyo, 27% in Fukushima, and only 5% when considering the whole world). From the vulnerable users only a handful obtain good protection. We could not find a clear pattern to predict which movement profiles would best benefit from this defense. For many users, especially those with a few points, removing points at random still yields high precision as the few measurements are very localized. Overall, we do not notice much influence of the fraction of hidden points on the privacy of the users.

Finally, the protection provided by Rounding (pink) depends on the rounding parameter. Keeping 4 decimals reduces accuracy by just 11 meters. Therefore, the adversary finds roughly the same clusters, i.e., for many users we observe high recall and precision after the defense (especially in Tokyo and Fukushima). On the contrary, rounding to 2 or 3 decimals significantly increases the size of inferred spatial areas, which leads to variable recall (depending on the users' movement patterns) and low precision.

Regarding the POI privacy gain (Figure 1, bottom), a first observation is that the amount of users vulnerable to the attack, i.e., points in the graph, is lower. This is because for many users the identified clusters do not contain any POI (according to the OSM API). Second, for the users who have POIs in their clusters, both recall and precision are higher than in the Spatial gain. This is because many of the large clusters that contribute to the low Spatial precision do not have POIs and thus do not contribute to the confusion of the adversary when identifying particular POIs. Furthermore, the clusters that the adversary finds after the LPPMs may cover a smaller area than the original clusters, but still contain most of the users' initial POIs. This provides a higher POI recall than Spatial recall. Third, in this case we observe a significant difference between Tokyo and Fukushima. The reason is twofold. First, the Fukushima prefecture is much larger than the area of Tokyo we consider. Second, Fukushima is a rural area and thus contains fewer POIs than Tokyo where even small clusters have many places of interest.

These observations reinforce previous insights that solely considering the spatial dimension may provide a false perception of privacy [80]. Considering a POI-privacy measure is necessary for providing a comprehensive picture of the privacy threat users face in MCS applications. We note that this perception also depends on DBSCAN parameters, which define the size of the regions found, and consequently the number of POIs, increasing the manual effort of the adversary. We discuss this effect in Appendix A.5.

Impact of the amount of measurements on privacy. We present in Figure 2 the Spatial gain for the three best LPPMs (all parameters combined) split by the amount of measurements users contributed. We discard Rounding 4 as it does not provide any privacy. We see that all LPPMs provide low precision and recall regardless of the users' contribution volume. The exception is Rounding which, as explained above, by definition provides variable recall and low precision.

Counterintuitively, the LPPMs perform worse for users who contribute fewer points. This is because the attack constructs more, and larger (on average 10 times bigger), clusters for people who share many points than for those sharing fewer points. These clusters are split after the LPPMs are put in place, as some reported locations are moved away from their original clusters while other measurements, perturbed with noise, concentrate to new places forming wrong clusters. For Rounding, where every cluster created after the LPPM has roughly the same size, users with a few measurements have higher recall because their initial small clusters are often covered by the large regions resulting from the LPPM.

Thwarting workplace inference. Finally, we evaluate the effectiveness of the different LPPMs at hiding workplaces. Recall that, without protection, we can identify the workplace of 21% (7 out of 34) of the users. Five defenses, GeoInd, Release-GeoInd, Release, and Rounding 2, protect all users from inferences. Random hiding requires heavy sampling to be effective (hiding only 20% permits the identification of 6 workplaces, and hiding 40% still reveals 1). Finally, unsurprisingly, Rounding to 4 decimals does not protect against work inference, and Rounding with 3 decimals only hides one workplace out of 7.

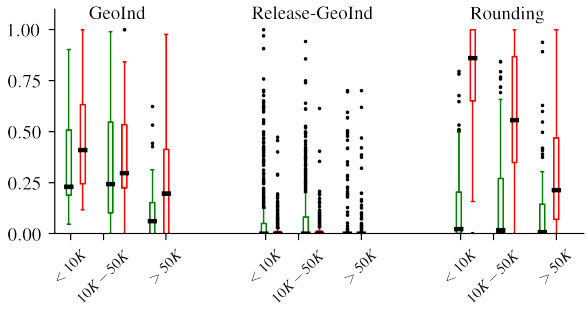


Figure 2: Precision (green) and recall (red), depending on the amount of measurements x per user for three selected defenses (all parameters combined).

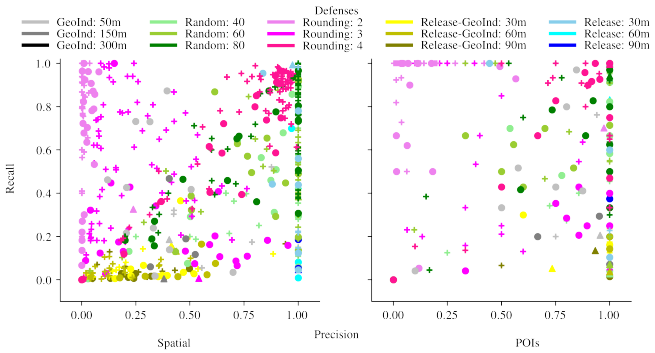


Figure 3: Spatial privacy gain (left part) and POI privacy gain (right part) in Radiocells. Amount of measurements per user + : <10k, ● : [10k,50k], ▲ : >50k. Each point on the graphs represents one user.

4.3.2 Radiocells

Users in Radiocells have on average fewer measurements than those in Safecast, and clustering requiring 75 points yields very few clusters. Hence, for this dataset we loosened the DBSCAN requirement to 25 points per cluster.

We see in Figure 3 that GeoInd-based mechanisms behave similarly to the Safecast case in terms of Spatial gain: GeoInd provides highly variable protection, and Release-GeoInd yields low recall while precision depends on the user behavior. Vanilla GeoInd decreases the number of vulnerable users by 14%, and Release-GeoInd by 2%. Given that only 16% of the users were initially vulnerable, this reduction is significant. For the hiding mechanisms, the Random and the Release mechanisms decrease the number of vulnerable users by 7% and 14%, respectively. For the vulnerable users, contrary to Safecast, these mechanisms consistently yield high precision, i.e., they offer poor privacy protection for Radiocell's users movement profiles. Finally, the Rounding mechanisms with parameters 2 and 3 offer reasonable privacy. Regarding POIs, we observe similar behavior to the Safecast dataset.

Overall, the results in Radiocells are consistent with our findings in the Safecast dataset, confirming the trends regarding the LPPMs behavior in the MCS setting.

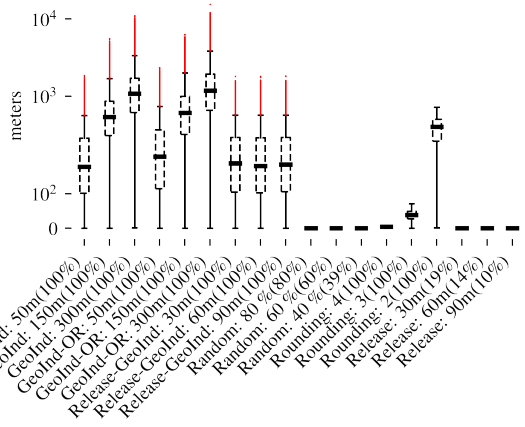


Figure 4: Measurement error in Tokyo using a distance-based metric. This can be interpreted either as privacy gain or utility loss.

4.4 Privacy-Utility Trade-Off

4.4.1 Safecast

Distance-based metric vs Aggregate statistics for MCS. We first evaluate the utility loss incurred by the LPPMs measured using the LBS-oriented distance-based metric described in Section 3.3. This utility metric is based on the distance between reported and real locations, but disregards the (radiation) values that Safecast cares about. Figure 4 displays the results for users in the Safecast-Tokyo dataset. The y-axis indicates the distance in meters, and the x-axis the LPPM and the percentage of points that are released. Random and Release LPPMs, which add no noise, are the best in terms of error; and GeoInd LPPMs offer the worst performance as they tend to spread locations — sometimes more than a kilometer away from the initial measurements (see Figure 10 in Appendix A.4).

Next we consider the utility loss for aggregated statistics, i.e., utility measured as the difference between radiation values to be plotted on the generated map. We plot per grid-point utility loss for Tokyo and Fukushima in Figures 5 and 6, respectively. We observe that the loss is similar in both regions, though in Fukushima the median loss is slightly higher and there are more, and larger (up to 10^4 radiation offset with respect to the original value), outliers than Tokyo. Because of the interpolation step, in this case all GeoInd variants offer roughly the same utility loss on average. Still, Hiding and Rounding strategies offer better performance, with small

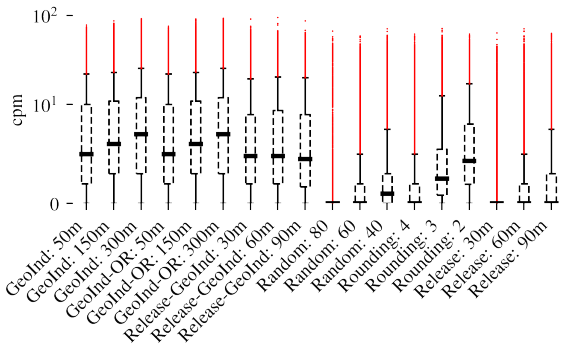


Figure 5: Absolute difference in Tokyo's radiation values with Safecast dataset.

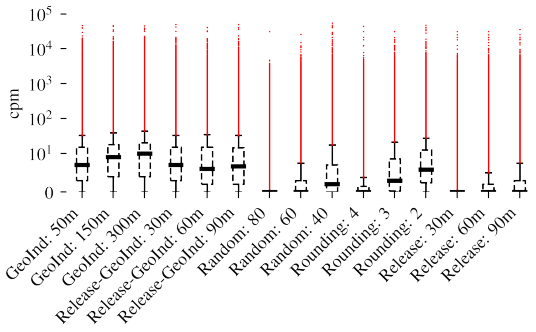


Figure 6: Absolute difference in Fukushima's radiation values with the Safecast dataset.

median error for the least protective parameters.

If we compare the distance-based results (Figure 4) to the aggregated statistics utility loss (Figure 5), we observe significant differences. First, the interpolation step results in LPPMs based on GeoInd to fare much better in terms of aggregates than in terms of distance. Second, distance-based metrics underestimate the utility loss of hiding LPPMs (Random and Release). While it is true that the released points have no error in distance, hiding points comes at a cost not reflected in the metric. This is made evident by the aggregated metrics, which show that the more points are hidden, the larger is the utility loss. We note that relying on Markov mobility models such as in [51, 81] could help interpolate the hidden locations. Yet, this would not help recover the (radiation) values attached to them and the utility loss would remain. For the generalization mechanisms, distance-based metrics consistently report larger median loss, but have less variance and less outliers.

In summary, distance-based metrics provide a very different perception of the LPPM performance than considering utility functions computed on the geo-located values, overestimating the performance of some methods (e.g., hiding strategies) and underestimating others (e.g., GeoInd-based LPPMs). We conclude that traditional LBS-oriented metrics are inadequate for measuring utility in MCS scenarios.

Semantic interpretation. The absolute difference in cpm of measurements before and after the defense gives a rough idea about the utility loss, but it is difficult to interpret. Is it significant? What is the effect of outliers? Does reporting the values after the defense have any implication on the danger for human health? To answer these questions, we study how the variance introduced by the defenses can change the interpretation of the risk at a given location. To this end, we rely on the cpm safety scale [26] provided with one of the top-seller Geiger counters (radiation measurement devices) on the market. This scale contains five categories:

- Category 1: 0-50 cpm. Normal radiation background.
- Category 2: 51-99 cpm. Medium level.
- Category 3: >100 cpm. High level.
- Category 4: >1000 cpm. Very high level, leave area.
- Category 5: >2000 cpm. Extremely high level, immediate evacuation.

Table 3: Danger category changes after applying Geo-Ind ($r = 300$ meters) in Fukushima.

Geo-Ind: 300m	**1**	**2**	**3**	**4**	**5**	**Number**
Original						**of points**
1	79.7%	19.3%	1%	0.003%	0.001%	1,354,110
2	41.5%	49.5%	9%	0.023%	0.01%	650,486
3	8.7%	35.9%	52.2%	2.3%	0.9%	229,848
4	2.5%	3.3%	49.3%	29.8%	15.1%	10,489
5	3.9%	1.7%	34.7%	29.3%	30.4%	5,067

We select the prefecture of Fukushima and two defenses that produce a good level of privacy: GeoInd 300m and Rounding 2. For each of the 2.25 million grid-points on Safecast's radiation map for Fukushima, we compute their radiation category according to the safety scale before and after each defense. For GeoInd 300m, which is of probabilistic nature, we repeat the procedure 10 times and report the average. We present the results in Tables 3 and 4. We observe that the majority of the points either stay in their original category or move to a nearby. However, we observe some extreme category jumps from the first category (safe radiation levels) to the fourth and fifth (high danger). For instance, GeoInd causes 53 places to be marked as dangerous instead of safe. Even more alarming, 283 locations that should be marked as extremely dangerous are marked as safe or slightly elevated (categories 1 and 2). On the contrary, the Rounding mechanism limits the number of extreme changes. For instance, there is a category jump from 5 to 1 and 2 only for 45 grid-points.

Why optimal remapping does not work for MCS. Even though GeoInd-OR was designed to increase utility while preserving privacy, we observe that, in the MCS case, utility roughly stays the same (Figure 5), and privacy slightly increases, both in decreasing the number of vulnerable users and in increasing the spatial gain. The reason for this mismatch is that this mechanism was designed in the context of

Table 4: Danger category changes after applying the Rounding mechanism (2 decimals) in Fukushima.

Rounding: 2 / Original	1	2	3	4	5	Number of points
1	89.3%	10.3%	0.3%	-	0.001%	1,354,110
2	30.2%	64%	5.8%	0.003%	-	650,486
3	0.7%	22.6%	74.8%	1.6%	0.3%	229,847
4	0.2%	0.01%	43.3%	39.6%	16.9%	10,490
5	0.9%	-	9.3%	42.1%	47.6%	5,067

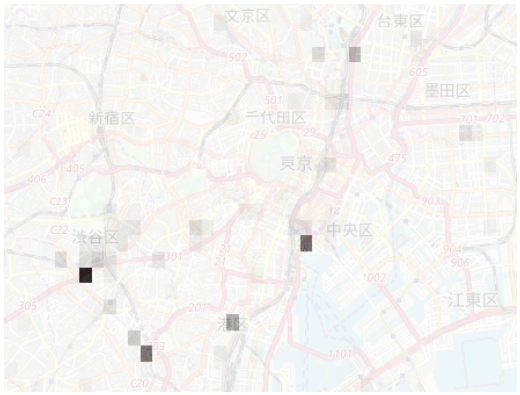


Figure 7: Prior probability of visiting locations in Tokyo (white - low probability, black - high probability).

LBSs, where remapping locations to places where the user is likely to be is bound to provide good utility on average. However, in Safecast, the utility does not depend on the locations themselves, but on the associated measurements. Remapping the location, however, concentrates measurements in these popular locations, effectively polluting the measurements. We illustrate this effect in Figure 7, which represents the prior probability of users' locations over all locations in Tokyo (low in white, high in black). In the low probability areas, most locations have the same probability, thus remapping has a randomizing effect. However, when there is a location with high probability, all locations are remapped to this popular location. We note that, while significantly hurting utility, this effect creates artificial clusters that reduce the adversary's precision and recall, thus increasing privacy.

The case of high precision measurements. Safecast also uses the crowdsourced measurements to monitor radiation *hotspots* that could be dangerous for public health. For this case, location precision is highly important, both to understand the dangers it can cause and to keep low costs if experts have to be sent to study the origin of the abnormality.

We study the impact of LPPMs on hotspot localization by looking for locations with more than 100 cpm radiation after averaging the measurements over the last 270 days but *before interpolating the data*. This is to avoid that interpolation modifies the position of the hotspots, or even eliminates them. We show the results of detection when using the raw measurements (top left), and after the application of Release-GeoInd 30m (bottom left), GeoInd 300m (top right), and Rounding 2 (bottom right) in Figure 8. We see that noise-based mechanisms spread the measurements and, as the noise increases, create additional hotspots. Thus, these mechanisms are useless for hotspot detection: the results cannot be properly interpreted. Imagine a hotspot in a place known to present high radiation, thus being already closely monitored by the authorities. Finding such hotspot is not alarming. However, after spreading, the finding of hotspots conveys a much different message, especially when they appear in zones that had low radiation in the past.

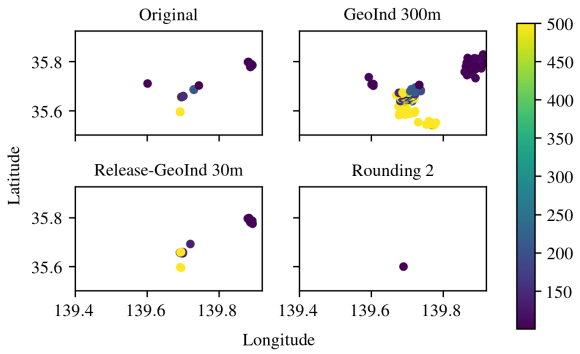


Figure 8: Safecast: Hotspot detection for areas with at least 100 cpm. Comparison of various defenses vs the original hotspots.

Generalization such as Rounding 2, which provides a good privacy-utility tradeoff for aggregated statistics, also performs poorly. In this case, the defense causes hotspots to disappear, potentially causing a dangerous situation if a high radiation zone is marked as safe. We also carry out experiments with hiding mechanisms and find that, similarly to Rounding, they miss some of the original hotspots.

Safecast takeaways. Considering only the privacy loss, GeoInd variants (except GeoInd 50m) and Rounding to 2 decimals seem to offer the best performance, while Random sampling and Release's protection is generally bad in terms of precision, and also too dependent on users' movement profiles. However, an analysis of the utility impact indicates that *none of the existent LPPMs is well suited* for the Safecast setting. The semantic interpretation results indicate that even if two defenses produce similar average results, the outliers they create can convey opposite messages. Furthermore, even a slight addition of noise or generalization can hinder the project's ability to correctly locate abnormal events. These limitations effectively prevent Safecast from deploying them to protect their users' privacy.[11]

[11] This statement was verified in communication with Safecast.

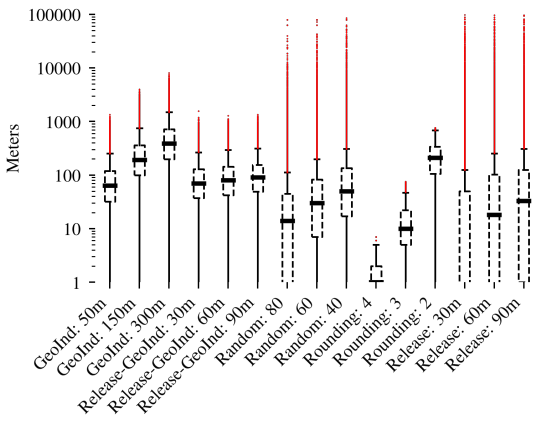


Figure 9: Radiocells: Utility loss (distance to tower location).

4.4.2 Radiocells

Radiocells' utility function is rather different than the one for Safecast. Instead of averaging measurements associated to a location, Radiocells averages all reported coordinates associated to an antenna to derive its position. We show the related utility loss for different LPPMs in Figure 9.

All GeoInd variants induce high utility loss, with medians between 80 and 400 meters, and with outliers beyond 2 kilometers. Surprisingly, in this use case hiding mechanisms (Release and Random) have many outliers. After manual inspection, we found out that several users had inconsistent measurements. For instance, a user was swapping her measurements' longitudes and latitudes in a random pattern. Other outliers are caused by providers moving their antennas IDs creating mixed measurements for a given ID. Furthermore, hiding defenses also influence the number of antennas located. In our dataset, we detect from 10.2% up to 18.6% fewer antennas when the Release defense is used, and the Random mechanism eliminates from 2.6% up to 13.7% of them.

The best mechanism in the Radiocells dataset is Release GeoInd which offers on average lower utility loss than other LPPMs and provides acceptable privacy. However, some antennas might be moved over a kilometer away. The next best alternative is Rounding 2 that has a higher median utility loss but no outliers. However, as the goal of the project is to *accurately* detect antennas in order to give individuals the ability to geolocate themselves offline or to enable scientific studies, a median error of 100 meters (Release GeoInd) or 200 meters (Rounding 2) is considered too large and precludes Radiocells from deploying them.

5 What's Next?

In this section, we elaborate on technical and non-technical steps to enhance privacy at smaller utility cost in the context of MCS applications.

5.1 Towards Effective Defenses

We first discuss possible strategies to improve the trade-off between users' privacy and MCS utility.

An unexplored approach is the use of advanced cryptographic protocols to compute the values of interest for MCS without revealing the users' individual values to the providers [41]. For instance, users could use multi-party computation to collaboratively compute aggregates and only report the result to the provider. However, cryptographic approaches require high computational power on the users' side and increase the bandwidth needs to perform the joint computation. Furthermore, this would limit the availability of raw measurements for analysis other than those predefined by the cryptographic protocols, which is at odds with the principles of open data and open science defended by most of the MCS platforms.

In our evaluation, we only considered spatial generalization. Another avenue to explore would be to also generalize the time dimension. On its own time obfuscation cannot hide patterns revealed by repeated visits. However, combined with full de-identification and hiding of users could reduce the inference power of the adversary. For instance, the MCS service provider could release a batch of measurements once a day or once a week without linking these to any user identifier. These techniques would be cheaper than the use of cryptography, but require trust on the service provider to properly apply sanitization and protect the raw data.

A third research path is the co-design of defenses and aggregation algorithms. In this paper, we have considered that the output of the LPPMs is directly input to the utility functions currently used by MCS providers. However, it would be possible that the providers adapt their data processing to account for noise, using statistical methods or machine learning, as done in fields that rely on noisy sensors [79] or train in different settings from which they are deployed [43, 72, 85].

Finally, MCS could provide users with dedicated local software (e.g., building on our evaluation method) to alert them regarding the privacy dangers of publishing raw location data. Such a system would allow them to selectively hide some of their measurements, reducing the confidence of inference attacks. We note that, when building such a tool, one would like to consider attacks beyond the POI-based inferences considered in this paper. For instance, it has been shown that co-locations can unveil social links [38, 44]. We run a preliminary evaluation to learn whether our MCS setting is also prone to such an attack. We identified 50 unique pairs of users with real names and at least one co-location (similar latitude, longitude, and time) in the Safecast dataset. We could validate 16 of these pairs as real friendships using information available on online social networks, i.e., yielding a 32% correct inference rate. Note that many of the other pairs could not be verified because either users were not part of any social network or they did not publicly reveal their social links. More advanced methods, such as measuring the amount of time

two users are co-located or the number of different locations where two users jointly report their locations [30,38,87] could further improve these results. Therefore, new defenses need to also obfuscate co-locations [71].

5.2 Privacy Considerations for Developers

In our study, we identified a number of issues related to the collection and sharing of data that, even though cannot fully prevent inference, could make inference attacks detectable and could render potential attackers accountable.

A first consideration to make is the type of policy under which MCS publish the collected data. While making large datasets available to everyone for unrestricted use is admirable, and certainly of high value for the academic community, it can have serious implications for the altruistic contributors. To reduce this risk, developers could add clauses to the policies that not only mandate that use of the data is properly acknowledged, but also that it is well documented, implying that researchers or other individuals have to disclose how they have processed the data, and for which purpose.

Second, both Safecast and Radiocells datasets are available for download without the need for authentication. This hinders traceability of who has the data, and thus enables stealthy attacks where nor the users neither the applications are aware of the danger. Like in other projects that make data available for research and other purposes (e.g., the Drebin project[12]), these sites could require simple registration to maintain a log of who has had access to the datasets. Together with the previous requirement, which would include documentation of sharing, it should help mitigate the risks.

Third, these applications typically do not perform any control on who are the contributors. This poses a particular problem when it comes to children. In many jurisdictions, children's data are subject to particular legislation [37,47], and in particular require the parents' consent to be collected and processed. The lack of control upon collection implies that the datasets could contain children's geo-located data collected illegally. Adding control would solve this problem and also support the previous two points.

Finally, the datasets we studied contain data from users from all over the world. These users, therefore, are subject to different legislations that regulate how their data can be processed. While this may not be a problem for corporations or criminals that want to exploit the datasets, it creates a hurdle for researchers who have to obtain approval from their institution for data processing. This problem arised during our discussions with our institution's Ethical Review Committee, and almost caused us to stop the project. In other words, lack of proper documentation may limit the free use of the data for science, effectively hindering one of the main goals of these applications. Better documentation as to the origin of data and its use possibilities would greatly facilitate the process.

[12] https://www.sec.cs.tu-bs.de/~danarp/drebin/

6 Related Work

We have covered the related work on LPPMs in Section 3.1 and the previous work on privacy quantification in Section 3.2. We complete this review of the literature with previous research on human mobility and its privacy implications.

Similar to [36, 49, 58, 59, 64, 66, 67, 69, 86], our POIs extraction attack is based on machine learning. Gambs and Killijian [52] also rely on POIs inference to build mobility Markov chains and de-anonymize traces. Gonzalez et al. [54] and Song et al. [82] study anonymized mobile phone data. Their results indicate that human trajectories have a high degree of temporal and spatial regularity, and that an individual's location data history is a unique identifier. De Montjoye et al. [39] investigate how the uniqueness of mobility traces decays depending on their resolution. They show that uniqueness cannot be avoided by lowering the resolution of a dataset. While these works aim at understanding the uniqueness of individuals or de-anonymize them, we focused on inferences that rely on labeled traces.

Similar to us, Gambs et al. [50] develop a platform for evaluating various sanitization methods and attacks on geo-located data. They focused on evaluating LBSs, while we evaluate the effectiveness of defenses on MCS applications. We also use different privacy metrics, and utility functions, tailored to the MCS scenario. Finally, Drakonakis et al. [42] explore the privacy loss stemming from by public location metadata. They propose a tool to infer users' regions of interest and, by experimenting with data gathered from Twitter, they illustrate the accuracy of their tool in pinpointing users' sensitive locations. Furthermore, they highlight how the spatial data provide additional context on the information shared by the user. We use similar techniques to prove that these inferences are also possible in MCS. For further information about the security and privacy landscape of location data we refer the reader to the surveys in [53, 65, 76, 84].

7 Conclusion

Mobile crowdsourcing is an increasingly popular way to collect geo-located data from millions of contributors. We present the first study on privacy implications of MCS applications. We study the applicability of well-established location privacy defenses created for LBSs. We show that neither the location privacy and utility metrics typically found in the literature nor the existing privacy-preserving mechanisms are well-suited for the MCS case. On the one hand, given the persistent patterns stemming from continuous collection, these solutions provide less privacy than in the case of LBSs where locations are revealed once. Second, the existing mechanisms are optimized to provide utility regarding the location of the users, but MCS applications rely on measurements associated to these locations, or on some function of the locations. Therefore, state-of-the-art defenses have a detrimental impact on

the MCS utility.

In conclusion, we identify an underexplored space in the location privacy literature, that is of practical relevance for many new applications. We have outlined promising lines to improve the situation. We hope that our findings spawn new research that soon enables the deployment of privacy-preserving crowdsourcing applications.

Acknowledgments

This work has been funded by the German Science Foundation (DFG) as part of the project A1 within the RTG 2050 "Privacy and Trust for Mobile Users".

References

[1] URL: https://www.spotteron.net/apps.

[2] URL: https://support.google.com/wifi/answer/6246642.

[3] URL: https://privacy.microsoft.com/en-us/windows-10-location-and-privacy.

[4] URL: https://location.services.mozilla.com.

[5] URL: https://www.openstreetmap.org.

[6] URL: https://opensignal.com/.

[7] URL: https://blog.safecast.org.

[8] URL: https://www.opencellid.org.

[9] URL: https://radiocells.org.

[10] URL: https://www.skyhookwireless.com.

[11] URL: https://www.sensorly.com.

[12] URL: http://www.cellumap.com.

[13] URL: https://www.mapillary.com.

[14] URL: https://play.google.com/store/apps/details?id=com.opensignal.weathersignal.

[15] URL: https://www.waze.com.

[16] URL: https://www.qualcomm.com/solutions/automotive/drive-data-platform.

[17] URL: https://www.gokamino.com.

[18] URL: http://www.app-store.es/stereopublic.

[19] URL: https://www.wired.com/story/strava-heat-map-military-bases-fitness-trackers-privacy/.

[20] URL: https://www.bellingcat.com/resources/articles/2018/07/08/strava-polar-revealing-homes-soldiers-spies/.

[21] URL: http://www.whosdrivingyou.org/blog/ubers-deleted-rides-of-glory-blog-post.

[22] URL: https://github.com/SpatialVision/differential_privacy.

[23] URL: https://en.wikipedia.org/wiki/Fukushima_Daiichi_nuclear_disaster.

[24] URL: https://github.com/spring-epfl/MCSAuditing.

[25] URL: https://developer.mapquest.com/documentation.

[26] URL: http://www.gqelectronicsllc.com/GMC_Safty_Guide.jpg.

[27] URL: http://earthpy.org/interpolation_between_grids_with_ckdtree.html.

[28] Miguel E Andrés, Nicolás E Bordenabe, Konstantinos Chatzikokolakis, and Catuscia Palamidessi. Geo-indistinguishability: Differential privacy for location-based systems. In *CCS*, 2013.

[29] Enrique Estellés Arolas and Fernando González-Ladrón-de-Guevara. Towards an integrated crowdsourcing definition. *J INF SCI*, 2012.

[30] Michael Backes, Mathias Humbert, Jun Pang, and Yang Zhang. walk2friends: Inferring social links from mobility profiles. In *CCS*, 2017.

[31] Bhuvan Bamba, Ling Liu, Péter Pesti, and Ting Wang. Supporting anonymous location queries in mobile environments with privacygrid. In *WWW*, 2008.

[32] Vincent Bindschaedler and Reza Shokri. Synthesizing plausible privacy-preserving location traces. In *IEEE S&P*, 2016.

[33] Konstantinos Chatzikokolakis, Ehab Elsalamouny, and Catuscia Palamidessi. Efficient utility improvement for location privacy. *PETS*, 2017.

[34] Konstantinos Chatzikokolakis, Catuscia Palamidessi, and Marco Stronati. A predictive differentially-private mechanism for mobility traces. In *PETS*, 2014.

[35] Rui Chen, Gergely Ács, and Claude Castelluccia. Differentially private sequential data publication via variable-length n-grams. In *CCS*, 2012.

[36] Sung-Bae Cho. Exploiting machine learning techniques for location recognition and prediction with smartphone logs. *NEUROCOMPUTING*, 2016.

[37] U.S. federal trade commission. complying with coppa: Frequently asked questions, 2015.

[38] David J Crandall, Lars Backstrom, Dan Cosley, Siddharth Suri, Daniel Huttenlocher, and Jon Kleinberg. Inferring social ties from geographic coincidences. *P NATL ACAD SCI USA*, 2010.

[39] Yves-Alexandre De Montjoye, César A Hidalgo, Michel Verleysen, and Vincent D Blondel. Unique in the crowd: The privacy bounds of human mobility. *SCIENTIFIC REPORTS*, 2013.

[40] Deloitte. The three billion, enterprice crowdsourcing and the growing fragmentation of work. URL: https://www2.deloitte.com/content/dam/Deloitte/de/Documents/Innovation/us-cons-enterprise-crowdsourcing-and-growing-fragmentation-of-work%20(3).pdf.

[41] Daniel Demmler, Thomas Schneider, and Michael Zohner. Aby-a framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[42] Kostas Drakonakis, Panagiotis Ilia, Sotiris Ioannidis, and Jason Polakis. Please forget where i was last summer: The privacy risks of public location (meta) data. In *NDSS*, 2019.

[43] Greg Durrett, Jonathan K. Kummerfeld, Taylor Berg-Kirkpatrick, Rebecca S. Portnoff, Sadia Afroz, Damon McCoy, Kirill Levchenko, and Vern Paxson. Identifying products in online cybercrime marketplaces: A dataset for fine-grained domain adaptation. In *Conference on Empirical Methods in Natural Language Processing, EMNLP*, pages 2598–2607, 2017.

[44] Nathan Eagle, Alex Sandy Pentland, and David Lazer. Inferring friendship network structure by using mobile phone data. *P NATL ACAD SCI USA*, 2009.

[45] Hariton Efstathiades, Demetris Antoniades, George Pallis, and Marios D. Dikaiakos. Identification of key locations based on online social network activity. In *ASONAM*, pages 218–225. ACM, 2015.

[46] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, 1996.

[47] Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). *Official Journal of the European Union*, 2016.

[48] Kassem Fawaz and Kang G Shin. Location privacy protection for smartphone users. In *CCS*, 2014.

[49] Julien Freudiger, Reza Shokri, and Jean-Pierre Hubaux. Evaluating the privacy risk of location-based services. In *FC*, 2011.

[50] Sébastien Gambs, Marc-Olivier Killijian, and Miguel Núñez del Prado Cortez. Show me how you move and i will tell you who you are. In *SPRINGL*, 2010.

[51] Sébastien Gambs, Marc-Olivier Killijian, and Miguel Núñez del Prado Cortez. Next place prediction using mobility markov chains. In *MPM*, 2012.

[52] Sébastien Gambs, Marc-Olivier Killijian, and Miguel Núñez del Prado Cortez. De-anonymization attack on geolocated data. *TRUSTCOM*, 2013.

[53] Gabriel Ghinita. Privacy for location-based services. *Synthesis Lectures on Information Security, Privacy, & Trust*, 2013.

[54] Marta C Gonzalez, Cesar A Hidalgo, and Albert-Laszlo Barabasi. Understanding individual human mobility patterns. *Nature*, 2008.

[55] Marco Gruteser and Dirk Grunwald. Anonymous usage of location-based services through spatial and temporal cloaking. In *MobiSys*, 2003.

[56] Nicolas Haderer, Romain Rouvoy, Christophe Ribeiro, and Lionel Seinturier. Apisense: Crowd-sensing made easy. *ERCIM News*, 2013.

[57] Wajih Ul Hassan, Saad Hussain, and Adam Bates. Analysis of privacy protections in fitness tracking social networks-or-you can run, but can you hide? In *USENIX*, 2018.

[58] Min-Oh Heo, Myung-Gu Kang, Byoung-Kwon Lim, Kyu-Baek Hwang, Young-Tack Park, and Byoung-Tak Zhang. Real-time route inference and learning for smartphone users using probabilistic graphical models. *Journal of KIISE*, 2012.

[59] Baik Hoh, Marco Gruteser, Hui Xiong, and Ansaf Alrabady. Enhancing security and privacy in traffic-monitoring systems. *IEEE PERVAS COMPUT*, 2006.

[60] Baik Hoh, Marco Gruteser, Hui Xiong, and Ansaf Alrabady. Preserving privacy in gps traces via uncertainty-aware path cloaking. In *CCS*, 2007.

[61] Leping Huang, Hiroshi Yamane, Kanta Matsuura, and Kaoru Sezaki. Silent cascade: Enhancing location privacy without communication qos degradation. In *SPC*.

[62] Huan Feng Kassem Fawaz and Kang G Shin. Anatomization and protection of mobile apps' location privacy threats. In *USENIX*, 2015.

[63] Youssef Khazbak and Guohong Cao. Deanonymizing mobility traces with co-location information. In *CNS*, 2017.

[64] John Krumm. Inference attacks on location tracks. In *PERVASIVE*, 2007.

[65] John Krumm. A survey of computational location privacy. *PERS UBIQUIT COMPUT*, 2009.

[66] L. Liao, D. Fox and H. Kautz. Learning and inferring transportation routines. *AAAI*, 2004.

[67] Lin Liao, Dieter Fox, and Henry Kautz. Location-based activity recognition. In *NIPS*, 2006.

[68] Changsha Ma and Chang Wen Chen. Nearby friend discovery with geo-indistinguishability to stalkers. *FNC/MobiSPC*, 2014.

[69] Wesley Mathew, Ruben Raposo, and Bruno Martins. Predicting future locations with hidden markov models. In *UbiComp*, 2012.

[70] Alexandra-Mihaela Olteanu, Kévin Huguenin, Reza Shokri, Mathias Humbert, and Jean-Pierre Hubaux. Quantifying interdependent privacy risks with location data. *TMC*, 2017.

[71] Alexandra-Mihaela Olteanu, Mathias Humbert, Kévin Huguenin, and Jean-Pierre Hubaux. The (co-)location sharing game. In *PoPETs*, 2019.

[72] Rebekah Overdorf and Rachel Greenstadt. Blogs, twitter feeds, and reddit comments: Cross-domain authorship attribution. *PoPETs*, 2016(3):155–171, 2016.

[73] Simon Oya, Carmela Troncoso, and Fernando Pérez-González. Back to the drawing board: Revisiting the design of optimal location privacy-preserving mechanisms. In *CCS*, 2017.

[74] Dau Pelleg and Andrew Moore. X-means: Extending k-means with efficient estimation of the number of clusters. In *ICML*, 2000.

[75] Layla Pournajaf, Li Xiong, Vaidy Sunderam, and Xiaofeng Xu. Stac: Spatial task assignment for crowd sensing with cloaked participant locations. In *SIGSPATIAL/GIS*, 2015.

[76] Vincent Primault, Antoine Boutet, Sonia Ben Mokhtar, and Lionel Brunie. The long road to computational location privacy: A survey. *IEEE Commun. Surv. Tutor.*, 2018.

[77] Apostolos Pyrgelis, Carmela Troncoso, and Emiliano De Cristofaro. Knock knock, who's there? membership inference on aggregate location data. *NDSS*, 2018.

[78] Apostolos Pyrgelis, Carmela Troncoso, and Emiliano De Cristofaro. What does the crowd say about you? evaluating aggregation-based location privacy. *PETS*, 2017.

[79] Jing Shi, Rui Zhang, Yunzhong Liu, and Yanchao Zhang. Prisense: privacy-preserving data aggregation in people-centric urban sensing systems. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. IEEE, 2010.

[80] Reza Shokri, Julien Freudiger, Murtuza Jadliwala, and Jean-Pierre Hubaux. A distortion-based metric for location privacy. In *WPES*, 2009.

[81] Reza Shokri, George Theodorakopoulos, Jean-Yves Le Boudec, and Jean-Pierre Hubaux. Quantifying location privacy. In *IEEE S&P*, 2011.

[82] Chaoming Song, Zehui Qu, Nicholas Blumm, and Albert-László Barabási. Limits of predictability in human mobility. *SCIENCE*, 2010.

[83] Fung Global Retail & Technology. Crowdsourcing:seeking the wisdom of crowds. URL: http://www.deborahweinswig.com/wp-content/uploads/2016/07/Crowdsourcing-Report-by-Fung-Global-Retail-Tech-July-12-2016.pdf.

[84] Manolis Terrovitis. Privacy preservation in the dissemination of location data. *SIGKDD Explorations*, 2011.

[85] Devis Tuia, Claudio Persello, and Lorenzo Bruzzone. Domain adaptation for the classification of remote sensing data: An overview of recent advances. *IEEE Geoscience and Remote sensing magazine*, 4(2):41–57, 2016.

[86] Jorim Urner, Dominik Bucher, Jing Yang, and David Jonietz. Assessing the influence of spatio-temporal context for next place prediction using different machine learning approaches. *ISPRS INT GEO-INF*, 2018.

[87] Hongjian Wang, Zhenhui Li, and Wang-Chien Lee. Pgt: Measuring mobility relationship using personal, global and temporal factors. In *ICDM*, 2014.

[88] Yonghui Xiao and Li Xiong. Protecting locations with differential privacy under temporal correlations. In *CCS*, 2015.

[89] Hui Zang and Jean Bolot. Anonymization of location data does not work: A large-scale measurement study. In *MobiCom*, 2011.

A Appendix

A.1 Density Based Clustering (DBSCAN)

The algorithm receives as input all locations (also referred to as points) reported by a user, the minimum required amount of points per cluster, and the maximum allowed distance between the cluster's points. It outputs a label for every point, indicating to which cluster it belongs, or if it has been labeled as noise.

DBSCAN starts by randomly selecting a point c. Then, it finds all points p that are in distance ε from this point. Then, from the points p reachable from the first point, it tries to find more points q where q are reachable directly from p but not from c. If at the end of this procedure the minimum points have not been reached, it moves to another random point and starts all over again. In order to use our locations which are in latitudes and longitude, we converted the distance ε to radians first. Moreover, we used a ball tree data structure to speed up the neighbors queries.

A.2 Geo-Indistinguishability

The noise is drawn by first transforming the location to polar coordinates. Then, the angle is drawn randomly between 0 and 2π while the distance is drawn from

$$C^{-1}(\rho) = -\frac{1}{\varepsilon}(W^{-1}(\frac{\rho-1}{e})+1)$$

with W^{-1} denoting the -1 branch of the Lambert W function. Finally, the generated distance and angle are added to the original location.

A.3 Optimal Remapping

For the optimal remapping technique we follow these steps; For performance reasons, we first round each location to 3 digits, in order to merge nearby locations together. Then, we calculate the probability of each coordinate. Afterwards, we convert all coordinates to a Cartesian system using their distance from the center of the Earth. A useful tutorial on this can be found in [27]. Using the Cartesian coordinates we build a KD-Tree for efficient nearest neighbor calculations. Then, for every location where GeoInd has been applied, we query all nearest neighbors in a region r'. This r' is set to be as the 99% percentile of the distribution that generated the parameter r used in GeoInd. In other words, the user has 99% chance of being remapped somewhere within this distance. For all neighboring points, we compute the posterior and then, we calculate the geometric median of those coordinates using the iterative Weiszfeld's algorithm. The geometric median minimizes the average Euclidean distance and hence, returning us the new, optimal (in terms of utility as privacy should remain the same) location.

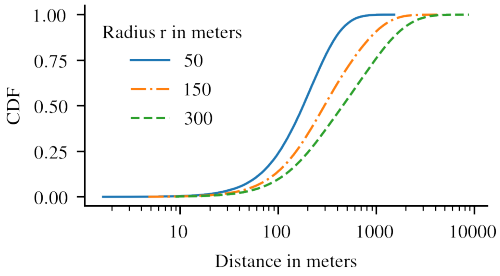


Figure 10: GeoInd noise magnitude for different radius ($l = \ln(1.6)$).

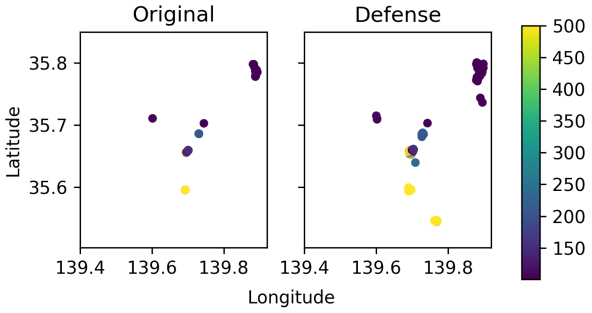


Figure 11: Safecast: Hotspot detection for areas with at least 100 cpm. The presented defense is GeoInd with 50m parameter.

A.4 Defenses evaluation

We include three more figures to complement the defense evaluation:

- Figure 10 portrays the CDF of the noise added by GeoInd. This noise is added on all GeoInd variants (Optimal Remapping and Release-GeoInd) and it is controlled by either the radius (r) or the privacy parameter (l).
- Figure 11 illustrates the hotspot detection results when GeoInd 50m is used. Even a slight addition of noise spreads the locations, not allowing Safecast to accurately detect elevated radiation regions.
- In Figure 12 we present the privacy gain results for each of the defense mechanisms for the whole Safecast dataset.

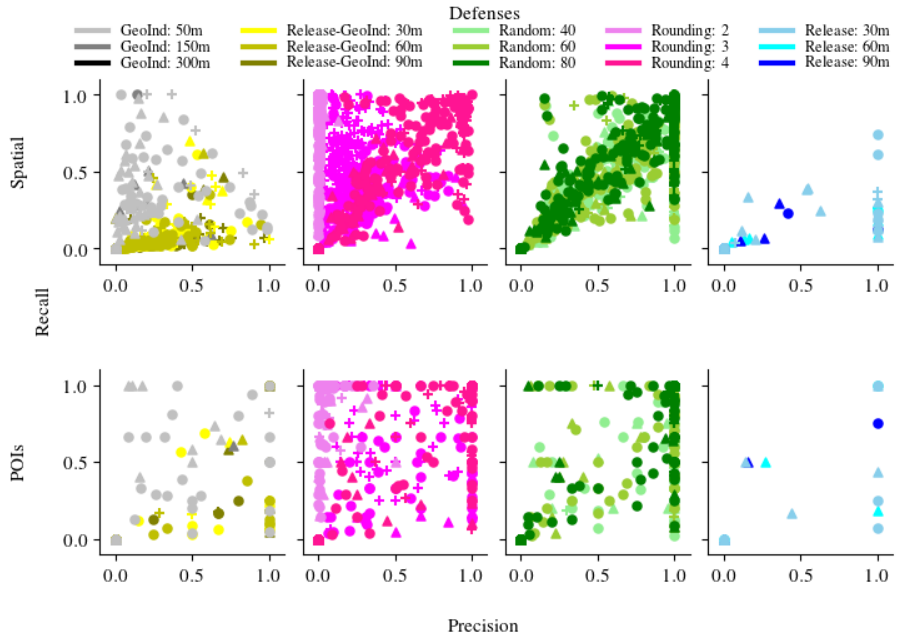


Figure 12: Safecast: Privacy gain for each of the defense mechanism (whole dataset).

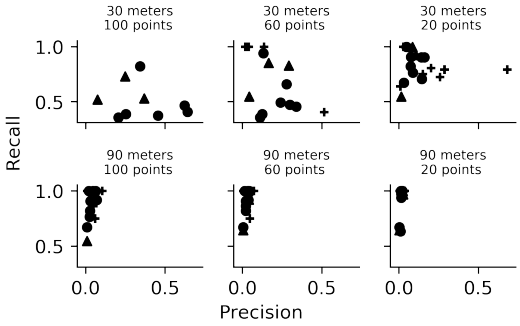


Figure 13: Precision and recall vs. clustering parameters for GeoInd ($r = 50$m) in Tokyo.

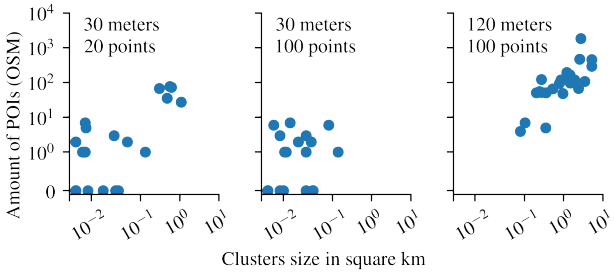


Figure 14: Clusters' size and amount of POIs per cluster vs. clustering parameters with GeoInd ($r = 50$m) in Tokyo.

A.5 Experimental results

Adjusting the clustering parameters. We now study the influence of the DBSCAN clustering parameters on our results. We show the difference in precision and recall for GeoInd (r=50 meters) when we vary both the maximum distance and the minimum number of point per cluster in Figure 13. As we increase the maximum distance between points and decrease the minimum required points per cluster, the results concentrate on the upper left corner of the diagram. This is because as the parameters become 'looser', the resulting clusters grow in size increasing recall (more likelihood of covering all users' original clusters) but reducing precision due to many false positives. Furthermore, increasing the cluster size increases the adversary's cost, as the clusters contain a larger number of POIs (Figure 14) which requires more filtering and increases the probability of having false positives.

Utility-Optimized Local Differential Privacy Mechanisms for Distribution Estimation *

Takao Murakami
AIST

Yusuke Kawamoto
AIST

Abstract

LDP (Local Differential Privacy) has been widely studied to estimate statistics of personal data (e.g., distribution underlying the data) while protecting users' privacy. Although LDP does not require a trusted third party, it regards all personal data equally sensitive, which causes excessive obfuscation hence the loss of utility. In this paper, we introduce the notion of *ULDP (Utility-optimized LDP)*, which provides a privacy guarantee equivalent to LDP only for sensitive data. We first consider the setting where all users use the same obfuscation mechanism, and propose two mechanisms providing ULDP: *utility-optimized randomized response* and *utility-optimized RAPPOR*. We then consider the setting where the distinction between sensitive and non-sensitive data can be different from user to user. For this setting, we propose a *personalized ULDP mechanism with semantic tags* to estimate the distribution of personal data with high utility while keeping secret what is sensitive for each user. We show theoretically and experimentally that our mechanisms provide much higher utility than the existing LDP mechanisms when there are a lot of non-sensitive data. We also show that when most of the data are non-sensitive, our mechanisms even provide almost the same utility as non-private mechanisms in the low privacy regime.

1 Introduction

DP (Differential Privacy) [21,22] is becoming a gold standard for data privacy; it enables big data analysis while protecting users' privacy against adversaries with arbitrary background knowledge. According to the underlying architecture, DP can be categorized into the one in the *centralized model* and the one in the *local model* [22]. In the centralized model, a "trusted" database administrator, who can access to all users' personal data, obfuscates the data (e.g., by adding noise, generalization) before providing them to a (possibly malicious) data analyst. Although DP was extensively studied for the centralized model at the beginning, the original personal data in this model can be leaked from the database by illegal access or internal fraud. This issue is critical in recent years, because the number of data breach incidents is increasing [15].

The local model does not require a "trusted" administrator, and therefore does not suffer from the data leakage issue explained above. In this model, each user obfuscates her personal data by herself, and sends the obfuscated data to a data collector (or data analyst). Based on the obfuscated data, the data collector can estimate some statistics (e.g., histogram, heavy hitters [45]) of the personal data. DP in the local model, which is called *LDP* (*Local Differential Privacy*) [19], has recently attracted much attention in the academic field [5, 12, 24, 29, 30, 39, 43, 45, 46, 50, 56], and has also been adopted by industry [16, 23, 49].

However, LDP mechanisms regard all personal data as equally sensitive, and leave a lot of room for increasing data utility. For example, consider questionnaires such as: "Have you ever cheated in an exam?" and "Were you with a prostitute in the last month?" [11]. Obviously, "Yes" is a sensitive response to these questionnaires, whereas "No" is not sensitive. A RR (Randomized Response) method proposed by Mangat [37] utilizes this fact. Specifically, it reports "Yes" or "No" as follows: if the true answer is "Yes", always report "Yes"; otherwise, report "Yes" and "No" with probability p and $1-p$, respectively. Since the reported answer "Yes" may come from both the true answers "Yes" and "No", the confidentiality of the user reporting "Yes" is not violated. Moreover, since the reported answer "No" is always come from the true answer "No", the data collector can estimate a distribution of true answers with higher accuracy than Warner's RR [52], which simply flips "Yes" and "No" with probability p. However, Mangat's RR does not provide LDP, since LDP regards both "Yes" and "No" as equally sensitive.

There are a lot of "non-sensitive" data for other types of data. For example, locations such as hospitals and home can be sensitive, whereas visited sightseeing places, restaurants, and coffee shops are non-sensitive for many users. Divorced people may want to keep their divorce secret, while the oth-

*This study was supported by JSPS KAKENHI JP19H04113, JP17K12667, and by Inria under the project LOGIS.

ers may not care about their marital status. The distinction between sensitive and non-sensitive data can also be different from user to user (e.g., home address is different from user to user; some people might want to keep secret even the sightseeing places). To explain more about this issue, we briefly review related work on LDP and variants of DP.

Related work. Since Dwork [21] introduced DP, a number of its variants have been studied to provide different types of privacy guarantees; e.g., LDP [19], d-privacy [8], Pufferfish privacy [32], dependent DP [36], Bayesian DP [53], mutual-information DP [14], Rényi DP [38], and distribution privacy [31]. In particular, LDP [19] has been widely studied in the literature. For example, Erlingsson *et al.* [23] proposed the RAPPOR as an obfuscation mechanism providing LDP, and implemented it in Google Chrome browser. Kairouz *et al.* [29] showed that under the l_1 and l_2 losses, the randomized response (generalized to multiple alphabets) and RAPPOR are order optimal among all LDP mechanisms in the low and high privacy regimes, respectively. Wang *et al.* [51] generalized the RAPPOR and a random projection-based method [6], and found parameters that minimize the variance of the estimate.

Some studies also attempted to address the non-uniformity of privacy requirements among records (rows) or among items (columns) in the centralized DP: Personalized DP [28], Heterogeneous DP [3], and One-sided DP [17]. However, obfuscation mechanisms that address the non-uniformity among input values in the "local" DP have not been studied, to our knowledge. In this paper, we show that data utility can be significantly increased by designing such local mechanisms.

Our contributions. The goal of this paper is to design obfuscation mechanisms in the local model that achieve high data utility while providing DP for sensitive data. To achieve this, we introduce the notion of *ULDP (Utility-optimized LDP)*, which provides a privacy guarantee equivalent to LDP only for sensitive data, and obfuscation mechanisms providing ULDP. As a task for the data collector, we consider *discrete distribution estimation* [2, 23, 24, 27, 29, 39, 46, 56], where personal data take discrete values. Our contributions are as follows:

- We first consider the setting in which all users use the same obfuscation mechanism, and propose two ULDP mechanisms: *utility-optimized RR* and *utility-optimized RAPPOR*. We prove that when there are a lot of non-sensitive data, our mechanisms provide much higher utility than two state-of-the-art LDP mechanisms: the RR (for multiple alphabets) [29, 30] and RAPPOR [23]. We also prove that when most of the data are non-sensitive, our mechanisms even provide almost the same utility as a non-private mechanism that does not obfuscate the personal data in the low privacy regime where the privacy budget is $\varepsilon = \ln|\mathcal{X}|$ for a set $\mathcal{X}$ of personal data.

- We then consider the setting in which the distinction between sensitive and non-sensitive data can be different from user to user, and propose a *PUM (Personalized ULDP Mechanism) with semantic tags*. The PUM keeps secret what is sensitive for each user, while enabling the data collector to estimate a distribution using some background knowledge about the distribution conditioned on each tag (e.g., geographic distributions of homes). We also theoretically analyze the data utility of the PUM.

- We finally show that our mechanisms are very promising in terms of utility using two large-scale datasets.

The proofs of all statements in the paper are given in the extended version of the paper [40].

Cautions and limitations. Although ULDP is meant to protect sensitive data, there are some cautions and limitations.

First, we assume that each user sends a single datum and that each user's personal data is independent (see Section 2.1). This is reasonable for a variety of personal data (e.g., locations, age, sex, marital status), where each user's data is irrelevant to most others' one. However, for some types of personal data (e.g., flu status [48]), each user can be highly influenced by others. There might also be a correlation between sensitive data and non-sensitive data when a user sends multiple data (on a related note, non-sensitive attributes may lead to re-identification of a record [41]). A possible solution to these problems would be to incorporate ULDP with *Pufferfish privacy* [32, 48], which is used to protect correlated data. We leave this as future work (see Section 7 for discussions on the case of multiple data per user and the correlation issue).

We focus on a scenario in which it is easy for users to decide what is sensitive (e.g., cheating experience, location of home). However, there is also a scenario in which users do not know what is sensitive. For the latter scenario, we cannot use ULDP but can simply apply LDP.

Apart from the sensitive/non-sensitive data issue, there are scenarios in which ULDP does not cover. For example, ULDP does not protect users who have a sensitivity about "information disclosure" itself (i.e., those who will not disclose any information). We assume that users have consented to information disclosure. To collect as much data as possible, we can provide an incentive for the information disclosure; e.g., provide a reward or point-of-interest (POI) information nearby a reported location. We also assume that the data collector obtains a consensus from users before providing reported data to third parties. Note that these cautions are common to LDP.

There might also be a risk of discrimination; e.g., the data collector might discriminate against all users that provide a yes-answer, and have no qualms about small false positives. False positives decrease with increase in ε. We note that LDP also suffer from this attack; the false positive probability is the same for both ULDP and LDP with the same ε.

In summary, ULDP provides a privacy guarantee equivalent to LDP for sensitive data under the assumption of the data independence. We consider our work as a building-block of broader DP approaches or the basis for further development.

2 Preliminaries

2.1 Notations

Let $\mathbb{R}_{\geq 0}$ be the set of non-negative real numbers. Let n be the number of users, $[n] = \{1, 2, \ldots, n\}$, $\mathcal{X}$ (resp. $\mathcal{Y}$) be a finite set of personal (resp. obfuscated) data. We assume continuous data are discretized into bins in advance (e.g., a location map is divided into some regions). We use the superscript "(i)" to represent the i-th user. Let $X^{(i)}$ (resp. $Y^{(i)}$) be a random variable representing personal (resp. obfuscated) data of the i-th user. The i-th user obfuscates her personal data $X^{(i)}$ via her obfuscation mechanism $\mathbf{Q}^{(i)}$, which maps $x \in \mathcal{X}$ to $y \in \mathcal{Y}$ with probability $\mathbf{Q}^{(i)}(y|x)$, and sends the obfuscated data $Y^{(i)}$ to a data collector. Here we assume that each user sends a single datum. We discuss the case of multiple data in Section 7.

We divide personal data into two types: *sensitive data* and *non-sensitive data*. Let $\mathcal{X}_S \subseteq \mathcal{X}$ be a set of sensitive data common to all users, and $\mathcal{X}_N = \mathcal{X} \setminus \mathcal{X}_S$ be the remaining personal data. Examples of such "common" sensitive data $x \in \mathcal{X}_S$ are the regions including public sensitive locations (e.g., hospitals) and obviously sensitive responses to questionnaires described in Section 1[1].

Furthermore, let $\mathcal{X}_S^{(i)} \subseteq \mathcal{X}_N$ ($i \in [n]$) be a set of sensitive data specific to the i-th user (here we do not include $\mathcal{X}_S$ into $\mathcal{X}_S^{(i)}$ because $\mathcal{X}_S$ is protected for all users in our mechanisms). $\mathcal{X}_S^{(i)}$ is a set of personal data that is possibly non-sensitive for many users but sensitive for the i-th user. Examples of such "user-specific" sensitive data $x \in \mathcal{X}_S^{(i)}$ are the regions including private locations such as their home and workplace. (Note that the majority of working population can be uniquely identified from their home/workplace location pairs [25].)

In Sections 3 and 4, we consider the case where all users divide $\mathcal{X}$ into the same sets of sensitive data and of non-sensitive data, i.e., $\mathcal{X}_S^{(1)} = \cdots = \mathcal{X}_S^{(n)} = \emptyset$, and use the same obfuscation mechanism $\mathbf{Q}$ (i.e., $\mathbf{Q} = \mathbf{Q}^{(1)} = \cdots = \mathbf{Q}^{(n)}$). In Section 5, we consider a general setting that can deal with the user-specific sensitive data $\mathcal{X}_S^{(i)}$ and user-specific mechanisms $\mathbf{Q}^{(i)}$. We call the former case a *common-mechanism scenario* and the latter a *personalized-mechanism scenario*.

We assume that each user's personal data $X^{(i)}$ is independently and identically distributed (i.i.d.) with a probability distribution $\mathbf{p}$, which generates $x \in \mathcal{X}$ with probability $\mathbf{p}(x)$. Let $\mathbf{X} = (X^{(1)}, \cdots, X^{(n)})$ and $\mathbf{Y} = (Y^{(1)}, \cdots, Y^{(n)})$ be tuples of all personal data and all obfuscated data, respectively. The data collector estimates $\mathbf{p}$ from $\mathbf{Y}$ by a method described in Section 2.5. We denote by $\hat{\mathbf{p}}$ the estimate of $\mathbf{p}$. We further denote by $\mathcal{C}$ the probability simplex; i.e., $\mathcal{C} = \{\mathbf{p} | \sum_{x \in \mathcal{X}} \mathbf{p}(x) = 1, \mathbf{p}(x) \geq 0 \text{ for any } x \in \mathcal{X}\}$.

[1] Note that these data might be sensitive for many/most users but not for all in practice (e.g., some people might not care about their cheating experience). However, we can regard these data as sensitive for all users (i.e., be on the safe side) by allowing a small loss of data utility.

2.2 Privacy Measures

LDP (Local Differential Privacy) [19] is defined as follows:

Definition 1 (ε-LDP). *Let $\varepsilon \in \mathbb{R}_{\geq 0}$. An obfuscation mechanism $\mathbf{Q}$ from $\mathcal{X}$ to $\mathcal{Y}$ provides ε-LDP if for any $x, x' \in \mathcal{X}$ and any $y \in \mathcal{Y}$,*

$$\mathbf{Q}(y|x) \leq e^{\varepsilon} \mathbf{Q}(y|x'). \tag{1}$$

LDP guarantees that an adversary who has observed y cannot determine, for any pair of x and x', whether it is come from x or x' with a certain degree of confidence. As the privacy budget ε approaches to 0, all of the data in $\mathcal{X}$ become almost equally likely. Thus, a user's privacy is strongly protected when ε is small.

2.3 Utility Measures

In this paper, we use the l_1 loss (i.e., absolute error) and the l_2 loss (i.e., squared error) as utility measures. Let l_1 (resp. l_2^2) be the l_1 (resp. l_2) loss function, which maps the estimate $\hat{\mathbf{p}}$ and the true distribution $\mathbf{p}$ to the loss; i.e., $l_1(\hat{\mathbf{p}}, \mathbf{p}) = \sum_{x \in \mathcal{X}} |\hat{\mathbf{p}}(x) - \mathbf{p}(x)|$, $l_2^2(\hat{\mathbf{p}}, \mathbf{p}) = \sum_{x \in \mathcal{X}} (\hat{\mathbf{p}}(x) - \mathbf{p}(x))^2$. It should be noted that $\mathbf{X}$ is generated from $\mathbf{p}$ and $\mathbf{Y}$ is generated from $\mathbf{X}$ using $\mathbf{Q}^{(1)}, \cdots, \mathbf{Q}^{(n)}$. Since $\hat{\mathbf{p}}$ is computed from $\mathbf{Y}$, both the l_1 and l_2 losses depend on $\mathbf{Y}$.

In our theoretical analysis in Sections 4 and 5, we take the expectation of the l_1 loss over all possible realizations of $\mathbf{Y}$. In our experiments in Section 6, we replace the expectation of the l_1 loss with the sample mean over multiple realizations of $\mathbf{Y}$ and divide it by 2 to evaluate the TV (Total Variation). In Appendix C, we also show that the l_2 loss has similar results to the ones in Sections 4 and 6 by evaluating the expectation of the l_2 loss and the MSE (Mean Squared Error), respectively.

2.4 Obfuscation Mechanisms

We describe the RR (Randomized Response) [29, 30] and a generalized version of the RAPPOR [51] as follows.

Randomized response. The RR for $|\mathcal{X}|$-ary alphabets was studied in [29, 30]. Its output range is identical to the input domain; i.e., $\mathcal{X} = \mathcal{Y}$.

Formally, given $\varepsilon \in \mathbb{R}_{\geq 0}$, the *$\varepsilon$-RR* is an obfuscation mechanism that maps x to y with the probability:

$$\mathbf{Q}_{RR}(y|x) = \begin{cases} \frac{e^{\varepsilon}}{|\mathcal{X}| + e^{\varepsilon} - 1} & (\text{if } y = x) \\ \frac{1}{|\mathcal{X}| + e^{\varepsilon} - 1} & (\text{otherwise}). \end{cases} \tag{2}$$

It is easy to check by (1) and (2) that $\mathbf{Q}_{RR}$ provides ε-LDP.

Generalized RAPPOR. The RAPPOR (Randomized Aggregatable Privacy-Preserving Ordinal Response) [23] is an obfuscation mechanism implemented in Google Chrome browser. Wang *et al.* [51] extended its simplest configuration called the basic one-time RAPPOR by generalizing two

probabilities in perturbation. Here we call it the *generalized RAPPOR* and describe its algorithm in detail.

The generalized RAPPOR is an obfuscation mechanism with the input alphabet $\mathcal{X} = \{x_1, x_2, \cdots, x_{|\mathcal{X}|}\}$ and the output alphabet $\mathcal{Y} = \{0,1\}^{|\mathcal{X}|}$. It first deterministically maps $x_i \in \mathcal{X}$ to $e_i \in \{0,1\}^{|\mathcal{X}|}$, where e_i is the i-th standard basis vector. It then probabilistically flips each bit of e_i to obtain obfuscated data $y = (y_1, y_2, \cdots, y_{|\mathcal{X}|}) \in \{0,1\}^{|\mathcal{X}|}$, where $y_i \in \{0,1\}$ is the i-th element of y. Wang *et al.* [51] compute ε from two parameters $\theta \in [0,1]$ (representing the probability of keeping 1 unchanged) and $\psi \in [0,1]$ (representing the probability of flipping 0 into 1). In this paper, we compute ψ from two parameters θ and ε.

Specifically, given $\theta \in [0,1]$ and $\varepsilon \in \mathbb{R}_{\geq 0}$, the (θ, ε)-*generalized RAPPOR* maps x_i to y with the probability:

$$\mathbf{Q}_{RAP}(y|x_i) = \prod_{1 \leq j \leq |\mathcal{X}|} \Pr(y_j|x_i),$$

where $\Pr(y_j|x_i) = \theta$ if $i = j$ and $y_j = 1$, and $\Pr(y_j|x_i) = 1-\theta$ if $i = j$ and $y_j = 0$, and $\Pr(y_j|x_i) = \psi = \frac{\theta}{(1-\theta)e^{\varepsilon}+\theta}$ if $i \neq j$ and $y_j = 1$, and $\Pr(y_j|x_i) = 1-\psi$ otherwise. The basic one-time RAPPOR [23] is a special case of the generalized RAPPOR where $\theta = \frac{e^{\varepsilon/2}}{e^{\varepsilon/2}+1}$. $\mathbf{Q}_{RAP}$ also provides ε-LDP.

2.5 Distribution Estimation Methods

Here we explain the empirical estimation method [2, 27, 29] and the EM reconstruction method [1, 2]. Both of them assume that the data collector knows the obfuscation mechanism $\mathbf{Q}$ used to generate $\mathbf{Y}$ from $\mathbf{X}$.

Empirical estimation method. The empirical estimation method [2, 27, 29] computes an empirical estimate $\hat{\mathbf{p}}$ of $\mathbf{p}$ using an empirical distribution $\hat{\mathbf{m}}$ of the obfuscated data $\mathbf{Y}$. Note that $\hat{\mathbf{p}}$, $\hat{\mathbf{m}}$, and $\mathbf{Q}$ can be represented as an $|\mathcal{X}|$-dimensional vector, $|\mathcal{Y}|$-dimensional vector, and $|\mathcal{X}| \times |\mathcal{Y}|$ matrix, respectively. They have the following equation:

$$\hat{\mathbf{p}}\mathbf{Q} = \hat{\mathbf{m}}. \tag{3}$$

The empirical estimation method computes $\hat{\mathbf{p}}$ by solving (3).

Let $\mathbf{m}$ be the true distribution of obfuscated data; i.e., $\mathbf{m} = \mathbf{p}\mathbf{Q}$. As the number of users n increases, the empirical distribution $\hat{\mathbf{m}}$ converges to $\mathbf{m}$. Therefore, the empirical estimate $\hat{\mathbf{p}}$ also converges to $\mathbf{p}$. However, when the number of users n is small, many elements in $\hat{\mathbf{p}}$ can be negative. To address this issue, the studies in [23, 51] kept only estimates above a significance threshold determined via Bonferroni correction, and discarded the remaining estimates.

EM reconstruction method. The EM (Expectation-Maximization) reconstruction method [1, 2] (also called the iterative Bayesian technique [2]) regards $\mathbf{X}$ as a hidden variable and estimates $\mathbf{p}$ from $\mathbf{Y}$ using the EM algorithm [26] (for details of the algorithm, see [1, 2]). Let $\hat{\mathbf{p}}_{EM}$ be an estimate of $\mathbf{p}$ by the EM reconstruction method. The feature of this algorithm is that $\hat{\mathbf{p}}_{EM}$ is equal to the maximum likelihood estimate in the probability simplex $\mathcal{C}$ (see [1] for the proof). Since this property holds irrespective of the number of users n, the elements in $\hat{\mathbf{p}}_{EM}$ are always non-negative.

In this paper, our theoretical analysis uses the empirical estimation method for simplicity, while our experiments use the empirical estimation method, the one with the significance threshold, and the EM reconstruction method.

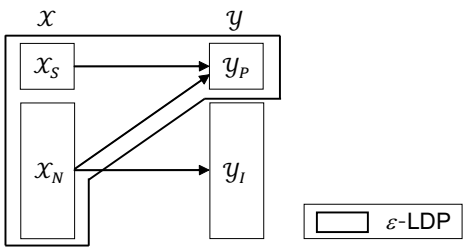


Figure 1: Overview of ULDP. It has no transitions from $\mathcal{X}_S$ to $\mathcal{Y}_I$, and every output in $\mathcal{Y}_I$ reveals the corresponding input in $\mathcal{X}_N$. It also provides ε-LDP for $\mathcal{Y}_P$.

3 Utility-Optimized LDP (ULDP)

In this section, we focus on the common-mechanism scenario (outlined in Section 2.1) and introduce ULDP (Utility-optimized Local Differential Privacy), which provides a privacy guarantee equivalent to ε-LDP only for sensitive data. Section 3.1 provides the definition of ULDP. Section 3.2 shows some theoretical properties of ULDP.

3.1 Definition

Figure 1 shows an overview of ULDP. An obfuscation mechanism providing ULDP, which we call the utility-optimized mechanism, divides obfuscated data into *protected data* and *invertible data*. Let $\mathcal{Y}_P$ be a set of protected data, and $\mathcal{Y}_I = \mathcal{Y} \setminus \mathcal{Y}_P$ be a set of invertible data.

The feature of the utility-optimized mechanism is that it maps sensitive data $x \in \mathcal{X}_S$ to only protected data $y \in \mathcal{Y}_P$. In other words, *it restricts the output set, given the input* $x \in \mathcal{X}_S$, *to* $\mathcal{Y}_P$. Then it provides ε-LDP for $\mathcal{Y}_P$; i.e., $\mathbf{Q}(y|x) \leq e^{\varepsilon}\mathbf{Q}(y|x')$ for any $x, x' \in \mathcal{X}$ and any $y \in \mathcal{Y}_P$. By this property, a privacy guarantee equivalent to ε-LDP is provided for any sensitive data $x \in \mathcal{X}_S$, since the output set corresponding to $\mathcal{X}_S$ is restricted to $\mathcal{Y}_P$. In addition, every output in $\mathcal{Y}_I$ reveals the corresponding input in $\mathcal{X}_N$ (as in Mangat's randomized response [37]) to optimize the estimation accuracy.

We now formally define ULDP and the utility-optimized mechanism:

Definition 2 ($(\mathcal{X}_S, \mathcal{Y}_P, \varepsilon)$-ULDP)**.** *Given* $\mathcal{X}_S \subseteq \mathcal{X}$, $\mathcal{Y}_P \subseteq \mathcal{Y}$, *and* $\varepsilon \in \mathbb{R}_{\geq 0}$, *an obfuscation mechanism* $\mathbf{Q}$ *from* $\mathcal{X}$ *to* $\mathcal{Y}$ *provides* $(\mathcal{X}_S, \mathcal{Y}_P, \varepsilon)$*-ULDP if it satisfies the following properties:*

1. *For any $y \in \mathcal{Y}_I$, there exists an $x \in \mathcal{X}_N$ such that*

$$\mathbf{Q}(y|x) > 0 \text{ and } \mathbf{Q}(y|x') = 0 \text{ for any } x' \neq x. \quad (4)$$

2. *For any $x, x' \in \mathcal{X}$ and any $y \in \mathcal{Y}_P$,*

$$\mathbf{Q}(y|x) \leq e^{\varepsilon}\mathbf{Q}(y|x'). \quad (5)$$

We refer to an obfuscation mechanism $\mathbf{Q}$ providing $(\mathcal{X}_S, \mathcal{Y}_P, \varepsilon)$-ULDP as the $(\mathcal{X}_S, \mathcal{Y}_P, \varepsilon)$-utility-optimized mechanism.

Example. For an intuitive understanding of Definition 2, we show that Mangat's randomized response [37] provides $(\mathcal{X}_S, \mathcal{Y}_P, \varepsilon)$-ULDP. As described in Section 1, this mechanism considers binary alphabets (i.e., $\mathcal{X} = \mathcal{Y} = \{0,1\}$), and regards the value 1 as sensitive (i.e., $\mathcal{X}_S = \mathcal{Y}_P = \{1\}$). If the input value is 1, it always reports 1 as output. Otherwise, it reports 1 and 0 with probability p and $1-p$, respectively. Obviously, this mechanism does not provide ε-LDP for any $\varepsilon \in [0, \infty)$. However, it provides $(\mathcal{X}_S, \mathcal{Y}_P, \ln \frac{1}{p})$-ULDP.

$(\mathcal{X}_S, \mathcal{Y}_P, \varepsilon)$-ULDP provides a privacy guarantee equivalent to ε-LDP for any sensitive data $x \in \mathcal{X}_S$, as explained above. On the other hand, no privacy guarantees are provided for non-sensitive data $x \in \mathcal{X}_N$ because every output in $\mathcal{Y}_I$ reveals the corresponding input in $\mathcal{X}_N$. However, it does not matter since non-sensitive data need not be protected. Protecting only minimum necessary data is the key to achieving locally private distribution estimation with high data utility.

We can apply any ε-LDP mechanism to the sensitive data in $\mathcal{X}_S$ to provide $(\mathcal{X}_S, \mathcal{Y}_P, \varepsilon)$-ULDP as a whole. In Sections 4.1 and 4.2, we propose a utility-optimized RR (Randomized Response) and utility-optimized RAPPOR, which apply the ε-RR and ε-RAPPOR, respectively, to the sensitive data $\mathcal{X}_S$.

In Appendix B, we also consider OSLDP (One-sided LDP), a local model version of OSDP introduced in a preprint [17], and explain the reason for using ULDP in this paper.

It might be better to generalize ULDP so that different levels of ε can be assigned to different sensitive data. We leave introducing such granularity as future work.

Remark. It should also be noted that the data collector needs to know $\mathbf{Q}$ to estimate $\mathbf{p}$ from $\mathbf{Y}$ (as described in Section 2.5), and that the $(\mathcal{X}_S, \mathcal{Y}_P, \varepsilon)$-utility-optimized mechanism $\mathbf{Q}$ itself includes the information on *what is sensitive for users* (i.e., the data collector learns whether each $x \in \mathcal{X}$ belongs to $\mathcal{X}_S$ or not by checking the values of $\mathbf{Q}(y|x)$ for all $y \in \mathcal{Y}$). This does not matter in the common-mechanism scenario, since the set $\mathcal{X}_S$ of sensitive data is common to all users (e.g., public hospitals). However, in the personalized-mechanism scenario, the $(\mathcal{X}_S \cup \mathcal{X}_S^{(i)}, \mathcal{Y}_P, \varepsilon)$-utility-optimized mechanism $\mathbf{Q}^{(i)}$, which expands the set $\mathcal{X}_S$ of personal data to $\mathcal{X}_S \cup \mathcal{X}_S^{(i)}$, includes the information on *what is sensitive for the i-th user*. Therefore, the data collector learns whether each $x \in \mathcal{X}_N$ belongs to $\mathcal{X}_S^{(i)}$ or not by checking the values of $\mathbf{Q}^{(i)}(y|x)$ for all $y \in \mathcal{Y}$, despite the fact that the i-th user wants to hide her user-specific sensitive data $\mathcal{X}_S^{(i)}$ (e.g., home, workplace). We address this issue in Section 5.

3.2 Basic Properties of ULDP

Previous work showed some basic properties of differential privacy (or its variant), such as compositionality [22] and immunity to post-processing [22]. We briefly explain theoretical properties of ULDP including the ones above.

Sequential composition. ULDP is preserved under adaptive sequential composition when the composed obfuscation mechanism maps sensitive data to pairs of protected data. Specifically, consider two mechanisms $\mathbf{Q}_0$ from $\mathcal{X}$ to $\mathcal{Y}_0$ and $\mathbf{Q}_1$ from $\mathcal{X}$ to $\mathcal{Y}_1$ such that $\mathbf{Q}_0$ (resp. $\mathbf{Q}_1$) maps sensitive data $x \in \mathcal{X}_S$ to protected data $y_0 \in \mathcal{Y}_{0P}$ (resp. $y_1 \in \mathcal{Y}_{1P}$). Then the sequential composition of $\mathbf{Q}_0$ and $\mathbf{Q}_1$ maps sensitive data $x \in \mathcal{X}_S$ to pairs (y_0, y_1) of protected data ranging over:

$$(\mathcal{Y}_0 \times \mathcal{Y}_1)_P = \{(y_0, y_1) \in \mathcal{Y}_0 \times \mathcal{Y}_1 \mid y_0 \in \mathcal{Y}_{0P} \text{ and } y_1 \in \mathcal{Y}_{1P}\}.$$

Then we obtain the following compositionality.

Proposition 1 (Sequential composition)**.** *Let $\varepsilon_0, \varepsilon_1 \geq 0$. If $\mathbf{Q}_0$ provides $(\mathcal{X}_S, \mathcal{Y}_{0P}, \varepsilon_0)$-ULDP and $\mathbf{Q}_1(y_0)$ provides $(\mathcal{X}_S, \mathcal{Y}_{1P}, \varepsilon_1)$-ULDP for each $y_0 \in \mathcal{Y}_0$, then the sequential composition of $\mathbf{Q}_0$ and $\mathbf{Q}_1$ provides $(\mathcal{X}_S, (\mathcal{Y}_0 \times \mathcal{Y}_1)_P, \varepsilon_0 + \varepsilon_1)$-ULDP.*

For example, if we apply an obfuscation mechanism providing $(\mathcal{X}_S, \mathcal{Y}_P, \varepsilon)$-ULDP for t times, then we obtain $(\mathcal{X}_S, (\mathcal{Y}_P)^t, \varepsilon t)$-ULDP in total (this is derived by repeatedly using Proposition 1).

Post-processing. ULDP is immune to the post-processing by a randomized algorithm that *preserves data types*: protected data or invertible data. Specifically, if a mechanism $\mathbf{Q}_0$ provides $(\mathcal{X}_S, \mathcal{Y}_P, \varepsilon)$-ULDP and a randomized algorithm $\mathbf{Q}_1$ maps protected data over $\mathcal{Y}_P$ (resp. invertible data) to protected data over $\mathcal{Z}_P$ (resp. invertible data), then the composite function $\mathbf{Q}_1 \circ \mathbf{Q}_0$ provides $(\mathcal{X}_S, \mathcal{Z}_P, \varepsilon)$-ULDP.

Note that $\mathbf{Q}_1$ needs to preserve data types for utility; i.e., to make all $y \in \mathcal{Y}_I$ invertible (as in Definition 2) after post-processing. The DP guarantee for $y \in \mathcal{Y}_P$ is preserved by any post-processing algorithm. See Appendix A.1 for details.

Compatibility with LDP. Assume that data collectors A and B adopt a mechanism providing ULDP and a mechanism providing LDP, respectively. In this case, all protected data in the data collector A can be combined with all obfuscated data in the data collector B (i.e., data integration) to perform data analysis under LDP. See Appendix A.2 for details.

Lower bounds on the l_1 and l_2 losses. We present lower bounds on the l_1 and l_2 losses of any ULDP mechanism by using the fact that ULDP provides (5) for any $x, x' \in \mathcal{X}_S$ and any $y \in \mathcal{Y}_P$. Specifically, Duchi *et al.* [20] showed that for $\varepsilon \in [0,1]$, the lower bounds on the l_1 and l_2 losses (minimax rates) of any ε-LDP mechanism can be expressed as $\Theta(\frac{|\mathcal{X}|}{\sqrt{n\varepsilon^2}})$

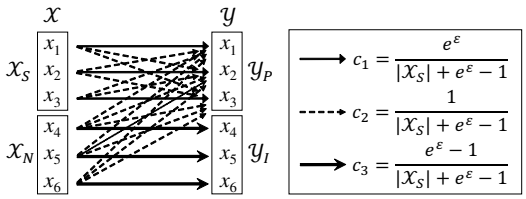


Figure 2: Utility-optimized RR in the case where $\mathcal{X}_S = \mathcal{Y}_P = \{x_1, x_2, x_3\}$ and $\mathcal{X}_N = \mathcal{Y}_I = \{x_4, x_5, x_6\}$.

and $\Theta(\frac{|\mathcal{X}|}{n\varepsilon^2})$, respectively. By directly applying these bounds to $\mathcal{X}_S$ and $\mathcal{Y}_P$, the lower bounds on the l_1 and l_2 losses of any $(\mathcal{X}_S, \mathcal{Y}_P, \varepsilon)$-ULDP mechanisms for $\varepsilon \in [0,1]$ can be expressed as $\Theta(\frac{|\mathcal{X}_S|}{\sqrt{n\varepsilon^2}})$ and $\Theta(\frac{|\mathcal{X}_S|}{n\varepsilon^2})$, respectively. In Section 4.3, we show that our utility-optimized RAPPOR achieves these lower bounds when ε is close to 0 (i.e., high privacy regime).

4 Utility-Optimized Mechanisms

In this section, we focus on the common-mechanism scenario and propose the *utility-optimized RR (Randomized Response)* and *utility-optimized RAPPOR* (Sections 4.1 and 4.2). We then analyze the data utility of these mechanisms (Section 4.3).

4.1 Utility-Optimized Randomized Response

We propose the utility-optimized RR, which is a generalization of Mangat's randomized response [37] to $|\mathcal{X}|$-ary alphabets with $|\mathcal{X}_S|$ sensitive symbols. As with the RR, the output range of the utility-optimized RR is identical to the input domain; i.e., $\mathcal{X} = \mathcal{Y}$. In addition, we divide the output set in the same way as the input set; i.e., $\mathcal{X}_S = \mathcal{Y}_P$, $\mathcal{X}_N = \mathcal{Y}_I$.

Figure 2 shows the utility-optimized RR with $\mathcal{X}_S = \mathcal{Y}_P = \{x_1, x_2, x_3\}$ and $\mathcal{X}_N = \mathcal{Y}_I = \{x_4, x_5, x_6\}$. The utility-optimized RR applies the ε-RR to $\mathcal{X}_S$. It maps $x \in \mathcal{X}_N$ to $y \in \mathcal{Y}_P$ $(= \mathcal{X}_S)$ with the probability $\mathbf{Q}(y|x)$ so that (5) is satisfied, and maps $x \in \mathcal{X}_N$ to itself with the remaining probability. Formally, we define the utility-optimized RR (uRR) as follows:

Definition 3 ($(\mathcal{X}_S, \varepsilon)$-utility-optimized RR)**.** *Let $\mathcal{X}_S \subseteq \mathcal{X}$ and $\varepsilon \in \mathbb{R}_{\geq 0}$. Let $c_1 = \frac{e^\varepsilon}{|\mathcal{X}_S| + e^\varepsilon - 1}$, $c_2 = \frac{1}{|\mathcal{X}_S| + e^\varepsilon - 1}$, and $c_3 = 1 - |\mathcal{X}_S| c_2 = \frac{e^\varepsilon - 1}{|\mathcal{X}_S| + e^\varepsilon - 1}$. Then the $(\mathcal{X}_S, \varepsilon)$-utility-optimized RR (uRR) is an obfuscation mechanism that maps $x \in \mathcal{X}$ to $y \in \mathcal{Y}$ $(= \mathcal{X})$ with the probability $\mathbf{Q}_{uRR}(y|x)$ defined as follows:*

$$\mathbf{Q}_{uRR}(y|x) = \begin{cases} c_1 & (\text{if } x \in \mathcal{X}_S,\ y = x) \\ c_2 & (\text{if } x \in \mathcal{X}_S,\ y \in \mathcal{X}_S \setminus \{x\}) \\ c_2 & (\text{if } x \in \mathcal{X}_N,\ y \in \mathcal{X}_S) \\ c_3 & (\text{if } x \in \mathcal{X}_N,\ y = x) \\ 0 & (\text{otherwise}). \end{cases} \tag{6}$$

Proposition 2. *The $(\mathcal{X}_S, \varepsilon)$-uRR provides $(\mathcal{X}_S, \mathcal{X}_S, \varepsilon)$-ULDP.*

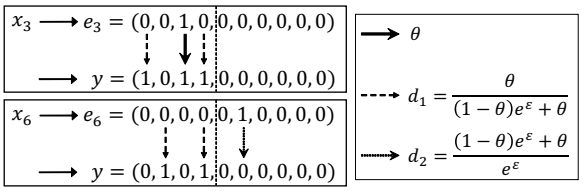


Figure 3: Utility-optimized RAPPOR in the case where $\mathcal{X}_S = \{x_1, \cdots, x_4\}$ and $\mathcal{X}_N = \{x_5, \cdots, x_{10}\}$.

4.2 Utility-Optimized RAPPOR

Next, we propose the utility-optimized RAPPOR with the input alphabet $\mathcal{X} = \{x_1, x_2, \cdots, x_{|\mathcal{X}|}\}$ and the output alphabet $\mathcal{Y} = \{0,1\}^{|\mathcal{X}|}$. Without loss of generality, we assume that $x_1, \cdots, x_{|\mathcal{X}_S|}$ are sensitive and $x_{|\mathcal{X}_S|+1}, \cdots, x_{|\mathcal{X}|}$ are non-sensitive; i.e., $\mathcal{X}_S = \{x_1, \cdots, x_{|\mathcal{X}_S|}\}$, $\mathcal{X}_N = \{x_{|\mathcal{X}_S|+1}, \cdots, x_{|\mathcal{X}|}\}$.

Figure 3 shows the utility-optimized RAPPOR with $\mathcal{X}_S = \{x_1, \cdots, x_4\}$ and $\mathcal{X}_N = \{x_5, \cdots, x_{10}\}$. The utility-optimized RAPPOR first deterministically maps $x_i \in \mathcal{X}$ to the i-th standard basis vector e_i. It should be noted that if x_i is sensitive data (i.e., $x_i \in \mathcal{X}_S$), then the last $|\mathcal{X}_N|$ elements in e_i are always zero (as shown in the upper-left panel of Figure 3). Based on this fact, the utility-optimized RAPPOR regards obfuscated data $y = (y_1, y_2, \ldots, y_{|\mathcal{X}|}) \in \{0,1\}^{|\mathcal{X}|}$ such that $y_{|\mathcal{X}_S|+1} = \cdots = y_{|\mathcal{X}|} = 0$ as protected data; i.e.,

$$\mathcal{Y}_P = \{(y_1, \ldots, y_{|\mathcal{X}_S|}, 0, \cdots, 0) | y_1, \ldots, y_{|\mathcal{X}_S|} \in \{0,1\}\}. \tag{7}$$

Then it applies the (θ, ε)-generalized RAPPOR to $\mathcal{X}_S$, and maps $x \in \mathcal{X}_N$ to $y \in \mathcal{Y}_P$ (as shown in the lower-left panel of Figure 3) with the probability $\mathbf{Q}(y|x)$ so that (5) is satisfied. We formally define the utility-optimized RAPPOR (uRAP):

Definition 4 ($(\mathcal{X}_S, \theta, \varepsilon)$-utility-optimized RAPPOR)**.** *Let $\mathcal{X}_S \subseteq \mathcal{X}$, $\theta \in [0,1]$, and $\varepsilon \in \mathbb{R}_{\geq 0}$. Let $d_1 = \frac{\theta}{(1-\theta)e^\varepsilon + \theta}$, $d_2 = \frac{(1-\theta)e^\varepsilon + \theta}{e^\varepsilon}$. Then the $(\mathcal{X}_S, \theta, \varepsilon)$-utility-optimized RAPPOR (uRAP) is an obfuscation mechanism that maps $x_i \in \mathcal{X}$ to $y \in \mathcal{Y} = \{0,1\}^{|\mathcal{X}|}$ with the probability $\mathbf{Q}_{uRAP}(y|x)$ given by:*

$$\mathbf{Q}_{uRAP}(y|x_i) = \textstyle\prod_{1 \leq j \leq |\mathcal{X}|} \Pr(y_j|x_i), \tag{8}$$

where $\Pr(y_j|x_i)$ is written as follows:

(i) if $1 \leq j \leq |\mathcal{X}_S|$:

$$\Pr(y_j|x_i) = \begin{cases} 1-\theta & (\text{if } i = j,\ y_j = 0) \\ \theta & (\text{if } i = j,\ y_j = 1) \\ 1-d_1 & (\text{if } i \neq j,\ y_j = 0) \\ d_1 & (\text{if } i \neq j,\ y_j = 1). \end{cases} \tag{9}$$

(ii) if $|\mathcal{X}_S| + 1 \leq j \leq |\mathcal{X}|$:

$$\Pr(y_j|x_i) = \begin{cases} d_2 & (\text{if } i = j,\ y_j = 0) \\ 1-d_2 & (\text{if } i = j,\ y_j = 1) \\ 1 & (\text{if } i \neq j,\ y_j = 0) \\ 0 & (\text{if } i \neq j,\ y_j = 1). \end{cases} \tag{10}$$

Proposition 3. *The $(\mathcal{X}_S, \theta, \varepsilon)$-uRAP provides $(\mathcal{X}_S, \mathcal{Y}_P, \varepsilon)$-ULDP, where $\mathcal{Y}_P$ is given by (7).*

Although we used the generalized RAPPOR in $\mathcal{X}_S$ and $\mathcal{Y}_P$ in Definition 4, hereinafter we set $\theta = \frac{e^{\varepsilon/2}}{e^{\varepsilon/2}+1}$ in the same way as the original RAPPOR [23]. There are two reasons for this. First, it achieves "order" optimal data utility among all $(\mathcal{X}_S, \mathcal{Y}_P, \varepsilon)$-ULDP mechanisms in the high privacy regime, as shown in Section 4.3. Second, it maps $x_i \in \mathcal{X}_N$ to $y \in \mathcal{Y}_I$ with probability $1 - d_2 = 1 - e^{-\varepsilon/2}$, which is close to 1 when ε is large (i.e., low privacy regime). Wang *et al.* [51] showed that the generalized RAPPOR with parameter $\theta = \frac{1}{2}$ minimizes the variance of the estimate. However, our uRAP with parameter $\theta = \frac{1}{2}$ maps $x_i \in \mathcal{X}_N$ to $y \in \mathcal{Y}_I$ with probability $1 - d_2 = \frac{e^{\varepsilon}-1}{2e^{\varepsilon}}$ which is less than $1 - e^{-\varepsilon/2}$ for any $\varepsilon > 0$ and is less than $\frac{1}{2}$ even when ε goes to infinity. Thus, our uRAP with $\theta = \frac{e^{\varepsilon/2}}{e^{\varepsilon/2}+1}$ maps $x_i \in \mathcal{X}_N$ to $y \in \mathcal{Y}_I$ with higher probability, and therefore achieves a smaller estimation error over all non-sensitive data. We also consider that an optimal θ for our uRAP is different from the optimal θ ($= \frac{1}{2}$) for the generalized RAPPOR. We leave finding the optimal θ for our uRAP (with respect to the estimation error over all personal data) as future work.

We refer to the $(\mathcal{X}_S, \theta, \varepsilon)$-uRAP with $\theta = \frac{e^{\varepsilon/2}}{e^{\varepsilon/2}+1}$ in shorthand as the $(\mathcal{X}_S, \varepsilon)$-*uRAP*.

4.3 Utility Analysis

We evaluate the l_1 loss of the uRR and uRAP when the empirical estimation method is used for distribution estimation[2]. In particular, we evaluate the l_1 loss when ε is close to 0 (i.e., high privacy regime) and $\varepsilon = \ln|\mathcal{X}|$ (i.e., low privacy regime). Note that ULDP provides a natural interpretation of the latter value of ε. Specifically, it follows from (5) that if $\varepsilon = \ln|\mathcal{X}|$, then for any $x \in \mathcal{X}$, the likelihood that the input data is x is almost equal to the sum of the likelihood that the input data is $x' \neq x$. This is consistent with the fact that the ε-RR with parameter $\varepsilon = \ln|\mathcal{X}|$ sends true data (i.e., $y = x$ in (2)) with probability about 0.5 and false data (i.e., $y \neq x$) with probability about 0.5, and hence provides plausible deniability [29].

uRR in the general case. We begin with the uRR:

Proposition 4 (l_1 loss of the uRR)**.** *Let $\varepsilon \in \mathbb{R}_{\geq 0}$, $u = |\mathcal{X}_S| + e^{\varepsilon} - 1$, $u' = e^{\varepsilon} - 1$, and $v = \frac{u}{u'}$. Then the expected l_1 loss of the $(\mathcal{X}_S, \varepsilon)$-uRR mechanism is given by:*

$$\mathbb{E}[l_1(\hat{\mathbf{p}}, \mathbf{p})] \approx \sqrt{\tfrac{2}{n\pi}} \Bigg(\sum_{x \in \mathcal{X}_S} \sqrt{\big(\mathbf{p}(x) + 1/u'\big)\big(v - \mathbf{p}(x) - 1/u'\big)} + \sum_{x \in \mathcal{X}_N} \sqrt{\mathbf{p}(x)\big(v - \mathbf{p}(x)\big)} \Bigg), \tag{11}$$

where $f(n) \approx g(n)$ represents $\lim_{n\to\infty} f(n)/g(n) = 1$.

Let $\mathbf{p}_{U_N}$ be the uniform distribution over $\mathcal{X}_N$; i.e., for any $x \in \mathcal{X}_S$, $\mathbf{p}_{U_N}(x) = 0$, and for any $x \in \mathcal{X}_N$, $\mathbf{p}_{U_N}(x) = \frac{1}{|\mathcal{X}_N|}$. Symmetrically, let $\mathbf{p}_{U_S}$ be the uniform distribution over $\mathcal{X}_S$.

For $0 < \varepsilon < \ln(|\mathcal{X}_N| + 1)$, the l_1 loss is maximized by $\mathbf{p}_{U_N}$:

Proposition 5. *For any $0 < \varepsilon < \ln(|\mathcal{X}_N| + 1)$ and $|\mathcal{X}_S| \leq |\mathcal{X}_N|$, (11) is maximized by $\mathbf{p}_{U_N}$:*

$$\begin{aligned} &\mathbb{E}[l_1(\hat{\mathbf{p}}, \mathbf{p})] \lesssim \mathbb{E}[l_1(\hat{\mathbf{p}}, \mathbf{p}_{U_N})] \\ &= \sqrt{\tfrac{2}{n\pi}} \left(\frac{|\mathcal{X}_S|\sqrt{|\mathcal{X}_S| + e^{\varepsilon} - 2}}{e^{\varepsilon} - 1} + \sqrt{\frac{|\mathcal{X}_S||\mathcal{X}_N|}{e^{\varepsilon} - 1} + |\mathcal{X}_N| - 1} \right), \end{aligned} \tag{12}$$

where $f(n) \lesssim g(n)$ represents $\lim_{n\to\infty} f(n)/g(n) \leq 1$.

For $\varepsilon \geq \ln(|\mathcal{X}_N| + 1)$, the l_1 loss is maximized by a mixture distribution of $\mathbf{p}_{U_N}$ and $\mathbf{p}_{U_S}$:

Proposition 6. *Let $\mathbf{p}^*$ be a distribution over $\mathcal{X}$ defined by:*

$$\mathbf{p}^*(x) = \begin{cases} \frac{1 - |\mathcal{X}_N|/(e^{\varepsilon}-1)}{|\mathcal{X}_S| + |\mathcal{X}_N|} & \text{(if } x \in \mathcal{X}_S\text{)} \\ \frac{1 + |\mathcal{X}_S|/(e^{\varepsilon}-1)}{|\mathcal{X}_S| + |\mathcal{X}_N|} & \text{(otherwise)} \end{cases} \tag{13}$$

Then for any $\varepsilon \geq \ln(|\mathcal{X}_N| + 1)$, (11) is maximized by $\mathbf{p}^*$:

$$\mathbb{E}[l_1(\hat{\mathbf{p}}, \mathbf{p})] \lesssim \mathbb{E}[l_1(\hat{\mathbf{p}}, \mathbf{p}^*)] = \sqrt{\tfrac{2(|\mathcal{X}|-1)}{n\pi}} \cdot \frac{|\mathcal{X}_S| + e^{\varepsilon} - 1}{e^{\varepsilon} - 1}, \tag{14}$$

where $f(n) \lesssim g(n)$ represents $\lim_{n\to\infty} f(n)/g(n) \leq 1$.

Next, we instantiate the l_1 loss in the high and low privacy regimes based on these propositions.

uRR in the high privacy regime. When ε is close to 0, we have $e^{\varepsilon} - 1 \approx \varepsilon$. Thus, the right-hand side of (12) in Proposition 5 can be simplified as follows:

$$\mathbb{E}[l_1(\hat{\mathbf{p}}, \mathbf{p}_{U_N})] \approx \sqrt{\tfrac{2}{n\pi}} \cdot \frac{|\mathcal{X}_S|\sqrt{|\mathcal{X}_S| - 1}}{\varepsilon}. \tag{15}$$

It was shown in [29] that the expected l_1 loss of the ε-RR is at most $\sqrt{\frac{2}{n\pi}} \frac{|\mathcal{X}|\sqrt{|\mathcal{X}|-1}}{\varepsilon}$ when $\varepsilon \approx 0$. The right-hand side of (15) is much smaller than this when $|\mathcal{X}_S| \ll |\mathcal{X}|$. Although both of them are "upper-bounds" of the expected l_1 losses, we show that the total variation of the $(\mathcal{X}_S, \varepsilon)$-uRR is also much smaller than that of the ε-RR when $|\mathcal{X}_S| \ll |\mathcal{X}|$ in Section 6.

uRR in the low privacy regime. When $\varepsilon = \ln|\mathcal{X}|$ and $|\mathcal{X}_S| \ll |\mathcal{X}|$, the right-hand side of (14) in Proposition 6 can be simplified by using $|\mathcal{X}_S|/|\mathcal{X}| \approx 0$:

$$\mathbb{E}[l_1(\hat{\mathbf{p}}, \mathbf{p}^*)] \approx \sqrt{\tfrac{2(|\mathcal{X}|-1)}{n\pi}}.$$

[2] We note that we use the empirical estimation method in the same way as [29], and that it might be possible that other mechanisms have better utility with a different estimation method. However, we emphasize that even with the empirical estimation method, the uRAP achieves the lower bounds on the l_1 and l_2 losses of any ULDP mechanisms when $\varepsilon \approx 0$, and the uRR and uRAP achieve almost the same utility as a non-private mechanism when $\varepsilon = \ln|\mathcal{X}|$ and most of the data are non-sensitive.

It should be noted that the expected l_1 loss of the non-private mechanism, which does not obfuscate the personal data at all, is at most $\sqrt{\frac{2(|\mathcal{X}|-1)}{n\pi}}$ [29]. Thus, when $\varepsilon = \ln|\mathcal{X}|$ and $|\mathcal{X}_S| \ll |\mathcal{X}|$, the $(\mathcal{X}_S,\varepsilon)$-uRR achieves almost the same data utility as the non-private mechanism, whereas the expected l_1 loss of the ε-RR is twice larger than that of the non-private mechanism [29].

uRAP in the general case. We then analyze the uRAP:

Proposition 7 (l_1 loss of the uRAP)**.** *Let* $\varepsilon \in \mathbb{R}_{\geq 0}$, $u' = e^{\varepsilon/2} - 1$, *and* $v_N = \frac{e^{\varepsilon/2}}{e^{\varepsilon/2}-1}$. *The expected* l_1*-loss of the* $(\mathcal{X}_S,\varepsilon)$*-uRAP mechanism is:*

$$\mathbb{E}[l_1(\hat{\mathbf{p}},\mathbf{p})] \approx \sqrt{\frac{2}{n\pi}}\left(\sum_{j=1}^{|\mathcal{X}_S|}\sqrt{\left(\mathbf{p}(x_j)+1/u'\right)\left(v_N-\mathbf{p}(x_j)\right)} + \sum_{j=|\mathcal{X}_S|+1}^{|\mathcal{X}|}\sqrt{\mathbf{p}(x_j)\left(v_N-\mathbf{p}(x_j)\right)}\right), \quad (16)$$

where $f(n) \approx g(n)$ *represents* $\lim_{n\to\infty} f(n)/g(n) = 1$.

When $0 < \varepsilon < 2\ln(\frac{|\mathcal{X}_N|}{2}+1)$, the l_1 loss is maximized by the uniform distribution $\mathbf{p}_{U_N}$ over $\mathcal{X}_N$:

Proposition 8. *For any* $0 < \varepsilon < 2\ln(\frac{|\mathcal{X}_N|}{2}+1)$ *and* $|\mathcal{X}_S| \leq |\mathcal{X}_N|$, *(16) is maximized when* $\mathbf{p} = \mathbf{p}_{U_N}$*:*

$$\mathbb{E}[l_1(\hat{\mathbf{p}},\mathbf{p})] \lesssim \mathbb{E}[l_1(\hat{\mathbf{p}},\mathbf{p}_{U_N})] = \sqrt{\frac{2}{n\pi}}\left(\frac{e^{\varepsilon/4}|\mathcal{X}_S|}{e^{\varepsilon/2}-1} + \sqrt{\frac{e^{\varepsilon/2}|\mathcal{X}_N|}{e^{\varepsilon/2}-1}-1}\right), \quad (17)$$

where $f(n) \lesssim g(n)$ *represents* $\lim_{n\to\infty} f(n)/g(n) \leq 1$.

Note that this proposition covers a wide range of ε. For example, when $|\mathcal{X}_S| \leq |\mathcal{X}_N|$, it covers both the high privacy regime ($\varepsilon \approx 0$) and low privacy regime ($\varepsilon = \ln|\mathcal{X}|$), since $\ln|\mathcal{X}| < 2\ln(\frac{|\mathcal{X}_N|}{2}+1)$. Below we instantiate the l_1 loss in the high and low privacy regimes based on this proposition.

uRAP in the high privacy regime. If ε is close to 0, we have $e^{\varepsilon/2} - 1 \approx \varepsilon/2$. Thus, the right-hand side of (17) in Proposition 8 can be simplified as follows:

$$\mathbb{E}[l_1(\hat{\mathbf{p}},\mathbf{p}_{U_N})] \approx \sqrt{\frac{2}{n\pi}} \cdot \frac{2|\mathcal{X}_S|}{\varepsilon}. \quad (18)$$

It is shown in [29] that the expected l_1 loss of the ε-RAPPOR is at most $\sqrt{\frac{2}{n\pi}} \cdot \frac{2|\mathcal{X}|}{\varepsilon}$ when $\varepsilon \approx 0$. Thus, by (18), the expected l_1 loss of the $(\mathcal{X}_S,\varepsilon)$-uRAP is much smaller than that of the ε-RAPPOR when $|\mathcal{X}_S| \ll |\mathcal{X}|$.

Moreover, by (18), the expected l_1 loss of the $(\mathcal{X}_S,\varepsilon)$-uRAP in the worst case is expressed as $\Theta(\frac{|\mathcal{X}_S|}{\sqrt{n\varepsilon^2}})$ in the high privacy regime. As described in Section 3.2, this is "order" optimal among all $(\mathcal{X}_S,\mathcal{Y}_P,\varepsilon)$-ULDP mechanisms (in Appendix C.1, we also show that the expected l_2 of the $(\mathcal{X}_S,\varepsilon)$-uRAP is expressed as $\Theta(\frac{|\mathcal{X}_S|}{n\varepsilon^2})$).

uRAP in the low privacy regime. If $\varepsilon = \ln|\mathcal{X}|$ and $|\mathcal{X}_S| \ll |\mathcal{X}|^{\frac{3}{4}}$, the right-hand side of (17) can be simplified, using $|\mathcal{X}_S|/|\mathcal{X}|^{\frac{3}{4}} \approx 0$, as follows:

$$\mathbb{E}[l_1(\hat{\mathbf{p}},\mathbf{p}_{U_N})] \approx \sqrt{\frac{2(|\mathcal{X}|-1)}{n\pi}}.$$

Thus, when $\varepsilon = \ln|\mathcal{X}|$ and $|\mathcal{X}_S| \ll |\mathcal{X}|^{\frac{3}{4}}$, the $(\mathcal{X}_S,\varepsilon)$-uRAP also achieves almost the same data utility as the non-private mechanism, whereas the expected l_1 loss of the ε-RAPPOR is $\sqrt{|\mathcal{X}|}$ times larger than that of the non-private mechanism [29].

Summary. In summary, the uRR and uRAP provide much higher utility than the RR and RAPPOR when $|\mathcal{X}_S| \ll |\mathcal{X}|$. Moreover, when $\varepsilon = \ln|\mathcal{X}|$ and $|\mathcal{X}_S| \ll |\mathcal{X}|$ (resp. $|\mathcal{X}_S| \ll |\mathcal{X}|^{\frac{3}{4}}$), the uRR (resp. uRAP) achieves almost the same utility as a non-private mechanism.

5 Personalized ULDP Mechanisms

We now consider the personalized-mechanism scenario (outlined in Section 2.1), and propose a *PUM (Personalized ULDP Mechanism)* to keep secret what is sensitive for each user while enabling the data collector to estimate a distribution.

Sections 5.1 describes the PUM. Section 5.2 explains its privacy properties. Section 5.3 proposes a method to estimate the distribution $\mathbf{p}$ from $\mathbf{Y}$ obfuscated using the PUM. Section 5.4 analyzes the data utility of the PUM.

5.1 PUM with κ Semantic Tags

Figure 4 shows the overview of the PUM $\mathbf{Q}^{(i)}$ for the i-th user ($i = 1,2,\ldots,n$). It first deterministically maps personal data $x \in \mathcal{X}$ to *intermediate data* using a *pre-processor* $f_{pre}^{(i)}$, and then maps the intermediate data to obfuscated data $y \in \mathcal{Y}$ using a utility-optimized mechanism $\mathbf{Q}_{cmn}$ common to all users. The pre-processor $f_{pre}^{(i)}$ maps user-specific sensitive data $x \in \mathcal{X}_S^{(i)}$ to one of κ bots: $\perp_1, \perp_2, \cdots$, or $\perp_\kappa$. The κ bots represent user-specific sensitive data, and each of them is associated with a *semantic tag* such as "home" or "workplace". The κ semantic tags are the same for all users, and are useful when the data collector has some background knowledge about $\mathbf{p}$ conditioned on each tag. For example, a distribution of POIs tagged as "home" or "workplace" can be easily obtained via the Fousquare venue API [54]. Although this is not a user distribution but a "POI distribution", it can be used to roughly approximate the distribution of users tagged as "home" or "workplace", as shown in Section 6. We define a set $\mathcal{Z}$ of intermediate data by $\mathcal{Z} = \mathcal{X} \cup \{\perp_1, \cdots, \perp_\kappa\}$, and a set $\mathcal{Z}_S$ of sensitive intermediate data by $\mathcal{Z}_S = \mathcal{X}_S \cup \{\perp_1, \cdots, \perp_\kappa\}$.

Formally, the PUM $\mathbf{Q}^{(i)}$ first maps personal data $x \in \mathcal{X}$ to intermediate data $z \in \mathcal{Z}$ using a pre-processor $f_{pre}^{(i)} : \mathcal{X} \to \mathcal{Z}$ specific to each user. The pre-processor $f_{pre}^{(i)}$ maps sensitive

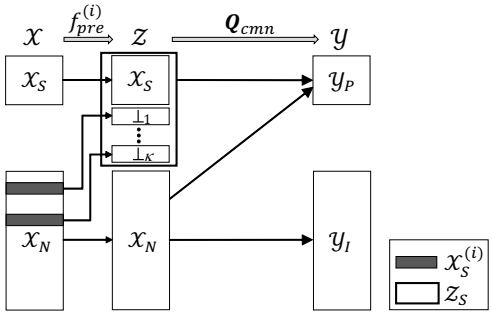


Figure 4: Overview of the PUM $\mathbf{Q}^{(i)}$ ($= \mathbf{Q}_{cmn} \circ f_{pre}^{(i)}$).

data $x \in \mathcal{X}_S^{(i)}$ associated with the k-th tag ($1 \leq k \leq \kappa$) to the corresponding bot $\perp_k$, and maps other data to themselves. Let $\mathcal{X}_{S,k}^{(i)}$ be a set of the i-th user's sensitive data associated with the k-th tag (e.g., set of regions including her primary home and second home). Then, $\mathcal{X}_S^{(i)}$ is expressed as $\mathcal{X}_S^{(i)} = \bigcup_{1 \leq k \leq \kappa} \mathcal{X}_{S,k}^{(i)}$, and $f_{pre}^{(i)}$ is given by:

$$f_{pre}^{(i)}(x) = \begin{cases} \perp_k & (\text{if } x \in \mathcal{X}_{S,k}^{(i)}) \\ x & (\text{otherwise}). \end{cases} \tag{19}$$

After mapping personal data $x \in \mathcal{X}$ to intermediate data $z \in \mathcal{Z}$, the $(\mathcal{Z}_S, \mathcal{Y}_P, \varepsilon)$-utility-optimized mechanism $\mathbf{Q}_{cmn}$ maps z to obfuscated data $y \in \mathcal{Y}$. Examples of $\mathbf{Q}_{cmn}$ include the $(\mathcal{Z}_S, \varepsilon)$-uRR (in Definition 3) and $(\mathcal{Z}_S, \varepsilon)$-uRAP (in Definition 4). As a whole, the PUM $\mathbf{Q}^{(i)}$ can be expressed as: $\mathbf{Q}^{(i)} = \mathbf{Q}_{cmn} \circ f_{pre}^{(i)}$. The i-th user stores $f_{pre}^{(i)}$ and $\mathbf{Q}_{cmn}$ in a device that obfuscates her personal data (e.g., mobile phone, personal computer). Note that if $f_{pre}^{(i)}$ is leaked, $x \in \mathcal{X}_N$ corresponding to each bot (e.g., home, workplace) is leaked. Thus, the user keeps $f_{pre}^{(i)}$ secret. To strongly prevent the leakage of $f_{pre}^{(i)}$, the user may deal with $f_{pre}^{(i)}$ using a tamper-resistant hardware/software. On the other hand, the utility-optimized mechanism $\mathbf{Q}_{cmn}$, which is common to all users, is available to the data collector.

The feature of the proposed PUM $\mathbf{Q}^{(i)}$ is two-fold: (i) the secrecy of the pre-processor $f_{pre}^{(i)}$ and (ii) the κ semantic tags. By the first feature, the i-th user can keep $\mathcal{X}_S^{(i)}$ (i.e., what is sensitive for her) secret, as shown in Section 5.2. The second feature enables the data collector to estimate a distribution $\mathbf{p}$ with high accuracy. Specifically, she estimates $\mathbf{p}$ from obfuscated data $\mathbf{Y}$ using $\mathbf{Q}_{cmn}$ and some background knowledge about $\mathbf{p}$ conditioned on each tag, as shown in Section 5.3.

In practice, it may happen that a user has her specific sensitive data $x \in \mathcal{X}_S^{(i)}$ that is not associated with any semantic tags. For example, if we prepare only tags named "home" and "workplace", then sightseeing places, restaurants, and any other places are not associated with these tags. One way to deal with such data is to create another bot associated with a tag named "others" (e.g., if $\perp_1$ and $\perp_2$ are associated with "home" and "workplace", respectively, we create $\perp_3$ associated with "others"), and map x to this bot. It would be difficult for the data collector to obtain background knowledge about $\mathbf{p}$ conditioned on such a tag. In Section 5.3, we will explain how to estimate $\mathbf{p}$ in this case.

5.2 Privacy Properties

We analyze the privacy properties of the PUM $\mathbf{Q}^{(i)}$. First, we show that it provides ULDP.

Proposition 9. *The PUM $\mathbf{Q}^{(i)}$ ($= \mathbf{Q}_{cmn} \circ f_{pre}^{(i)}$) provides $(\mathcal{X}_S \cup \mathcal{X}_S^{(i)}, \mathcal{Y}_P, \varepsilon)$-ULDP.*

We also show that our PUM provides DP in that an adversary who has observed $y \in \mathcal{Y}_P$ cannot determine, for any $i, j \in [n]$, whether it is obfuscated using $\mathbf{Q}^{(i)}$ or $\mathbf{Q}^{(j)}$, which means that $y \in \mathcal{Y}_P$ reveals almost no information about $\mathcal{X}_S^{(i)}$:

Proposition 10. *For any $i, j \in [n]$, any $x \in \mathcal{X}$, and any $y \in \mathcal{Y}_P$,*

$$\mathbf{Q}^{(i)}(y|x) \leq e^{\varepsilon} \mathbf{Q}^{(j)}(y|x).$$

We then analyze the secrecy of $\mathcal{X}_S^{(i)}$. The data collector, who knows the common-mechanism $\mathbf{Q}_{cmn}$, cannot obtain any information about $\mathcal{X}_S^{(i)}$ from $\mathbf{Q}_{cmn}$ and $y \in \mathcal{Y}_P$. Specifically, the data collector knows, for each $z \in \mathcal{Z}$, whether $z \in \mathcal{Z}_S$ or not by viewing $\mathbf{Q}_{cmn}$. However, she cannot obtain any information about $\mathcal{X}_S^{(i)}$ from $\mathcal{Z}_S$, because she does not know the mapping between $\mathcal{X}_S^{(i)}$ and $\{\perp_1, \cdots, \perp_\kappa\}$ (i.e., $f_{pre}^{(i)}$). In addition, Propositions 9 and 10 guarantee that $y \in \mathcal{Y}_P$ reveals almost no information about both input data and $\mathcal{X}_S^{(i)}$.

For example, assume that the i-th user obfuscates her home $x \in \mathcal{X}_S \cup \mathcal{X}_S^{(i)}$ using the PUM $\mathbf{Q}^{(i)}$, and sends $y \in \mathcal{Y}_P$ to the data collector. The data collector cannot infer either $x \in \mathcal{X}_S \cup \mathcal{X}_S^{(i)}$ or $z \in \mathcal{Z}_S$ from $y \in \mathcal{Y}_P$, since both $\mathbf{Q}_{cmn}$ and $\mathbf{Q}^{(i)}$ provide ULDP. This means that the data collector cannot infer *the fact that she was at home* from y. Furthermore, the data collector cannot infer *where her home is*, since $\mathcal{X}_S^{(i)}$ cannot be inferred from $\mathbf{Q}_{cmn}$ and $y \in \mathcal{Y}_P$ as explained above.

We need to take a little care when the i-th user obfuscates non-sensitive data $x \in \mathcal{X}_N \setminus \mathcal{X}_S^{(i)}$ using $\mathbf{Q}^{(i)}$ and sends $y \in \mathcal{Y}_I$ to the data collector. In this case, the data collector learns x from y, and therefore learns that x is not sensitive (i.e., $x \notin \mathcal{X}_S^{(i)}$). Thus, the data collector, who knows that the user wants to hide her home, would reduce the number of possible candidates for her home from $\mathcal{X}$ to $\mathcal{X} \setminus \{x\}$. However, if $|\mathcal{X}|$ is large (e.g., $|\mathcal{X}| = 625$ in our experiments using location data), the number $|\mathcal{X}| - 1$ of candidates is still large. Since the data collector cannot further reduce the number of candidates using $\mathbf{Q}_{cmn}$, her home is still kept strongly secret. In Section 7, we also explain that the secrecy of $\mathcal{X}_S^{(i)}$ is achieved under reasonable assumptions even when she sends multiple data.

5.3 Distribution Estimation

We now explain how to estimate a distribution $\mathbf{p}$ from data $\mathbf{Y}$ obfuscated using the PUM. Let $\mathbf{r}^{(i)}$ be a distribution of intermediate data for the i-th user:

$$\mathbf{r}^{(i)}(z) = \begin{cases} \sum_{x \in \mathcal{X}_{S,k}^{(i)}} \mathbf{p}(x) & (\text{if } z = \perp_k \text{ for some } k = 1, \dots, \kappa) \\ 0 & (\text{if } z \in \mathcal{X}_S^{(i)}) \\ \mathbf{p}(z) & (\text{otherwise}). \end{cases}$$

and $\mathbf{r}$ be the average of $\mathbf{r}^{(i)}$ over n users; i.e., $\mathbf{r}(z) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{r}^{(i)}(z)$ for any $z \in \mathcal{Z}$. Note that $\sum_{x \in \mathcal{X}} \mathbf{p}(x) = 1$ and $\sum_{z \in \mathcal{Z}} \mathbf{r}(z) = 1$. Furthermore, let π_k be a distribution of personal data $x \in \mathcal{X}$ conditioned on $\perp_k$ defined by:

$$\pi_k(x) = \frac{\sum_{i=1}^{n} \mathbf{p}_k^{(i)}(x)}{\sum_{x' \in \mathcal{X}} \sum_{i=1}^{n} \mathbf{p}_k^{(i)}(x')}, \tag{20}$$

$$\mathbf{p}_k^{(i)}(x) = \begin{cases} \mathbf{p}(x) & (\text{if } f_{pre}^{(i)}(x) = \perp_k) \\ 0 & (\text{otherwise}). \end{cases}$$

$\pi_k(x)$ in (20) is a normalized sum of the probability $\mathbf{p}(x)$ of personal data x whose corresponding intermediate data is $\perp_k$. Note that although $x \in \mathcal{X}$ is deterministically mapped to $z \in \mathcal{Z}$ for each user, we can consider the probability distribution π_k for n users. For example, if $\perp_k$ is tagged as "home", then π_k is a distribution of users at home.

We propose a method to estimate a distribution $\mathbf{p}$ from obfuscated data $\mathbf{Y}$ using some background knowledge about π_k as an estimate $\hat{\pi}_k$ of π_k (we explain the case where we have no background knowledge later). Our estimation method first estimates a distribution $\mathbf{r}$ of intermediate data from obfuscated data $\mathbf{Y}$ using $\mathbf{Q}_{cmn}$. This can be performed in the same way as the common-mechanism scenario. Let $\hat{\mathbf{r}}$ be the estimate of $\mathbf{r}$.

After computing $\hat{\mathbf{r}}$, our method estimates $\mathbf{p}$ using the estimate $\hat{\pi}_k$ (i.e., background knowledge about π_k) as follows:

$$\hat{\mathbf{p}}(x) = \hat{\mathbf{r}}(x) + \sum_{k=1}^{\kappa} \hat{\mathbf{r}}(\perp_k)\hat{\pi}_k(x), \quad \forall x \in \mathcal{X}. \tag{21}$$

Note that $\hat{\mathbf{p}}$ in (21) can be regarded as an empirical estimate of $\mathbf{p}$. Moreover, if both $\hat{\mathbf{r}}$ and $\hat{\pi}_k$ are in the probability simplex $\mathcal{C}$, then $\hat{\mathbf{p}}$ in (21) is always in $\mathcal{C}$.

If we do not have estimates $\hat{\pi}_k$ for some bots (like the one tagged as "others" in Section 5.1), then we set $\hat{\pi}_k(x)$ in proportion to $\hat{\mathbf{r}}(x)$ over $x \in \mathcal{X}_N$ (i.e., $\hat{\pi}_k(x) = \frac{\hat{\mathbf{r}}(x)}{\sum_{x' \in \mathcal{X}_N} \hat{\mathbf{r}}(x')}$) for such bots. When we do not have any background knowledge $\hat{\pi}_1, \cdots, \hat{\pi}_\kappa$ for all bots, it amounts to simply discarding the estimates $\hat{\mathbf{r}}(\perp_1), \cdots, \hat{\mathbf{r}}(\perp_\kappa)$ for κ bots and normalizing $\hat{\mathbf{r}}(x)$ over $x \in \mathcal{X}_N$ so that the sum is one.

5.4 Utility Analysis

We now theoretically analyze the data utility of our PUM. Recall that $\hat{\mathbf{p}}$, $\hat{\mathbf{r}}$, and $\hat{\pi}_k$ are the estimate of the distribution of personal data, intermediate data, and personal data conditioned on $\perp_k$, respectively. In the following, we show that the l_1 loss of $\hat{\mathbf{p}}$ can be upper-bounded as follows:

Theorem 1 (l_1 loss of the PUM)**.**

$$l_1(\hat{\mathbf{p}}, \mathbf{p}) \leq l_1(\hat{\mathbf{r}}, \mathbf{r}) + \sum_{k=1}^{\kappa} \hat{\mathbf{r}}(\perp_k) l_1(\hat{\pi}_k, \pi_k). \tag{22}$$

This means the upper-bound on the l_1 loss of $\hat{\mathbf{p}}$ can be decomposed into the l_1 loss of $\hat{\mathbf{r}}$ and of $\hat{\pi}_k$ weighted by $\hat{\mathbf{r}}(\perp_k)$.

The first term in (22) is the l_1 loss of $\hat{\mathbf{r}}$, which depends on $\mathbf{Q}_{cmn}$. For example, if we use the uRR or uRAP as $\mathbf{Q}_{cmn}$, the expectation of $l_1(\hat{\mathbf{r}}, \mathbf{r})$ is given by Propositions 4 and 7, respectively. In Section 6, we show they are very small.

The second term in (22) is the summation of the l_1 loss of $\hat{\pi}_k$ weighted by $\hat{\mathbf{r}}(\perp_k)$. If we accurately estimate π_k, the second term is very small. In other words, if we have enough background knowledge about π_k, we can accurately estimate $\mathbf{p}$ in the personalized-mechanism scenario.

It should be noted that when the probability $\hat{\mathbf{r}}(\perp_k)$ is small, the second term in (22) is small *even if we have no background knowledge about* π_k. For example, when only a small number of users map $x \in \mathcal{X}_S^{(i)}$ to a tag named "others", they hardly affect the accuracy of $\hat{\mathbf{p}}$. Moreover, the second term in (22) is upper-bounded by $2\sum_{k=1}^{\kappa} \hat{\mathbf{r}}(\perp_k)$, since the l_1 loss is at most 2. Thus, after computing $\hat{\mathbf{r}}$, the data collector can easily compute the worst-case value of the second term in (22) to know the effect of the estimation error of $\hat{\pi}_k$ on the accuracy of $\hat{\mathbf{p}}$.

Last but not least, the second term in (22) does not depend on ε (while the first term depends on ε). Thus, the effect of the second term is relatively small when ε is small (i.e., high privacy regime), as shown in Section 6.

Remark. Note that different privacy preferences might skew the distribution π_k. For example, doctors might not consider hospitals as sensitive as compared to patients. Consequently, the distribution π_k conditioned on "hospital" might be a distribution of patients (not doctors) in hospitals. This kind of systematic bias can increase the estimation error of $\hat{\pi}_k$. Theorem 1 and the above discussions are also valid in this case.

6 Experimental Evaluation

6.1 Experimental Set-up

We conducted experiments using two large-scale datasets:

Foursquare dataset. The Foursquare dataset (global-scale check-in dataset) [54] is one of the largest location datasets among publicly available datasets (e.g., see [10], [44], [55], [57]); it contains 33278683 check-ins all over the world, each of which is associated with a POI ID and venue category (e.g., restaurant, shop, hotel, hospital, home, workplace).

We used 359054 check-ins in Manhattan, assuming that each check-in is from a different user. Then we divided Manhattan into 25×25 regions at regular intervals and used them

as input alphabets; i.e., $|\mathcal{X}| = 625$. The size of each region is about 400m (horizontal) $\times$ 450m (vertical). We assumed a region that includes a hospital visited by at least ten users as a sensitive region common to all users. The number of such regions was $|\mathcal{X}_S| = 15$. In addition, we assumed a region in $\mathcal{X}_N$ that includes a user's home or workplace as her user-specific sensitive region. The number of users at home and workplace was 5040 and 19532, respectively.

US Census dataset. The US Census (1990) dataset [35] was collected as part of the 1990 U.S. census. It contains responses from 2458285 people (each person provides one response), each of which contains 68 attributes.

We used the responses from all people, and used age, income, marital status, and sex as attributes. Each attribute has 8, 5, 5, and 2 categories, respectively. (See [35] for details about the value of each category ID.) We regarded a tuple of the category IDs as a total category ID, and used it as an input alphabet; i.e., $|\mathcal{X}| = 400$ $(= 8 \times 5 \times 5 \times 2)$. We considered the fact that "divorce" and "unemployment" might be sensitive for many users [34], and regarded such categories as sensitive for all users (to be on the safe side, as described in Section 2.1). Note that people might be students until their twenties and might retire in their fifties or sixties. Children of age twelve and under cannot get married. We excluded such categories from sensitive ones. The number of sensitive categories was $|\mathcal{X}_S| = 76$.

We used a frequency distribution of all people as a true distribution $\mathbf{p}$, and randomly chose a half of all people as users who provide their obfuscated data; i.e., $n = 179527$ and 1229143 in the Foursquare and US Census datasets, respectively. Here we did not use all people, because we would like to evaluate the non-private mechanism that does not obfuscate the personal data; i.e., the non-private mechanism has an estimation error in our experiments due to the random sampling from the population.

As utility, we evaluated the TV (Total Variation) by computing the sample mean over a hundred realizations of $\mathbf{Y}$.

6.2 Experimental Results

Common-mechanism scenario. We first focused on the common-mechanism scenario, and evaluated the RR, RAPPOR, uRR, and uRAP. As distribution estimation methods, we used empirical estimation, empirical estimation with the significance threshold, and EM reconstruction (denoted by "emp", "emp+thr", and "EM", respectively). In "emp+thr", we set the significance level α to be $\alpha = 0.05$, and uniformly assigned the remaining probability to each of the estimates below the significance threshold in the same way as [51].

Figure 5 shows the results in the case where ε is changed from 0.1 to 10. "no privacy" represents the non-private mechanism. It can be seen that our mechanisms outperform the existing mechanisms by one or two orders of magnitude. Our mechanisms are effective especially in the Foursquare

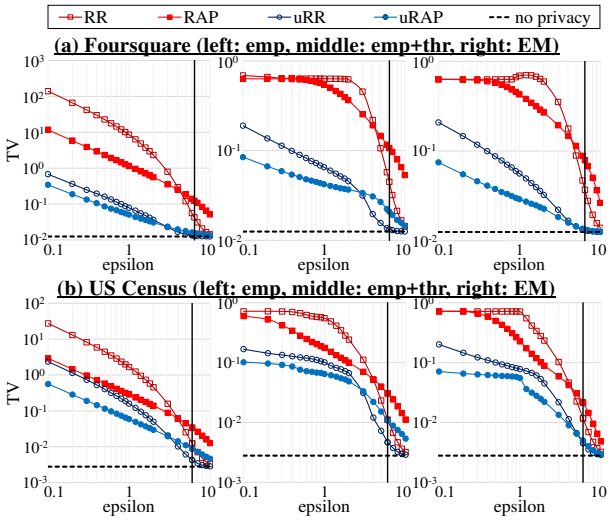


Figure 5: ε vs. TV (common-mechanism). A bold line parallel to the y-axis represents $\varepsilon = \ln|\mathcal{X}|$.

dataset, since the proportion of sensitive regions is very small $(15/625 = 0.024)$. Moreover, the uRR provides almost the same performance as the non-private mechanism when $\varepsilon = \ln|\mathcal{X}|$, as described in Section 4.3. It can also be seen that "emp+thr" and "EM" significantly outperform "emp", since the estimates in "emp+thr" and "EM" are always non-negative. Although "EM" outperforms "emp+thr" for the RAPPOR and uRAP when ε was large, the two estimation methods provide very close performance as a whole.

We then evaluated the relationship between the number of sensitive regions/categories and the TV. To this end, we randomly chose $\mathcal{X}_S$ from $\mathcal{X}$, and increased $|\mathcal{X}_S|$ from 1 to $|\mathcal{X}|$ (only in this experiment). We attempted one hundred cases for randomly choosing $\mathcal{X}_S$ from $\mathcal{X}$, and evaluated the TV by computing the sample mean over one hundred cases.

Figure 6 shows the results for $\varepsilon = 0.1$ (high privacy regime) or $\ln|\mathcal{X}|$ (low privacy regime). Here we omit the performance of "emp+thr", since it is very close to that of "EM" in the same way as in Figure 5. The uRAP and uRR provide the best performance when $\varepsilon = 0.1$ and $\ln|\mathcal{X}|$, respectively. In addition, the uRR provides the performance close to the non-private mechanism when $\varepsilon = \ln|\mathcal{X}|$ and the number $|\mathcal{X}_S|$ of sensitive regions/categories is less than 100. The performance of the uRAP is also close to that of the non-private mechanism when $|\mathcal{X}_S|$ is less than 20 (note that $|\mathcal{X}|^{\frac{3}{4}} = 125$ and 89 in the Foursquare and US Census datasets, respectively). However, it rapidly increases with increase in $|\mathcal{X}_S|$. Overall, our theoretical results in Section 4.3 hold for the two real datasets.

We also evaluated the performance when the number of attributes was increased from 4 to 9 in the US Census dataset. We added, one by one, five attributes as to whether or not a user has served in the military during five periods ("Sept80", "May75880", "Vietnam", "Feb55", and "Korean" in [18]; we added them in this order). We assumed that these attributes

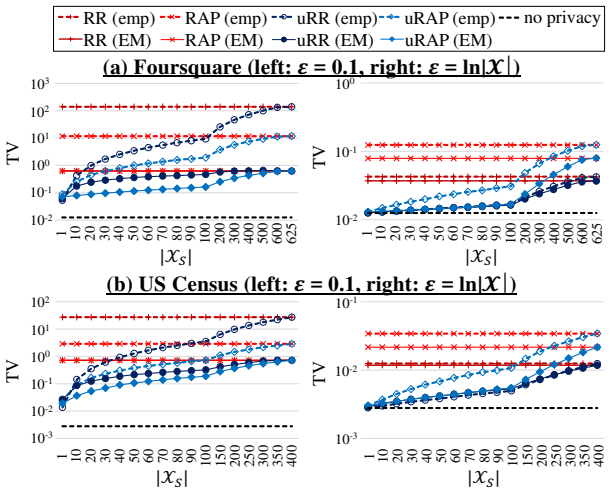


Figure 6: $|\mathcal{X}_S|$ vs. TV when $\varepsilon = 0.1$ or $\ln|\mathcal{X}|$.

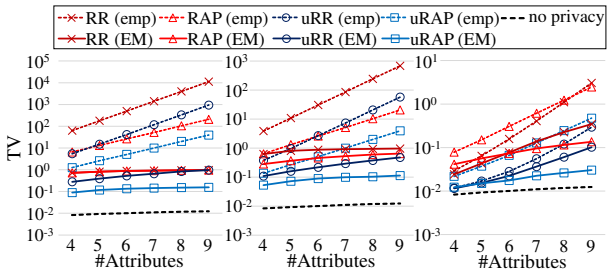


Figure 7: Number of attributes vs. TV (US Census dataset; left: $\varepsilon = 0.1$, middle: $\varepsilon = 1.0$, right: $\varepsilon = 6.0$).

are non-sensitive. Since each of the five attributes had two categories (1: yes, 0: no), $|\mathcal{X}|$ (resp. $|\mathcal{X}_S|$) was changed from 400 to 12800 (resp. from 76 to 2432). We randomly chose $n = 240000$ people as users who provide obfuscated data, and evaluated the TV by computing the sample mean over ten realizations of $\mathbf{Y}$ (only in this experiment).

Figure 7 shows the results in the case where $\varepsilon = 0.1$, 1.0, or 6.0 (=ln 400). Here we omit the performance of "emp+thr" in the same way as Figure 6. Although the TV increases with an increase in the number of attributes, overall our utility-optimized mechanisms remain effective, compared to the existing mechanisms. One exception is the case where $\varepsilon = 0.1$ and the number of attributes is 9; the TV of the RR (EM), RAPPOR (EM), and uRR (EM) is almost 1. Note that when we use the EM reconstruction method, the worst value of the TV is 1. Thus, as with the RR and RAPPOR, the uRR fails to estimate a distribution in this case. On the other hand, the TV of the uRAP (EM) is much smaller than 1 even in this case, which is consistent with the fact that the uRAP is order optimal in the high privacy regime. Overall, the uRAP is robust to the increase of the attributes at the same value of ε (note that for large $|\mathcal{X}|$, $\varepsilon = 1.0$ or 6.0 is a medium privacy regime where $0 \ll \varepsilon \ll \ln|\mathcal{X}|$).

We also measured the running time (i.e., time to estimate $\mathbf{p}$ from $\mathbf{Y}$) of "EM" (which sets the estimate by "emp+thr" as

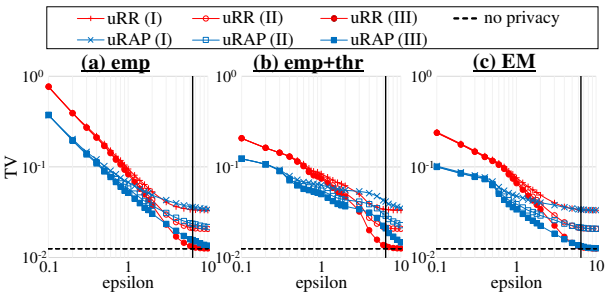


Figure 8: ε vs. TV (personalized-mechanism) ((I): w/o knowledge, (II): POI distribution, (III): true distribution).

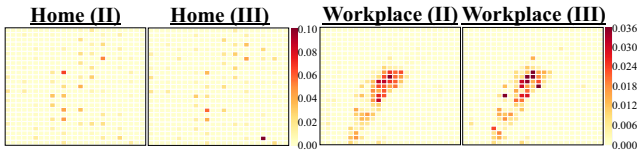


Figure 9: Visualization of the distributions ((II): POI distribution, (III): true distribution).

an initial value of $\hat{\mathbf{p}}$) on an Intel Xeon CPU E5-2620 v3 (2.40 GHz, 6 cores, 12 logical processors) with 32 GB RAM. We found that the running time increases roughly linearly with the number of attributes. For example, when $\varepsilon = 6.0$ and the number of attributes is 9, the running time of "EM" required 3121, 1258, 5225, and 1073 seconds for "RR", "uRR", "RAP", and "uRAP", respectively. We also measured the running time of 'emp" and "emp+thr", and found that they required less than one second even when the number of attributes is 9. Thus, if "EM" requires too much time for a large number of attributes, "emp+thr" would be a good alternative to "EM".

Personalized-mechanism scenario. We then focused on the personalized-mechanism scenario, and evaluated our utility-optimized mechanisms using the Foursquare dataset. We used the PUM with $\kappa = 2$ semantic tags (described in Section 5.1), which maps "home" and 'workplace" to bots $\perp_1$ and $\perp_2$, respectively. As the background knowledge about the bot distribution π_k ($1 \le k \le 2$), we considered three cases: (I) we do not have any background knowledge; (II) we use a distribution of POIs tagged as "home" (resp. "workplace"), which is computed from the POI data in [54], as an estimate of the bot probability $\hat{\pi}_1$ (resp. $\hat{\pi}_2$); (III) we use the true distributions (i.e., $\hat{\pi}_k = \pi_k$). Regarding (II), we emphasize again that it is not a user distribution but a "POI distribution", and can be easily obtained via the Foursquare venue API [54].

Figure 8 shows the results. We also show the POI and true distributions in Figure 9. It can be seen that the performance of (II) lies in between that of (I) and (III), which shows that the estimate $\hat{\pi}_k$ of the bot distribution affects utility. However, when ε is smaller than 1, all of (I), (II), and (III) provide almost the same performance, since the effect of the estimation error of $\hat{\pi}_k$ does not depend on ε, as described in Section 5.4.

We also computed the l_1 loss $l_1(\hat{\mathbf{p}}, \mathbf{p})$ and the first and second terms in the right-hand side of (22) to investigate

Table 1: l_1 loss $l_1(\hat{\mathbf{p}}, \mathbf{p})$ and the first and second terms in the right-hand side of (22) in the case where $\varepsilon = \ln|\mathcal{X}|$ and the EM reconstruction method is used.

Method	$l_1(\hat{\mathbf{p}}, \mathbf{p})$	first term	second term
uRR (**I**)	6.73×10^{-2}	2.70×10^{-2}	7.34×10^{-2}
uRR (**II**)	4.24×10^{-2}	2.70×10^{-2}	2.96×10^{-2}
uRR (**III**)	2.62×10^{-2}	2.70×10^{-2}	0
uRAP (**I**)	6.77×10^{-2}	2.76×10^{-2}	7.35×10^{-2}
uRAP (**II**)	4.28×10^{-2}	2.76×10^{-2}	2.96×10^{-2}
uRAP (**III**)	2.67×10^{-2}	2.76×10^{-2}	0

whether Theorem 1 holds. Table 1 shows the results (we averaged the values over one hundred realizations of $\mathbf{Y}$). It can be seen that $l_1(\hat{\mathbf{p}}, \mathbf{p})$ is smaller than the summation of the first and second terms in all of the methods, which shows that Theorem 1 holds in our experiments.

From these experimental results, we conclude that our proposed methods are very effective in both the common-mechanism and personalized-mechanism scenarios. In Appendix C.2, we show the MSE has similar results to the TV.

7 Discussions

On the case of multiple data per user. We have so far assumed that each user sends only a single datum. Now we discuss the case where each user sends multiple data based on the compositionality of ULDP described in Section 3.2. Specifically, when a user sends t (> 1) data, we obtain $(\mathcal{X}_S, (\mathcal{Y}_P)^t, \varepsilon)$-ULDP in total by obfuscating each data using the $(\mathcal{X}_S, \mathcal{Y}_P, \varepsilon/t)$-utility-optimized mechanism. Note, however, that the amount of noise added to each data increases with increase in t. Consequently, for $\varepsilon \in [0, t]$, the lower bound on the l_1 (resp. l_2) loss (described in Section 3.2) can be expressed as $\Theta(\frac{\sqrt{t}|\mathcal{X}_S|}{\sqrt{n\varepsilon^2}})$ (resp. $\Theta(\frac{t|\mathcal{X}_S|}{n\varepsilon^2})$), which increases with increase in t. Thus, t cannot be large for distribution estimation in practice. This is also common to all LDP mechanisms.

Next we discuss the secrecy of $\mathcal{X}_S^{(i)}$. Assume that the i-th user obfuscates t data using different seeds, and sends t_P protected data in $\mathcal{Y}_P$ and t_I invertible data in $\mathcal{Y}_I$, where $t = t_P + t_I > 1$ (she can also use the same seed for the same data to reduce t_I as in [23]). If all the t_I data in $\mathcal{Y}_I$ are different from each other, the data collector learns t_I original data in $\mathcal{X}_N$. However, t_I $(\leq t)$ cannot be large in practice, as explained above. In addition, in many applications, a user's personal data is highly non-uniform and sparse. In locations data, for example, a user often visits only a small number of regions in the whole map $\mathcal{X}$. Let $\mathcal{T}^{(i)} \subseteq \mathcal{X}_N$ be a set of possible input values for the i-th user in $\mathcal{X}_N$. Then, even if t_I is large, the data collector cannot learn more than $|\mathcal{T}^{(i)}|$ data in $\mathcal{X}_N$.

Moreover, the t_P data in $\mathcal{Y}_P$ reveal almost no information about $\mathcal{X}_S^{(i)}$, since $\mathbf{Q}^{(i)}$ provides $(\mathcal{X}_S, (\mathcal{Y}_P)^t, \varepsilon)$-ULDP. $\mathbf{Q}_{cmn}$ provides no information about $\mathcal{X}_S^{(i)}$, since $f_{pre}^{(i)}$ is kept secret. Thus, the data collector, who knows that the user wants to hide her home, cannot reduce the number of candidates for her home from $\max\{|\mathcal{X}| - t_I, |\mathcal{X}| - |\mathcal{T}^{(i)}|\}$ using the t_P data and $\mathbf{Q}_{cmn}$. If either t_I or $|\mathcal{T}^{(i)}|$ is much smaller than $|\mathcal{X}|$, her home is kept strongly secret.

Note that $\mathbf{p}$ can be estimated even if $\mathcal{X}_S^{(i)}$ changes over time. $\mathcal{X}_S^{(i)}$ is also kept strongly secret if t_I or $|\mathcal{T}^{(i)}|$ is small.

On the correlation between $\mathcal{X}_S$ and $\mathcal{X}_N$. It should also be noted that there might be a correlation between sensitive data $\mathcal{X}_S$ and non-sensitive data $\mathcal{X}_N$. For example, if a user discloses a non-sensitive region close to a sensitive region including her home, the adversary might infer approximate information about the original location (e.g., the fact that the user lives in Paris). However, we emphasize that if the size of each region is large, the adversary cannot infer the exact location such as the exact home address. Similar approaches can be seen in a state-of-the-art location privacy measure called *geo-indistinguishability* [4, 7, 42, 47]. Andrés *et al.* [4] considered privacy protection within a radius of 200m from the original location, whereas the size of each region in our experiments was about 400m $\times$ 450m (as described in Section 6.1). We can protect the exact location by setting the size of each region to be large enough, or setting all regions close to a user's sensitive location to be sensitive.

There might also be a correlation between two attributes (e.g., income and marital status) in the US Census dataset. However, we combined the four category IDs into a total category ID for each user as described in Section 6.1. Thus, there is only "one" category ID for each user. Assuming that each user's data is independent, there is no correlation between data. Therefore, we conclude that the sensitive data are strongly protected in both the Foursquare and US Census datasets in our experiments.

It should be noted, however, that the number of total category IDs increases exponentially with the number of attributes. Thus, when there are many attributes as in Figure 7, the estimation accuracy might be increased by obfuscating each attribute independently (rather than obfuscating a total ID) while considering the correlation among attributes. We also need to consider a correlation among "users" for some types of personal data (e.g., flu status). For rigorously protecting such correlated data, we should incorporate Pufferfish privacy [32, 48] into ULDP, as described in Section 1.

8 Conclusion

In this paper, we introduced the notion of ULDP that guarantees privacy equivalent to LDP for only sensitive data. We proposed ULDP mechanisms in both the common and personalized mechanism scenarios. We evaluated the utility of our mechanisms theoretically and demonstrated the effectiveness of our mechanisms through experiments.

References

[1] D. Agrawal and C. C. Aggarwal. On the design and quantification of privacy preserving data mining algorithms. In *Proc. PODS*, pages 247–255, 2001.

[2] R. Agrawal, R. Srikant, and D. Thomas. Privacy preserving OLAP. In *Proc. SIGMOD*, pages 251–262, 2005.

[3] M. Alaggan, S. Gambs, and A.-M. Kermarrec. Heterogeneous differential privacy. *Journal of Privacy and Confidentiality*, 7(2):127–158, 2017.

[4] M. E. Andrés, N. E. Bordenabe, K. Chatzikokolakis, and C. Palamidessi. Geo-indistinguishability: Differential privacy for location-based systems. In *Proc. CCS*, pages 901–914, 2013.

[5] B. Avent, A. Korolova, D. Zeber, T. Hovden, and B. Livshits. BLENDER: Enabling local search with a hybrid differential privacy model. In *Proc. USENIX Security*, pages 747–764, 2017.

[6] R. Bassily and A. Smith. Local, private, efficient protocols for succinct histograms. In *Proc. STOC*, pages 127–135, 2015.

[7] N. E. Bordenabe, K. Chatzikokolakis, and C. Palamidessi. Optimal geo-indistinguishable mechanisms for location privacy. In *Proc. CCS*, pages 251–262, 2014.

[8] K. Chatzikokolakis, M. E. André, N. E. Bordenabe, and C. Palamidessi. Broadening the scope of differential privacy using metrics. In *Proc. PETS*, pages 82–102, 2013.

[9] X. Chen, A. Guntuboyina, and Y. Zhang. On Bayes risk lower bounds. *J. Mach. Learn. Res.*, 17(219):1–58, 2016.

[10] E. Cho, S. A. Myers, and J. Leskovec. Friendship and mobility: User movement in location-based social networks. In *Proc. KDD*, pages 1082–1090, 2011.

[11] J. E. Cohen. Statistical concepts relevant to AIDS. In *Proc. Symposium on Statistics in Science, Industry, and Public Policy*, pages 43–51, 1989.

[12] G. Cormode, T. Kulkarni, and D. Srivastava. Marginal release under local differential privacy. In *Proc. SIGMOD*, pages 131–146, 2018.

[13] T. M. Cover and J. A. Thomas. *Elements of Information Theory, Second Edition*. Wiley-Interscience, 2006.

[14] P. Cuff and L. Yu. Differential privacy as a mutual information constraint. In *Proc. CCS*, pages 43–54, 2016.

[15] Data Breaches Increase 40 Percent in 2016, Finds New Report from Identity Theft Resource Center and CyberScout. http://www.idtheftcenter.org/2016databreaches.html, 2017.

[16] B. Ding, J. Kulkarni, and S. Yekhanin. Collecting telemetry data privately. In *Proc. NIPS*, pages 3574–3583, 2017.

[17] S. Doudalis, I. Kotsoginannis, S. Haney, A. Machanavajjhala, and S. Mehrotra. One-sided differential privacy. *CoRR*, abs/1712.05888, 2017.

[18] D. Dua and E. K. Taniskidou. UCI machine learning repository. http://archive.ics.uci.edu/ml, 2017.

[19] J. C. Duchi, M. I. Jordan, and M. J. Wainwright. Local privacy and statistical minimax rates. In *Proc. FOCS*, pages 429–438, 2013.

[20] J. C. Duchi, M. I. Jordan, and M. J. Wainwright. Local privacy, data processing inequalities, and minimax rates. *CoRR*, abs/1302.3203, 2013.

[21] C. Dwork, F. Mcsherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Proc. TCC*, pages 265–284, 2006.

[22] C. Dwork and A. Roth. *The Algorithmic Foundations of Differential Privacy*. Now Publishers, 2014.

[23] U. Erlingsson, V. Pihur, and A. Korolova. RAPPOR: Randomized aggregatable privacy-preserving ordinal response. In *Proc. CCS*, pages 1054–1067, 2014.

[24] G. Fanti, V. Pihur, and U. Erlingsson. Building a RAPPOR with the unknown: Privacy-preserving learning of associations and data dictionaries. *PoPETs*, 2016(3):1–21, 2016.

[25] P. Golle and K. Partridge. On the anonymity of home/work location pairs. In *Proc. Pervasive*, pages 390–397, 2009.

[26] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Spinger, 2nd edition, 2009.

[27] Z. Huang and W. Du. OptRR: Optimizing randomized response schemes for privacy-preserving data mining. In *Proc. ICDE*, pages 705–714, 2008.

[28] Z. Jorgensen, T. Yu, and G. Cormode. Conservative or liberal? Personalized differential privacy. In *Proc. ICDE*, pages 1023–1034, 2015.

[29] P. Kairouz, K. Bonawitz, and D. Ramage. Discrete distribution estimation under local privacy. In *Proc. ICML*, pages 2436–2444, 2016.

[30] P. Kairouz, S. Oh, and P. Viswanath. Extremal mechanisms for local differential privacy. *J. Mach. Learn. Res.*, 17(1):492–542, 2016.

[31] Y. Kawamoto and T. Murakami. Differentially private obfuscation mechanisms for hiding probability distributions. *CoRR*, abs/1812.00939, 2018.

[32] D. Kifer and A. Machanavajjhala. Pufferfish: A framework for mathematical privacy definitions. *ACM Trans. Database Syst.*, 39(1):1–36, 2014.

[33] S. Krishnan, J. Wang, M. J. Franklin, K. Goldberg, and T. Kraska. PrivateClean: Data cleaning and differential privacy. In *Proc. SIGMOD*, pages 937–951, 2016.

[34] R. L. Leahy. Feeling ashamed of being unemployed - am I afraid of telling people that I am out of work? https://www.psychologytoday.com/us/blog/anxiety-files/201310/feeling-ashamed-being-unemployed, 2013.

[35] M. Lichman. UCI machine learning repository, 2013.

[36] C. Liu, S. Chakraborty, and P. Mittal. Dependence makes you vulnerable: Differential privacy under dependent tuples. In *Proc. NDSS*, 2016.

[37] N. S. Mangat. An improved ranomized response strategy. *J. Royal Stat. Soc. Series B (Methodological)*, 56(1):93–95, 1994.

[38] I. Mironov. Rényi differential privacy. In *Proc. CSF*, pages 263–275, 2017.

[39] T. Murakami, H. Hino, and J. Sakuma. Toward distribution estimation under local differential privacy with small samples. *PoPETs*, 3:84–104, 2017.

[40] T. Murakami and Y. Kawamoto. Utility-optimized local differential privacy mechanisms for distribution estimation. *CoRR*, abs/1807.11317, 2019.

[41] A. Narayanan and V. Shmatikov. Myths and fallacies of "personally identifiable information". *Commun. ACM*, 53(6):24–26, 2010.

[42] S. Oya, C. Troncoso, and F. Pérez-González. Back to the drawing board: Revisiting the design of optimal location privacy-preserving mechanisms. In *Proc. CCS*, pages 1959–1972, 2017.

[43] A. Pastore and M. Gastpar. Locally differentially-private distribution estimation. In *Proc. ISIT*, pages 2694–2698, 2016.

[44] M. Piorkowski, N. Sarafijanovic-Djukic, and M. Grossglauser. CRAWDAD dataset epfl/mobility (v. 2009-02-24). http://crawdad.org/epfl/mobility/20090224, 2009.

[45] Z. Qin, Y. Yang, T. Yu, I. Khalil, X. Xiao, and K. Ren. Heavy hitter estimation over set-valued data with local differential privacy. In *Proc. CCS*, pages 192–203, 2016.

[46] Y. Sei and A. Ohusuga. Differential private data collection and analysis based on randomized multiple dummies for untrusted mobile crowdsensing. *IEEE Trans. Inf. Forensics Secur.*, 12(4):926–939, 2017.

[47] R. Shokri. Privacy games: Optimal user-centric data obfuscation. *PoPETs*, 2015(2):299–315, 2015.

[48] S. Song, Y. Wang, and K. Chaudhuri. Pufferfish privacy mechanisms for correlated data. In *Proc. SIGMOD*, pages 1291–1306, 2017.

[49] A. G. Thakurta, A. H. Vyrros, U. S. Vaishampayan, G. Kapoor, J. Freudiger, V. R. Sridhar, and D. Davidson. Learning New Words, US Patent 9,594,741, Mar. 14 2017.

[50] N. Wang, X. Xiao, T. D. Hoang, H. Shin, J. Shin, and G. Yu. PrivTrie: Effective frequent term discovery under local differential privacy. In *Proc. ICDE*, 2018.

[51] T. Wang, J. Blocki, N. Li, and S. Jha. Locally differentially private protocols for frequency estimation. In *Proc. USENIX Security*, pages 729–745, 2017.

[52] S. L. Warner. Randomized response: A survey technique for eliminating evasive answer bias. *J. Am. Stat. Assoc.*, 60(309):63–69, 1965.

[53] B. Yang, I. Sato, and H. Nakagawa. Bayesian differential privacy on correlated data. In *Proc. SIGMOD*, pages 747–762, 2015.

[54] D. Yang, D. Zhang, and B. Qu. Participatory cultural mapping based on collective behavior data in location based social network. *ACM Trans. Intell. Syst. Technol.*, 7(3):30:1–30:23, 2016.

[55] D. Yang, D. Zhang, V. W. Zheng, and Z. Yu. Modeling user activity preference by leveraging user spatial temporal characteristics in LBSNs. *IEEE Trans. Syst., Man, Cybern., Syst.*, 45(1):129–142, 2015.

[56] M. Ye and A. Barg. Optimal schemes for discrete distribution estimation under local differential privacy. In *Proc. ISIT*, pages 759–763, 2017.

[57] Y. Zheng, X. Xie, and W.-Y. Ma. GeoLife: A collaborative social networking service among user, location and trajectory. *IEEE Data Eng. Bull.*, 32(2):32–40, 2010.

A Properties of ULDP

In this section, we describe the properties of ULDP (the immunity to post-processing and the compatibility with LDP) in more details.

A.1 Post-processing

We first define a class of post-processing randomized algorithms that preserve data types:

Definition 5 (Preservation of data types)**.** Let $\mathcal{Y}_P$ and $\mathcal{Z}_P$ be sets of protected data, and $\mathcal{Y}_I$ and $\mathcal{Z}_I$ be sets of invertible data. Given a randomized algorithm $\mathbf{Q}_1$ from $\mathcal{Y}_P \cup \mathcal{Y}_I$ to $\mathcal{Z}_P \cup \mathcal{Z}_I$, we say that $\mathbf{Q}_1$ *preserves data types* if it satisfies:

- for any $z \in \mathcal{Z}_P$ and any $y \in \mathcal{Y}_I$, $\mathbf{Q}_1(z|y) = 0$, and
- for any $z \in \mathcal{Z}_I$, there exists a $y \in \mathcal{Y}_I$ such that $\mathbf{Q}_1(z|y) > 0$ and $\mathbf{Q}_1(z|y') = 0$ for any $y' \neq y$.

Then we show that ULDP is immune to the post-processing by this class of randomized algorithms.

Proposition 11 (Post-processing)**.** *Let $\varepsilon \geq 0$. Let $\mathcal{Z}_P$ and $\mathcal{Z}_I$ be sets of protected and invertible data respectively, and $\mathcal{Z} = \mathcal{Z}_P \cup \mathcal{Z}_I$. Let $\mathbf{Q}_1$ be a randomized algorithm from $\mathcal{Y}$ to $\mathcal{Z}$ that preserves data types. If an obfuscation mechanism $\mathbf{Q}_0$ from $\mathcal{X}$ to $\mathcal{Y}$ provides $(\mathcal{X}_S, \mathcal{Y}_P, \varepsilon)$-ULDP then the composite function $\mathbf{Q}_1 \circ \mathbf{Q}_0$ provides $(\mathcal{X}_S, \mathcal{Z}_P, \varepsilon)$-ULDP.*

For example, ULDP is immune to data cleaning operations (e.g., transforming values, merging disparate values) [33] as long as they are represented as $\mathbf{Q}_1$ explained above.

Note that $\mathbf{Q}_1$ needs to preserve data types for utility (i.e., to make all $y \in \mathcal{Y}_I$ invertible, as in Definition 2, after post-processing), and the DP guarantee for $y \in \mathcal{Y}_P$ is preserved by any post-processing algorithm. Specifically, by (5), for any randomized post-processing algorithm $\mathbf{Q}_1^*$, any obfuscated data $z \in \mathcal{Z}$ obtained from $y \in \mathcal{Y}_P$ via $\mathbf{Q}_1^*$, and any $x, x' \in \mathcal{X}$, we have: $\Pr(z|x) \leq e^{\varepsilon} \Pr(z|x')$.

A.2 Compatibility with LDP

Assume that data collectors A and B adopt a mechanism $\mathbf{Q}_A$ providing $(\mathcal{X}_S, \mathcal{Y}_P, \varepsilon_A)$-ULDP and a mechanism $\mathbf{Q}_B$ providing ε_B-LDP, respectively. In this case, all protected data in the data collector A can be combined with all obfuscated data in the data collector B (i.e., data integration) to perform data analysis under LDP. More specifically, assume that Alice transforms her sensitive personal data in $\mathcal{X}_S$ into $y_A \in \mathcal{Y}_P$ (resp. $y_B \in \mathcal{Y}$) using $\mathbf{Q}_A$ (resp. $\mathbf{Q}_B$), and sends y_A (resp. y_B) to the data collector A (resp. B) to request two different services (e.g., location check-in for A and point-of-interest search for B). Then, the composition $(\mathbf{Q}_A, \mathbf{Q}_B)$ in parallel has the following property:

Proposition 12 (Compatibility with LDP)**.** *If $\mathbf{Q}_A$ and $\mathbf{Q}_B$ respectively provide $(\mathcal{X}_S, \mathcal{Y}_P, \varepsilon_A)$-ULDP and ε_B-LDP, then for any $x, x' \in \mathcal{X}$, $y_A \in \mathcal{Y}_P$, and $y_B \in \mathcal{Y}$, we have:*

$$(\mathbf{Q}_A, \mathbf{Q}_B)(y_A, y_B|x) \leq e^{\varepsilon_A + \varepsilon_B} (\mathbf{Q}_A, \mathbf{Q}_B)(y_A, y_B|x').$$

Proposition 12 implies that Alice's sensitive personal data in $\mathcal{X}_S$ is protected by $(\varepsilon_A + \varepsilon_B)$-LDP after the data integration.

B Relationship between LDP, ULDP and OSLDP

In this section, we introduce the notion of OSLDP (One-sided LDP), a local model version of OSDP (One-sided DP) proposed in a preprint [17]:

Definition 6 ($(\mathcal{X}_S, \varepsilon)$-OSLDP)**.** *Given $\mathcal{X}_S \subseteq \mathcal{X}$ and $\varepsilon \in \mathbb{R}_{\geq 0}$, an obfuscation mechanism $\mathbf{Q}$ from $\mathcal{X}$ to $\mathcal{Y}$ provides $(\mathcal{X}_S, \varepsilon)$-OSLDP if for any $x \in \mathcal{X}_S$, any $x' \in \mathcal{X}$ and any $y \in \mathcal{Y}$, we have*

$$\mathbf{Q}(y|x) \leq e^{\varepsilon} \mathbf{Q}(y|x'). \tag{23}$$

OSLDP is a special case of OSDP [17] that takes as input personal data of a single user. Unlike ULDP, OSLDP allows the transition probability $\mathbf{Q}(y|x')$ from non-sensitive data $x' \in \mathcal{X}_N$ to be very large for any $y \in \mathcal{Y}$, and hence does not provide ε-LDP for $\mathcal{Y}$ (whereas ULDP provides ε-LDP for $\mathcal{Y}_P$). Thus, OSLDP can be regarded as a "relaxation" of ULDP. In fact, the following proposition holds:

Proposition 13. *If an obfuscation mechanism $\mathbf{Q}$ provides $(\mathcal{X}_S, \mathcal{Y}_P, \varepsilon)$-ULDP, then it also provides $(\mathcal{X}_S, \varepsilon)$-OSLDP.*

It should be noted that if an obfuscation mechanism provides ε-LDP, then it obviously provides $(\mathcal{X}_S, \mathcal{Y}_P, \varepsilon)$-ULDP, where $\mathcal{Y}_P = \mathcal{Y}$. Therefore, $(\mathcal{X}_S, \mathcal{Y}_P, \varepsilon)$-ULDP is a privacy measure that lies between ε-LDP and $(\mathcal{X}_S, \varepsilon)$-OSLDP.

We use ULDP instead of OSLDP for the following two reasons. The first reason is that ULDP is compatible with LDP, and makes it possible to perform data integration and data analysis under LDP (Proposition 12). OSLDP does not have this property in general, since it allows the transition probability $\mathbf{Q}(y|x')$ from non-sensitive data $x' \in \mathcal{X}_N$ to be very large for any $y \in \mathcal{Y}$, as explained above.

The second reason, which is more important, is that *the utility of OSLDP is not better than that of ULDP*. Intuitively, it can be explained as follows. First, although $\mathcal{Y}_P$ is not explicitly defined in OSLDP, we can define $\mathcal{Y}_P$ in OSLDP as the *image of* $\mathcal{X}_S$, and $\mathcal{Y}_I$ as $\mathcal{Y}_I = \mathcal{Y} \setminus \mathcal{Y}_P$, analogously to ULDP. Then, OSLDP differs from ULDP in the following two points: (i) it allows the transition probability $\mathbf{Q}(y|x')$ from $x' \in \mathcal{X}_N$ to $y \in \mathcal{Y}_P$ to be very large (i.e., (5) may not satisfied); (ii) it allows $y \in \mathcal{Y}_I$ to be non-invertible. (i.e., (4) may not satisfied). Regarding (i), it is important to note that the transition probability from $x' \in \mathcal{X}_N$ to $\mathcal{Y}_I$ decreases with increase in the transition probability from x' to $\mathcal{Y}_P$. Thus, (i) and (ii) only allow us to mix non-sensitive data with sensitive data or other non-sensitive data, and reduce the amount of output data $y \in \mathcal{Y}_I$ that can be inverted to $x \in \mathcal{X}_N$.

Then, each OSLDP mechanism can be decomposed into a ULDP mechanism and a randomized post-processing that mixes non-sensitive data with sensitive data or other non-sensitive data. Note that this post-processing does not preserve data types (in Definition 5), and hence OSLDP does not have a compatibility with LDP as explained above. In

addition, although the post-processing might improve privacy for non-sensitive data, we would like to protect sensitive data in this paper and ULDP is sufficient for this purpose; i.e., it guarantees ε-LDP for sensitive data.

Since the information is generally lost (never gained) by mixing data via the randomized post-processing, the utility of OSLDP is not better than that of ULDP (this holds for the information-theoretic utility such as mutual information and f-divergences [30] because of the data processing inequality [9, 13]; we also show this for the expected l_1 and l_2 losses at the end of Appendix B). Thus, it suffices to consider ULDP for our goal of designing obfuscation mechanisms that achieve high utility while providing LDP for sensitive data (as tdescribed in Section 1).

We now formalize our claim as follows:

Proposition 14. *Let $\mathcal{M}_O$ be the class of all mechanisms from $\mathcal{X}$ to $\mathcal{Y}$ providing $(\mathcal{X}_S,\varepsilon)$-OSLDP. For any $\mathbf{Q}_O \in \mathcal{M}_O$, there exist two sets $\mathcal{Z}$ and $\mathcal{Z}_P$, a $(\mathcal{X}_S,\mathcal{Z}_P,\varepsilon)$-ULDP mechanism $\mathbf{Q}_U$ from $\mathcal{X}$ to $\mathcal{Z}$, and a randomized algorithm $\mathbf{Q}_R$ from $\mathcal{Z}$ to $\mathcal{Y}$ such that:*

$$\mathbf{Q}_O = \mathbf{Q}_R \circ \mathbf{Q}_U. \tag{24}$$

From Proposition 14, we show that the expected l_1 and l_2 losses of OSLDP are not better than those of ULDP as follows. For any OSLDP mechanism $\mathbf{Q}_O \in \mathcal{M}_O$ and any estimation method λ_O from data in $\mathcal{Y}$, we can construct a ULDP mechanism $\mathbf{Q}_U$ in (24) and an estimation method λ_U that perturbs data in $\mathcal{Z}$ via $\mathbf{Q}_R$ and then estimates a distribution from data in $\mathcal{Y}$ via λ_O. $\mathbf{Q}_U$ and λ_U provide the same expected l_1 and l_2 losses as $\mathbf{Q}_O$ and λ_O, and there might also exist ULDP mechanisms and estimation methods from data in $\mathcal{Z}$ that provide smaller expected l_1 and l_2 losses. Thus, the expected l_1 and l_2 losses of OSLDP are not better than those of ULDP.

C L2 loss of the utility-optimized Mechanisms

C.1 Utility Analysis

uRR in the general case. We first present the l_2 loss of the $(\mathcal{X}_S,\varepsilon)$-uRR.

Proposition 15 (l_2 loss of the uRR)**.** *The expected l_2 loss of the $(\mathcal{X}_S,\varepsilon)$-uRR mechanism is given by:*

$$\mathbb{E}[l_2^2(\hat{\mathbf{p}},\mathbf{p})] = \frac{2(e^\varepsilon-1)(|\mathcal{X}_S|-\mathbf{p}(\mathcal{X}_S))+|\mathcal{X}_S|(|\mathcal{X}_S|-1)}{n(e^\varepsilon-1)^2} + \frac{1}{n}\Big(1-\sum_{x\in\mathcal{X}}\mathbf{p}(x)^2\Big). \tag{25}$$

When $0<\varepsilon<\ln(|\mathcal{X}_N|+1)$, the l_2 loss is maximized by the uniform distribution $\mathbf{p}_{U_N}$ over $\mathcal{X}_N$.

Proposition 16. *For any $0<\varepsilon<\ln(|\mathcal{X}_N|+1)$, (25) is maximized by $\mathbf{p}_{U_N}$:*

$$\mathbb{E}\left[l_2^2(\hat{\mathbf{p}},\mathbf{p})\right] \le \mathbb{E}\left[l_2^2(\hat{\mathbf{p}},\mathbf{p}_{U_N})\right] = \frac{|\mathcal{X}_S|(|\mathcal{X}_S|+2e^\varepsilon-3)}{n(e^\varepsilon-1)^2} + \frac{1}{n}\Big(1-\frac{1}{|\mathcal{X}_N|}\Big). \tag{26}$$

When $\varepsilon \ge \ln(|\mathcal{X}_N|+1)$, the l_2 loss is maximized by a mixture of the uniform distribution $\mathbf{p}_{U_S}$ over $\mathcal{X}_S$ and the uniform distribution $\mathbf{p}_{U_N}$ over $\mathcal{X}_N$.

Proposition 17. *For any $\varepsilon \ge \ln(|\mathcal{X}_N|+1)$, (25) is maximized by $\mathbf{p}^*$ in (13):*

$$\mathbb{E}\left[l_2^2(\hat{\mathbf{p}},\mathbf{p})\right] \le \mathbb{E}\left[l_2^2(\hat{\mathbf{p}},\mathbf{p}^*)\right] = \frac{(|\mathcal{X}_S|+e^\varepsilon-1)^2}{n(e^\varepsilon-1)^2}\Big(1-\frac{1}{|\mathcal{X}|}\Big).$$

uRR in the high privacy regime. Consider the high privacy regime where $\varepsilon \approx 0$. In this case, $e^\varepsilon - 1 \approx \varepsilon$. By using this approximation, the right-hand side of (26) in Proposition 16 can be simplified as follows:

$$\mathbb{E}\left[l_2^2(\hat{\mathbf{p}},\mathbf{p})\right] \le \mathbb{E}\left[l_2^2(\hat{\mathbf{p}},\mathbf{p}_{U_N})\right] \approx \frac{|\mathcal{X}_S|(|\mathcal{X}_S|-1)}{n\varepsilon^2}.$$

It is shown in [29] that the expected l_2 loss of the ε-RR is at most $\frac{|\mathcal{X}|(|\mathcal{X}|-1)}{n\varepsilon^2}$ when $\varepsilon \approx 0$. Thus, the expected l_2 loss of the $(\mathcal{X}_S,\varepsilon)$-uRR is much smaller than that of the ε-RR when $|\mathcal{X}_S| \ll |\mathcal{X}|$.

uRR in the low privacy regime. Consider the low privacy regime where $\varepsilon = \ln|\mathcal{X}|$ and $|\mathcal{X}_S| \ll |\mathcal{X}|$. By Proposition 17, the expected l_2^2 loss of the $(\mathcal{X}_S,\varepsilon)$-uRR is given by:

$$\mathbb{E}\left[l_2^2(\hat{\mathbf{p}},\mathbf{p})\right] \le \mathbb{E}\left[l_2^2(\hat{\mathbf{p}},\mathbf{p}^*)\right] \approx \frac{1}{n}.$$

It should be noted that the expected l_2 loss of the non-private mechanism is at most $\frac{1}{n}(1-\frac{1}{|\mathcal{X}|})$ [29], and that $\frac{1}{n}(1-\frac{1}{|\mathcal{X}|}) \approx \frac{1}{n}$ when $|\mathcal{X}| \gg 1$. Thus, when $\varepsilon = \ln|\mathcal{X}|$ and $|\mathcal{X}_S| \ll |\mathcal{X}|$, the $(\mathcal{X}_S,\varepsilon)$-uRR achieves almost the same data utility as the non-private mechanism, whereas the expected l_1 loss of the ε-RR is four times larger than that of the non-private mechanism [29].

Utility-optimized RAPPOR in the general case. We then present the l_2 loss of the $(\mathcal{X}_S,\varepsilon)$-uRAP.

Proposition 18 (l_2 loss of the uRAP)**.** *Then the expected l_2-loss of the $(\mathcal{X}_S,\varepsilon)$-uRAP mechanism is given by:*

$$\mathbb{E}\left[l_2^2(\hat{\mathbf{p}},\mathbf{p})\right] = \frac{1}{n}\Big(1+\frac{(|\mathcal{X}_S|+1)e^{\varepsilon/2}-1}{(e^{\varepsilon/2}-1)^2} - \frac{1}{e^{\varepsilon/2}-1}\mathbf{p}(\mathcal{X}_S) - \sum_{j=1}^{|\mathcal{X}|}\mathbf{p}(x_j)^2\Big). \tag{27}$$

For any $0<\varepsilon<2\ln(\frac{|\mathcal{X}_N|}{2}+1)$, the l_2 loss is maximized by the uniform distribution $\mathbf{p}_{U_N}$ over $\mathcal{X}_N$.

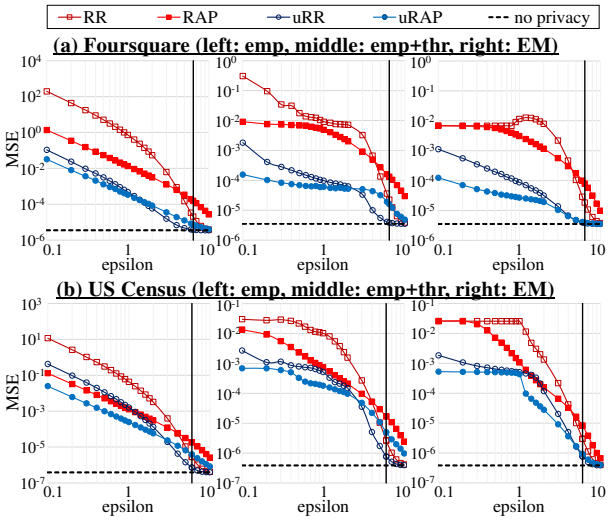


Figure 10: ε vs. MSE (common-mechanism). A bold line parallel to the y-axis represents $\varepsilon = \ln|\mathcal{X}|$.

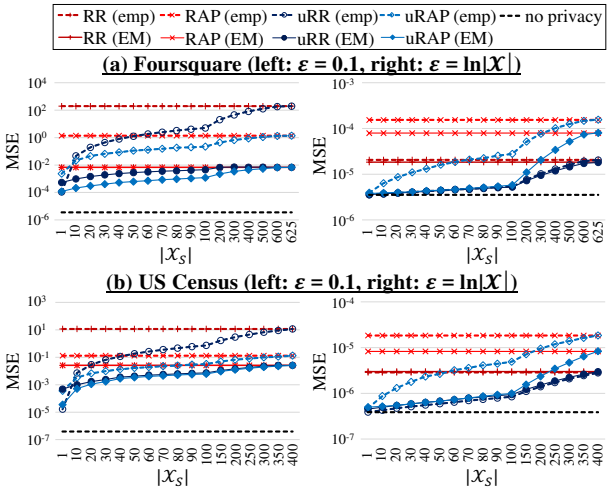


Figure 11: $|\mathcal{X}_S|$ vs. MSE when $\varepsilon = 0.1$ or $\ln|\mathcal{X}|$.

Proposition 19. *For any* $0 < \varepsilon < 2\ln(\frac{|\mathcal{X}_N|}{2} + 1)$*, the* l_2*-loss* $\mathbb{E}\left[l_2^2(\hat{\mathbf{p}}, \mathbf{p})\right]$ *is maximized when* $\mathbf{p} = \mathbf{p}_{U_N}$*:*

$$\mathbb{E}\left[l_2^2(\hat{\mathbf{p}}, \mathbf{p})\right] \leq \mathbb{E}\left[l_2^2(\hat{\mathbf{p}}, \mathbf{p}_{U_N})\right] = \frac{1}{n}\left(1 + \frac{(|\mathcal{X}_S|+1)e^{\varepsilon/2}-1}{(e^{\varepsilon/2}-1)^2} - \frac{1}{|\mathcal{X}_N|}\right). \quad (28)$$

uRAP in the high privacy regime. Consider the high privacy regime where $\varepsilon \approx 0$. In this case, $e^{\varepsilon/2} - 1 \approx \varepsilon/2$. By using this approximation, the right-hand side of (28) in Proposition 19 can be simplified as:

$$\mathbb{E}\left[l_2^2(\hat{\mathbf{p}}, \mathbf{p})\right] \leq \mathbb{E}\left[l_2^2(\hat{\mathbf{p}}, \mathbf{p}_{U_N})\right] \approx \frac{4|\mathcal{X}_S|}{n\varepsilon^2}.$$

Thus, the expected l_2 loss of the uRAP is at most $\frac{4|\mathcal{X}_S|}{n\varepsilon^2}$ in the high privacy regime. It is shown in [29] that the expected

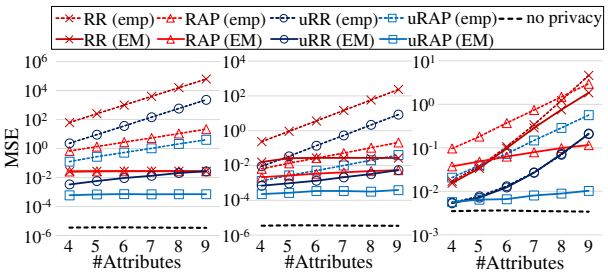


Figure 12: Number of attributes vs. MSE (US Census dataset; left: $\varepsilon = 0.1$, middle: $\varepsilon = 1.0$, right: $\varepsilon = 6.0$).

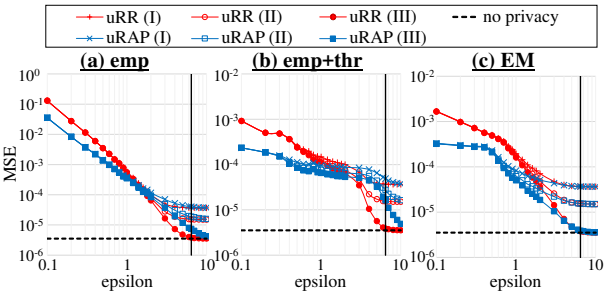


Figure 13: ε vs. MSE (personalized-mechanism) ((I): w/o knowledge, (II) POI distribution, (III) true distribution).

l_2 loss of the ε-RAPPOR is at most $\frac{4|\mathcal{X}|}{n\varepsilon^2}(1 - \frac{1}{|\mathcal{X}|})$ when $\varepsilon \approx 0$. Thus, the expected l_2 loss of the $(\mathcal{X}_S, \varepsilon)$-uRAP is much smaller than that of the ε-RAPPOR when $|\mathcal{X}_S| \ll |\mathcal{X}|$.

Note that the expected l_2 loss of the uRAP in the worst case can also be expressed as $\Theta(\frac{|\mathcal{X}_S|}{n\varepsilon^2})$ in this case. As described in Section 3.2, this is "order" optimal among all ULDP mechanisms.

uRAP in the low privacy regime. If $\varepsilon = \ln|\mathcal{X}|$ and $|\mathcal{X}_S| \ll \sqrt{|\mathcal{X}|}$, the right-hand side of (28) in Proposition 19 can be simplified as follows:

$$\mathbb{E}\left[l_2^2(\hat{\mathbf{p}}, \mathbf{p})\right] \leq \mathbb{E}\left[l_2^2(\hat{\mathbf{p}}, \mathbf{p}_{U_N})\right] \approx \frac{1}{n}. \quad (29)$$

Note that the expected l_2 loss of the non-private mechanism is at most $\frac{1}{n}(1 - \frac{1}{|\mathcal{X}|})$ [29], and that $\frac{1}{n}(1 - \frac{1}{|\mathcal{X}|}) \approx \frac{1}{n}$ when $|\mathcal{X}| \gg 1$. Thus, when $\varepsilon = \ln|\mathcal{X}|$ and $|\mathcal{X}_S| \ll \sqrt{|\mathcal{X}|}$, the $(\mathcal{X}_S, \varepsilon)$-uRAP achieves almost the same data utility as the non-private mechanism, whereas the expected l_2 loss of the ε-RAPPOR is $\sqrt{|\mathcal{X}|}$ times larger than that of the non-private mechanism [29].

C.2 Experimental Results of the MSE

Figures 10, 11, 12, and 13 show the results of the MSE corresponding to Figures 5, 6, 7, and 8, respectively. It can be seen that a tendency similar to the results of the TV is obtained for the results of the MSE, meaning that our proposed methods are effective in terms of both the l_1 and l_2 losses.

Evaluating Differentially Private Machine Learning in Practice

Bargav Jayaraman and David Evans
Department of Computer Science
University of Virginia

Abstract

Differential privacy is a strong notion for privacy that can be used to prove formal guarantees, in terms of a privacy budget, ϵ, about how much information is leaked by a mechanism. When used in privacy-preserving machine learning, the goal is typically to limit what can be inferred from the model about individual training records. However, the calibration of the privacy budget is not well understood. Implementations of privacy-preserving machine learning often select large values of ϵ in order to get acceptable utility of the model, with little understanding of the impact of such choices on meaningful privacy. Moreover, in scenarios where iterative learning procedures are used, relaxed definitions of differential privacy are often used which appear to reduce the needed privacy budget but present poorly understood trade-offs between privacy and utility. In this paper, we quantify the impact of these choices on privacy in experiments with logistic regression and neural network models. Our main finding is that there is no way to obtain privacy for free—relaxed definitions of differential privacy that reduce the amount of noise needed to improve utility also increase the measured privacy leakage. Current mechanisms for differentially private machine learning rarely offer acceptable utility-privacy trade-offs for complex learning tasks: settings that provide limited accuracy loss provide little effective privacy, and settings that provide strong privacy result in useless models.

1 Introduction

Differential privacy has become a de facto privacy standard, and nearly all works on privacy-preserving machine learning use some form of differential privacy. These works include designs for differentially private versions of prominent machine learning algorithms including empirical risk minimization [11, 12] and deep neural networks [1, 60].

While many methods for achieving differential privacy have been proposed, it is not well understood how to use these methods in practice. In particular, there is little concrete guidance on how to choose an appropriate privacy budget ϵ, and limited understanding of how variants of the differential privacy definition impact privacy in practice. As a result, privacy-preserving machine learning implementations tend to choose arbitrary values for ϵ as needed to achieve acceptable model utility. For instance, the implementation of Shokri and Shmatikov [60] requires ϵ proportional to the size of the target deep learning model, which could be in the order of few millions. Setting ϵ to such arbitrarily large values severely undermines privacy, although there is no consensus on a hard threshold value for ϵ above which formal guarantees differential privacy provides become meaningless in practice.

One proposed way to improve utility for a given privacy budget is to relax the definition of differential privacy. Several relaxed definitions of differential privacy have been proposed that are shown to provide better utility even for small ϵ values [9, 18, 49]. How much privacy leakage these relaxations allow in adversarial scenarios, however, is not well understood. We shed light on this question by evaluating the relaxed differential privacy notions for different choices of ϵ values and empirically measuring privacy leakage, including how many individual training records are exposed by membership inference attacks on different models.

Contributions. Our main contribution is the evaluation of differential privacy mechanisms for machine learning to understand the impact of different choices of ϵ and different relaxations of differential privacy on both utility and privacy. We focus our evaluation on gradient perturbation mechanisms, which are applicable to a wide class of machine learning algorithms including empirical risk minimization (ERM) algorithms such as logistic regression and deep learning (Section 2.2). Our experiments cover three popular differential privacy relaxations: differential privacy with advanced composition, zero-concentrated differential privacy [9], and Rényi differential privacy [49] (described in Section 2.1). These variations allow for tighter analysis of cumulative privacy loss, thereby reducing the noise that must be added in the training process. We evaluate the concrete privacy loss of these variations using

membership inference attacks [61, 74] and attribute inference attacks [74] (Section 3). While the model utility increases with the privacy budget, increasing the privacy budget also increases the success rate of inference attacks. Hence, we aim to find the range of values of ϵ which achieves a balance between utility and privacy, and also to evaluate the concrete privacy leakage in terms of the number of individual members of the training data at risk of exposure. We study both logistic regression and neural network models, on two multi-class classification data sets. Our key findings (Section 4) quantify the practical risks of using different differential privacy notions across a range of privacy budgets.

Related work. Orthogonal to our work, Ding et al. [13] and Hay et al. [26] evaluate the existing differential privacy implementations for the *correctness* of implementation. Whereas, we assume correct implementations and aim to evaluate the impact of the privacy budget and choice of differential privacy variant. While Carlini et al. [10] also explore the effectiveness of differential privacy against attacks, they do not explicitly answer what values of ϵ should be used nor do they evaluate the privacy leakage of the relaxed definitions. Li et al. [42] raise concerns about relaxing the differential privacy notion in order to achieve better overall utility, but do not evaluate the leakage. We perform a thorough evaluation of the differential privacy variations and quantify their leakage for different privacy budgets. The work of Rahman et al. [58] is most closely related to our work. It evaluates differential privacy implementations against membership inference attacks, but does not evaluate the privacy leakage of relaxed variants of differential privacy. Ours is the first work to experimentally measure the excess privacy leakage due to the relaxed notions of differential privacy.

2 Differential Privacy for Machine Learning

Next, we review the definition of differential privacy and its relaxed variants. Section 2.2 surveys mechanisms for achieving differentially private machine learning. Section 2.3 summarizes applications of differential privacy to machine learning and surveys implementations' choices about privacy budgets.

2.1 Background on Differential Privacy

Differential privacy is a probabilistic privacy mechanism that provides an information-theoretic security guarantee. Dwork [16] gives the following definition:

Definition 2.1 ((ϵ, δ)-Differential Privacy)**.** Given two neighboring data sets D and D' differing by one record, a mechanism $\mathcal{M}$ preserves (ϵ, δ)-differential privacy if

$$Pr[\mathcal{M}(D) \in S] \leq Pr[\mathcal{M}(D') \in S] \times e^{\epsilon} + \delta$$

where ϵ is the privacy budget and δ is the failure probability.

When $\delta = 0$ we achieve a strictly stronger notion of ϵ-differential privacy.

The quantity

$$\ln \frac{Pr[\mathcal{M}(D) \in S]}{Pr[\mathcal{M}(D') \in S]}$$

is called the *privacy loss*.

One way to achieve ϵ-DP and (ϵ, δ)-DP is to add noise sampled from Laplace and Gaussian distributions respectively, where the noise is proportional to the *sensitivity* of the mechanism $\mathcal{M}$:

Definition 2.2 (Sensitivity)**.** For two neighboring data sets D and D' differing by one record, the sensitivity of $\mathcal{M}$ is the maximum change in the output of $\mathcal{M}$ over all possible inputs:

$$\Delta\mathcal{M} = \max_{D, D', \|D - D'\|_1 = 1} \|\mathcal{M}(D) - \mathcal{M}(D')\|$$

where $\|\cdot\|$ is a norm of the vector. Throughout this paper we assume ℓ_2-sensitivity which considers the upper bound on the ℓ_2-norm of $\mathcal{M}(D) - \mathcal{M}(D')$.

Composition. Differential privacy satisfies a simple composition property: when two mechanisms with privacy budgets ϵ_1 and ϵ_2 are performed on the same data, together they consume a privacy budget of $\epsilon_1 + \epsilon_2$. Thus, composing multiple differentially private mechanisms leads to a linear increase in the privacy budget (or corresponding increases in noise to maintain a fixed ϵ total privacy budget).

Relaxed Definitions. Dwork [17] showed that this linear composition bound on ϵ can be reduced at the cost of *slightly* increasing the failure probability δ. In essence, this relaxation considers the linear composition of *expected* privacy loss of mechanisms which can be converted to a cumulative privacy budget ϵ with high probability bound. Dwork defines this as the *advanced composition theorem*, and proves that it applies to any differentially private mechanism.

Three commonly-used subsequent relaxed versions of differential privacy are Concentrated Differential Privacy [18], Zero Concentrated Differential Privacy [9], and Rényi Differential Privacy [49]. All of these achieve tighter analysis of cumulative privacy loss by taking advantage of the fact that the privacy loss random variable is strictly centered around an *expected* privacy loss. The cumulative privacy budget obtained from these analyses bounds the worst case privacy loss of the composition of mechanisms with all but δ failure probability. This reduces the noise required and hence improves utility over multiple compositions. However, it is important to consider the actual impact these relaxations have on the privacy leakage, which is a main focus of this paper.

Dwork et al. [18] note that the privacy loss of a differentially private mechanism follows a sub-Gaussian distribution. In other words, the privacy loss is strictly distributed around the expected privacy loss and the spread is controlled by the variance of the sub-Gaussian distribution. Multiple compositions of differentially private mechanisms thus result in the

	Advanced Comp.	Concentrated (CDP)	Zero-Concentrated (zCDP)	Rényi (RDP)
Expected Loss	$\epsilon(e^\epsilon - 1)$	$\mu = \frac{\epsilon(e^\epsilon-1)}{2}$	$\zeta + \rho = \frac{\epsilon^2}{2}$	$2\epsilon^2$
Variance of Loss	ϵ^2	$\tau^2 = \epsilon^2$	$2\rho = \epsilon^2$	ϵ^2
Convert from ϵ-DP	-	$(\frac{\epsilon(e^\epsilon-1)}{2}, \epsilon)$-CDP	$(\frac{\epsilon^2}{2})$-zCDP	(α, ϵ)-RDP
Convert to DP	-	$(\mu + \tau\sqrt{2\log(1/\delta)}, \delta)$-DP†	$(\zeta + \rho + 2\sqrt{\rho\log(1/\delta)}, \delta)$-DP	$(\epsilon + \frac{\log(1/\delta)}{\alpha-1}, \delta)$-DP
Composition of k ϵ-DP Mechanisms	$(\epsilon\sqrt{2k\log(1/\delta)} + k\epsilon(e^\epsilon - 1), \delta)$-DP	$(\epsilon\sqrt{2k\log(1/\delta)} + k\epsilon(e^\epsilon - 1)/2, \delta)$-DP	$(\epsilon\sqrt{2k\log(1/\delta)} + k\epsilon^2/2, \delta)$-DP	$(4\epsilon\sqrt{2k\log(1/\delta)}, \delta)$-DP‡

Table 1: Comparison of Different Variations of Differential Privacy
Advanced composition is an implicit property of DP and hence there is no conversion to and from DP.
†. Derived indirectly via zCDP. ‡. When $\log(1/\delta) \geq \epsilon^2 k$.

aggregation of corresponding mean and variance values of the individual sub-Gaussian distributions. This can be converted to a cumulative privacy budget similar to the advanced composition theorem, which in turn reduces the noise that must be added to the individual mechanisms. The authors call this *concentrated differential privacy* [18]:

Definition 2.3 (Concentrated Differential Privacy (CDP))**.** A randomized algorithm $\mathcal{M}$ is (μ, τ)-concentrated differentially private if, for all pairs of adjacent data sets D and D',

$$\mathcal{D}_{subG}(\mathcal{M}(D) \parallel \mathcal{M}(D')) \leq (\mu, \tau)$$

where the sub-Gaussian divergence, $\mathcal{D}_{subG}$, is defined such that the expected privacy loss is bounded by μ and after subtracting μ, the resulting centered sub-Gaussian distribution has standard deviation τ. Any ϵ-DP algorithm satisfies $(\epsilon \cdot (e^\epsilon - 1)/2, \epsilon)$-CDP, however the converse is not true.

A variation on CDP, *zero-concentrated differential privacy* (zCDP) [9] uses Rényi divergence as a different method to show that the privacy loss random variable follows a sub-Gaussian distribution.

Definition 2.4 (Zero-Concentrated Differential Privacy (zCDP))**.** A randomized mechanism $\mathcal{M}$ is (ξ, ρ)-zero-concentrated differentially private if, for all neighbouring data sets D and D' and all $\alpha \in (1, \infty)$,

$$\mathcal{D}_\alpha(\mathcal{M}(D) \parallel \mathcal{M}(D')) \leq \xi + \rho\alpha$$

where $\mathcal{D}_\alpha(\mathcal{M}(D) \parallel \mathcal{M}(D'))$ is the α-Rényi divergence between the distribution of $\mathcal{M}(D)$ and the distribution of $\mathcal{M}(D')$.

$\mathcal{D}_\alpha$ also gives the α-th moment of the privacy loss random variable. For example, $\mathcal{D}_1$ gives the first order moment which is the mean or the expected privacy loss, and $\mathcal{D}_2$ gives the second order moment or the variance of privacy loss. There is a direct relation between DP and zCDP. If $\mathcal{M}$ satisfies ϵ-DP, then it also satisfies $(\frac{1}{2}\epsilon^2)$-zCDP. Furthermore, if $\mathcal{M}$ provides ρ-zCDP, it is $(\rho + 2\sqrt{\rho\log(1/\delta)}, \delta)$-DP for any $\delta > 0$.

The Rényi divergence allows zCDP to be mapped back to DP, which is not the case for CDP. However, Bun and Steinke [9] give a relationship between CDP and zCDP, which allows an indirect mapping from CDP to DP (Table 1).

The use of Rényi divergence as a metric to bound the privacy loss leads to the formulation of a more generic notion of Rényi differential privacy that is applicable to any individual moment of privacy loss random variable:

Definition 2.5 (Rényi Differential Privacy (RDP) [49])**.** A randomized mechanism $\mathcal{M}$ is said to have ϵ-Rényi differential privacy of order α (which can be abbreviated as (α, ϵ)-RDP), if for any adjacent data sets D, D' it holds that

$$\mathcal{D}_\alpha(\mathcal{M}(D) \parallel \mathcal{M}(D')) \leq \epsilon.$$

The main difference is that CDP and zCDP linearly bound *all* positive moments of privacy loss, whereas RDP bounds one moment at a time, which allows for a more accurate numerical analysis of privacy loss [49]. If $\mathcal{M}$ is an (α, ϵ)-RDP mechanism, it also satisfies $(\epsilon + \frac{\log 1/\delta}{\alpha-1}, \delta)$-DP for any $0 < \delta < 1$.

Table 1 compares the relaxed variations of differential privacy. For all the variations, the privacy budget grows sub-linearly with the number of compositions k.

Moments Accountant. Motivated by relaxations of differential privacy, Abadi et al. [1] propose the *moments accountant* (MA) mechanism for bounding the cumulative privacy loss of differentially private algorithms. The moments accountant keeps track of a bound on the moments of the privacy loss random variable during composition. Though the authors do not formalize this as a relaxed definition, their definition of the moments bound is analogous to the Rényi divergence [49]. Thus, the moments accountant can be considered as an instantiation of Rényi differential privacy. The moments accountant is widely used for differentially private deep learning due to its practical implementation in the TensorFlow Privacy library [2] (see Section 2.3 and Table 4).

2.2 Differential Privacy Methods for ML

This section summarizes methods for modifying machine learning algorithms to satisfy differential privacy. First, we

Data: Training data set (X, y)
Result: Model parameters θ
$\theta \leftarrow$ Init(0)
#1. Add noise here: *objective perturbation*
$J(\theta) = \frac{1}{n}\sum_{i=1}^{n} \ell(\theta, X_i, y_i) + \lambda R(\theta) + \beta$
for *epoch* **in** *epochs* **do**
  #2. Add noise here: *gradient perturbation*
  $\theta = \theta - \eta(\nabla J(\theta) + \beta)$
end
#3. Add noise here: *output perturbation*
return $\theta + \beta$

Algorithm 1: Privacy noise mechanisms.

review convex optimization problems, such as empirical risk minimization (ERM) algorithms, and show several methods for achieving differential privacy during the learning process. Next, we discuss methods that can be applied to non-convex optimization problems, including deep learning.

ERM. Given a training data set (X, y), where X is a feature matrix and y is the vector of class labels, an ERM algorithm aims to reduce the convex objective function of the form,

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n} \ell(\theta, X_i, y_i) + \lambda R(\theta),$$

where $\ell(\cdot)$ is a convex loss function (such as mean square error (MSE) or cross-entropy loss) that measures the training loss for a given θ, and $R(\cdot)$ is a regularization function. Commonly used regularization functions include ℓ_1 penalty, which makes the vector θ sparse, and ℓ_2 penalty, which shrinks the values of θ vector.

The goal of the algorithm is to find the optimal θ^* that minimizes the objective function: $\theta^* = \arg\min_\theta J(\theta)$. While many first order [14, 37, 57, 76] and second order [40, 43] methods exist to solve this minimization problem, the most basic procedure is gradient descent where we iteratively calculate the gradient of $J(\theta)$ with respect to θ and update θ with the gradient information. This process is repeated until $J(\theta) \approx 0$ or some other termination condition is met.

There are three obvious candidates for where to add privacy-preserving noise during this training process, demarcated in Algorithm 1. First, we could add noise to the objective function $J(\theta)$, which gives us the *objective perturbation mechanism* (#1 in Algorithm 1). Second, we could add noise to the gradients at each iteration, which gives us the *gradient perturbation mechanism* (#2). Finally, we can add noise to θ^* obtained after the training, which gives us the *output perturbation mechanism* (#3). While there are other methods of achieving differential privacy such as input perturbation [15], sample-aggregate framework [51], exponential mechanism [48] and teacher ensemble framework [52]. We focus our experimental analysis on gradient perturbation since it is applicable to all machine learning algorithms in general and is widely used for deep learning with differential privacy.

The amount of noise that must be added depends on the sensitivity of the machine learning algorithm. For instance, consider logistic regression with ℓ_2 regularization penalty. The objective function is of the form:

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n} \log(1 + e^{-X_i^\top \theta y_i}) + \frac{\lambda}{2} \| \theta \|_2^2$$

Assume that the training features are bounded, $\|X_i\|_2 \leq 1$ and $y_i \in \{-1, 1\}$. Chaudhuri et al. [12] prove that for this setting, objective perturbation requires sampling noise in the scale of $\frac{2}{n\epsilon}$, and output perturbation requires sampling noise in the scale of $\frac{2}{n\lambda\epsilon}$. The gradient of the objective function is:

$$\nabla J(\theta) = \frac{1}{n}\sum_{i=1}^{n} \frac{-X_i y_i}{1 + e^{X_i^\top \theta y_i}} + \lambda\theta$$

which has a sensitivity of $\frac{2}{n}$. Thus, gradient perturbation requires sampling noise in the scale of $\frac{2}{n\epsilon}$ at each iteration.

Deep learning. Deep learning follows the same learning procedure as in Algorithm 1, but the objective function is non-convex. As a result, the sensitivity analysis methods of Chaudhuri et al. [12] do not hold as they require a strong convexity assumption. Hence, their output and objective perturbation methods are not applicable. An alternative approach is to replace the non-convex function with a convex polynomial function [55, 56], and then use the standard objective perturbation. This approach requires carefully designing convex polynomial functions that can approximate the non-convexity, which can still limit the model's learning capacity. Moreover, it would require a considerable change in the existing machine learning infrastructure.

A simpler and more popular approach is to add noise to the gradients. Application of gradient perturbation requires a bound on the gradient norm. Since the gradient norm can be unbounded in deep learning, gradient perturbation can be used after manually clipping the gradients at each iteration. As noted by Abadi et al. [1], norm clipping provides a sensitivity bound on the gradients which is required for generating noise in gradient perturbation.

2.3 Implementing Differential Privacy

This section surveys how differential privacy has been used in machine learning applications, with a particular focus on the compromises implementers have made to obtain satisfactory utility. While the effective privacy provided by differential privacy mechanisms depends crucially on the choice of privacy budget ϵ, setting the ϵ value is discretionary and higher privacy budgets provide better utility.

Some of the early data analytics works on frequent pattern mining [7, 41], decision trees [21], private record linkage [30]

	Perturbation	Data Set	n	d	ϵ
Chaudhuri et al. [12]	Output and Objective	Adult	45,220	105	0.2
		KDDCup99	70,000	119	0.2
Pathak et al. [54]	Output	Adult	45,220	105	0.2
Hamm et al. [25]	Output	KDDCup99	493,000	123	1.0
		URL	200,000	50	1.0
Zhang et al. [78]	Objective	US	370,000	14	0.8
		Brazil	190,000	14	0.8
Jain and Thakurta [33]	Objective	CoverType	500,000	54	0.5
		KDDCup2010	20,000	2M	0.5
Jain and Thakurta [34]	Output and Objective	URL	100,000	20M	0.1
		COD-RNA	60,000	8	0.1
Song et al. [63]	Gradient	KDDCup99	50,000	9	1.0
		MNIST†	60,000	15	1.0
Wu et al. [70]	Output	Protein	72,876	74	0.05
		CoverType	498,010	54	0.05
Jayaraman et al. [35]	Output	Adult	45,220	104	0.5
		KDDCup99	70,000	122	0.5

Table 2: Simple ERM Methods which achieve High Utility with Low Privacy Budget.
† While MNIST is normally a 10-class task, Song et al. [63] use this for '1 vs rest' binary classification.

and recommender systems [47] were able to achieve both high utility and privacy with ϵ settings close to 1. These methods rely on finding frequency counts as a sub-routine, and hence provide ϵ-differential privacy by either perturbing the counts using Laplace noise or by releasing the top frequency counts using the exponential mechanism [48]. Machine learning, on the other hand, performs much more complex data analysis, and hence requires higher privacy budgets to maintain utility.

Next, we cover simple binary classification works that use small privacy budgets ($\epsilon \leq 1$). Then we survey complex classification tasks which seem to require large privacy budgets. Finally, we summarize recent works that aim to perform complex tasks with low privacy budgets by using relaxed definitions of differential privacy.

Binary classification. The first practical implementation of a private machine learning algorithm was proposed by Chaudhuri and Monteleoni [11]. They provide a novel sensitivity analysis under strong convexity constraints, allowing them to use output and objective perturbation for binary logistic regression. Chaudhuri et al. [12] subsequently generalized this method for ERM algorithms. This sensitivity analysis method has since been used by many works for binary classification tasks under different learning settings (listed in Table 2). While these applications can be implemented with low privacy budgets ($\epsilon \leq 1$), they only perform learning in restricted settings such as learning with low dimensional data, smooth objective functions and strong convexity assumptions, and are only applicable to simple binary classification tasks.

There has also been considerable progress in generalizing privacy-preserving machine learning to more complex scenarios such as learning in high-dimensional settings [33, 34, 64], learning without strong convexity assumptions [65], or relaxing the assumptions on data and objective functions [62, 68, 77]. However, these advances are mainly of theoretical interest and only a few works provide implementations [33, 34].

Complex learning tasks. All of the above works are limited to convex learning problems with binary classification tasks. Adopting their approaches to more complex learning tasks requires higher privacy budgets (see Table 3). For instance, the online version of ERM as considered by Jain et al. [32] requires ϵ as high as 10 to achieve acceptable utility. From the definition of differential privacy, we can see that $\Pr[\mathcal{M}(D) \in S] \leq e^{10} \times \Pr[\mathcal{M}(D') \in S]$. In other words, even if the model's output probability is 0.0001 on a data set D' that doesn't contain the target record, the model's output probability can be as high as 0.9999 on a neighboring data set D that contains the record. This allows an adversary to infer the presence or absence of a target record from the training data with high confidence. Adopting these binary classification methods for multi-class classification tasks requires even higher ϵ values. As noted by Wu et al. [70], it would require training a separate binary classifier for each class. Finally, high privacy budgets are required for non-convex learning algorithms, such as deep learning [60, 79]. Since the output and objective perturbation methods of Chaudhuri et al. [12] are not applicable to non-convex settings, implementations of differentially private deep learning rely on gradient pertur-

	Task	Perturbation	Data Set	n	d	C	ϵ
Jain et al. [32]	Online ERM	Objective	Year	500,000	90	2	10
			CoverType	581,012	54	2	10
Iyengar et al. [31]	Binary ERM	Objective	Adult	45,220	104	2	10
	Binary ERM		KDDCup99	70,000	114	2	10
	Multi-Class ERM		CoverType	581,012	54	7	10
	Multi-Class ERM		MNIST	65,000	784	10	10
	High Dimensional ERM		Gisette	6,000	5,000	2	10
Phan et al. [55, 56]	Deep Learning	Objective	YesiWell	254	30	2	1
			MNIST	60,000	784	10	1
Shokri and Shmatikov [60]	Deep Learning	Gradient	MNIST	60,000	1,024	10	369,200
			SVHN	100,000	3,072	10	369,200
Zhao et al. [79]	Deep Learning	Gradient	US	500,000	20	2	100
			MNIST	60,000	784	10	100

Table 3: Classification Methods for Complex Tasks

	Task	DP Relaxation	Data Set	n	d	C	ϵ
Huang et al. [28]	ERM	MA	Adult	21,000	14	2	0.5
Jayaraman et al. [35]	ERM	zCDP	Adult	45,220	104	2	0.5
			KDDCup99	70,000	122	2	0.5
Park et al. [53]	ERM	zCDP and MA	Stroke	50,345	100	2	0.5
			LifeScience	26,733	10	2	2.0
			Gowalla	1,256,384	2	2	0.01
			OlivettiFace	400	4,096	2	0.3
Lee [39]	ERM	zCDP	Adult	48,842	124	2	1.6
			US	40,000	58	2	1.6
			Brazil	38,000	53	2	1.6
Geumlek et al. [23]	ERM	RDP	Abalone	2,784	9	2	1.0
			Adult	32,561	100	2	0.05
			MNIST	7,988	784	2	0.14
Beaulieu et al. [6]	Deep Learning	MA	eICU	4,328	11	2	3.84
			TCGA	994	500	2	6.11
Abadi et al. [1]	Deep Learning	MA	MNIST	60,000	784	10	2.0
			CIFAR	60,000	3,072	10	8.0
Yu et al. [75]	Deep Learning	MA	MNIST	60,000	784	10	21.5
			CIFAR	60,000	3,072	10	21.5
Papernot et al. [52]	Deep Learning	MA	MNIST	60,000	784	10	2.0
			SVHN	60,000	3,072	10	8.0
Geyer et al. [24]	Deep Learning	MA	MNIST	60,000	784	10	8.0
Bhowmick et al. [8]	Deep Learning	MA	MNIST	60,000	784	10	3.0
			CIFAR	60,000	3,072	10	3.0
Hynes et al. [29]	Deep Learning	MA	CIFAR	50,000	3,072	10	4.0

Table 4: Gradient Perturbation based Classification Methods using Relaxed Notion of Differential Privacy

bation in their iterative learning procedure. These methods do not scale to large numbers of training iterations due to the composition theorem of differential privacy which causes the privacy budget to accumulate across iterations. The only exceptions are the works of Phan et al. [55, 56] that replace the non-linear functions in deep learning with polynomial approximations and then apply objective perturbation. With this transformation, they achieve high model utility for $\epsilon = 1$, as shown in Table 3. However, we note that this polynomial approximation is a non-standard approach to deep learning which can limit the model's learning capacity, and thereby diminish the model's accuracy for complex tasks.

Machine learning with relaxed DP definitions. To avoid the stringent composition property of differential privacy, several proposed privacy-preserving deep learning methods adopt the relaxed privacy definitions introduced in Section 2.1. Table 4 lists works that use gradient perturbation with relaxed notions of differential to reduce the overall privacy budget during iterative learning. The utility benefit of using relaxation is evident from the fact that the privacy budget for deep learning algorithms is significantly less than the prior works of Shokri and Shmatikov [60] and Zhao et al. [79] which do not use any relaxation.

While these *relaxed* definitions of differential privacy make complex iterative learning feasible for reasonable ϵ values, they might lead to more privacy leakage in practice. The main goal of our study is to evaluate the impact of implementation decisions regarding the privacy budget and relaxed definitions of differential privacy on the concrete privacy leakage that can be exploited by an attacker in practice. We do this by experimenting with various inference attacks, described in the next section.

3 Inference Attacks on Machine Learning

This section surveys the two types of inference attacks, *membership inference* (Section 3.1) and *attribute inference* (Section 3.2), and explains why they are useful metrics for evaluating privacy leakage. Section 3.3 briefly summarizes other relevant privacy attacks on machine learning.

3.1 Membership Inference

The aim of a *membership inference* attack is to infer whether or not a given record is present in the training set. Membership inference attacks can uncover highly sensitive information from training data. An early membership inference attack showed that it is possible to identify individuals contributing DNA to studies that analyze a mixture of DNA from many individuals, using a statistical distance measure to determine if a known individual is in the mixture [27].

Membership inference attacks can either be completely black-box where an attacker only has query access to the target model [61], or can assume that the attacker has full white-box access to the target model, along with some auxiliary information [74]. The first membership inference attack on machine learning was proposed by Shokri et al. [61]. They consider an attacker who can query the target model in a black-box way to obtain confidence scores for the queried input. The attacker tries to exploit the confidence score to determine whether the query input was present in the training data. Their attack method involves first training shadow models on a labelled data set, which can be generated either via black-box queries to the target model or through assumptions about the underlying distribution of training set. The attacker then trains an attack model using the shadow models to distinguish whether or not an input record is in the shadow training set. Finally, the attacker makes API calls to the target model to obtain confidence scores for each given input record and infers whether or not the input was part of the target model's training set. The inference model distinguishes the target model's predictions for inputs that are in its training set from those it did not train on. The key assumption is that the confidence score of the target model is higher for the training instances than it would be for arbitrary instances not present in the training set. This can be due to the generalization gap, which is prominent in models that overfit to training data.

A more targeted approach was proposed by Long et al. [44] where the shadow models are trained with and without a targeted input record t. At inference time, the attacker can check if the input record t was present in the training set of target model. This approach tests the membership of a specific record more accurately than Shokri et al.'s approach [61]. Recently, Salem et al. [59] proposed more generic membership inference attacks by relaxing the requirements of Shokri et al. [61]. In particular, requirements on the number of shadow models, knowledge of training data distribution and the target model architecture can be relaxed without substantially degrading the effectiveness of the attack.

Yeom et al. [74] recently proposed a more computationally efficient membership inference attack when the attacker has access to the target model and knows the average training loss of the model. To test the membership of an input record, the attacker evaluates the loss of the model on the input record and then classifies it as a member if the loss is smaller than the average training loss.

Connection to Differential Privacy. Differential privacy, by definition, aims to obfuscate the presence or absence of a record in the data set. On the other hand, membership inference attacks aim to identify the presence or absence of a record in the data set. Thus, intuitively these two notions counteract each other. Li et al. [42] point to this fact and provide a direct relationship between differential privacy and membership inference attacks. Backes et al. [4] studied membership inference attacks on microRNA studies and showed that differential privacy can reduce the success of membership

inference attacks, but at the cost of utility.

Yeom et al. [74] formally define a membership inference attack as an adversarial game where a data element is selected from the distribution, which is randomly either included in the training set or not. Then, an adversary with access to the trained model attempts to determine if that element was used in training. The *membership advantage* is defined as the difference between the adversary's true and false positive rates for this game. The authors prove that if the learning algorithm satisfies ϵ-differential privacy, then the adversary's advantage is bounded by $e^{\epsilon} - 1$. Hence, it is natural to use membership inference attacks as a metric to evaluate the privacy leakage of differentially private algorithms.

3.2 Attribute Inference

The aim of an *attribute inference* attack (also called *model inversion*) is to learn hidden sensitive attributes of a test input given at least API access to the model and information about the non-sensitive attributes. Fredrikson et al. [20] formalize this attack in terms of maximizing the posterior probability estimate of the sensitive attribute. More concretely, for a test record x where the attacker knows the values of its non-sensitive attributes $x_1, x_2, \cdots x_{d-1}$ and all the prior probabilities of the attributes, the attacker obtains the output of the model, $f(x)$, and attempts to recover the value of the sensitive attribute x_d. The attacker essentially searches for the value of x_d that maximizes the posterior probability $P(x_d \mid x_1, x_2, \cdots x_{d-1}, f(x))$. The success of this attack is based on the correlation between the sensitive attribute, x_d, and the model output, $f(x)$.

Yeom et al. [74] also propose an attribute inference attack using the same principle they use for their membership inference attack. The attacker evaluates the model's empirical loss on the input instance for different values of the sensitive attribute, and reports the value which has the maximum posterior probability of achieving the empirical loss. The authors define the *attribute advantage* similarly to their definition of membership advantage for membership inference.

Fredrikson et al. [20] demonstrated attribute inference attacks that could identify genetic markers based on warfarin dosage output by a model with just black-box access to model API.[1] With additional access to confidence scores of the model (noted as white-box information by Wu et al. [69]), more complex tasks have been performed, such as recovering faces from the training data [19].

Connection to Differential Privacy. Differential privacy is mainly tailored to obfuscate the presence or absence of a record in a data set, by limiting the effect of any single record on the output of differential private model trained on the data set. Logically this definition also extends to attributes or features of a record. In other words, by adding sufficient differential privacy noise, we should be able to limit the effect of a sensitive attribute on the model's output. This relationship between records and attributes is discussed by Yeom et al. [74]. Hence, we include these attacks in our experiments.

[1] This application has stirred some controversy based on the warfarin dosage output by the model itself being sensitive information correlated to the sensitive genetic markers, hence the assumption on attacker's prior knowledge of warfarin dosage is somewhat unrealistic [46].

3.3 Other Attacks on Machine Learning

Apart from inference attacks, many other attacks have been proposed in the literature which try to infer specific information from the target model. The most relevant are memorization attacks, which try to exploit the ability of high capacity models to memorize certain sensitive patterns in the training data [10]. These attacks have been found to be thwarted by differential privacy mechanisms with very little noise ($\epsilon = 10^9$) [10].

Other privacy attacks include model stealing, hyperparameter stealing, and property inference attacks. A model stealing attack aims to recover the model parameters via black-box access to the target model, either by adversarial learning [45] or by equation solving attacks [66]. Hyperparameter stealing attacks try to recover the underlying hyperparameters used during the model training, such as regularization coefficient [67] or model architecture [72]. These hyperparameters are intellectual property of commercial organizations that deploy machine learning models as a service, and hence these attacks are regarded as a threat to valuable intellectual property. A property inference attack tries to infer whether the training data set has a specific property, given a white-box access to the trained model. For instance, given access to a speech recognition model, an attacker can infer if the training data set contains speakers with a certain accent. Here the attacker can use the shadow training method of Shokri et al. [61] for distinguishing the presence and absence of a target property. These attacks have been performed on HMM and SVM models [3] and neural networks [22].

Though all these attacks may leak sensitive information about the target model or training data, the information leaked tends to be application-specific and is not clearly defined in a general way. For example, a property inference attack leaks some statistical property of the training data that is surprising to the model developer. Of course, the overall purpose of the model is to learn statistical properties from the training data. So, there is no general definition of a property inference attack without a prescriptive decision about which statistical properties of the training data should be captured by the model and which are sensitive to leak. In addition, the attacks mentioned in this section do not closely follow the threat model of differential privacy. Thus, we only consider inference attacks for our experimental evaluation.

In addition to these attacks, several poisoning and adversarial training attacks have been proposed [5, 50, 71, 73] which require an adversary that can actively interfere with the model

training process. We consider these out of scope for this paper, and assume a clean training process not under the control of the adversary.

4 Empirical Evaluation

To quantify the privacy leakage of the differentially private implementations for machine learning, we conduct experiments to measure how much an adversary can infer from a model. As motivated in Section 3, we measure privacy leakage using membership and attribute inference in our experiments. Note, however, that the conclusions we can draw from experiments like this are limited to showing a lower bound on the information leakage since they are measuring the effectiveness of a particular attack. Such experimental results cannot be used to make strong claims about what the best possible attack would be able to infer, especially in cases where an adversary has auxiliary information to help guide the attack. Evidence from our experiments, however, does provide clear evidence for when implemented privacy protections do not appear to provide sufficient privacy.

4.1 Experimental Setup

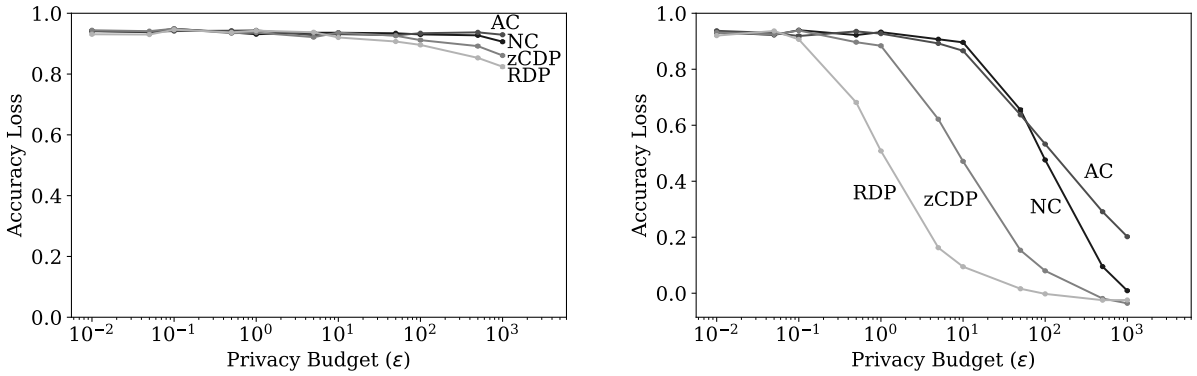
We evaluate the privacy leakage of two differentially private algorithms using gradient perturbation: logistic regression for empirical risk minimization (Section 4.2) and neural networks for non-convex learning (Section 4.3). For both, we consider the different relaxed notions of differential privacy and compare their privacy leakage. The variations that we implement are naïve composition (NC), advanced composition (AC), zero-concentrated differential privacy (zCDP) and Rényi differential privacy (RDP) (see Section 2.1 for details). We do not include CDP as it has the same composition property as zCDP (Table 1). For RDP, we use the RDP accountant (previously moments accountant) of TF Privacy framework [2].

We evaluate the models on two main metrics: *accuracy loss*, the model's accuracy loss on test set with respect to the non-private baseline, and *privacy leakage*, the attacker's advantage as defined by Yeom et al. [74]. To evaluate out the inference attack, we provide the attacker with a set of 20,000 records consisting of 10,000 records from training set and 10,000 records from the test set. We call records in the training set *members*, and the other records *non-members*. These labels are not known to the attacker. The task of the attacker is to predict whether or not a given input record belongs to the training set (i.e., if it is a member). The privacy leakage metric is calculated by taking the difference between the true positive rate (TPR) and the false positive rate (FPR) of the inference attack. Thus the privacy leakage metric is always between 0 and 1, where the value of 0 indicates that there is no leakage. For example, if an attacker performs membership inference on a model and obtains a privacy leakage of 0.7 then it implies that for every 100 wrong membership predictions made by the attacker, 170 'true' members are revealed to the attacker. In other words, 170 training records are revealed to the attacker. To better understand the potential impact of leakage, we also conduct experiments to estimate the actual number of members who are at risk for disclosure in a membership inference attack.

Data sets. We evaluate our models over two data sets for multi-class classification tasks: CIFAR-100 [38] and Purchase-100 [36]. CIFAR-100 consists of 28×28 images of 100 real world objects, with 500 instances of each object class. We use PCA to reduce the dimensionality of records to 50. The Purchase-100 data set consists of 200,000 customer purchase records of size 100 each (corresponding to the 100 frequently-purchased items) where the records are grouped into 100 classes based on the customers' purchase style. For both data sets, we use 10,000 randomly-selected instances for training and 10,000 randomly-selected non-training instances for the test set. The remaining records are used for training shadow models and inference model.

Attacks. For our experiments, we use the attack frameworks of Shokri et al. [61] and Yeom et al. [74] for membership inference and the method proposed by Yeom et al. [74] for attribute inference. In Shokri et al.'s framework [61], multiple shadow models are trained on data that is sampled from the same distribution as the private data set. These shadow models are used to train an inference model to identify whether an input record belongs to the private data set. The inference model is trained using a set of records used to train the shadow models, a set of records randomly selected from the distribution that are not part of the shadow model training, along with the confidence scores output by the shadow models for all of the input records. Using these inputs, the inference model learns to distinguish the training records from the non-training records. At the inference stage, the inference model takes an input record along with the confidence score of the target model on the input record, and outputs whether the input record belongs to the target model's private training data set. The intuition is that if the target model overfits on its training set, its confidence score for a training record will be higher than its confidence score for an otherwise similar input that was not used in training. The inference model tries to exploit this property. In our instantiation of the attack framework, we use five shadow models which all have the same model architecture as the target model. Our inference model is a neural network with two hidden layers of size 64. This setting is consistent with the original work [61].

The attack framework of Yeom et al. [74] is simpler than Shokri et al.'s design. It assumes a white-box attacker with access to the target model's expected training loss on the private data set, in addition to having access to the target model. For membership inference, the attacker simply observes the target model's loss on the input record. The attacker classifies the record as a member if the loss is smaller than the target



(a) Batch gradient clipping

(b) Per-instance gradient clipping

Figure 1: Impact of clipping on accuracy loss of logistic regression (CIFAR-100).

model's expected training loss, otherwise the record is classified as a non-member. The same principle is used for attribute inference. Given an input record, the attacker brute-forces all possible values for the unknown private attribute and observes the target model's loss, outputting the value for which the loss is closest to the target's expected training loss. Since there are no attributes in our data sets that are explicitly annotated as private, we randomly choose five attributes, and perform the attribute inference attack on each attribute independently, and report the averaged results.

Hyperparameters. For both data sets, we train logistic regression and neural network models with ℓ_2 regularization. First, we train a non-private model and perform a grid search over the regularization coefficient λ to find the value that minimizes the classification error on the test set. For CIFAR-100, we found optimal values to be $\lambda = 10^{-5}$ for logistic regression and $\lambda = 10^{-4}$ for neural network. For Purchase-100, we found optimal values to be $\lambda = 10^{-5}$ for logistic regression and $\lambda = 10^{-8}$ for neural network. Next, we fix this setting to train differentially private models using gradient perturbation. We vary ϵ between 0.01 and 1000 while keeping $\delta = 10^{-5}$, and report the accuracy loss and privacy leakage. The choice of $\delta = 10^{-5}$ satisfies the requirement that δ should be smaller than the inverse of the training set size 10,000. We use the ADAM optimizer for training and fix the learning rate to 0.01 with a batch size of 200. Due to the random noise addition, all the experiments are repeated five times and the average results and standard errors are reported. We do not assume pre-trained model parameters, unlike the prior works of Abadi et al. [1] and Yu et al. [75].

Clipping. For gradient perturbation, clipping is required to bound the sensitivity of the gradients. We tried clipping at both the batch and per-instance level. Batch clipping is more computationally efficient and a standard practice in deep learning. On the other hand, per-instance clipping uses the privacy budget more efficiently, resulting in more accurate models for a given privacy budget. We use the TensorFlow Privacy framework [2] which implements both batch and per-instance clipping. We fix the clipping threshold at $C = 1$.

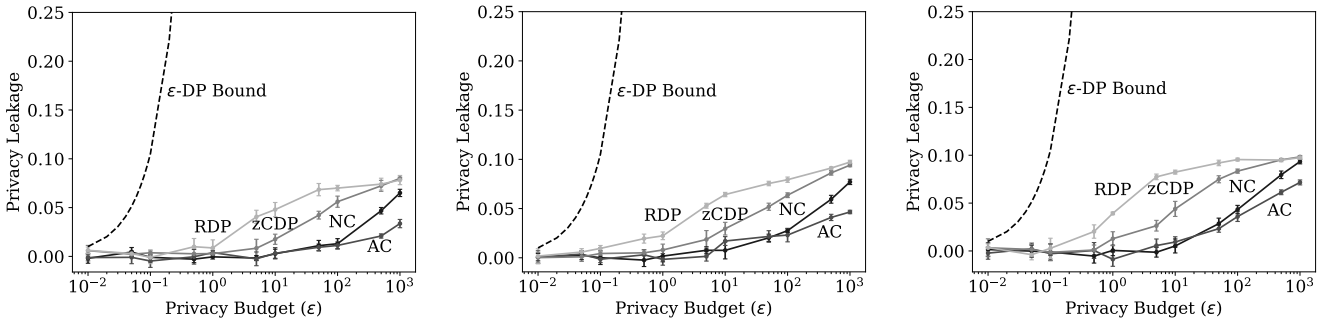
Figure 1 compares the accuracy loss of logistic regression models trained over CIFAR-100 data set with both batch clipping and per-instance clipping. Per-instance clipping allows learning more accurate models for all values of ϵ and amplifies the differences between the different mechanisms. For example, the model trained with RDP achieves accuracy close to the non-private model for $\epsilon = 100$ when performing per-instance clipping. Whereas, the models do not learn anything useful when using batch clipping. Hence, for the rest of the paper we only report the results for per-instance clipping.

4.2 Logistic Regression Results

We train ℓ_2-regularized logistic regression models on both the CIFAR-100 and Purchase-100 data sets.

CIFAR-100. The baseline model for non-private logistic regression achieves accuracy of 0.225 on training set and 0.155 on test set, which is competitive with the state-of-art neural network model [61] that achieves test accuracy close to 0.20 on CIFAR-100 after training on larger data set. Thus, there is a small generalization gap of 0.07, which the inference attacks try to exploit.

Figure 1(b) compares the accuracy loss for logistic regression models trained with different relaxed notions of differential privacy as we varying the privacy budget ϵ. The accuracy loss is normalized with respect to the accuracy of non-private model to clearly depict the model utility. An accuracy loss value of 1 means that the model has 100% loss and hence has no utility, whereas the value of 0 means that the model achieves same accuracy as the non-private baseline. As depicted in the figure, naïve composition achieves accuracy



(a) Shokri et al. membership inference (b) Yeom et al. membership inference (c) Yeom et al. attribute inference

Figure 2: Inference attacks on logistic regression (CIFAR-100).

close to 0.01 for $\epsilon \leq 10$ which is random guessing for 100-class classification. Naïve composition achieves accuracy loss close to 0 for $\epsilon = 1000$. Advanced composition adds more noise than naïve composition when privacy budget is greater than the number of training epochs ($\epsilon \geq 100$). The relaxations zCDP and RDP achieve accuracy loss close to 0 at $\epsilon = 500$ and $\epsilon = 50$ respectively, which is order of magnitudes smaller than the naïve composition. This is expected since the relaxed definitions require less added noise.

Figures 2(a) and 2(b) show the privacy leakage due to membership inference attacks on logistic regression models. Figure 2(a) shows results for the black-box attacker of Shokri et al. [61], which has access to the target model's confidence scores on the input record. Naïve composition achieves privacy leakage close to 0 for $\epsilon \leq 10$, and the leakage reaches 0.065 ± 0.004 for $\epsilon = 1000$. The relaxed variants RDP and zCDP have average leakage close to 0.080 ± 0.004 for $\epsilon = 1000$. As expected, the differential privacy variations have leakage in accordance with the amount of noise they add for a given ϵ. The plots also show the theoretical upper bound on the privacy leakage for ϵ-differential privacy, where the bound is $e^{\epsilon} - 1$ (see Section 3.1).

Figure 2(b) shows results for the white-box attacker of Yeom et al. [61], which has access to the target model's loss on the input record. As expected, zCDP and RDP relaxations leak the most. Naïve composition does not have any significant leakage for $\epsilon \leq 10$, but the leakage reaches 0.077 ± 0.003 for $\epsilon = 1000$. The observed leakage of all the variations is in accordance with the noise magnitude required for different differential privacy guarantees.

Figure 2(c) depicts the privacy leakage due to the attribute inference attack. The privacy leakage of RDP is highest, closely followed by zCDP. Naïve composition has low privacy leakage for $\epsilon \leq 10$ (attacker advantage of 0.005 ± 0.007 at $\epsilon = 10$), but it quickly increases to 0.093 ± 0.002 for $\epsilon = 1000$. But for meaningful privacy budgets, there is no significant leakage (< 0.02) for any of the methods. As expected, across all variations as privacy budgets increase both the attacker's advantage (privacy leakage) and the model utility (accuracy) increase. For this example, there is no choice of ϵ available that provides any effective privacy for a model that does better than random guessing.

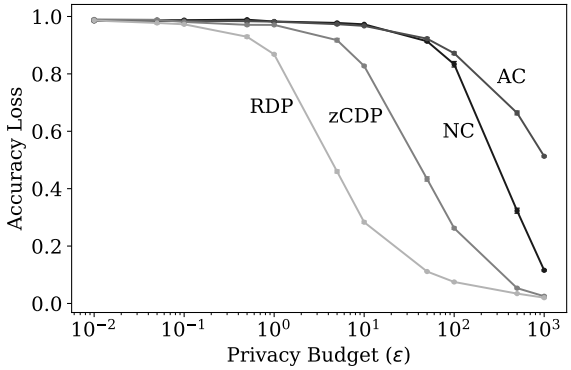


Figure 3: Accuracy loss of logistic regression (Purchase-100).

To gain more understanding of the impact of privacy leakage, Table 5 shows the actual number of training set members exposed to the attacker for different differential privacy variations. We assume the attacker has some limited tolerance for falsely exposing a member (that is, a bound on the acceptable false positive rate), and sets the required threshold score for the inference model output as the level needed to achieve that false positive rate. Then, we count the number of members in the private training data set for whom the inference model output exceeds that confidence threshold. Table 5 reports the number of members exposed to an adversary who tolerates false positive rates of 1%, 2%, and 5%. As we increase the tolerance threshold, there is a gradual increase in membership leakage for all the methods, and the leakage of relaxed variants increases drastically. Naïve composition and advanced composition are resistant to attack for $\epsilon \leq 10$, whereas zCDP is resistant to attack for $\epsilon \leq 1$. RDP is resistant up to $\epsilon = 0.05$.

	Naïve Composition				Advanced Composition				zCDP				RDP			
ϵ	Loss	1%	2%	5%	Loss	1%	2%	5%	Loss	1%	2%	5%	Loss	1%	2%	5%
0.01	.93	0	0	0	.94	0	0	0	.93	0	0	0	.92	0	0	0
0.05	.92	0	0	0	.93	0	0	0	.92	0	0	0	.94	0	0	0
0.1	.94	0	0	0	.92	0	0	0	.94	0	0	0	.91	0	0	1
0.5	.92	0	0	0	.94	0	0	0	.90	0	0	0	.68	0	3	27
1.0	.93	0	0	0	.93	0	0	0	.88	0	0	0	.51	4	21	122
5.0	.91	0	0	0	.89	0	0	0	.62	2	11	45	.16	39	95	304
10.0	.90	0	0	0	.87	0	0	0	.47	15	38	137	.09	55	109	329
50.0	.65	0	2	16	.64	19	31	73	.15	44	102	291	.02	70	142	445
100.0	.48	6	29	152	.53	18	47	138	.08	58	121	362	.00	76	158	456
500.0	.10	53	112	328	.29	42	88	256	.00	80	159	487	.00	86	166	516
1,000.0	.01	65	138	413	.20	57	111	301	.00	86	172	514	.00	93	185	530

Table 5: Number of individuals (out of 10,000) exposed by Yeom et al. membership inference attack on logistic regression (CIFAR-100). The non-private ($\epsilon = \infty$) model leaks 129, 240 and 704 members for 1%, 2% and 5% FPR respectively.

Purchase-100. The baseline model for non-private logistic regression achieves accuracy of 0.942 on the training set and 0.695 on test set. In comparison, Google ML platform's black-box trained model achieves a test accuracy of 0.656 for Purchase-100 (see Shokri et al. [61] for details).

Figure 3 shows the accuracy loss of all differential privacy variants on Purchase-100 data set. Naïve composition and advanced composition have essentially no utility until ϵ exceeds 100. At $\epsilon = 1000$, naïve composition achieves accuracy loss of 0.116 ± 0.003, the advanced composition achieves accuracy loss of 0.513 ± 0.003 and the other variants achieve accuracy loss close to 0.02. RDP achieves the best utility across all ϵ values. zCDP performs better than advanced composition and naïve composition.

Figure 4 compares the privacy leakage of the variants against the inference attacks. The leakage is in accordance to the noise each variant adds and it increases proportionally to the model utility. Hence, if a model has reasonable utility, it is bound to leak membership information. The white-box membership inference attack of Yeom et al. is relatively more effective than the black-box membership inference attack of Shokri et al. as shown in Figures 4(a) and 4(b). Table 6 shows the number of individual members exposed, with similar results to the findings for CIFAR-100.

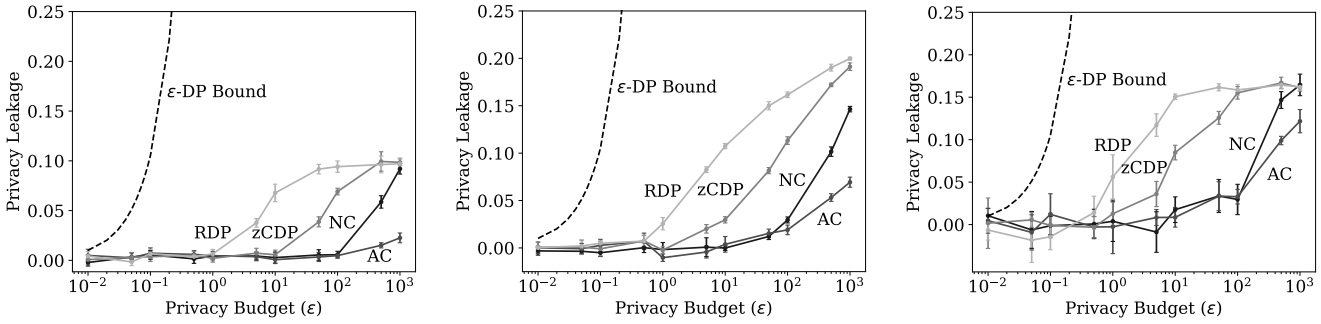
4.3 Neural Networks

We train a neural network model consisting of two hidden layers and an output layer. The hidden layers have 256 neurons that use ReLU activation. The output layer is a softmax layer with 100 neurons, each corresponding to a class label. This architecture is similar to the one used by Shokri et al. [61].

CIFAR-100. The baseline non-private neural network model achieves accuracy of 1.000 on the training set and 0.168 on test set, which is competitive to the neural network model of Shokri et al. [61]. Their model is trained on a training set of size 29,540 and achieves test accuracy of 0.20, whereas our model is trained on 10,000 training instances. There is a huge generalization gap of 0.832, which the inference attacks can exploit. Figure 5(a) compares the accuracy loss of neural network models trained with different relaxed notions of differential privacy with varying privacy budget ϵ. The model trained with naïve composition does not learn anything useful until $\epsilon = 100$ (accuracy loss of 0.907 ± 0.004), at which point the advanced composition also has accuracy loss close to 0.935 and the other variants achieve accuracy loss close to 0.24. None of the variants approach zero accuracy loss, even for $\epsilon = 1000$. The relative performance is similar to that of logistic regression model discussed in Section 4.2.

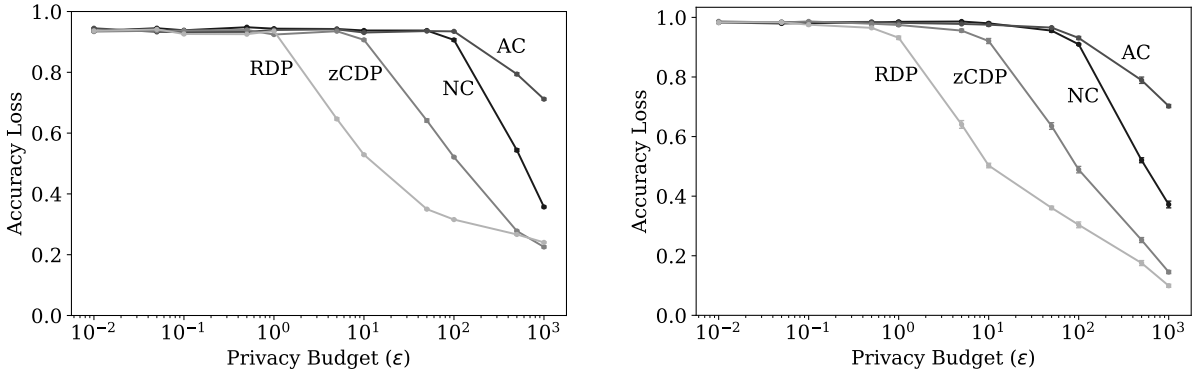
Figures 6(a) and 6(b) shows the privacy leakage due to membership inference attacks on neural network models trained with different relaxed notions for both attacks. The privacy leakage for each variation of differential privacy accords with the amount of noise it adds to the model. The leakage is significant for relaxed variants at higher ϵ values due to model overfitting. For $\epsilon = 1000$, with the Shokri et al. attack, naïve composition has leakage of 0.034 compared to 0.002 for advanced composition, 0.219 for zCDP, and 0.277 for RDP (above the region shown in the plot). For the white-box attacker of Yeom et al. [74], RDP leaks the most for $\epsilon = 1000$ (membership advantage of 0.399) closely followed by zCDP. This is because these relaxed variations add considerably less noise in comparison to naïve composition. Naïve composition and advanced composition achieve strong privacy against membership inference attackers, but fail to learning anything useful. No option appears to provide both acceptable model utility and meaningful privacy.

Like we did for logistic regression, we report the actual number of training set members exposed to the attacker in Table 7. The impact of privacy leakage is far more severe for the non-private neural network model due to model overfitting—



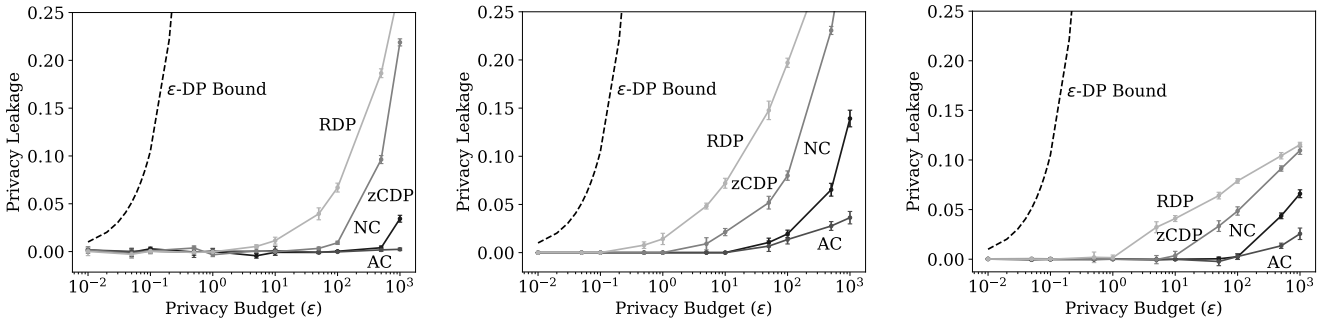
(a) Shokri et al. membership inference (b) Yeom et al. membership inference (c) Yeom et al. attribute inference

Figure 4: Inference attacks on logistic regression (Purchase-100).



(a) CIFAR-100 (b) Purchase-100

Figure 5: Accuracy loss of neural networks.



(a) Shokri et al. membership inference (b) Yeom et al. membership inference (c) Yeom et al. attribute inference

Figure 6: Inference attacks on neural network (CIFAR-100).

	Naïve Composition				Advanced Composition				zCDP				RDP			
ϵ	Loss	1%	2%	5%	Loss	1%	2%	5%	Loss	1%	2%	5%	Loss	1%	2%	5%
0.01	.98	0	0	0	.99	0	0	0	.99	0	0	0	.99	0	0	0
0.05	.99	0	0	0	.98	0	0	0	.99	0	0	0	.98	0	0	0
0.1	.99	0	0	0	.98	0	0	0	.98	0	0	0	.97	0	0	0
0.5	.98	0	0	0	.98	0	0	0	.97	0	0	0	.93	0	0	2
1.0	.98	0	0	0	.98	0	0	0	.97	0	0	0	.87	0	0	23
5.0	.98	0	0	0	.97	0	0	0	.92	0	0	4	.46	42	72	174
10.0	.97	0	0	0	.97	0	0	0	.83	1	7	35	.28	53	101	270
50.0	.91	0	0	1	.92	0	0	1	.43	38	65	187	.11	72	154	406
100.0	.83	0	0	28	.87	0	1	10	.26	55	113	289	.08	84	160	473
500.0	.32	45	95	227	.66	18	34	84	.05	77	183	487	.03	75	181	533
1,000.0	.12	81	164	427	.51	34	58	145	.02	87	184	530	.02	94	189	566

Table 6: Number of members (out of 10,000) exposed by Yeom et al. membership inference attack on logistic regression (Purchase-100). The non-private ($\epsilon = \infty$) model leaks 102, 262 and 716 members for 1%, 2% and 5% FPR respectively.

	Naïve Composition				Advanced Composition				zCDP				RDP			
ϵ	Loss	1%	2%	5%	Loss	1%	2%	5%	Loss	1%	2%	5%	Loss	1%	2%	5%
0.01	.94	0	0	0	.94	0	0	0	.93	0	0	0	.94	0	0	0
0.05	.94	0	0	0	.93	0	0	0	.94	0	0	0	.94	0	0	0
0.1	.94	0	0	0	.93	0	0	0	.94	0	0	0	.93	0	0	0
0.5	.95	0	0	0	.93	0	0	0	.94	0	0	0	.92	0	0	0
1.0	.94	0	0	0	.94	0	0	0	.92	0	0	0	.94	0	0	0
5.0	.94	0	0	0	.94	0	0	0	.94	0	0	0	.65	11	24	79
10.0	.94	0	0	0	.93	0	0	0	.91	0	0	2	.53	9	33	108
50.0	.94	0	0	0	.94	0	0	0	.64	2	12	65	.35	28	65	185
100.0	.91	0	0	0	.93	0	0	0	.52	13	31	98	.32	21	67	205
500.0	.54	3	21	58	.79	4	7	31	.28	8	41	210	.27	5	54	278
1,000.0	.36	20	48	131	.71	8	16	74	.22	12	42	211	.24	10	37	269

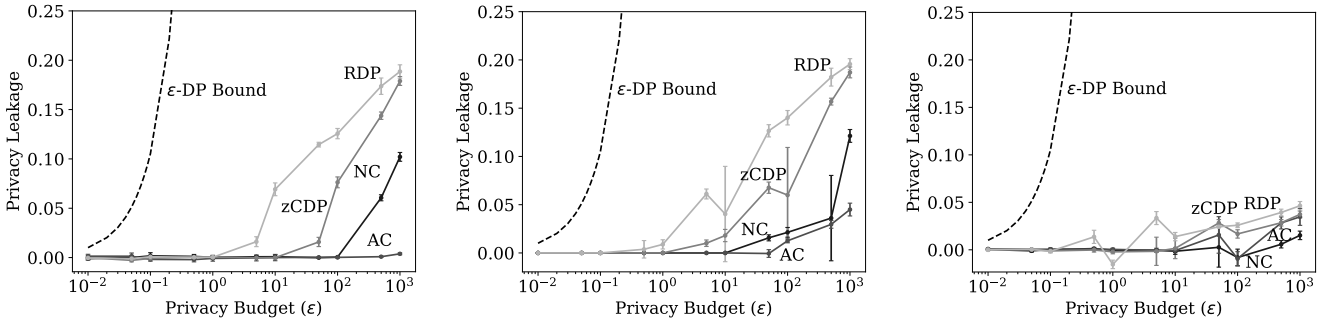
Table 7: Number of members (out of 10,000) exposed by Yeom et al. membership inference attack on neural network (CIFAR-100). The non-private ($\epsilon = \infty$) model leaks 0, 556 and 7349 members for 1%, 2% and 5% FPR respectively.

exposing over 73% of training set members at 5% false positive rate, compared to only 7% for the logistic regression model.[2] The privacy mechanisms provide substantial reduction in exposure, even with high ϵ budgets, but the relaxed variants expose more members compared to naïve composition and advanced composition.

Figure 6(c) depicts the privacy leakage due to attribute inference attack on the neural network models. Naïve composition and advanced composition are both resistant to the attack for $\epsilon \leq 100$, but the relaxed variants reveal some privacy leakage for lower privacy budgets.

Purchase-100. The baseline non-private neural network model achieves accuracy of 0.982 on the training set and 0.605 on test set. In comparison, the neural network model of Shokri et al. [61] trained on a similar data set (but with 600 attributes instead of 100 as in our data set) achieves 0.670 test accuracy. Figure 5(b) compares the accuracy loss, and Figure 7 the privacy leakage, of neural network models trained with different variants of differential privacy. The trends for both accuracy and privacy are similar to those for the logistic regression models (Figure 3). The relaxed variants achieve model utility close to the non-private baseline for $\epsilon = 1000$, while naïve composition continues to suffer from high accuracy loss (0.372). Advanced composition has higher accuracy loss of 0.702 for $\epsilon = 1000$ as it requires addition of more noise than naïve composition when ϵ is greater than the number of training epochs. Figure 7 shows the privacy leakage comparison of the variants against the inference attacks. The results are consistent with those observed for CIFAR-100.

[2] Curiously, this appears to be contradicted at 1% FPR where no members are revealed by non-private NN model but some are revealed by the privacy-preserving models. This is due to the number of extremely high-confidence incorrect outputs of the non-private model, meaning that there is no confidence threshold that does not include at least 1% false positives.



(a) Shokri et al. membership inference (b) Yeom et al. membership inference (c) Yeom et al. attribute inference

Figure 7: Inference attacks on neural network (Purchase-100).

4.4 Discussion

While the tighter cumulative noise bounds provided by relaxed variants of differential privacy improve model utility for a given privacy budget, the reduction in noise increases vulnerability to inference attacks. Thus, privacy does not come for free, and the relaxations of the differential privacy definition that result in lower noise requirements come with additional privacy risks. While these relaxed definitions still satisfy the (ϵ, δ)-differential privacy guarantees, the concrete value of these guarantees diminishes rapidly with high ϵ values and non-zero δ. Although the theoretical guarantees provided by differential privacy are very appealing, once ϵ values exceed small values, the practical value of these guarantees is insignificant—in most of our inference attack figures, the theoretical bound given by ϵ-DP falls off the graph before any measurable privacy leakage occurs (and at levels well before models provide acceptable utility). The value of these privacy mechanisms comes not from the theoretical guarantees, but from the impact of the mechanism on what realistic adversaries can infer.

We note that in our inference attack experiments, we use equal numbers of member and non-member records which provides 50-50 prior success probability to the attacker. Thus, even an ϵ-DP implementation might leak even for small ϵ values, though we did not observe any such leakage. Alternatively, a skewed prior probability may lead to smaller leakage even for large ϵ values. Our goal in this work is to evaluate scenarios where risk of inference is high, so the use of 50-50 prior probability is justified. We also emphasis that our results show the privacy leakage due to two particular membership inference attacks. Attacks only get better, so future attacks may be able to infer more than is shown in our experiments.

5 Conclusion

Differential privacy has earned a well-deserved reputation providing principled and powerful mechanisms for ensuring provable privacy. However, when it is implemented for challenging tasks such as machine learning, compromises must be made to preserve utility. It is essential that the privacy impact of those compromises is well understood when differential privacy is deployed to protect sensitive data. Our results are a step towards improving that understanding, and reveal that the commonly-used relaxations of differential privacy may provide unacceptable utility-privacy trade-offs. We hope our study will encourage more careful assessments of the practical privacy value of formal claims based on differential privacy, and lead to deeper understanding of the privacy impact of design decisions when deploying differential privacy, and eventually to solutions that provide desirable, and well understood, utility-privacy trade-offs.

Availability

Open source code for reproducing all of our experiments is available at https://github.com/bargavj/EvaluatingDPML.

Acknowledgments

The authors are deeply grateful to Úlfar Erlingsson for pointing out some key misunderstandings in an early version of this work and for convincing us of the importance of per-instance gradient clipping, and to Úlfar, Ilya Mironov, and Shuang Song for help validating and improving the work. We thank Vincent Bindschaedler for shepherding our paper. We thank Youssef Errami and Jonah Weissman for contributions to the experiments, and Ben Livshits for feedback on the work. Atallah Hezbor, Faysal Shezan, Tanmoy Sen, Max Naylor, Joshua Holtzman and Nan Yang helped systematize the related works. Finally, we thank Congzheng Song and Samuel Yeom for providing their implementation of inference attacks. This work was partially funded by grants from the National Science Foundation SaTC program (#1717950, #1915813) and support from Intel and Amazon.

References

[1] Martin Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *ACM Conference on Computer and Communications Security*, 2016.

[2] Galen Andrew, Steve Chien, and Nicolas Papernot. TensorFlow Privacy. https://github.com/tensorflow/privacy.

[3] Giuseppe Ateniese, Luigi Mancini, Angelo Spognardi, Antonio Villani, Domenico Vitali, and Giovanni Felici. Hacking smart machines with smarter ones: How to extract meaningful data from machine learning classifiers. *International Journal of Security and Networks*, 2015.

[4] Michael Backes, Pascal Berrang, Mathias Humbert, and Praveen Manoharan. Membership privacy in MicroRNA-based studies. In *ACM Conference on Computer and Communications Security*, 2016.

[5] Eugene Bagdasaryan, Andreas Veit, Yiqing Hua, Deborah Estrin, and Vitaly Shmatikov. How to backdoor federated learning. *arXiv:1807.00459*, 2018.

[6] Brett K Beaulieu-Jones, William Yuan, Samuel G Finlayson, and Zhiwei Steven Wu. Privacy-preserving distributed deep learning for clinical data. *arXiv:1812.01484*, 2018.

[7] Raghav Bhaskar, Srivatsan Laxman, Adam Smith, and Abhradeep Thakurta. Discovering frequent patterns in sensitive data. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2010.

[8] Abhishek Bhowmick, John Duchi, Julien Freudiger, Gaurav Kapoor, and Ryan Rogers. Protection against reconstruction and its applications in private federated learning. *arXiv:1812.00984*, 2018.

[9] Mark Bun and Thomas Steinke. Concentrated differential privacy: Simplifications, extensions, and lower bounds. In *Theory of Cryptography Conference*, 2016.

[10] Nicholas Carlini, Chang Liu, Jernej Kos, Úlfar Erlingsson, and Dawn Song. The Secret Sharer: Evaluating and testing unintended memorization in neural networks. In *USENIX Security Symposium*, 2019.

[11] Kamalika Chaudhuri and Claire Monteleoni. Privacy-preserving logistic regression. In *Advances in Neural Information Processing Systems*, 2009.

[12] Kamalika Chaudhuri, Claire Monteleoni, and Anand D. Sarwate. Differentially private Empirical Risk Minimization. *Journal of Machine Learning Research*, 2011.

[13] Zeyu Ding, Yuxin Wang, Guanhong Wang, Danfeng Zhang, and Daniel Kifer. Detecting violations of differential privacy. In *ACM Conference on Computer and Communications Security*, 2018.

[14] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 2011.

[15] John C Duchi, Michael I Jordan, and Martin J Wainwright. Local privacy and statistical minimax rates. In *Symposium on Foundations of Computer Science*, 2013.

[16] Cynthia Dwork. Differential Privacy: A Survey of Results. In *International Conference on Theory and Applications of Models of Computation*, 2008.

[17] Cynthia Dwork and Aaron Roth. The Algorithmic Foundations of Differential Privacy. *Foundations and Trends in Theoretical Computer Science*, 2014.

[18] Cynthia Dwork and Guy N. Rothblum. Concentrated differential privacy. *arXiv:1603.01887*, 2016.

[19] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *ACM Conference on Computer and Communications Security*, 2015.

[20] Matthew Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing. In *USENIX Security Symposium*.

[21] Arik Friedman and Assaf Schuster. Data mining with differential privacy. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2010.

[22] Karan Ganju, Qi Wang, Wei Yang, Carl A Gunter, and Nikita Borisov. Property inference attacks on fully connected neural networks using permutation invariant representations. In *ACM Conference on Computer and Communications Security*, 2018.

[23] Joseph Geumlek, Shuang Song, and Kamalika Chaudhuri. Rényi differential privacy mechanisms for posterior sampling. In *Advances in Neural Information Processing Systems*, 2017.

[24] Robin C Geyer, Tassilo Klein, and Moin Nabi. Differentially private federated learning: A client level perspective. *arXiv:1712.07557*, 2017.

[25] Jihun Hamm, Paul Cao, and Mikhail Belkin. Learning privately from multiparty data. In *International Conference on Machine Learning*, 2016.

[26] Michael Hay, Ashwin Machanavajjhala, Gerome Miklau, Yan Chen, and Dan Zhang. Principled evaluation of differentially private algorithms using DPBench. In *ACM SIGMOD Conference on Management of Data*, 2016.

[27] Nils Homer et al. Resolving individuals contributing trace amounts of DNA to highly complex mixtures us-

ing high-density SNP genotyping microarrays. *PLoS Genetics*, 2008.

[28] Zonghao Huang, Rui Hu, Yanmin Gong, and Eric Chan-Tin. DP-ADMM: ADMM-based distributed learning with differential privacy. *arXiv:1808.10101*, 2018.

[29] Nick Hynes, Raymond Cheng, and Dawn Song. Efficient deep learning on multi-source private data. *arXiv:1807.06689*, 2018.

[30] Ali Inan, Murat Kantarcioglu, Gabriel Ghinita, and Elisa Bertino. Private record matching using differential privacy. In *International Conference on Extending Database Technology*, 2010.

[31] Roger Iyengar, Joseph P Near, Dawn Song, Om Thakkar, Abhradeep Thakurta, and Lun Wang. Towards practical differentially private convex optimization. In *IEEE Symposium on Security and Privacy*, 2019.

[32] Prateek Jain, Pravesh Kothari, and Abhradeep Thakurta. Differentially private online learning. In *Annual Conference on Learning Theory*, 2012.

[33] Prateek Jain and Abhradeep Thakurta. Differentially private learning with kernels. In *International Conference on Machine Learning*, 2013.

[34] Prateek Jain and Abhradeep Guha Thakurta. (Near) Dimension independent risk bounds for differentially private learning. In *International Conference on Machine Learning*, 2014.

[35] Bargav Jayaraman, Lingxiao Wang, David Evans, and Quanquan Gu. Distributed learning without distress: Privacy-preserving Empirical Risk Minimization. In *Advances in Neural Information Processing Systems*, 2018.

[36] Kaggle, Inc. Acquire Valued Shoppers Challenge. https://kaggle.com/c/acquire-valued-shoppers-challenge/data, 2014.

[37] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

[38] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

[39] Jaewoo Lee. Differentially private variance reduced stochastic gradient descent. In *International Conference on New Trends in Computing Sciences*, 2017.

[40] Dong-Hui Li and Masao Fukushima. A modified BFGS method and its global convergence in nonconvex minimization. *Journal of Computational and Applied Mathematics*, 2001.

[41] Ninghui Li, Wahbeh Qardaji, Dong Su, and Jianneng Cao. PrivBasis: Frequent itemset mining with differential privacy. *The VLDB Journal*, 2012.

[42] Ninghui Li, Wahbeh Qardaji, Dong Su, Yi Wu, and Weining Yang. Membership privacy: A unifying framework for privacy definitions. In *ACM Conference on Computer and Communications Security*, 2013.

[43] Dong C Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical programming*, 1989.

[44] Yunhui Long, Vincent Bindschaedler, and Carl A. Gunter. Towards measuring membership privacy. *arXiv:1712.09136*, 2017.

[45] Daniel Lowd and Christopher Meek. Adversarial learning. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2005.

[46] Frank McSherry. Statistical inference considered harmful. https://github.com/frankmcsherry/blog/blob/master/posts/2016-06-14.md, 2016.

[47] Frank McSherry and Ilya Mironov. Differentially private recommender systems: Building privacy into the Netflix prize contenders. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2009.

[48] Frank McSherry and Kunal Talwar. Mechanism design via differential privacy. In *Symposium on Foundations of Computer Science*, 2007.

[49] Ilya Mironov. Rényi differential privacy. In *IEEE Computer Security Foundations Symposium*, 2017.

[50] Luis Munoz-González, Battista Biggio, Ambra Demontis, Andrea Paudice, Vasin Wongrassamee, Emil C. Lupu, and Fabio Roli. Towards poisoning of deep learning algorithms with back-gradient optimization. In *ACM Workshop on Artificial Intelligence and Security*, 2017.

[51] Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. Smooth sensitivity and sampling in private data analysis. In *ACM Symposium on Theory of Computing*, 2007.

[52] Nicolas Papernot, Martín Abadi, Úlfar Erlingsson, Ian Goodfellow, and Kunal Talwar. Semi-supervised knowledge transfer for deep learning from private training data. In *International Conference on Learning Representations*, 2017.

[53] Mijung Park, Jimmy Foulds, Kamalika Chaudhuri, and Max Welling. DP-EM: Differentially private expectation maximization. In *Artificial Intelligence and Statistics*, 2017.

[54] Manas Pathak, Shantanu Rane, and Bhiksha Raj. Multiparty Differential Privacy via Aggregation of Locally

Trained Classifiers. In *Advances in Neural Information Processing Systems*, 2010.

[55] NhatHai Phan, Yue Wang, Xintao Wu, and Dejing Dou. Differential privacy preservation for deep auto-encoders: An application of human behavior prediction. In *AAAI Conference on Artificial Intelligence*, 2016.

[56] NhatHai Phan, Xintao Wu, and Dejing Dou. Preserving differential privacy in convolutional deep belief networks. *Machine Learning*, 2017.

[57] Boris T Polyak and Anatoli B Juditsky. Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization*, 1992.

[58] Md Atiqur Rahman, Tanzila Rahman, Robert Laganière, Noman Mohammed, and Yang Wang. Membership inference attack against differentially private deep learning model. *Transactions on Data Privacy*, 2018.

[59] Ahmed Salem, Yang Zhang, Mathias Humbert, Pascal Berrang, Mario Fritz, and Michael Backes. ML-Leaks: Model and data independent membership inference attacks and defenses on machine learning models. In *Network and Distributed Systems Security Symposium*.

[60] Reza Shokri and Vitaly Shmatikov. Privacy-preserving deep learning. In *ACM Conference on Computer and Communications Security*, 2015.

[61] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *IEEE Symposium on Security and Privacy*, 2017.

[62] Adam Smith and Abhradeep Thakurta. Differentially Private Feature Selection via Stability Arguments, and the Robustness of the Lasso. In *Proceedings of Conference on Learning Theory*, 2013.

[63] Shuang Song, Kamalika Chaudhuri, and Anand D Sarwate. Stochastic gradient descent with differentially private updates. In *IEEE Global Conference on Signal and Information Processing*, 2013.

[64] Kunal Talwar, Abhradeep Thakurta, and Li Zhang. Private Empirical Risk Minimization beyond the worst case: The effect of the constraint set geometry. *arXiv:1411.5417*, 2014.

[65] Kunal Talwar, Abhradeep Thakurta, and Li Zhang. Nearly Optimal Private LASSO. In *Advances in Neural Information Processing Systems*, 2015.

[66] Florian Tramèr, Fan Zhang, Ari Juels, Michael Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction APIs. In *USENIX Security Symposium*, 2016.

[67] Binghui Wang and Neil Zhenqiang Gong. Stealing hyperparameters in machine learning. In *IEEE Symposium on Security and Privacy*, 2018.

[68] Di Wang, Minwei Ye, and Jinhui Xu. Differentially private Empirical Risk Minimization revisited: Faster and more general. In *Advances in Neural Information Processing Systems*, 2017.

[69] Xi Wu, Matthew Fredrikson, Somesh Jha, and Jeffrey F Naughton. A methodology for formalizing model-inversion attacks. In *IEEE Computer Security Foundations Symposium*, 2016.

[70] Xi Wu, Fengan Li, Arun Kumar, Kamalika Chaudhuri, Somesh Jha, and Jeffrey Naughton. Bolt-on differential privacy for scalable stochastic gradient descent-based analytics. In *ACM SIGMOD Conference on Management of Data*, 2017.

[71] Huang Xiao, Battista Biggio, Blaine Nelson, Han Xiao, Claudia Eckert, and Fabio Roli. Support vector machines under adversarial label contamination. *Neurocomputing*, 2015.

[72] Mengjia Yan, Christopher Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. *arXiv:1808.04761*, 2018.

[73] Chaofei Yang, Qing Wu, Hai Li, and Yiran Chen. Generative poisoning attack method against neural networks. *arXiv:1703.01340*, 2017.

[74] Samuel Yeom, Irene Giacomelli, Matt Fredrikson, and Somesh Jha. Privacy risk in machine learning: Analyzing the connection to overfitting. In *IEEE Computer Security Foundations Symposium*, 2018.

[75] Lei Yu, Ling Liu, Calton Pu, Mehmet Emre Gursoy, and Stacey Truex. Differentially private model publishing for deep learning. In *IEEE Symposium on Security and Privacy*, 2019.

[76] Matthew D Zeiler. ADADELTA: An adaptive learning rate method. *arXiv:1212.5701*, 2012.

[77] Jiaqi Zhang, Kai Zheng, Wenlong Mou, and Liwei Wang. Efficient private ERM for smooth objectives. In *International Joint Conference on Artificial Intelligence*, 2017.

[78] Jun Zhang, Zhenjie Zhang, Xiaokui Xiao, Yin Yang, and Marianne Winslett. Functional mechanism: Regression analysis under differential privacy. *The VLDB Journal*, 2012.

[79] Lingchen Zhao, Yan Zhang, Qian Wang, Yanjiao Chen, Cong Wang, and Qin Zou. Privacy-preserving collaborative deep learning with irregular participants. *arXiv:1812.10113*, 2018.

FUZZIFICATION: Anti-Fuzzing Techniques

Jinho Jung, Hong Hu, David Solodukhin, Daniel Pagan, Kyu Hyung Lee[†], Taesoo Kim

Georgia Institute of Technology
[†] *University of Georgia*

Abstract

Fuzzing is a software testing technique that quickly and automatically explores the input space of a program without knowing its internals. Therefore, developers commonly use fuzzing as part of test integration throughout the software development process. Unfortunately, it also means that such a blackbox and the automatic natures of fuzzing are appealing to adversaries who are looking for zero-day vulnerabilities.

To solve this problem, we propose a new *mitigation* approach, called FUZZIFICATION, that helps developers protect the released, binary-only software from attackers who are capable of applying state-of-the-art fuzzing techniques. Given a performance budget, this approach aims to hinder the fuzzing process from adversaries as much as possible. We propose three FUZZIFICATION techniques: 1) SpeedBump, which amplifies the slowdown in normal executions by hundreds of times to the fuzzed execution, 2) BranchTrap, interfering with feedback logic by hiding paths and polluting coverage maps, and 3) AntiHybrid, hindering taint-analysis and symbolic execution. Each technique is designed with best-effort, defensive measures that attempt to hinder adversaries from bypassing FUZZIFICATION.

Our evaluation on popular fuzzers and real-world applications shows that FUZZIFICATION effectively reduces the number of discovered paths by 70.3% and decreases the number of identified crashes by 93.0% from real-world binaries, and decreases the number of detected bugs by 67.5% from LAVA-M dataset while under user-specified overheads for common workloads. We discuss the robustness of FUZZIFICATION techniques against adversarial analysis techniques. We open-source our FUZZIFICATION system to foster future research.

1 Introduction

Fuzzing is a software testing technique that aims to find software bugs automatically. It keeps running the program with randomly generated inputs and waits for bug-exposing behaviors such as crashing or hanging. It has become a standard practice to detect security problems in complex, modern software [40, 72, 37, 25, 23, 18, 9]. Recent research has built several efficient fuzzing tools [57, 52, 29, 34, 6, 64] and found a large number of security vulnerabilities [51, 72, 59, 26, 10].

Unfortunately, advanced fuzzing techniques can also be used by malicious attackers to find zero-day vulnerabilities. Recent studies [61, 58] confirm that attackers predominantly prefer fuzzing tools over others (*e.g.*, reverse engineering) in finding vulnerabilities. For example, a survey of information security experts [28] shows that fuzzing techniques discover 4.83 times more bugs than static analysis or manual detection. Therefore, developers might want to apply *anti-fuzzing* techniques on their products to hinder fuzzing attempts by attackers, similar in concept to using obfuscation techniques to cripple reverse engineering [12, 13].

In this paper, we propose a new direction of binary protection, called FUZZIFICATION, that hinders attackers from effectively finding bugs. Specifically, attackers may still be able to find bugs from the binary protected by FUZZIFICATION, but with significantly more effort (*e.g.*, CPU, memory, and time). Thus, developers or other trusted parties who get the original binary are able to detect program bugs and synthesize patches before attackers widely abuse them. An effective FUZZIFICATION technique should enable the following three features. First, it should be effective for hindering existing fuzzing tools, finding fewer bugs within a fixed time; second, the protected program should still run efficiently in normal usage; third, the protection code should not be easily identified or removed from the protected binary by straightforward analysis techniques.

No existing technique can achieve all three goals simultaneously. First, software obfuscation techniques, which impede static program analysis by randomizing binary representations, seem to be effective in thwarting fuzzing attempts [12, 13]. However, we find that it falls short of FUZZIFICATION in two ways. Obfuscation introduces unacceptable overhead to normal program executions. Figure 1(a) shows that obfuscation slows the execution by at least 1.7 times when using UPX [60] and up to 25.0 times when using

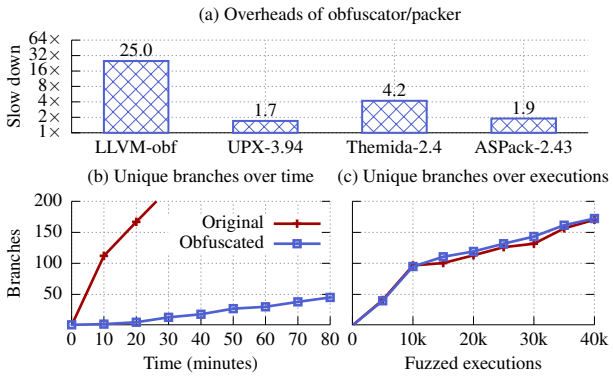


Figure 1: Impact of obfuscation techniques on fuzzing. (a) Obfuscation techniques introduce 1.7×-25.0× execution slow down. (b) and (c) fuzzing obfuscated binaries discovers fewer program paths over time, but gets a similar number of paths over executions.

LLVM-obfuscator [33]. Also, obfuscation cannot effectively hinder fuzzers in terms of path exploration. It can slow each fuzzed execution, as shown in Figure 1(b), but the path discovery per execution is almost identical to that of fuzzing the original binary, as shown in Figure 1(c). Therefore, obfuscation is not an ideal FUZZIFICATION technique. Second, software diversification changes the structure and interfaces of the target application to distribute diversified versions [35, 3, 53, 50]. For example, the technique of N-version software [3] is able to mitigate exploits because attackers often depend on clear knowledge of the program states. However, software diversification is powerless on hiding the original vulnerability from the attacker's analysis; thus it is not a good approach for FUZZIFICATION.

In this paper, we propose three FUZZIFICATION techniques for developers to protect their programs from malicious fuzzing attempts: SpeedBump, BranchTrap, and AntiHybrid. The SpeedBump technique aims to slow program execution during fuzzing. It injects delays to *cold* paths, which normal executions rarely reach but that fuzzed executions frequently visit. The BranchTrap technique inserts a large number of input-sensitive jumps into the program so that any input drift will significantly change the execution path. This will induce coverage-based fuzzing tools to spend their efforts on injected bug-free paths instead of on the real ones. The AntiHybrid technique aims to thwart hybrid fuzzing approaches that incorporate traditional fuzzing methods with dynamic taint analysis and symbolic execution.

We develop defensive mechanisms to hinder attackers identifying or removing our techniques from protected binaries. For SpeedBump, instead of calling the `sleep` function, we inject randomly synthesized CPU-intensive operations to cold paths and create control-flow and data-flow dependencies between the injected code and the original code. We reuse existing binary code to realize BranchTrap to prevent an adversary from identifying the injected branches.

To evaluate our FUZZIFICATION techniques, we apply them on the LAVA-M dataset and nine real-world applications, including `libjpeg`, `libpng`, `libtiff`, `pcre2`, `readelf`, `objdump`, `nm`, `objcopy`, and `MuPDF`. These programs are extensively used to evaluate the effectiveness of fuzzing tools [19, 11, 48, 67]. Then, we use four popular fuzzers —AFL, HonggFuzz, VUzzer, and QSym— to fuzz the original programs and the protected ones for the same amount of time. On average, fuzzers detect 14.2 times more bugs from the original binaries and 3.0 times more bugs from the LAVA-M dataset than those from "fuzzified" ones. At the same time, our FUZZIFICATION techniques decrease the total number of discovered paths by 70.3%, and maintain user-specified overhead budget. This result shows that our FUZZIFICATION techniques successfully decelerate fuzzing performance on vulnerability discovery. We also perform an analysis to show that data-flow and control-flow analysis techniques cannot easily disarm our techniques.

In this paper, we make the following contributions:

- We first shed light on the new research direction of anti-fuzzing schemes, so-called, FUZZIFICATION.
- We develop three FUZZIFICATION techniques to slow each fuzzed execution, to hide path coverage, and to thwart dynamic taint-analysis and symbolic execution.
- We evaluate our techniques on popular fuzzers and common benchmarks. Our results show that the proposed techniques hinder these fuzzers, finding 93% fewer bugs from the real-world binaries and 67.5% fewer bugs from the LAVA-M dataset, and 70.3% less coverage while maintaining the user-specified overhead budget.

We will release the source code of our work at https://github.com/sslab-gatech/fuzzification.

2 Background and Problem

2.1 Fuzzing Techniques

The goal of fuzzing is to automatically detect program bugs. For a given program, a fuzzer first creates a large number of inputs, either by random mutation or by format-based generation. Then, it runs the program with these inputs to see whether the execution exposes unexpected behaviors, such as a crash or an incorrect result. Compared to manual analysis or static analysis, fuzzing is able to execute the program orders of magnitude more times and thus can explore more program states to maximize the chance of finding bugs.

2.1.1 Fuzzing with Fast Execution

A straightforward way to improve fuzzing efficiency is to make each execution faster. Current research highlights several fast execution techniques, including (1) customized system and hardware to accelerate fuzzed execution and (2) parallel fuzzing to amortize the absolute execution time in

large-scale. Among these techniques, AFL uses the fork server and persistent mode to avoid the heavy process creation and can accelerate fuzzing by a factor of two or more [68, 69]. AFL-PT, kAFL, and HonggFuzz utilize hardware features such as Intel Process Tracing (PT) and Branch Trace Store (BTS) to collect code coverage efficiently to guide fuzzing [65, 54, 23]. Recently, Xu *et al*. designed new operating system primitives, like efficient system calls, to speed up fuzzing on multi-core machines [64].

2.1.2 Fuzzing with Coverage-guidance

Coverage-guided fuzzing collects the code coverage for each fuzzed execution and prioritizes fuzzing the input that has triggered new coverage. This fuzzing strategy is based on two empirical observations: (1) a higher path coverage indicates a higher chance of exposing bugs; and (2) mutating inputs that ever trigger new paths is likely to trigger another new path. Most popular fuzzers take code coverage as guidance, like AFL, HonggFuzz, and LibFuzzer, but with different methods for coverage representation and coverage collection.

Coverage representation. Most fuzzers take basic blocks or branches to represent the code coverage. For example, HonggFuzz and VUzzer use basic block coverage, while AFL instead considers the branch coverage, which provides more information about the program states. Angora [11] combines branch coverage with the call stack to further improve coverage accuracy. However, the choice of representation is a trade-off between coverage accuracy and performance, as more fine-grained coverage introduces higher overhead to each execution and harms the fuzzing efficiency.

Coverage collection. If the source code is available, fuzzers can instrument the target program during compilation or assembly to record coverage at runtime, like in AFL-LLVM mode and LibFuzzer. Otherwise, fuzzers have to utilize either static or dynamic binary instrumentation to achieve a similar purpose, like in AFL-QEMU mode [70]. Also, several fuzzers leverage hardware features to collect the coverage [65, 54, 23]. Fuzzers usually maintain their own data structure to store coverage information. For example, AFL and HonggFuzz use a fixed-size array and VUzzer utilizes a *Set* data structure in Python to store their coverage. However, the size of the structure is also a trade-off between accuracy and performance: an overly small memory cannot capture every coverage change, while an overly large memory introduces significant overhead. For example, AFL's performance drops 30% if the bitmap size is changed from 64KB to 1MB [19].

2.1.3 Fuzzing with Hybrid Approaches

Hybrid approaches are proposed to help solve the limitations of existing fuzzers. First, fuzzers do not distinguish input bytes with different types (*e.g.*, magic number, length specifier) and thus may waste time mutating less important bytes that cannot affect any control flow. In this case, taint analysis is used to help find which input bytes are used to determine branch conditions, like VUzzer [52]. By focusing on the mutation of these bytes, fuzzers can quickly find new execution paths. Second, fuzzers cannot easily resolve complicated conditions, such as comparison with magic value or checksum. Several works [57, 67] utilize symbolic execution to address this problem, which is good at solving complicated constraints but incurs high overhead.

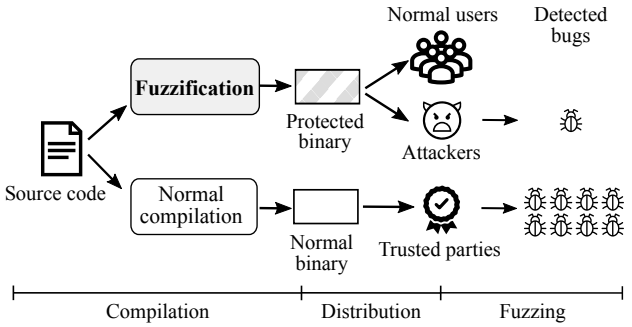


Figure 2: Workflow of FUZZIFICATION protection. Developers create a protected binary with FUZZIFICATION techniques and release it to public. Meanwhile, they send the normally compiled binary to trusted parties. Attackers cannot find many bugs from the protected binary through fuzzing, while trusted parties can effectively find significantly more bugs and developers can patch them in time.

2.2 FUZZIFICATION Problem

Program developers may want to completely control the bug-finding process, as any bug leakage can bring attacks and lead to financial loss [45]. They demand exposing bugs by themselves or by trusted parties, but not by malicious end-users. Anti-fuzzing techniques can help to achieve that by decelerating unexpected fuzzing attempts, especially from malicious attackers.

We show the workflow of FUZZIFICATION in Figure 2. Developers compile their code in two versions. One is compiled with FUZZIFICATION techniques to generate a protected binary, and the other is compiled normally to generate a normal binary. Then, developers distribute the protected binary to the public, including normal users and malicious attackers. Attackers fuzz the protected binary to find bugs. However, with the protection of FUZZIFICATION techniques, they cannot find as many bugs quickly. At the same time, developers distribute the normal binary to trusted parties. The trusted parties can launch fuzzing on the normal binary with the native speed and thus can find more bugs in a timely manner. Therefore, developers who receive bug reports from trusted parties can fix the bug before attackers widely abuse it.

Anti-fuzz candidates	Effective	Generic	Efficient	Robust
Pack & obfuscation	✔	✔	✗	✔
Bug injection	✔	✔	✗	✗
Fuzzer identification	✔	✗	✔	✗
Emulator bugs	✔	✗	✔	✔
FUZZIFICATION	✔	✔	✔	✔

Table 1: Possible design choices and evaluation with our goals.

2.2.1 Threat Model

We consider motivated attackers who attempt to find software vulnerabilities through state-of-the-art fuzzing techniques, but with limited resources like computing power (at most similar resources as trusted parties). Adversaries have the binary protected by FUZZIFICATION and they have knowledge of our FUZZIFICATION techniques. They can use off-the-shelf binary analysis techniques to disarm FUZZIFICATION from the protected binary. Adversaries who have access to the unprotected binary or even to program source code (*e.g.*, inside attackers, or through code leakage) are out of the scope of this study.

2.2.2 Design Goals and Choices

A FUZZIFICATION technique should achieve the following four goals simultaneously:

- **Effective:** It should effectively reduce the number of bugs found in the protected binary, compared to that found in the original binary.
- **Generic:** It tackles the fundamental principles of fuzzing and is generally applicable to most fuzzers.
- **Efficient:** It introduces minor overhead to the normal program execution.
- **Robust:** It is resistant to the adversarial analysis trying to remove it from the protected binary.

With these goals in mind, we examine four design choices for hindering malicious fuzzing, shown in Table 1. Unfortunately, no method can satisfy all goals.

Packing/obfuscation. Software packing and obfuscation are mature techniques against reverse engineering, both generic and robust. However, they usually introduce higher performance overhead to program executions, which not only hinders fuzzing, but also affects the use of normal users.

Bug injection. Injecting arbitrary code snippets that trigger non-exploitable crashes can cause additional bookkeeping overhead and affect end users in unexpected ways [31].

Fuzzer identification. Detecting the fuzzer process and changing the execution behavior accordingly can be bypassed easily (*e.g.*, by changing fuzzer name). Also, we cannot enumerate all fuzzers or fuzzing techniques.

Emulator bugs. Triggering bugs in dynamic instrumentation tools [4, 14, 38] can interrupt fuzzing [42, 43]. However, it requires strong knowledge of the fuzzer, so it is not generic.

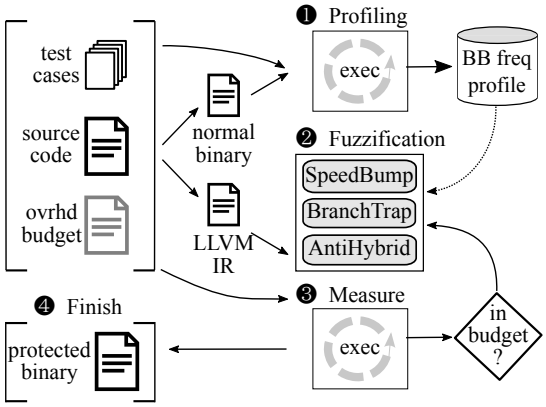


Figure 3: Overview of FUZZIFICATION process. It first runs the program with given test cases to get the execution frequency profile. With the profile, it instruments the program with three techniques. The protected binary is released if it satisfies the overhead budget.

2.3 Design Overview

We propose three FUZZIFICATION techniques – SpeedBump, BranchTrap, and AntiHybrid– to target each fuzzing technique discussed in §2.1. First, SpeedBump injects fine-grained delay primitives into cold paths that fuzzed executions frequently touch but normal executions rarely use (§3). Second, BranchTrap fabricates a number of input-sensitive branches to induce the coverage-based fuzzers to waste their efforts on fruitless paths (§4). Also, it intentionally saturates the code coverage storage with frequent path collisions so that the fuzzer cannot identify interesting inputs that trigger new paths. Third, AntiHybrid transforms explicit data-flows into implicit ones to prevent data-flow tracking through taint analysis, and inserts a large number of spurious symbols to trigger path explosion during the symbolic execution (§5).

Figure 3 shows an overview of our FUZZIFICATION system. It takes the program source code, a set of commonly used test cases, and an overhead budget as input and produces a binary protected by FUZZIFICATION techniques. Note that FUZZIFICATION relies on developers to determine the appropriate overhead budget, whatever they believe will create a balance between the functionality and security of their production. ❶ We compile the program to generate a normal binary and run it with the given normal test cases to collect basic block frequencies. The frequency information tells us which basic blocks are rarely used by normal executions. ❷ Based on the profile, we apply three FUZZIFICATION techniques to the program and generate a temporary protected binary. ❸ We measure the overhead of the temporary binary with the given normal test cases again. If the overhead is over the budget, we go back to step ❷ to reduce the slow down to the program, such as using shorter delay and adding less instrumentation. If the overhead is far below the budget, we increase the overhead accordingly. Otherwise, ❹ we generate the protected binary.

3 SpeedBump: Amplifying Delay in Fuzzing

We propose a technique called SpeedBump to slow the fuzzed execution while minimizing the effect to normal executions. Our observation is that the fuzzed execution frequently falls into paths such as error-handling (*e.g.*, wrong MAGIC bytes) that the normal executions rarely visit. We call them the *cold* paths. Injecting delays in cold paths will significantly slow fuzzed executions but will not affect regular executions that much. We first identify cold paths from normal executions with the given test cases and then inject crafted delays into least-executed code paths. Our tool automatically determines the number of code paths to inject delays and the length of each delay so that the protected binary has overhead under the user-defined budget during normal executions.

Basic block frequency profiling. FUZZIFICATION generates a basic block frequency profile to identify cold paths. The profiling process follows three steps. First, we instrument the target programs to count visited basic blocks during the execution and generate a binary for profiling. Second, with the user-provided test cases, we run this binary and collect the basic blocks visited by each input. Third, FUZZIFICATION analyzes the collected information to identify basic blocks that are rarely executed or never executed by valid inputs. These blocks are treated as cold paths in delay injection.

Our profiling does not require the given test cases to cover 100% of all legitimate paths, but just to trigger the commonly used functionalities. We believe this is a practical assumption, as experienced developers should have a set of test cases covering most of the functionalities (*e.g.*, regression test-suites). Optionally, if developers can provide a set of test cases that trigger uncommon features, our profiling results will be more accurate. For example, for applications parsing well-known file formats (*e.g.*, `readelf` parses ELF binaries), collecting valid/invalid dataset is straightforward.

Configurable delay injection. We perform the following two steps repeatedly to determine the set of code blocks to inject delays and the length of each delay:

- We start by injecting a 30ms delay to 3% of the least-executed basic blocks in the test executions. We find that this setting is close enough to the final evaluation result.
- We measure the overhead of the generated binary. If it does not exceed the user-defined overhead budget, we go to the previous step to inject more delay into more basic blocks. Otherwise, we use the delay in the previous round as the final result.

Our SpeedBump technique is especially useful for developers who generally have a good understanding of their applications, as well as the requirements for FUZZIFICATION. We provide five options that developers can use to finely tune SpeedBump's effect. Specifically, `MAX_OVERHEAD` defines the overhead budget. Developers can specify any value as long as they feel comfortable with the overhead. `DELAY_LENGTH` specifies the range of delays. We use 10ms to 300ms in the evaluation. `INCLUDE_INCORRECT` determines whether or not to inject delays to error-handling basic blocks (*i.e.*, locations that are *only* executed by invalid inputs), which is enabled by default. `INCLUDE_NON_EXEC` and `NON_EXEC_RATIO` specify whether to inject delays into how ever many basic blocks are never executed during test execution. This is useful when developers do not have a large set of test cases.

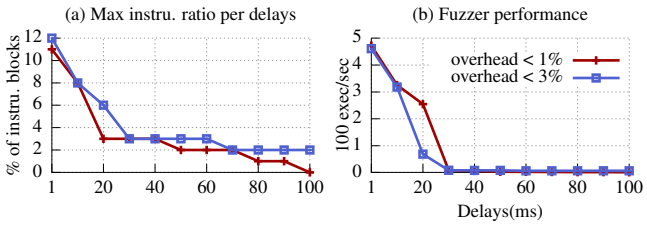


Figure 4: Protecting `readelf` with different overhead budgets. While satisfying the overhead budget, (a) demonstrates the maximum ratio of instrumentation for each delay length, and (b) displays the execution speed of AFL-QEMU on protected binaries.

Figure 4 demonstrates the impact of different options on protecting the `readelf` binary with SpeedBump. We collect 1,948 ELF files on the Debian system as valid test cases and use 600 text and image files as invalid inputs. Figure 4(a) shows the maximum ratio of basic blocks that we can inject delay into while introducing overhead less than 1% and 3%. For a 1ms delay, we can instrument 11% of the least-executed basic blocks for a 1% overhead budget and 12% for 3% overhead. For a 120ms delay, we cannot inject any blocks for 1% overhead and can inject only 2% of the cold paths for 3% overhead. Figure 4(b) shows the actual performance of AFL-QEMU when it fuzzes SpeedBump-protected binaries. The ratio of injected blocks is determined as in Figure 4(a). The result shows that SpeedBump with a 30ms delay slows the fuzzer by more than 50×. Therefore, we use 30ms and the corresponding 3% instrumentation as the starting point.

3.1 Analysis-resistant Delay Primitives

As attackers may use program analysis to identify and remove simple delay primitives (*e.g.*, calling `sleep`), we design robust primitives that involve arithmetic operations and are connected with the original code base. Our primitives are based on CSmith [66], which can generate random and bug-free code snippets with refined options. For example, CSmith can generate a function that takes parameters, performs arithmetic operations, and returns a specific type of value. We modified CSmith to generate code that has data dependencies and code dependencies to the original code. Specifically, we pass a variable from the original code to the generated code as an argument, make a reference from the generated code to the original one, and use the return value to modify a global variable of the original code. Figure 5 shows an example of our delay primitives. It declares a local variable `PASS_VAR`

```
1  //Predefined global variables
2  int32_t GLOBAL_VAR1 = 1, GLOBAL_VAR2 = 2;
3  //Randomly generated code
4  int32_t * func(int32_t p6) {
5     int32_t *l0[1000];
6     GLOBAL_VAR1 = 0x4507L; // affect global var.
7     int32_t *l1 = &g8[1][0];
8     for (int i = 0; i < 1000; i++)
9        l0[i] = p6;         // affect local var from argv.
10    (*g7) = func2(g6++);
11    (*g5) |= ~(!func3(**g4 = ~0UL));
12    return l1;             // affect global var.
13 }
14 //Inject above function for delay
15 int32_t PASS_VAR = 20;
16 GLOBAL_VAR2 = func(PASS_VAR);
```

Figure 5: Example delay primitive. Function `func` updates global variables to build data-flow dependency with original program.

and modifies global variables `GLOBAL_VAR1` and `GLOBAL_VAR2`. In this way, we introduce data-flow dependency between the original code and the injected code (line 6, 9 and 12), and change the program state without affecting the original program. Although the code is randomly generated, it is tightly coupled with the original code via data-flow and control-flow dependencies. Therefore, it is non-trivial for common binary analysis techniques, like dead-code elimination, to distinguish it from the original code. We repeatedly run the modified CSmith to find appropriate code snippets that take a specific time (*e.g.*, 10ms) for delay injection.

Safety of delay primitives. We utilize the safety checks from CSmith and FUZZIFICATION to guarantee that the generated code is bug-free. First, we use CSmith's default safety checks, which embed a collection of tests in the code, including integer, type, pointer, effect, array, initialization, and global variable. For example, CSmith conducts pointer analysis to detect any access to an out-of-scope stack variable or null pointer dereference, uses explicit initialization to prevent uninitialized usage, applies math wrapper to prevent unexpected integer overflow, and analyzes qualifiers to avoid any mismatch. Second, FUZZIFICATION also has a separate step to help detect bad side effects (*e.g.*, crashes) in delay primitives. Specifically, we run the code 10 times with fixed arguments and discard it if the execution shows any error. Finally, FUZZIFICATION embeds the generated primitives with the same fixed argument to avoid errors.

Fuzzers aware of error-handling blocks. Recent fuzzing proposals, like VUzzer [52] and T-Fuzz [48], identify error-handling basic blocks through profiling and exclude them from the code coverage calculation to avoid repetitive executions. This may affect the effectiveness of our SpeedBump technique, which uses a similar profiling step to identify cold paths. Fortunately, the cold paths from SpeedBump include not only error-handling basic blocks, but also rarely executed functional blocks. Further, we use similar methods to identify error-handling blocks from the cold paths and provide developers the option to choose not to instrument these blocks. Thus, our FUZZIFICATION will focus on instrumenting rarely executed functional blocks to maximize its effectiveness.

4 BranchTrap: Blocking Coverage Feedback

Code coverage information is widely used by fuzzers to find and prioritize interesting inputs [72, 37, 23]. We can make these fuzzers *diligent fools* if we insert a large number of conditional branches whose conditions are sensitive to slight input changes. When the fuzzing process falls into these branch traps, coverage-based fuzzers will waste their resources to explore (a huge number of) worthless paths. Therefore, we propose the technique of BranchTrap to deceive coverage-based fuzzers by misleading or blocking the coverage feedback.

4.1 Fabricating Fake Paths on User Input

The first method of BranchTrap is to *fabricate* a large number of conditional branches and indirect jumps, and inject them into the original program. Each fabricated conditional branch relies on some input bytes to determine to take the branch or not, while indirect jumps calculate their targets based on user input. Thus, the program will take different execution paths even when the input slightly changes. Once a fuzzed execution triggers the fabricated branch, the fuzzer will set a higher priority to mutate that input, resulting in the detection of more fake paths. In this way, the fuzzer will keep wasting its resources (*i.e.*, CPU and memory) to inspect fruitless but bug-free fake paths.

To effectively induce the fuzzers focusing on fake branches, we consider the following four design aspects. First, BranchTrap should fabricate a sufficient number of fake paths to affect the fuzzing policy. Since the fuzzer generates various variants from one interesting input, fake paths should provide different coverage and be directly affected by the input so that the fuzzer will keep unearthing the trap. Second, the injected new paths introduce minimal overhead to regular executions. Third, the paths in BranchTrap should be deterministic regarding user input, which means that the same input should go through the same path. The reason is that some fuzzers can detect and ignore non-deterministic paths (*e.g.*, AFL ignores one input if two executions with it take different paths). Finally, BranchTrap cannot be easily identified or removed by adversaries.

A trivial implementation of BranchTrap is to inject a jump table and use some input bytes as the index to access the table (*i.e.*, different input values result in different jump targets). However, this approach can be easily nullified by simple adversarial analysis. We design and implement a robust BranchTrap with code-reuse techniques, similar in concept to the well-known return-oriented programming (ROP) [55].

4.1.1 BranchTrap with CFG Distortion

To harden BranchTrap, we diversify the return addresses of each injected branch according to the user input. Our idea is inspired by ROP, which reuses existing code for malicious at-

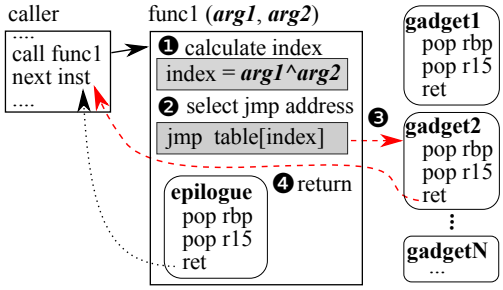


Figure 6: BranchTrap by reusing the existing ROP gadgets in the original binary. Among functionally equivalent gadgets, BranchTrap picks the one based on function arguments.

tacks by chaining various small code snippets. Our approach can heavily distort the program control-flow and makes nullifying BranchTrap more challenging for adversaries. The implementation follows three steps. First, BranchTrap collects function epilogues from the program assembly (generated during program compilation). Second, function epilogues with the same instruction sequence are grouped into one jump table. Third, we rewrite the assembly so that the function will retrieve one of several equivalent epilogues from the corresponding jump table to realize the original function return, using some input bytes as the jump table index. As we replace the function epilogue with a functional equivalent, it guarantees the identical operations as the original program.

Figure 6 depicts the internal of the BranchTrap implementation at runtime. For one function, BranchTrap ❶ calculates the `XOR`ed value of all arguments. BranchTrap uses this value for indexing the jump table (*i.e.*, candidates for epilogue address). ❷ BranchTrap uses this value as the index to visit the jump table and obtains the concrete address of the epilogue. To avoid out-of-bounds array access, BranchTrap divides the `XOR`ed value by the length of the jump table and takes the remainder as the index. ❸ After determining the target jump address, the control-flow is transferred to the gadget (*e.g.*, the same `pop rbp; pop r15; ret` gadget). ❹ Finally, the execution returns to the original return address.

The ROP-based BranchTrap has three benefits:

- **Effective:** Control-flow is constantly and sensitively changed together with the user input mutation; thus FUZZIFICATION can introduce a sufficient number of unproductive paths and make coverage feedback less effective. Also, BranchTrap guarantees the same control-flow on the same input (*i.e.*, deterministic path) so that the fuzzer will not ignore these fake paths.
- **Low overhead:** BranchTrap introduces low overhead to normal user operations (*e.g.*, less than 1% overhead) due to its lightweight operations (Store argument; XOR; Resolve jump address; Jump to gadget).
- **Robust:** The ROP-based design significantly increases the complexity for an adversary to identify or patch the binary. We evaluate the robustness of BranchTrap against adversarial analysis in §6.4.

4.2 Saturating Fuzzing State

The second method of BranchTrap is to saturate the *fuzzing state*, which blocks the fuzzers from learning the progress in the code coverage. Different from the first method, which induces fuzzers focusing on fruitless inputs, our goal here is to prevent the fuzzers from finding real interesting ones. To achieve this, BranchTrap inserts a massive number of branches to the program, and exploits the coverage representation mechanism of each fuzzer to mask new findings. BranchTrap is able to introduce an extensive number (*e.g.*, 10K to 100K) of deterministic branches to some rarely visited basic blocks. Once the fuzzer reaches these basic blocks, its coverage table will quickly fill up. In this way, most of the newly discovered paths in the following executions will be treated as *visited*, and thus the fuzzer will discard the input that in fact explores interesting paths. For example, AFL maintains a fixed-size bitmap (*i.e.*, 64KB) to track edge coverage. By inserting a large number of distinct branches, we significantly increase the probability of bitmap collision and thus reduce the coverage inaccuracy.

Figure 7(a) demonstrates the impact of bitmap saturation on fuzzing `readelf`. Apparently, a more saturated bitmap leads to fewer path discoveries. Starting from an empty bitmap, AFL identifies over 1200 paths after 10 hours of fuzzing. For the 40% saturation rate, it only finds around 950 paths. If the initial bitmap is highly filled, such as 80% saturation, AFL detects only 700 paths with the same fuzzing effort.

Fuzzers with collision mitigation. Recent fuzzers, like CollAFL [19], propose to mitigate the coverage collision issue by assigning a unique identifier to each path coverage (*i.e.*, branch in case of CollAFL). However, we argue that these techniques will not effectively undermine the strength of our BranchTrap technique on saturating coverage storage for two reasons. First, current collision mitigation techniques require program source code to assign unique identifiers during the linking time optimization [19]. In our threat model, attackers cannot obtain the program source code or the original binary – they only have a copy of the protected binary, which makes it significantly more challenging to apply similar ID-assignment algorithms. Second, these fuzzers still have to adopt a fixed size storage of coverage because of the overhead of large storage. Therefore, if we can saturate 90% of the storage, CollAFL can only utilize the remaining 10% for ID-assignment; thus the fuzzing performance will be significantly affected.

4.3 Design Factors of BranchTrap

We provide developers an interface to configure ROP-based BranchTrap and coverage saturation for optimal protection. First, the number of generated fake paths of ROP-based BranchTrap is configurable. BranchTrap depends on the number of functions to make a distorted control-flow. Therefore, injected BranchTrap is effective when the original program

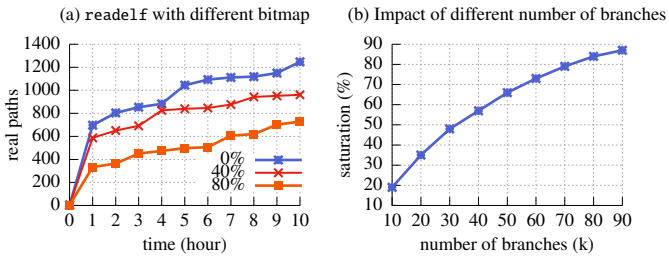


Figure 7: (a) AFL performance with different initial bitmap saturation. (b) Impact on bitmap with different number of branches.

contains plenty of functions. For binaries with fewer functions, we provide an option for developers to split existing basic blocks into multiple ones, each connected with conditional branches. Second, the size of the injected branches for saturating the coverage is also controllable. Figure 7(b) shows how the bitmap can be saturated in AFL by increasing the branch number. It clearly shows that more branches can fill up more bitmap entries. For example, 100K branches can fill up more than 90% of a bitmap entry. Injecting a massive number of branches into the program increases the output binary size. When we inject 100k branches, the size of the protected binary is 4.6MB larger than the original binary. To avoid high code size overhead, we inject a huge number of branches into only one or two of the most rarely executed basic blocks. As long as one fuzzed execution reaches such branches, the coverage storage will be filled and the following fuzzing will find fewer interesting inputs.

5 AntiHybrid: Thwarting Hybrid Fuzzers

A hybrid fuzzing method utilizes either symbolic execution or dynamic taint analysis to improve fuzzing efficiency. Symbolic (or concolic) execution is good at solving complicated branch conditions (*e.g.*, magic number and checksum), and therefore can help fuzzers bypass these hard-to-mutate roadblocks. DTA (Dynamic Taint Analysis) helps find input bytes that are related to branch conditions. Recently, several hybrid fuzzing methods have been proposed and successfully discovered security-critical bugs. For example, Driller [57] adapted selective symbolic execution and proved its efficacy during the DARPA Cyber Grand Challenge (CGC). VUzzer [52] utilized dynamic taint analysis to identify path-critical input bytes for effective input mutation. QSym [67] suggested a fast concolic execution technique that can be scalable on real-world applications.

Nevertheless, hybrid approaches have well-known weaknesses. First, both symbolic execution and taint analysis consume a large amount of resources such as CPU and memory, limiting them to analyzing simple programs. Second, symbolic execution is limited by the path explosion problem. If complex operation is required for processing symbols, the symbolic execution engine has to exhaustively explore and evaluate all execution states; then, most of the symbolic execution engines fail to run to the end of the execution path. Third, DTA analysis has difficulty in tracking implicit data dependencies, such as covert channels, control channels, or timing-based channels. For example, to cover data dependency through a control channel, the DTA engine has to aggressively propagate the taint attribute to any variable after a conditional branch, making the analysis more expensive and the result less accurate.

```
1  char input[] = ...; /* user input */
2  int  value   = ...; /* user input */
3
4  // 1. using implicit data-flow to copy input to antistr
5  //    original code: if (!strcmp(input, "condition!")) { ... }
6  char antistr[strlen(input)];
7  for (int i = 0; i<strlen(input); i++){
8    int ch = 0, temp = 0, temp2 = 0;
9    for (int j = 0; j<8; j++){
10     temp = input[i];
11     temp2 = temp & (1<<j);
12     if (temp2 != 0) ch |= 1<<j;
13   }
14   antistr[i] = ch;
15 }
16 if (!strcmp(antistr, "condition!")) { ... }
17
18 // 2. exploding path constraints
19 //    original code: if (value == 12345)
20 if (CRC_LOOP(value) == OUTPUT_CRC) { ... }
```

Figure 8: Example of AntiHybrid techniques. We use implicit data-flow (line 6-15) to copy strings to hinder dynamic taint analysis. We inject hash function around equal comparison (line 20) to cripple symbolic execution engine.

Introducing implicit data-flow dependencies. We transform the explicit data-flows in the original program into implicit data-flows to hinder taint analysis. FUZZIFICATION first identifies branch conditions and interesting information sinks (*e.g.*, strcmp) and then injects data-flow transformation code according to the variable type. Figure 8 shows an example application of AntiHybrid, where array `input` is used to decide branch condition and `strcmp` is an interesting sink function. Therefore, FUZZIFICATION uses implicit data-flows to copy the array (line 6-15) and replaces the original variable to the new one (line 16). Due to the transformed implicit data-flow, the DTA technique cannot identify the correct input bytes that affect the branch condition at line 16.

Implicit data-flow hinders data-flow analysis that tracks direct data propagation. However, it cannot prevent data dependency inference through differential analysis. For example, recent work, RedQueen [2], infers the potential relationship between input and branch conditions through pattern matching, and thus can bypass the implicit data-flow transformation. However, RedQueen requires the branch condition value to be explicitly shown in the input, which can be easily fooled through simple data modification (*e.g.*, adding the same constant value to both operands of the comparison).

Exploding path constraints. To hinder hybrid fuzzers using symbolic execution, FUZZIFICATION injects multiple code chunks to intentionally trigger path explosions. Specifi-

Project	Version	Program	Arg.	Seeds	Overhead (Binary size)				Overhead (CPU)			
					Speed	BranchTrap	AntiHybrid	All	Speed	BranchTrap	AntiHybrid	All
libjpeg	2017.7	djpeg		GIT	9.0% (0.1M)	101.5% (1.2M)	0.3% (0.0M)	103.2% (1.3M)	1.5%	0.9%	0.3%	2.4%
libpng	1.6.27	readpng		GIT	6.2% (0.1M)	56.0% (1.3M)	0.9% (0.0M)	65.7% (1.5M)	1.8%	2.0%	0.3%	4.0%
libtiff	4.0.6	tiffinfo		GIT	9.2% (0.2M)	72.5% (1.5M)	0.8% (0.0M)	77.3% (1.6M)	1.0%	2.1%	0.5%	4.8%
pcre2	10	pcre2test		built-in	12.9% (0.2M)	85.3% (1.3M)	0.8% (0.0M)	108.6% (1.7M)	1.2%	1.2%	1.0%	3.1%
binutils	2.23	readelf	-a	ELF files	9.6% (0.2M)	77.3% (1.3M)	0.2% (0.0M)	81.0% (1.4M)	1.0%	0.9%	0.9%	3.1%
		objdump	-d		1.4% (0.1M)	17.0% (1.3M)	0.1% (0.0M)	17.5% (1.3M)	1.6%	2.0%	0.9%	4.6%
		nm			1.9% (0.1M)	23.1% (1.2M)	0.1% (0.0M)	23.3% (1.2M)	1.8%	1.6%	1.1%	4.5%
		objcopy	-S		1.7% (0.1M)	20.2% (1.3M)	0.1% (0.0M)	20.6% (1.3M)	1.7%	0.8%	0.5%	2.9%
Average					6.5%	56.6%	0.4%	62.1%	1.4%	1.4%	0.7%	3.7%

Table 2: Code size overhead and performance overhead of fuzzified binaries. `GIT` means Google Image Test-suite. We set performance overhead budget as 5%. For size overhead, we show the percentage and the increased size.

cally, we replace each comparison instruction by comparing the hash values of the original comparison operands. We adopt the hash function because symbolic execution cannot easily determine the original operand with the given hash value. As hash functions usually introduce non-negligible overhead to program execution, we utilize the lightweight cyclic redundancy checking (CRC) loop iteration to transform the branch condition to reduce performance overhead. Although theoretically CRC is not as strong as hash functions for hindering symbolic execution, it also introduces significant slow down. Figure 8 shows an example of the path explosion instrumentation. To be specific, FUZZIFICATION changes the original condition `(value == 12345)` to `(CRC_LOOP(value) == OUTPUT_CRC)` (at line 20). If symbolic execution decides to solve the constraint of the CRC, it will mostly return a timeout error due to the complicated mathematics. For example, QSym, a state-of-the-art fast symbolic execution engine, is armed with many heuristics to scale on real-world applications. When QSym first tries to solve the complicated constraint that we injected, it will fail due to the timeout or path explosion. Once injected codes are run by the fuzzer multiple times, QSym identifies the repetitive basic blocks (*i.e.*, injected hash function) and performs *basic block pruning*, which decides not to generate a further constraint from it to assign resources into a new constraint. After that, QSym will not explore the condition with the injected hash function; thus, the code in the branch can be explored rarely.

6 Evaluation

We evaluate our FUZZIFICATION techniques to understand their effectiveness on hindering fuzzers from exploring program code paths (§6.1) and detecting bugs (§6.2), their practicality of protecting real-world large programs (§6.3), and their robustness against adversarial analysis techniques (§6.4).

Implementation. Our FUZZIFICATION framework is implemented in a total of 6,559 lines of Python code and 758 lines of C++ code. We implement the SpeedBump technique as an LLVM pass and use it to inject delays into cold blocks during the compilation. For the BranchTrap, we analyze the assembly code and modify it directly. For the AntiHybrid technique, we use an LLVM pass to introduce the path explosion and utilize a python script to automatically inject implicit data-flows. Currently, our system supports all three FUZZIFICATION techniques on 64bit applications, and is able to protect 32bit applications except for the ROP-based BranchTrap.

Tasks	Target	AFL	HonggFuzz	QSym	VUzzer
Coverage	8 binaries	O,S,B,H,A	O,S,B,H,A	O,S,B,H,A	–
	MuPDF	O,A	O,A	O,A	–
Crash	4 binaries	O,A	O,A	O,A	–
	LAVA-M	O,A	O,A	O,A	O,A

Table 3: Experiments summary. Protection options: **O**riginal, **S**peedBump, **B**ranchTrap, Anti**H**ybrid, **A**ll. We use 4 binutils binaries, 4 binaries from Google OSS project and `MuPDF` to measure the code coverage. We use binutils binaries and LAVA-M programs to measure the number of unique crashes.

Experimental setup. We evaluate FUZZIFICATION against four state-of-the-art fuzzers that work on binaries, specifically, AFL in QEMU mode, HonggFuzz in Intel-PT mode, VUzzer 32[1], and QSym with AFL-QEMU. We set up the evaluation on two machines, one with Intel Xeon CPU E7-8890 v4@2.20GHz, 192 processors and 504 GB of RAM, and another with Intel Xeon CPU E7-4820@2.00GHz, 32 processors and 128 GB of RAM.

To get reproducible results, we tried to eliminate the non-deterministic factors from fuzzers: we disable the address space layout randomization of the experiment machine and force the deterministic mode for AFL. However, we have to leave the randomness in HonggFuzz and VUzzer, as they do not support deterministic fuzzing. Second, we used the same set of test cases for basic block profiling in FUZZIFICATION, and fed the same seed inputs for different fuzzers. Third,

[1]We also tried to use VUzzer64 to fuzz different programs, but it did not find any crashes even for any original binary after three-day fuzzing. Since VUzzer64 is still experimental, we will try the stable version in the future.

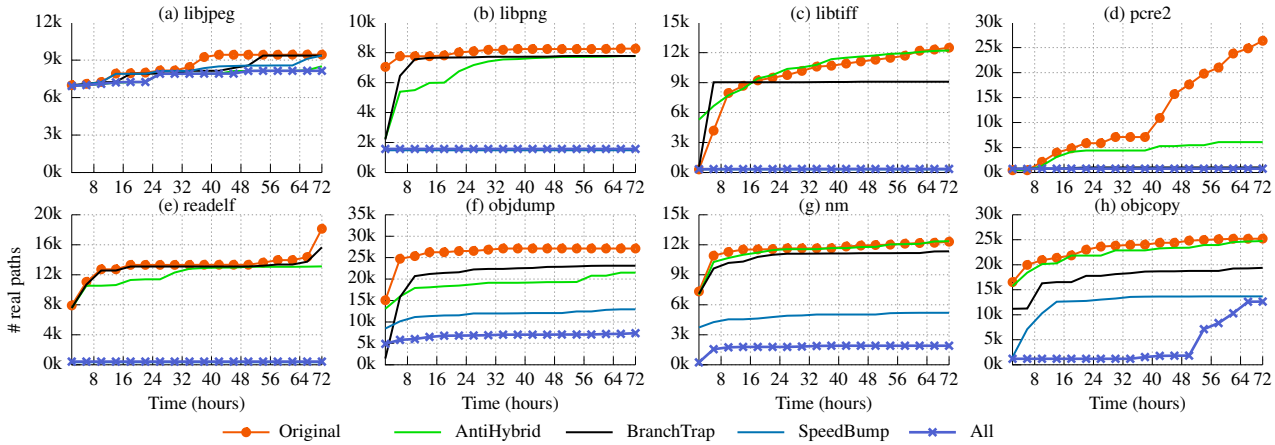


Figure 9: Paths discovered by AFL-QEMU from real-world programs. Each program is compiled with five settings: original (no protection), SpeedBump, BranchTrap, AntiHybrid, and all protections. We fuzz them with AFL-QEMU for three days.

Category	Option	Design Choice
SpeedBump	max_overhead	2%
	delay_length	10ms to 300ms
	include_invalid	True
	include_non_exec	True (5%)
BranchTrap	max_overhead	2%
	bitmap_saturation	40% of 64k bitmap
AntiHybrid	max_overhead	1%
	include_non_exec	True (5%)
Overall	max_overhead	5%

Table 4: Our configuration values for the evaluation.

we used identical FUZZIFICATION techniques and configurations when we conducted code instrumentation and binary rewriting for each target application. Last, we pre-generated FUZZIFICATION primitives (*e.g.*, SpeedBump codes for 10ms to 300ms and BranchTrap codes with deterministic branches), and used the primitives for all protections. Note that developers should use different primitives for the actual releasing binary to avoid code pattern matching analysis.

Target applications. We select the LAVA-M data set [17] and nine real-world applications as the fuzzing targets, which are commonly used to evaluate the performance of fuzzers [11, 19, 64, 52]. The nine real-world programs include four applications from the Google fuzzer test-suite [24], four programs from the binutils [20] (shown in Table 2), and the PDF reader `MuPDF`. We perform two sets of experiments on these binaries, summarized in Table 3. First, we fuzz nine real-world programs with three fuzzers (all except VUzzer[2]) to measure the impact of FUZZIFICATION on finding code paths. Specifically, we compile eight real-world programs (all except `MuPDF`) with five different settings: original (no protection), SpeedBump, BranchTrap, AntiHybrid, and a combination of three techniques (full protection). We compile `MuPDF` with two settings for simplicity: no protection and full protection. Second, we use three fuzzers to fuzz four binutils programs and all four fuzzers to fuzz LAVA-M programs to evaluate the impact of FUZZIFICATION on unique bug finding. All fuzzed programs in this step are compiled in two versions: with no protection and with full protection. We compiled the LAVA-M program to a 32bit version in order to be comparable with previous research. Table 4 shows the configuration of each technique used in our compilation. We changed the fuzzer's timeout if the binaries cannot start with the default timeout (*e.g.*, 1000 ms for AFL-QEMU).

Evaluation metric. We use two metrics to measure the effectiveness of FUZZIFICATION: code coverage in terms of discovered *real path*s, and unique crashes. Real path is the execution path shown in the original program, excluding the fake ones introduced by BranchTrap. We further excluded the real paths triggered by seed inputs so that we can focus on the ones discovered by fuzzers. Unique crash is measured as the input that can make the program crash with a distinct real path. We filter out duplicate crashes that are defined in AFL [71] and are widely used by other fuzzers [11, 36].

6.1 Reducing Code Coverage

6.1.1 Impact on Normal Fuzzers

We measure the impact of FUZZIFICATION on reducing the number of real paths against AFL-QEMU and HonggFuzz-Intel-PT. Figure 9 shows the 72-hour fuzzing result from AFL-QEMU on different programs with five protection settings. The result of HonggFuzz-Intel-PT is similar and we leave it in Appendix A.

In summary, with all three techniques, FUZZIFICATION can reduce discovered real paths by 76% to AFL, and by

[2]Due to time limit, we only use VUzzer 32 to finding bugs from LAVA-M programs. We plan to do other evaluations in the future.

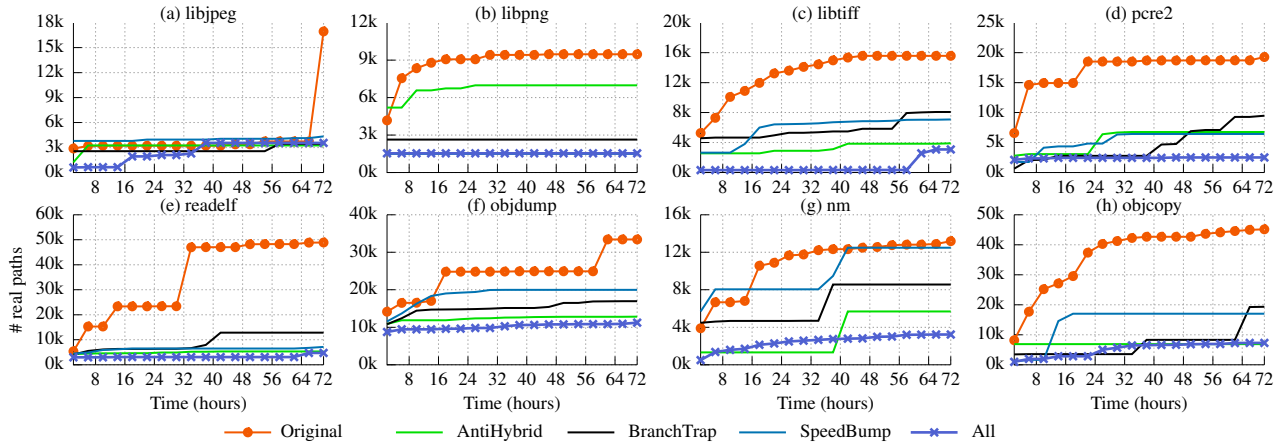


Figure 10: Paths discovered by QSym from real-world programs. Each program is compiled with the same five settings as in Figure 9. We fuzz these programs for three days, using QSym as the symbolic execution engine and AFL-QEMU as the native fuzzer.

67% to HonggFuzz, on average. For AFL, the reduction rate varies from 14% to 97% and FUZZIFICATION reduces over 90% of path discovery for `libtiff`, `pcre2` and `readelf`. For HonggFuzz, the reduction rate is between 38% to 90% and FUZZIFICATION only reduces more than 90% of paths for `pcre2`. As FUZZIFICATION automatically determines the details for each protection to satisfy the overhead budget, its effect varies for different programs.

Table 5 shows the effect of each technique on hindering path discovery. Among them, SpeedBump achieves the best protection against normal fuzzers, followed by BranchTrap and AntiHybrid. Interestingly, although AntiHybrid is developed to hinder hybrid approaches, it also helps reduce the discovered paths in normal fuzzers. We believe this is mainly caused by the slow down in fuzzed executions.

We measured the overhead by different FUZZIFICATION techniques, on program size and execution speed. The result is given in Table 2. In summary, FUZZIFICATION satisfies the user-specified overhead budget, but shows relatively high space overhead. On average, binaries armed with FUZZIFICATION are 62.1% larger than the original ones. The extra code mainly comes from the BranchTrap technique, which inserts massive branches to achieve bitmap saturation. Note that the extra code size is almost the same across different programs. Therefore, the size overhead is high for small programs, but is negligible for large applications. For example, the size overhead is less than 1% for LibreOffice applications, as we show in Table 7. Further, BranchTrap is configurable, and developers may inject a smaller number of fake branches to small programs to avoid large-size overhead.

Analysis on less effective results. FUZZIFICATION shows less effectiveness on protecting the `libjpeg` application. Specifically, it decreases the number of real paths on `libjpeg` by 13% to AFL and by 37% to HonggFuzz, whereas the average reduction is 76% and 67%, respectively. We analyzed

	SpeedBump	BranchTrap	AntiHybrid	All
AFL-QEMU	-66%	-23%	-18%	-74%
HonggFuzz (PT)	-44%	-14%	-7%	-61%
QSym (AFL-QEMU)	-59%	-58%	-67%	-80%
Average	-56%	-31%	-30%	-71%

Table 5: Reduction of discovered paths by FUZZIFICATION techniques. Each value is an average of the fuzzing result from eight real-world programs, as shown in Figure 9 and Figure 10.

FUZZIFICATION on `libjpeg` and find that SpeedBump and BranchTrap cannot effectively protect `libjpeg`. Specifically, these two techniques only inject nine basic blocks within the user-specified overhead budget (2% for SpeedBump and 2% for BranchTrap), which is less than 0.1% of all basic blocks. To address this problem, developers may increase the overhead budget so that FUZZIFICATION can insert more roadblocks to protect the program.

6.1.2 Impact on Hybrid Fuzzers

We also evaluated FUZZIFICATION's impact on code coverage against QSym, a hybrid fuzzer that utilizes symbolic execution to help fuzzing. Figure 10 shows the number of real paths discovered by QSym from the original and protected binaries. Overall, with all three techniques, FUZZIFICATION can reduce the path coverage by 80% to QSym on average, and shows consistent high effectiveness on all tested programs. Specifically, the reduction rate varies between 66% (`objdump`) to 90% (`readelf`). The result of libjpeg shows an interesting pattern: QSym finds a large number of real paths from the original binary in the last 8 hours, but it did not get the same result from any protected binary. Table 5 shows that AntiHybrid achieves the best effect (67% path reduction) against hybrid fuzzers, followed by SpeedBump (59%) and BranchTrap (58%).

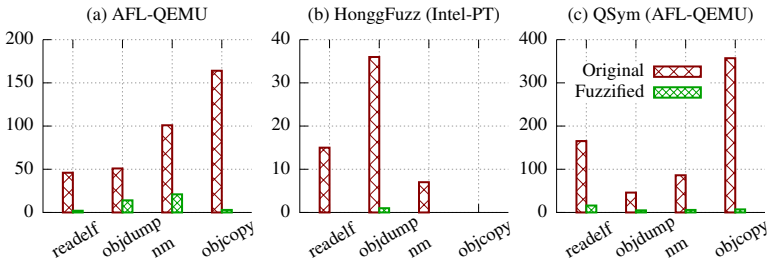


Figure 11: Crashes found by different fuzzers from binutils programs. Each program is compiled as original (no protection) and fuzzified (three techniques) and is fuzzed for three days.

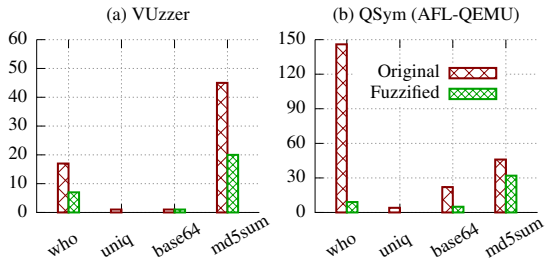


Figure 12: Bugs found by VUzzer and QSym from LAVA-M dataset. HonggFuzz discovers three bugs from the original `uniq`. AFL does not find any bug.

Comparison with normal fuzzing result. QSym uses efficient symbolic execution to help find new paths in fuzzing, and therefore it is able to discover 44% more real paths than AFL from original binaries. As we expect, AntiHybrid shows the most impact on QSym (67% reduction), and less effect on AFL (18%) and HonggFuzz (7%). With our FUZZIFICATION techniques, QSym shows less advantage over normal fuzzers, reduced from 44% to 12%.

	who	uniq	base64	md5sum	Average
Overhead (Size)	17.1% (0.3M)	220.6% (0.3M)	220.0% (0.3M)	210.7% (0.3M)	167.1%
Overhead (CPU)	22.7%	13.2%	21.1%	6.5%	15.9%

Table 6: Overhead of FUZZIFICATION on LAVA-M binaries (all protections except ROP-based BranchTrap) . The overhead is higher as LAVA-M binaries are relatively small (*e.g.*, ≈ 200KB).

6.2 Hindering Bug Finding

We measure the number of unique crashes that fuzzers find from the original and protected binaries. Our evaluation first fuzzes four binutils programs and LAVA-M applications with three fuzzers (all but VUzzer). Then we fuzz LAVA-M programs with VUzzer, where we compiled them into 32bit versions and excluded the protection of ROP-based BranchTrap, which is not implemented yet for 32bit programs.

6.2.1 Impact on Real-World Applications

Figure 11 shows the total number of unique crashes discovered by three fuzzers in 72 hours. Overall, FUZZIFICATION reduces the number of discovered crashes by 93%, specifically, by 88% to AFL, by 98% to HonggFuzz, and by 94% to QSym. If we assume a consistent crash-discovery rate along the fuzzing process, fuzzers have to take 40 times more effort to detect the same number of crashes from the protected binaries. As the crash-discovery rate usually reduces over time in real-world fuzzing, fuzzers will have to take much more effort. Therefore, FUZZIFICATION can effectively hinder fuzzers and makes them spend significantly more time discovering the same number of crash-inducing inputs.

6.2.2 Impact on LAVA-M Dataset

Compared with other tested binaries, LAVA-M programs are smaller in size and simpler in operation. If we inject a 1ms delay on 1% of rarely executed basic block on `who` binary, the program will suffer a slow down of more than 40 times. To apply FUZZIFICATION on the LAVA-M dataset, we allow higher overhead budget and apply more fine-grained FUZZIFICATION. Specifically, we used tiny delay primitives (*i.e.*, 10 µs to 100 µs), tuned the ratio of basic block instrumentation from 1% to 0.1%, reduced the number of applied AntiHybrid components, and injected smaller deterministic branches to reduce the code size overhead. Table 6 shows the run-time and space overhead of the generated LAVA-M programs with FUZZIFICATION techniques.

After fuzzing the protected binaries for 10 hours, AFL-QEMU does not find any crash. HonggFuzz detects three crashes from the original `uniq` binary and cannot find any crash from any protected binary. Figure 12 illustrates the fuzzing result of VUzzer and QSym. Overall, FUZZIFICATION can reduce 56% of discovered bugs to VUzzer and 78% of discovered bugs to QSym. Note that the fuzzing result on the original binaries is different from the ones reported in the original papers [67, 52] for several reasons: VUzzer and QSym cannot eliminate non-deterministic steps during fuzzing; we run the AFL part of each tool in QEMU mode; LAVA-M dataset is updated with several bug fixes[3].

6.3 Anti-fuzzing on Realistic Applications

To understand the practicality of FUZZIFICATION on large and realistic applications, we choose six programs that have a graphical user interface (GUI) and depend on tens of libraries. As fuzzing large and GUI programs is a well-known challenging problem, our evaluation here focuses on measuring the overhead of FUZZIFICATION techniques and the functionality

[3] https://github.com/panda-re/lava/search?q=bugfix&type=Commits

Category	Program	Version	Overhead	
			Size	CPU
LibreOffice	Writer	6.2	< 1% (+1.3 MB)	0.4%
	Calc		< 1% (+1.3 MB)	0.4%
	Impress		< 1% (+1.3 MB)	0.2%
Music Player	Clementine	1.3	4.3% (+1.3 MB)	0.5%
PDF Reader	MuPDF	1.13	4.1% (+1.3 MB)	2.2%
Image Viewer	Nomacs	3.10	21% (+1.2 MB)	0.7%
Average			5.4%	0.73%

Table 7: FUZZIFICATION on GUI applications. The CPU overhead is calculated on the application launching time. Due to the fixed code injection, code size overhead is negligible for these large applications.

of protected programs. When applying the SpeedBump technique, we have to skip the basic block profiling step due to the lack of command-line interface (CLI) support (*e.g.*, `readelf` parses ELF file and displays results in command line); thus, we only insert slow down primitives into error-handling routines. For the BranchTrap technique, we choose to inject massive fake branches into basic blocks near the entry point. In this way, the program execution will always pass the injected component so that we can measure runtime overhead correctly. We apply the AntiHybrid technique directly.

For each protected application, we first manually run it with multiple inputs, including given test cases, and confirm that FUZZIFICATION does not affect the program's original functionality. For example, `MuPDF` successfully displays, edits, saves, and prints all tested PDF documents. Second, we measure the code size and runtime overhead of the protected binaries for given test cases. As shown in Table 7, on average, FUZZIFICATION introduces 5.4% code size overhead and 0.73% runtime overhead. Note that the code size overhead is much smaller than that of previous programs (*i.e.*, 62.1% for eight relatively small programs Table 2 and over 100% size overhead for simple LAVA-M programs Table 6).

Anti-fuzzing on MuPDF. We also evaluated the effectiveness of FUZZIFICATION on protecting `MuPDF` against three fuzzers – AFL, HonggFuzz, and QSym– as `MuPDF` supports the CLI interface through the tool called "mutool." We compiled the binary with the same parameter shown in Table 4 and performed basic block profiling using the CLI interface. After 72-hours of fuzzing, no fuzzer finds any bug from `MuPDF`. Therefore, we instead compare the number of real paths between the original binary and the protected one. As shown in Figure 13, FUZZIFICATION reduces the total paths by 55% on average, specifically, by 77% to AFL, by 36% to HonggFuzz, and 52% to QSym. Therefore, we believe it is more challenging for real-world fuzzers to find bugs from protected applications.

	Pattern matching	Control analysis	Data analysis	Manual analysis
SpeedBump	✔	✔	✔	-
BranchTrap	✔	✔	✔	-
AntiHybrid	-	✔	✔	-

Table 8: Defense against adversarial analysis. ✔ indicates that the FUZZIFICATION technique is resistant to that adversarial analysis.

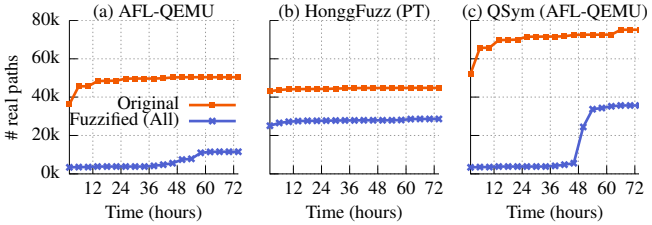


Figure 13: Paths discovered by different fuzzers from the original `MuPDF` and the one protected by three FUZZIFICATION techniques.

6.4 Evaluating Best-effort Countermeasures

We evaluate the robustness of FUZZIFICATION techniques against off-the-shelf program analysis techniques that adversaries may use to reverse our protections. However, the experiment results do not particularly indicate that FUZZIFICATION is robust against strong adversaries with incomparable computational resources.

Table 8 shows the analysis we covered and summarizes the evaluation result. First, attackers may search particular code patterns from the protected binary in order to identify injected protection code. To test anti-fuzzing against pattern matching, we examine a number of code snippets that are repeatedly used throughout the protected binaries. We found that the injected code by AntiHybrid crafts several observable patterns, like hash algorithms or data-flow reconstruction code, and thus could be detected by attackers. One possible solution to this problem is to use existing diversity techniques to eliminate the common patterns [35]. We confirm that no specific patterns can be found in SpeedBump and BranchTrap because we leverage CSmith [66] to randomly generate a new code snippet for each FUZZIFICATION process.

Second, control-flow analysis can identify unused code in a given binary automatically and thus automatically remove it (*i.e.*, dead code elimination). However, this technique cannot remove our FUZZIFICATION techniques, as all injected code is cross-referenced with the original code. Third, data-flow analysis is able to identify the data dependency. We run protected binaries inside the debugging tool, GDB, to inspect data dependencies between the injected code and the original code. We confirm that data dependencies always exist via global variables, arguments, and the return values of injected functions. Finally, we consider an adversary who is capable of conducting manual analysis for identifying the anti-fuzzing code with the knowledge of our techniques. It is worth noting that we do not consider strong adversaries who are capable

of analyzing the application logic for vulnerability discovery. Since FUZZIFICATION injected codes are supplemental to the original functions, we conclude that the manual analysis can eventually identify and nullify our techniques by evaluating the actual functionality of the code. However, since the injected code is functionally similar to normal arithmetic operations and has control- and data-dependencies on the original code, we believe that the manual analysis is time-consuming and error-prone, and thus we can deter the time for revealing real bugs.

7 Discussion and Future Work

In this section, we discuss the limitations of FUZZIFICATION and suggest provisional countermeasures against them.

Complementing attack mitigation system. The goal of anti-fuzzing is not to completely hide all vulnerabilities from adversaries. Instead, it introduces an expensive cost on the attackers' side when they try to fuzz the program to find bugs, and thus developers are able to detect bugs first and fix them in a timely manner. Therefore, we believe our anti-fuzzing technique is an important complement to the current attack mitigation ecosystem. Existing mitigation efforts either aim to avoid program bugs (*e.g.*, through type-safe language [32, 44]) or aim to prevent successful exploits, assuming attackers will find bugs anyway (*e.g.*, through control-flow integrity [1, 16, 30]). As none of these defenses can achieve 100% protection, our FUZZIFICATION techniques provide another level of defense that further enhances program security. However, we emphasize that FUZZIFICATION alone cannot provide the best security. Instead, we should keep working on all aspects of system security toward a completely secure computer system, including but not limited to secure development process, effective bug finding, and efficient runtime defense.

Best-effort protection against adversarial analysis. Although we examined existing generic analyses and believe they cannot completely disarm our FUZZIFICATION techniques, the defensive methods only provide a best-effort protection. First, if attackers have almost unlimited resources, such as when they launch APT (advanced persistent threat) attacks, no defense mechanism can survive the powerful adversarial analysis. For example, with extremely powerful binary-level control-flow analysis and data-flow analysis, attackers may finally identify the injected branches by BranchTrap and thus reverse it for an unprotected binary. However, it is hard to measure the amount of required resources to achieve this goal, and meanwhile, developers can choose more complicated branch logic to mitigate reversing. Second, we only examined currently existing techniques and cannot cover all possible analyses. It is possible that attackers who know the details of our FUZZIFICATION techniques propose a specific method to effectively bypass the protection, such as by utilizing our implementation bugs. But in this case, the anti-fuzzing technique will also get updated quickly to block the specific attack once we know the reversing technique. Therefore, we believe the anti-fuzzing technique will get improved continuously along the back-and-forth attack and defense progress.

Trade-off performance for security. FUZZIFICATION improves software security at the cost of a slight overhead, including code size increase and execution slow down. A similar trade-off has been shown in many defense mechanisms and affects the deployment of defense mechanisms. For example, address space layout randomization (ASLR) has been widely adopted by modern operating systems due to small overhead, while memory safety solutions still have a long way to go to become practical. Fortunately, the protection by FUZZIFICATION is quite flexible, where we provide various configuration options for developers to decide the optimal trade-off between security and performance, and our tool will automatically determine the maximum protection under the overhead budget.

Delay primitive on different H/W environments. We adopt CSmith-generated code as our delay primitives using measured delay on one machine (*i.e.*, developer's machine). This configuration implies that those injected delays might not be able to bring the expected slow down to the fuzzed execution with more powerful hardware support. On the other hand, the delay primitives can cause higher overhead than expected for regular users with less powerful devices. To handle this, we plan to develop an additional variation that can dynamically adjust the delay primitives at runtime. Specifically, we measure the CPU performance by monitoring a few instructions and automatically adjusting a loop counter in the delay primitives to realize the accurate delay in different hardware environments. However, the code may expose static pattern such as time measurement system call or a special instruction like `rdtsc`; thus we note that this variation has inevitable trade-off between adaptability and robustness.

8 Related Work

Fuzzing. Since the first proposal by Barton Miller in 1990 [40], fuzzing has evolved into a standard method for automatic program testing and bug finding. Various fuzzing techniques and tools have been proposed [57, 52, 29, 21, 34], developed [72, 37, 25, 23, 18, 9], and used to find a large number of program bugs [51, 72, 59, 26, 10]. There are continuous efforts to help improve fuzzing efficiency by developing a more effective feedback loop [6], proposing new OS primitives [64], and utilizing clusters for large-scale fuzzing [22, 24, 39].

Recently, researchers have been using fuzzing as a general way to explore program paths with specialties, such as maximizing CPU usage [49], reaching a particular code location [5], and verifying the deep learning result empiri-

cally [47]. All these works result in a significant improvement to software security and reliability. In this paper, we focus on the opposite side of the double-edged sword, where attackers abuse fuzzing techniques to find zero-day vulnerabilities and thus launch a sophisticated cyber attack. We build effective methods to hinder attackers on bug finding using FUZZIFICATION, which can provide developers and trusted researchers time to defeat the adversarial fuzzing effort.

Anti-fuzzing techniques. A few studies briefly discuss the concept of anti-fuzzing [63, 27, 41, 31]. Among them, Göransson *et al*. evaluated two straightforward techniques, *i.e*., crash masking to prevent fuzzers finding crashes and fuzzer detection to hide functionality when being fuzzed [27]. However, attackers can easily detect these methods and bypass them for effective fuzzing. Our system provides a fine-grained controllable method to slow the fuzzed execution and introduces effective ways to manipulate the feedback loop to fool fuzzers. We also consider defensive mechanisms to prevent attackers from removing our anti-fuzzing techniques.

Hu *et al*. proposed to hinder attacks by injecting provably (but not obviously) non-exploitable bugs to the program, called "Chaff Bugs" [31]. These bugs will confuse bug analysis tools and waste attackers' effort on exploit generation. Both chaff bugs and FUZZIFICATION techniques work on close-source programs. Differently, our techniques hinder bug finding in the first place, eliminating the chance for an attacker to analyze bugs or construct exploits. Further, both techniques may affect normal-but-rare usage of the program. However, our methods, at most, introduce slow down to the execution, while improper chaff bugs lead to crashes, thus harming the usability.

Anti-analysis techniques. Anti-symbolic-execution and anti-taint-analysis are well-known topics. Sharif *et al*. [56] designed a conditional code obfuscation that encrypts a conditional branch with cryptographic operations. Wang *et al*. [62] proposed a method to harden the binary from symbolic execution by using linear operations instead of cryptographic functions. However, neither of them considered performance overhead as an evaluation metric. SymPro [7] presented symbolic profiling, a method to identify and diagnose bottlenecks of the application under symbolic execution. Cavallaro *et al*. [8] showed a comprehensive collection of evading techniques on dynamic-taint-analysis.

Software obfuscation and diversity. Software obfuscation transforms the program code into obscure formats that are difficult to analyze so as to prevent unexpected reverse engineering [12, 13]. Various tools have been developed to obfuscate binaries [15, 60, 33, 46]. However, obfuscation is not effective to impede unexpected fuzzing because it focuses on evading static analysis and the original program logic is still revealed at runtime. Software diversity instead provides different implementations of the same program for different execution environments, aiming to either limit attacks on a specific version (usually a small set of all distributions), or significantly increase the effort to build generic exploits [35, 3, 53, 50]. Fuzzing one of many diversified versions could be less effective if the identified bug is specific to one version (which is likely caused by an implementation error of the diversity mechanism). However, for bugs stemming from a programming mistake, diversity cannot help hinder attackers finding them.

9 Conclusion

We propose a new attack mitigation system, called FUZZIFICATION, for developers to prevent adversarial fuzzing. We develop three principled ways to hinder fuzzing: injecting delays to slow fuzzed executions; inserting fabricated branches to confuse coverage feedback; transforming data-flows to prevent taint analysis and utilizing complicated constraints to cripple symbolic execution. We design robust anti-fuzzing primitives to hinder attackers from bypassing FUZZIFICATION. Our evaluation shows that FUZZIFICATION can reduce paths exploration by 70.3% and reduce bug discovery by 93.0% for real-world binaries, and reduce bug discovery by 67.5% for LAVA-M dataset.

10 Acknowledgment

We thank the anonymous reviewers, and our shepherd, Stephen McCamant, for their helpful feedback. This research was supported, in part, by the NSF award CNS-1563848, CNS-1704701, CRI-1629851 and CNS-1749711 ONR under grant N00014-18-1-2662, N00014-15-1-2162, N00014-17-1-2895, DARPA TC (No. DARPA FA8650-15-C-7556), and ETRI IITP/KEIT[B0101-17-0644], and gifts from Facebook, Mozilla and Intel.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, 2005.

[2] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2019.

[3] Algirdas Avizienis and Liming Chen. On the Implementation of N-version Programming for Software Fault Tolerance during Execution. *Proceedings of the IEEE COMPSAC*, pages 149–155, 1977.

[4] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the 2005 USENIX Annual Technical Conference (ATC)*, Anaheim, CA, April 2005.

[5] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed Greybox Fuzzing. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, October–November 2017.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, October 2016.

[7] James Bornholt and Emina Torlak. Finding Code that Explodes under Symbolic Evaluation. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), 2018.

[8] Lorenzo Cavallaro, Prateek Saxena, and R Sekar. Anti-taint-analysis: Practical Evasion Techniques against Information Flow based Malware Defense. Technical report, Stony Brook University, 2007.

[9] CENSUS. Choronzon - An Evolutionary Knowledge-based Fuzzer, 2015. ZeroNights Conference.

[10] Oliver Chang, Abhishek Arya, and Josh Armour. OSS-Fuzz: Five Months Later, and Rewarding Projects, 2018. https://security.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html.

[11] Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2018.

[12] Christian Collberg, Clark Thomborson, and Douglas Low. A Taxonomy of Obfuscating Transformations. Technical report, Department of Computer Science, University of Auckland, New Zealand, 1997.

[13] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998.

[14] Timothy Garnett Derek Bruening, Vladimir Kiriansky. Dynamic Instrumentation Tool Platform. http://www.dynamorio.org/, 2009.

[15] Theo Detristan, Tyll Ulenspiegel, Mynheer Superbus Von Underduk, and Yann Malcom. Polymorphic Shellcode Engine using Spectrum Analysis, 2003. http://phrack.org/issues/61/9.html.

[16] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. Efficient Protection of Path-Sensitive Control Security. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, August 2017.

[17] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. LAVA: Large-scale Automated Vulnerability Addition. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[18] Michael Eddington. Peach Fuzzing Platform. *Peach Fuzzer*, page 34, 2011.

[19] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. CollAFL: Path Sensitive Fuzzing. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2018.

[20] GNU Project. GNU Binutils Collection. https://www.gnu.org/software/binutils, 1996.

[21] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2008.

[22] Google. Fuzzing for Security, 2012. https://blog.chromium.org/2012/04/fuzzing-for-security.html.

[23] Google. Honggfuzz, 2016. https://google.github.io/honggfuzz/.

[24] Google. OSS-Fuzz - Continuous Fuzzing for Open Source Software, 2016. https://github.com/google/oss-fuzz.

[25] Google. Syzkaller - Linux Syscall Fuzzer, 2016. https://github.com/google/syzkaller.

[26] Google. Honggfuzz Found Bugs, 2018. https://github.com/google/honggfuzz#trophies.

[27] David Göransson and Emil Edholm. Escaping the Fuzz. Master's thesis, Chalmers University of Technology, Gothenburg, Sweden, 2016.

[28] Munawar Hafiz and Ming Fang. Game of Detections: How Are Security Vulnerabilities Discovered in the Wild? *Empirical Software Engineering*, 21(5):1920–1959, October 2016.

[29] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with Code Fragments. In *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, August 2012.

[30] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing Unique Code Target Property for Control-Flow Integrity. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, October 2018.

[31] Zhenghao Hu, Yu Hu, and Brendan Dolan-Gavitt. Chaff Bugs: Deterring Attackers by Making Software Buggier. *CoRR*, abs/1808.00659, 2018.

[32] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, 2002.

[33] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM – Software Protection for the Masses. In Brecht Wyseur, editor, *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection*. IEEE, 2015.

[34] Su Yong Kim, Sangho Lee, Insu Yun, Wen Xu, Byoungyoung Lee, Youngtae Yun, and Taesoo Kim. CAB-Fuzz: Practical Concolic Testing Techniques for COTS Operating Systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2017.

[35] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. SoK: Automated Software Diversity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[36] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: Program-state Based Binary Fuzzing. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[37] LLVM. LibFuzzer - A Library for Coverage-guided Fuzz Testing, 2017. http://llvm.org/docs/LibFuzzer.html.

[38] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, June 2005.

[39] Microsoft. Microsoft Previews Project Springfield, a Cloud-based Bug Detector, 2016. https://blogs.microsoft.com/next/2016/09/26/microsoft-previews-project-springfield-cloud-based-bug-detector.

[40] Barton P. Miller, Louis Fredriksen, and Bryan So. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM*, 33(12):32–44, December 1990.

[41] Charlie Miller. Anti-Fuzzing. https://www.scribd.com/document/316851783/anti-fuzzing-pdf, 2010.

[42] WinAFL Crashes with Testing Code. https://github.com/ivanfratric/winafl/issues/62, 2017.

[43] Unexplained Crashes in WinAFL. https://github.com/DynamoRIO/dynamorio/issues/2904, 2018.

[44] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.

[45] CSO online. Seven of the Biggest Recent Hacks on Crypto Exchanges, 2018. https://www.ccn.com/japans-16-licensed-cryptocurrency-exchanges-launch-self-regulatory-body/.

[46] Oreans Technologies. Themida, 2017. https://www.oreans.com/themida.php.

[47] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, October 2017.

[48] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: Fuzzing by Program Transformation. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2018.

[49] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, October–November 2017.

[50] Brian Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, (2):220–232, 1975.

[51] Michael Rash. A Collection of Vulnerabilities Discovered by the AFL Fuzzer, 2017. https://github.com/mrash/afl-cve.

[52] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February–March 2017.

[53] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. Software Diversity: State of the Art and Perspectives. *International Journal on Software Tools for Technology Transfer (STTT)*, 14(5):477–495, October 2012.

[54] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, August 2017.

[55] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, October–November 2007.

[56] Monirul I Sharif, Andrea Lanzi, Jonathon T Giffin, and Wenke Lee. Impeding Malware Analysis Using Conditional Code Obfuscation. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2008.

[57] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing through Selective Symbolic Execution. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[58] Synopsys. Where the Zero-days are, 2017. https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/state-of-fuzzing-2017.pdf.

[59] Syzkaller. Syzkaller Found Bugs - Linux Kernel, 2018. https://github.com/google/syzkaller/blob/master/docs/linux/found_bugs.md.

[60] UPX Team. The Ultimate Packer for eXecutables, 2017. https://upx.github.io.

[61] Daniel Votipka, Rock Stevens, Elissa M. Redmiles, Jeremy Hu, and Michelle L. Mazurek. Hackers vs. Testers A Comparison of Software Vulnerability Discovery Processes. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2018.

[62] Zhi Wang, Jiang Ming, Chunfu Jia, and Debin Gao. Linear Obfuscation to Combat Symbolic Execution. In *Proceedings of the 16th European Symposium on Research in Computer Security (ESORICS)*, Leuven, Belgium, September 2011.

[63] Ollie Whitehouse. Introduction to Anti-Fuzzing: A Defence in Depth Aid. https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2014/january/introduction-to-anti-fuzzing-a-defence-in-depth-aid/, 2014.

[64] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing New Operating Primitives to Improve Fuzzing Performance. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, October–November 2017.

[65] Zhou Xu. PTfuzzer, 2018. https://github.com/hunter-ht-2018/ptfuzzer.

[66] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *ACM SIGPLAN Notices*, volume 46, pages 283–294. ACM, 2011.

[67] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

[68] Michal Zalewski. Fuzzing Random Programs without execve(), 2014. https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html.

[69] Michal Zalewski. New in AFL: Persistent Mode, 2015. https://lcamtuf.blogspot.com/2015/06/new-in-afl-persistent-mode.html.

[70] Michal Zalewski. High-performance Binary-only Instrumentation for AFL-fuzz, 2016. https://github.com/mirrorer/afl/tree/master/qemu_mode.

[71] Michal Zalewski. Technical Whitepaper for AFL-fuzz, 2017. https://github.com/mirrorer/afl/blob/master/docs/technical_details.txt.

[72] Michal Zalewski. American Fuzzy Lop (2.52b), 2018. http://lcamtuf.coredump.cx/afl/.

Appendix

A HonggFuzz Intel-PT-mode Result

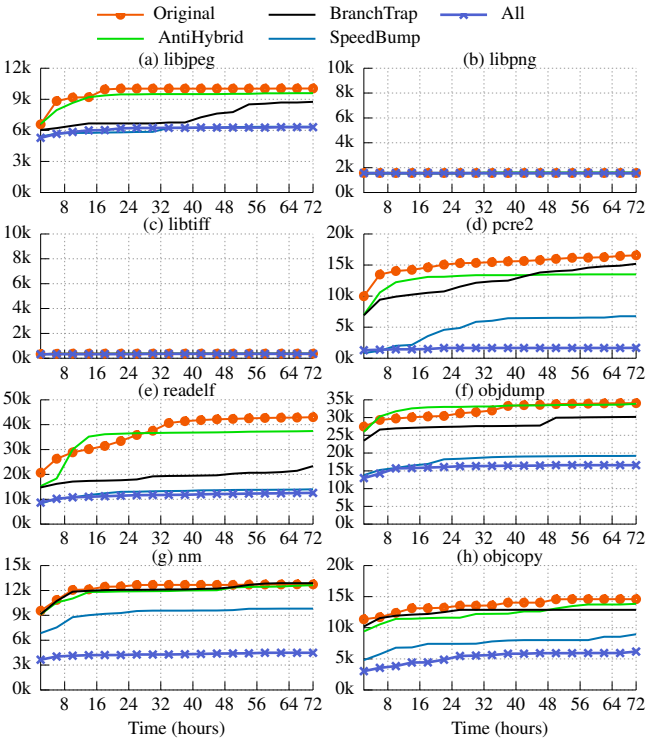


Figure 14: Paths discovered by HonggFuzz Intel-PT mode from real-world programs. Each program is compiled with five settings: original (no protection), SpeedBump, BranchTrap, AntiHybrid and all protections. We fuzz them for three days.

AntiFuzz: Impeding Fuzzing Audits of Binary Executables

Emre Güler, Cornelius Aschermann, Ali Abbasi, and Thorsten Holz
Ruhr-Universität Bochum

Abstract

A general defense strategy in computer security is to increase the cost of successful attacks in both computational resources as well as human time. In the area of binary security, this is commonly done by using obfuscation methods to hinder reverse engineering and the search for software vulnerabilities. However, recent trends in automated bug finding changed the modus operandi. Nowadays it is very common for bugs to be found by various fuzzing tools. Due to ever-increasing amounts of automation and research on better fuzzing strategies, large-scale, dragnet-style fuzzing of many hundreds of targets becomes viable. As we show, current obfuscation techniques are aimed at increasing the cost of human understanding and do little to slow down fuzzing.

In this paper, we introduce several techniques to protect a binary executable against an analysis with automated bug finding approaches that are based on fuzzing, symbolic/concolic execution, and taint-assisted fuzzing (commonly known as *hybrid fuzzing*). More specifically, we perform a systematic analysis of the fundamental assumptions of bug finding tools and develop general countermeasures for each assumption. Note that these techniques are not designed to target specific implementations of fuzzing tools, but address general assumptions that bug finding tools necessarily depend on. Our evaluation demonstrates that these techniques effectively impede fuzzing audits, while introducing a negligible performance overhead. Just as obfuscation techniques increase the amount of human labor needed to find a vulnerability, our techniques render automated fuzzing-based approaches futile.

1 Introduction

In recent years, fuzzing has proven a highly successful technique to uncover bugs in software in an automated way. Inspired by the large number of bugs found by fuzzers such as AFL [56], research recently focused heavily on improving the state-of-the-art in fuzzing techniques [10, 11, 22, 44, 54]. Previously, it was paramount to manually remove checksums and similar roadblocks from the fuzzing targets. Additionally, fuzzers typically required large, exhaustive seed corpora or a precise description of the input format in form of a grammar. In a push towards a greater degree of automation, research recently focused on avoiding these common roadblocks [14, 39, 44, 45, 48, 54]. This push toward automation greatly simplifies the usage of these tools. One can argue that, for the attacker, using a fuzzer is as easy as it is for the defender. In fact, recently the *Fuzzing Ex Machina* (FExM) [49] project managed to reduce the overhead of running fuzzers to a degree where they managed to fuzz the top 500 packages from the Arch Linux User Repository with no manual effort in seed selection or similar issues. This two day effort yielded crashes in 29 of the most popular packages of Arch Linux. It stands to reason that this kind of indiscriminate, dragnet-style searching for software bugs will become more prevalent in the future.

While the developers of a software system should typically thoroughly fuzz test every type of software, in practice they may want to maintain an asymmetric cost advantage. More specifically, it should be easier for the maintainers of a software project to fuzz their own software than for attackers. This can be achieved by adding mechanisms to the software such that the final binary executable is protected against fuzzing: the maintainers can then build an internal version that can be tested thoroughly, while an attacker can only access the protected binary which prohibits automated tests. In the past, similar asymmetric advantages in analysis and bug finding were introduced by obfuscation techniques. As we demonstrate, even very high levels of obfuscation will typically result only in a meager slowdown of current fuzzing techniques. This is due to the fact that obfuscation typically aims at protecting against program understanding and formal reasoning. On the other hand, fuzzers typically do not perform a significant amount of reasoning over the behaviour of the program. On the downside, these heavy obfuscation mechanisms will often incur a significant runtime overhead [19].

How software can be protected against fuzzing in an efficient and effective way is an open problem.

In this paper, we tackle this challenge and present several general methods to impede large scale, automated fuzzing audit of binary executables. We present several techniques that can be added during the compilation phase of a given software project such that the resulting binary executable withstands fuzzing and significantly hampers automated analysis. Our methods are based on a systematic analysis of 19 current bug finding tools with respect to their underlying assumptions. Note that we use the terms "fuzzer" and "bug finding tool" interchangeably to describe all kinds of tools that are analyzing programs to produce crashing inputs as opposed to static analysis tools and linters. We find that all of them rely on at least one of the following four basic assumptions: (i) coverage yields relevant feedback, (ii) crashes can be detected, (iii) many executions per second are achievable, and (iv) constraints are solvable with symbolic execution. Based on these insights, we develop fuzzing countermeasures and implement a lightweight protection scheme in the form of a configurable, auto-generated single C header file that developers can add to their application to impede fuzzers. For the evaluated programs, we had to change on average 29 lines of code, which took less than ten minutes. With these changes, attackers now need to spend a significant amount of time to manually remove these anti-fuzzing mechanisms from a protected binary executable (typically magnified by common obfuscation techniques on top), greatly increasing the cost of finding bugs as an attacker. Defenders, on the other hand, can still trivially fuzz the unmodified version of their software with no additional cost. Thus, only unwanted and unknown attackers are at a disadvantage.

We implemented a prototype of the proposed methods in a tool called AntiFuzz. We demonstrate in several experiments the effectiveness of our approach by showing that state-of-the-art fuzzers cannot find bugs in binary executables protected with AntiFuzz anymore. Moreover, we find that our approach introduces no observable, statistically significant performance overhead in the SPEC benchmark suite.

Contributions In summary, in this paper we make the following contributions:

- We present a survey of techniques employed by current fuzzers and systematically analyze the basic assumptions they make. We find that different fuzzing approaches rely on at least one of the fundamental assumptions which we identify.
- We demonstrate how small changes to a program nullify the main advantages of fuzzing by systematically violating the fundamental prerequisites. As a result, it becomes significantly harder (if not impossible with current approaches) to find bugs in a protected program without manual removal of our anti-fuzzing methods.
- We implemented our anti-fuzzing techniques in a tool called AntiFuzz that adds fuzzing countermeasures during the compilation phase. Our evaluation with several different programs shows that with a negligible performance overhead, AntiFuzz hardens a target binary executable such that none of the tested fuzzers are able to find any bugs.

To foster research on this topic, we release our implementation and the data sets used as part of the evaluation at https://github.com/RUB-SysSec/antifuzz.

2 Technical Background

Fuzzing (formerly known as random testing) has been around since at least 1981 [20]. In the beginning, fuzzers would simply try to execute programs with random inputs. While executing, the fuzzer observes the behavior of the program under test: if the program crashes, the fuzzer managed to find a bug and the input is stored for further evaluation. Even though this technique is surprisingly simple—particularly when compared to static program analysis techniques—with a sufficient number of executions per second it has been helpful at finding bugs in complex, real-world software in the past.

In recent years, the computer security community paid much more attention to improving the performance and scalability of fuzzing. For example, the OSS-Fuzz project has been fuzzing many highly-relevant pieces of software 24/7 since 2016 and exposed thousands of bugs [1]. FEXM automatized large parts of the setup and the authors were able to fuzz the top 500 packages from the Arch Linux User Repository [49]. To improve the usability of fuzzers in such scenarios, the biggest focus of the research community is to automatically overcome hard-to-fuzz code constructs that previous methods could not successfully solve with the goal of reaching deeper parts of the code. Particularly, common program analysis techniques were applied to the problem of fuzzing. For example, symbolic execution and its somewhat more scalable derivative concolic execution was used to overcome hard branches and trigger bugs that are only trigger-able by rare conditions [25–27, 31, 42, 48, 50, 54]. Other fuzzers use taint tracing to reduce the search space to mutations that actually influence interesting parts of the program [14, 23, 31, 42, 45]. A complementary line of work focused on improving the fuzzing process itself without falling back to (often costly) program analysis techniques. Many techniques propose improvements to the way AFL instruments the target [2, 22, 29, 47], or how inputs are scheduled and mutated [10, 11, 13, 46, 52]. Some methods go as far as removing hard parts from the target [44, 50]. Lastly, the effectiveness of machine learning models for efficient input generation was evaluated [9, 28, 32].

Generally speaking, existing methods for fuzzing can be categorized into the following three different categories based

on the techniques employed, which we explain in more detail in the following.

2.1 Blind Fuzzers

The oldest class of fuzzers are so-called *blind fuzzers*. Such fuzzers have to overcome the problem that random inputs will not exercise any interesting code within a given software. Two approaches were commonly applied: *mutational fuzzing* and *generational fuzzing*.

Mutational fuzzers require a good corpus of inputs to mutate. Generally, mutational fuzzers do not know which code regions depend on the input file and which inputs are necessary to reach more code regions. Instead, these fuzzers introduce some random mutations to the file and can only detect if the program has crashed or not. Mutational fuzzers need seed files that cover large parts of interesting code as they are unable to uncover new code effectively. In the past, these fuzzers were quite successful at uncovering bugs [33]. However, they typically need to perform a large number of executions per second to work properly. An example of mutational-only fuzzers are ZZUF [5] and RADAMSA [33].

The second approach is generational fuzzing: fuzzers which employ this technique need a formal specification to define the input format. Based on this specification, the fuzzer is able to produce many semi-valid inputs. This has the advantage that the fuzzer does not need to learn how to generate well-formed input files. However, manual human effort is necessary to create these definitions (e.g., a grammar that describes the input format). This task becomes hard for complex or unknown formats and the specification could still end up lacking certain features. The additional need for a formal specification makes this approach much less useful for large-scale bug hunting with little human interaction. An example of a generational fuzzer is PEACH [3].

In summary, the only thing a blind fuzzer is able to observe is whether its input led to a crash of the program or not. Therefore, these techniques have no indicator of their progress in exploring the programs state space and thus (especially in the case of mutational fuzzers), they are mostly limited to simple bugs even with non-empty and well-formed seed files.

2.2 Coverage-guided Fuzzers

To improve the performance of the mutational fuzzers, Zalewski introduced an efficient way to measure coverage-feedback of an application [56]. This led to a significant amount of research on coverage-guided fuzzers. These fuzzers typically use a feedback mechanism to receive information on how an input has affected the program under test. The key idea here is that this mechanism gives means by which to judge an input: Which (new) code regions were visited and how often? In contrast, a blind fuzzer introduces random mutations to the input without knowing how those mutations affect the program. It effectively relies on pure chance for finding crashing inputs, while a coverage-guided fuzzer could mutate the same input file iteratively to increase the code coverage and thus get closer to new regions where a crash could happen. Examples of coverage-based fuzzers are AFL [56], HONGGFUZZ [4], ANGORA [14], T-FUZZ [44], KAFL [47], REDQUEEN [8] and VUZZER [45]. These fuzzers use multiple ways to obtain coverage feedback:

Static Instrumentation: One of the fastest methods for obtaining code coverage is static compile time coverage (widely used by tools such as AFL, ANGORA, LIBFUZZER, and HONGGFUZZ). In this case, the compiler adds special code at the start of each basic block that stores the coverage information. From a defender's point of view, this kind of instrumentation is not relevant, as we assume that the attackers do not have access to the source code.

Dynamic Binary Instrumentation (DBI): If only a binary executable is available, fuzzer typically use dynamic binary instrumentation (DBI) to obtain coverage information. This is done by adding the relevant code at runtime. Examples of this approach are VUZZER and STEELIX [39], which both use PIN-based [40] instrumentation, and AFL which has multiple forks using QEMU, PIN, DYNAMORIO, or DYNINST for DBI. Fuzzers like DRILLER [48] and T-FUZZ use AFL under the hood and typically rely on the QEMU-based instrumentation.

Hardware Supported Tracing: Modern CPUs support various forms of hardware tracing. For Intel processors, two technologies can be used: Last Branch Record and Intel-PT. HONGGFUZZ is able to utilize both techniques, while fuzzers like KAFL only support Intel-PT.

2.2.1 Using Coverage Information:

Different fuzzers tend to use the coverage feedback obtained in different ways. To illustrate these differences, we select two well-known coverage-guided fuzzers; namely AFL and VUZZER. We then describe how these fuzzers are using coverage information internally. It is worth noting that by choosing AFL, we are basically covering the way various other fuzzers such as T-FUZZ, ANGORA, KAFL, STEELIX, DRILLER, LIBFUZZER, WINAFL, AFLFAST [11], and COLLAFL [22] are using coverage information. All of these fuzzers (except ANGORA) use the same underlying technique for leveraging coverage information. In contrast to AFL, no other fuzzer followed the path of VUZZER in coverage information usage. However, due to the unique usage of coverage information in VUZZER, we describe it as well.

AFL A key factor behind the success of AFL is an efficient, approximate representation of the code coverage. To reduce the memory footprint, AFL maps each basic block transition (edge) to one index in a fixed size array referred to as the "bitmap". Upon encountering a basic block transi-

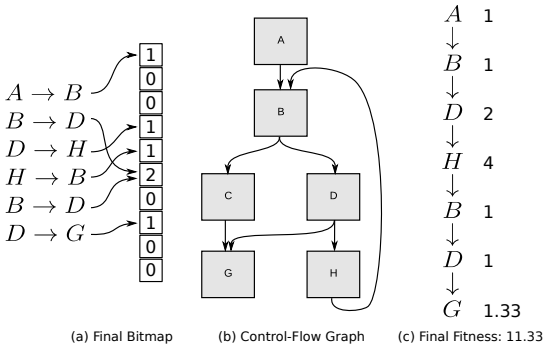


Figure 1: Using coverage information in AFL-like fuzzers versus Vuzzer in the same path of a given Control-Flow Graph (b).

tion, it increments the corresponding value in the bitmap as illustrated in Figure 1(a). The bitmap is typically limited to 64KiB, so it easily fits inside an L2 cache [55]. Although limiting the size of the bitmap allows very efficient updates, it also reduces its precision, since in some cases multiple edges share the same index in the bitmap. It is possible to increase the size of the bitmap, but at the cost of a significant decline in performance [22].

As mentioned earlier, ANGORA uses a very similar scheme with a slight difference: before updating the bitmap entry, ANGORA XORs the edge index with a hash of the call stack. This way, ANGORA can distinguish the same coverage in different contexts, while AFL can not. For example, in Listing 1, AFL cannot distinguish the coverage produced by lines 2 and 3 when called from line 10 from the coverage produced by the same lines (lines 2 and 3) in the second call. Therefore, AFL can use feedback to learn that the input should start with "fo", however, it cannot use the same information to learn that the input should continue with "ba". In contrast, ANGORA can identify the context (here "fo" and "ba") of the code and thus distinguish between these two calls. It is worth to mention that this drastically increases the number of entries in the bitmap, and therefore ANGORA might need a bigger bitmap.

Listing 1: A sample code which illustrates the differences between AFL and ANGORA on distinguishing coverage information

```
1  bool cmp(char *a, char *b){
2      if (a[0]==b[0]){
3          if (a[1]==b[1]){
4              return true;
5          }
6      }
7      return false;
8  }
9  ....
10 if (cmp(input,"fo")){
11     if (cmp(input+2, "ba")){
12     ....
13     }
14 }
```

Vuzzer AFL does not discriminate among edges. Therefore, an input that covers one previously unseen edge is just as interesting as an input which covers hundreds of unseen edges. This is the fundamental difference between VUZZER and AFL. Unlike AFL, VUZZER extracts the exact basic block coverage (instead of the bitmap) and enriches the feedback mechanism with additional data. For example, VUZZER uses a static disassembly to weight basic blocks according to how "deep" they are within a function (e.g., how many conditions have to be satisfied to reach this block). Higher scores are assigned to harder-to-reach blocks. To further improve the feedback mechanism, VUZZER excludes basic blocks that belong to error paths by measuring the coverage produced by random inputs. In the example shown in Figure 1(c), each basic block has a weight. As can be seen, basic block H has a much higher weight than basic block G because H is much less likely to be reached by a random walk across the control-flow graph (with back-edges removed). Finally, all the weights of all basic blocks in the path are added up to calculate a fitness value. VUZZER then uses an evolutionary algorithm to produce new mutations: inputs with a high fitness value produce more offspring. These newly created offspring are then used as the next generation.

2.3 Hybrid Fuzzers

While using coverage-based fuzzing already leads to interesting results on its own, there are code regions in a program which are hard to reach. This typically happens if only a very small percentage of the inputs satisfy some conditions. For example, a specific four-byte magic value that is checked by the program under the test makes it nearly impossible for coverage-based fuzzers to pass the check and therefore reach deeper code regions. To address this problem, various research suggest using a combination of program analysis techniques to assist the fuzzing process [44, 48, 54]. By using symbolic execution or taint analysis, a fuzzer is able to reason what inputs are necessary to cover new edges and basic blocks. Instead of only relying on random mutations and selection by information gathered through feedback mechanisms, these tools try to calculate and extract the correct input necessary for new code coverage. Examples of fuzzers which are using symbolic or concolic execution to assist the coverage-based fuzzer are DRILLER [48], QSYM [54], and T-FUZZ [44].

The archetypal hybrid fuzzer is DRILLER, which uses concolic execution to search for inputs that produce new coverage. It tries to provide a comprehensive analysis of the program's behaviour. In contrast, QSYM [54] identified this behavior as a weakness since the fuzzer can validate that the input proposed by the symbolic or concolic execution generates new coverage very cheaply. Therefore, an unsound symbolic or concolic execution engine can produce a large number of false positive proposals, without reducing the overall performance of the fuzzer. Building upon this insight, QSYM discards all but the last constraint in the concrete execution trace as well as the symbolic values produced by basic blocks that were executed frequently. Finally, it is worth mentioning that in the case of T-FUZZ, symbolic execution is not used for

the fuzzing process itself. Instead, T-FUZZ patches hard constraints. Once T-FUZZ finds a crashing input for the patched program, it uses symbolic execution to calculate an input that actually crashes the unpatched target program.

3 Analysis of Fuzzing Assumptions

Based on the categories described in the previous section, we now analyze and identify fundamental assumptions that fuzzers use to find bugs. The first insight is that while many aspects of fuzzing have changed since 1981, two basic assumptions which were originally made still apply to most modern fuzzers: these two basic original assumptions are *crash detection* and *high execution throughput*. However, to achieve better performance in modern fuzzers, additional assumptions were made in the past years, as we discuss next.

To evade not only current but also future bug finding methods, we analyze under which *core assumptions* all (or at least most) of the current tools operate. By systematically breaking assumptions shared by most fuzzers, we can develop a more universal defense against automated audits. Using this systematic approach, we avoid targeting specific implementations and therefore will hamper all future fuzzing methods built upon the same general assumptions. We divide the current fuzzing assumptions into the following four groups:

(A) Coverage Yields Relevant Feedback Coverage-guided fuzzers typically assume that novel code coverage also strongly correlates with novel behavior. Therefore, every time a modern coverage-guided fuzzer generates an input which traverses through a new code region, it assumes that the program behaves differently from previous inputs. Based on the coverage, the fuzzer decides how much time to allocate for generating further mutations of this input. For example, most current fuzzers such as AFL, VUZZER, DRILLER, QSYM, KAFL, ANGORA, T-FUZZ, and LIBFUZZER use this assumption for coverage-guided fuzzing.

(B) Crashes Can Be Detected Triggering security-relevant bugs will typically lead to a program crash. Thus, most bug finding tools need the ability to tell a crashing input apart from a non-crashing input in an efficient and scalable way. As a result, they require some techniques to detect if an application has crashed. Nearly all random testing tools share this assumption since 1981 [20]. In addition to the coverage-guided fuzzers, this assumption is also shared by blind fuzzers such as PEACH, RADAMSA, and ZZUF.

(C) Many Executions per Second To efficiently generate input files with great coverage, the number of executions per second needs to be as high as possible. In our experience, depending on the application and fuzzer, a range from few hundreds up to a few thousands of executions per second are typical. Slow executions will drastically degrade the performance. All fuzzers mentioned in the previous assumptions also fall into this class. Only pure symbolic execution tools such as KLEE do not fall into this category.

(D) Constraints Are Solvable with Symbolic Execution Hybrid fuzzers or tools based on symbolic execution such as DRILLER, KLEE, QSYM, and T-FUZZ need to be able to represent the program's behavior symbolically and solve the resulting formulas. Therefore, any symbolic or concolic execution-based tools only operate well when the semantics of the program under test are simple enough. This means that the internal representation of the state of the symbolic/concolic execution engine has to be small enough to store and the resulting constraints set has to be solvable by current solvers to avoid problems related to state explosion.

Summary We compiled a list of 19 different bug finding tools and systematically check which assumptions they rely on. An overview of the analyzed tools and their corresponding assumptions is shown in Table 1. It is worth mentioning that various tools in this table are based on AFL and thus share the same assumptions.

4 Impeding Fuzzing Audits

Based on the analysis results of the previous section, we now introduce techniques to break the identified assumptions of bug finding tools in a systematic and generic way. Moreover, we sketch how these techniques can be implemented; actual implementation details are provided in the next section.

Attacker Model Throughout this paper, we use the following attacker model. First, we assume that an attacker can only access the final protected binary executable and not the original source code of the software. She wants to find bugs in an automated way in the protected binary executable, while requiring only a minimum human intervention. Commonly there is the notion that source-based fuzzers significantly outperform binary-only fuzzers. Therefore, it is believed that defenders already have a significant cost advantage over attackers. However, recent advances in fuzzing have shown that this advantage is in decline. For example, recent binary-only fuzzing techniques paired with hardware acceleration technologies such as Intel PT have drastically reduced the performance gap between binary and source fuzzing. For example, Cisco Talos states that the overhead is only 5% to 15% [36] and similar numbers are reported for published Intel PT-based fuzzers such as KAFL [47]. Additionally, smart fuzzing techniques outperform source-based fuzzing even in binary-only targets [8, 54].

Although many relevant software projects are open source, a large part of all commercial software used in practice is not available in source code format (e.g., Windows, iOS and the vast majority of the embedded space). Nonetheless, some large software projects such as certain PDF viewer and hypervisors are not only well-tested by their developers, but also by whitehat attackers. This additional attention is an

Table 1: Bug finding tools and the assumptions they rely on.

	(A) Coverage Feedback	(B) Detectable Crashes	(C) Application Speed	(D) Solvable Constraints
AFL	✓	✓	✓	✗
KAFL	✓	✓	✓	✗
AFLFAST	✓	✓	✓	✗
COLLAFL	✓	✓	✓	✗
AFLGO	✓	✓	✓	✗
WINAFL	✓	✓	✓	✗
STEELIX	✓	✓	✓	✗
REDQUEEN	✓	✓	✓	✗
HONGGFUZZ	✓	✓	✓	✗
VUZZER	✓	✓	✓	✗
DRILLER	✓	✓	✓	✓
KLEE	✗	✗	✗	✓
ZZUF	✗	✓	✓	✗
PEACH	✗	✓	✓	✗
QSYM	✓	✓	✓	✓
T-FUZZ	✓	✓	✓	✓
ANGORA	✓	✓	✓	✗
RADAMSA	✗	✓	✓	✗
LIBFUZZER	✓	✓	✓	✗

important factor in their security model. Similarly, projects that have a history of helpful interactions with independent researchers should consider not to use ANTIFUZZ, to avoid scaring researchers away. As an alternative, projects with such a successful history of community integration can choose to release unprotected binaries to a set of trusted security researchers. On the other hand, the vast majority of software gets far less to no attention. These less well-known pieces of software are still used by many users and they might profit significantly from raising the bar against fuzzing (e.g., industrial controllers such as PLCs [6, 37] or other types of proprietary software).

Furthermore, in this paper, we consider the case that the attacker can use any state-of-the-art bug finding tool. However, we assume that she spends no time on manually reverse engineering the binary or building custom tooling. We are aware that in a more realistic scenario, the target application might be attacked by a human analyst. However, we assume that ANTIFUZZ is combined with other techniques that were developed to incur significant cost for human analyst during reverse engineering [16, 17, 21, 24, 41, 43, 53]. Therefore, to ensure that different concerns (defending against fuzzing *and* defending against analysis by a human) are separated, we explicitly exclude human analysts from our attacker model.

4.1 Attacking Coverage-guidance

As mentioned previously, the core assumption of coverage-guided fuzzers is that new coverage indicates new behavior in the program. To undermine this assumption, we modify the program which we want to defend against fuzzing by adding irrelevant code in such a way that its coverage information drowns out the actual signal. More specifically, by adding irrelevant code regions (which we call *fake code*), we deliberately disturb the code coverage tracking mechanisms within fuzzers. Thereby, we weaken the fuzzer's ability to use the feedback mechanism in any useful way and thus remove their advantage over blind fuzzers.

To introduce noise into the coverage information, we use two different techniques. The first technique aims at producing different "interesting" coverage for nearly all inputs. The rationale behind this is that according to the coverage-guide assumption, *any* new coverage means that the fuzzer found an input that causes new behavior. Therefore, if the program *always* displays new coverage (due to our fake code), the fuzzer cannot distinguish between legitimate new coverage and invalid fake coverage. As every single input seems to trigger new behavior, the fuzzer assumes that every input is interesting. Therefore, it spends a significant amount of time on generating mutations based on invalid input.

To implement this technique, we calculate the hash of the program input and based on this hash, we pick a small random subset of fake functions to call. Each fake function recursively calls the next fake function from a table of function pointers, in such a way that we introduce a large number of new edges in the protected program.

Since even a single bit flip in the input causes the hash to be completely different, nearly any input that the fuzzer generates displays new behavior. Fuzzers that are objective-driven and thus assign weights to more interesting code construct might find it easy to distinguish between this simple fake code and the actual application code. Since we cannot assume that future fuzzers will treat new coverage information in the same way as current fuzzers do, we introduce a second technique that aims at providing plausible-looking, semi-hard constraints. The second technique is designed to add fake code that looks like it belongs to the legitimate input handling

code of the original application. At the same time, this code should include a significant number of easy constraints as well as some very hard constraints. These hard constraints can draw the attention of different solving strategies, while the easy constraints allow us to add noise to the true coverage information. We create this fake code by creating random trees of nested conditions with conditions on the input ranging from simple to complicated.

Evasion Overall, the attack on the code-coverage assumption consists of a combination of these two techniques to fool the fuzzer into believing that most inputs lead to new code coverage and thus they are classified as "interesting". This fills up the attention mechanism of the fuzzer (e.g., AFL's bitmap or a queue) with random information which breaks the assumption that the feedback mechanism is helpful in determining which inputs will lead to interesting code.

4.2 Preventing Crash Detection

After applying our previous method, coverage-guided fuzzers are "blinded" and have few advantages left in comparison to blind fuzzers. To further reduce the ability of both coverage-guided and blind fuzzers to find bugs, we introduced two additional techniques that attack assumption B identified earlier.

There are multiple ways for a fuzzer to detect if a crash has happened. The three most common ways are (i) observing the exit status, (ii) catching the crashing signal by overwriting the signal handler, and (iii) using the operating system (OS) level debugging interfaces such as `ptrace`. To harden our protected program against fuzzers, we try to block these approaches by common anti-debugging measures as well as a custom signal handler that exits the application gracefully. After we install our custom signal handler, we intentionally trigger a segfault (*fake crash*) that our own signal handler recognizes and ignores. This way, if an outside entity is observing crashes that we try to mask, it will always observe a crash for each and every input. It is worth mentioning that by design, the fake crash is triggered at *every* program execution independent from the user input. Thus we do not introduce crashes based on user inputs.

Evasion We try to catch all crashes before they are reported to an outside entity. If the current application is under observation or analysis (i.e., where catching crashes is not allowed), the application is terminated. Typically, if it was deemed necessary to apply AntiFuzz to any application, there is likely no scenario where it would *also* be necessary to continue operating under the given conditions.

In all of these cases, no crashes will be detected even if they still occur, which breaks the assumption that a crashing input is detectable as such.

4.3 Delaying Execution

We found that fuzzing tools need many executions per second to operate efficiently. Our third countermeasure attacks this assumption, without reducing the overall performance of the protected program, as follows: we check whether the input is a well-formed input; if and only if we detect a malformed input, we enforce an artificial slowdown of the application. For most applications, this would not induce any slowdowns in real-world scenarios, where input files are typically well-formed. But at the same time, it would significantly reduce the execution speed for fuzzers, where most of the inputs will be incorrect. We believe that even if malformed input files occasionally happen in real scenarios, a slowdown of e.g., 250ms per invalid input is barely noticeable to the end user in most cases. In contrast, even such a small delay has drastic effects on fuzzing. Thus, only fuzzers are negatively affected by this technique.

Delaying the execution can happen through different means, the easiest way to cause a delay is using the `sleep()` function. However, to harden this technique against automated code analysis and patching tools, one can add a computationally-heavy task (e.g., encryption, hash calculation, or even cryptocurrency mining) to the protected program such that the resulting solution is necessary to continue the execution.

Evasion Most applications expect some kind of structure for their input files and have the ability to tell if the input adheres to this structure. Therefore, AntiFuzz does not need to rely on any formal specification; instead, our responses are triggered by existing error paths within the program. For the prototype implementation, we do not propose to detect error paths automatically, but instead insert them manually as a developer. If the input is malformed, we artificially slow down the execution speed of the program. This breaks the assumption that the application can be executed hundreds or thousands of times per second, thus severely limiting the chances of efficiently finding new code coverage.

4.4 Overloading Symbolic Execution Engines

To prevent program analysis techniques from extracting information to solve constraints and cover more code, we introduce two techniques. Both techniques are based on the idea that simple tasks can be rewritten in a way that it is a lot harder to reason about their behavior [51]. For example, we can replace an addition operation using an additive homomorphic encryption scheme. In the following, we introduce two practical techniques to achieve this goal.

First, we use hash comparisons. The idea is to replace all comparisons of input data to constants (e.g., magic bytes) with a comparison of their respective strong cryptographic hash values. While still practically equivalent (unless small collisions for current hashes are found), the resulting computation is significantly more complex. The resulting symbolic

expressions grow significantly, and the solvers fail to find a satisfying assignment for these equations; they become useless for finding correct inputs. However this technique has one weakness: If a seed file is provided that contains the correct value, a concolic execution engine might still be able to continue solving other branches.

As a second technique, we can encrypt and then decrypted the input with a block cipher. We later describe this technique in detail in Section 5.4.

Evasion By sending the input data through a strong block cipher and replacing direct comparisons of input data to magic bytes by hash operations, symbolic, concolic, and taint-based execution engines are significantly slowed down and hampered in their abilities to construct valid inputs. This breaks the assumption that constraints in the application are solvable. Even though the encryption/decryption combination is an identity transformation, it is very hard to prove automatically that the resulting output byte only depends on the corresponding input byte. Therefore, symbolic/concolic execution engines either carry very large expressions for each input byte, or they concretize every input byte, completely voiding the advantage they provide. Finally, common taint tracking engines will not be able to infer taint on the input, as the encryption thoroughly mixes the input bits.

5 Implementation Details

In this section, we provide an overview of the proof-of-concept implementation of our techniques in a tool called AntiFuzz. As explained above, the use case for AntiFuzz is a developer who has access to source code and wants to protect his application from attackers who use automatic bug finding tools to find bugs cost-effectively. Hence, an important objective was to keep the required modifications to the project at a minimum, so that AntiFuzz is easy to apply. The implementation consists of a Python script that automatically generates a single C header file that needs to be included in the target program. Furthermore, small changes need to be performed to instrumt a given application. For our experiments, we analyzed the time it took us to apply AntiFuzz to LAVA-M (which consists of the four programs `base64`, `md5sum`, `uniq`, and `who`). As we were already familiar with the code base of these tools, we could more closely resemble a developer who has a good understanding of the structure of the code. It took us four to ten minutes to apply AntiFuzz to each application. The number of lines that needed to be added or changed depends on the number of constant comparisons that need to be replaced by hash comparisons. `base64` was an outlier with 79 changed lines, 64 of which were necessary due to a check against every possible character in the `base64` alphabet. The three remaining applications required 6 (`uniq`), 7 (`who`), and 23 (`md5sum`) changed lines, respectively.

In the following, we describe technical details of how AntiFuzz is implemented.

5.1 Attacking Coverage-guidance

To prevent coverage-guided fuzzing, it is necessary to generate random constraints, edges, and constant comparisons, as detailed in Section 4.1. The core idea here is to use every byte of the input file in a way that could lead to a new basic block, e.g., by making it depend on some constraints or by comparing it to randomly generated constants. Depending on the configurable number of constraints and the size of the input file, every byte could be part of *multiple* constraints and constant comparisons.

Implementation-wise, although it is possible to generate code for AntiFuzz dynamically at runtime, this might cause problems for fuzzers relying on static code instrumentation (i.e., they might not be able to "see" code introduced by AntiFuzz). Thus, our template engine, implemented in 300 lines of Python code, generates a C file containing all randomly chosen constraints and constants, and further provides the ability to set configuration values (e.g., number of fake basic blocks).

The random edge generation is implemented through a shuffled array (where the input file seeds the randomness) consisting of functions that call each other based on their position in the array (up to a certain configurable depth).

AntiFuzz provides a function called `antifuzz_init()` that needs to be called with the input filename, ideally before the file is being processed by the application. This change needs to be done manually by the developer when he wants to protect his software against fuzzing: the developer needs to add one line that calls this function. The function implements all the techniques against coverage-guided fuzzers mentioned earlier and sets up signal handlers to prevent crash detection, as detailed in the next section.

5.2 Preventing Crash Detection

When `antifuzz_init()` is called, AntiFuzz has to confirm that no crashes can be observed. As detailed in Section 4.2, it is necessary to overwrite the crash signal handlers, as well as prevent it from being observed with `ptrace`.

In the former case, AntiFuzz first checks whether overwriting signals is possible: we register a custom signal handler and deliberately crash the application. If the custom signal handler was called, it ignores the crash and resumes execution. If the application does not survive the crash, it means that overwriting signals is not possible and, for our purposes, the resulting crash is a desirable side-effect. If the application survives the crash, evidently, signal overwriting is possible.

AntiFuzz then installs custom signal handlers for all common crash signals and overwrites these with either a timeout or a graceful exit (depending on the configuration). This will

keep some fuzzers from covering any code because they do not survive the artificial crash at the beginning of the application. This behavior could also be replaced by an exit or by calling additional functions that lead to fake code coverage to keep up a facade of a working fuzzer.

In the case of `ptrace`, we use a well-known anti-debugging technique [34] to detect if we are being observed by `ptrace`: we check whether we can `ptrace` our own process. If we can `ptrace` our own process, it means that no other process is `ptrace`ing it. However, if we are unable to `ptrace` our own process, it implies that another process is `ptrace`ing it and therefore ANTIFUZZ terminates the application.

5.3 Delaying Execution

As detailed in Section 4.3, ANTIFUZZ needs to know when an input is malformed to slow down the application and hamper the performance of fuzzers. The main idea, implementation-wise, is to allow the developer to inform ANTIFUZZ whenever an input is malformed. Most applications already have some kind of error handling for malformed input, which either discards the input or terminates the application. Within this error handling function of the to-be-protected program, the developer needs to add a single call to `antifuzz_onerror()`.

Upon invocation of `antifuzz_onerror()`, ANTIFUZZ delays the execution for a configurable amount of time using either of the mechanisms mentioned in Section 4.3.

5.4 Overloading Symbolic Execution Engines

There are two main parts to our countermeasures against symbolic/concolic execution and taint analysis engines: replacing constant comparisons with comparisons of their respective cryptographic hashes, and putting the input through a cryptographic block cipher before usage.

The first part is implemented via the SHA-512 hash function. The developer needs to replace important (i.e., input-based) comparisons with the hash functions provided by ANTIFUZZ. Due to the nature of cryptographic hashes, two hash values can only be checked for equality, and not whether one is larger or smaller than the other.

To encrypt and decrypt the input buffer, we use the AES-256 encryption function in ECB mode. The key is generated from a hash of the input at runtime. We provide a function that provides the encryption-decryption routine. We can use this function on any kind of input stream. We provide `antifuzz_fread()` as a convenience to make it easier to integrate the common cases. Any call to `fread()` needs to be replaced with its ANTIFUZZ-equivalent call.

Figure 2 illustrates the implementation of all described techniques using ANTIFUZZ in a simple program. Figure 2.a shows an unprotected application which is checking an input value. If the input is valid, it might lead to a program crash caused by a bug. Otherwise, the program will print some error and exit. Figure 2.b illustrates the same program which is now protected by ANTIFUZZ. Additional layers of fake edges and constraints are specifically targeting coverage-guided fuzzers. Further down the control-flow graph of the protected application, ANTIFUZZ added its input encryption/decryption routine. Next in the Figure 2.b, ANTIFUZZ installs its custom signal handler and then causes an intentional segmentation fault (fake crash). However, since ANTIFUZZ installed a custom signal handler, it receives the signal and checks whether it is the fake crash or not. If it is legitimate, it delays the execution and then exits gracefully. This step basically is the anti-crash detection implementation of ANTIFUZZ, which works together with an execution delay mechanism. Finally, in Figure 2.b, we harden the comparison against 1337 with a comparison of hashed values.

6 Evaluation

Our evaluation aims to answer the following five research questions (**RQs**):

- **RQ 1.** Are current obfuscation techniques efficient against automated bug-finding via fuzzing?
- **RQ 2.** Are the techniques we designed effective at disrupting the targeted fuzzing assumptions?
- **RQ 3.** Are the techniques effective at preventing fuzzers from finding bugs?
- **RQ 4.** Are the techniques effective at reducing the amount of code that is being tested?
- **RQ 5.** Do our techniques introduce any significant performance overhead?

To answer the first research question **RQ 1.**, we demonstrate that modifying a custom dummy application (which is illustrated in Listing 2) using the state-of-the-art obfuscation tool TIGRESS [15] does not yield a satisfying level of protection against current fuzzers.

Following the answer to **RQ 1.**, we test all our techniques individually on multiple fuzzers to demonstrate that they are effective if and only if the fuzzer employs the targeted assumptions. From this experiment, we can answer **RQ 2.** and conclude that our mitigations are working as intended. We use the same dummy application used in **RQ 1.** to evaluate eight fuzzers and bug-finding tools, namely: AFL 2.52b, VUZZER, HONGGFUZZ 1.6, DRILLER commit 66a3428, ZZUF 0.15, PEACH 3.1.124, and QSYM commit d4bf407. Besides the aforementioned fuzzers, we consider one purely symbolic execution based tool to complete the set of automatic bug finding techniques: KLEE 1.4.0.0 [12].

To answer **RQ 3.**, we test a subset of these fuzzers against the LAVA-M dataset to demonstrate that ANTIFUZZ is able to prevent bug finding in real-world applications.

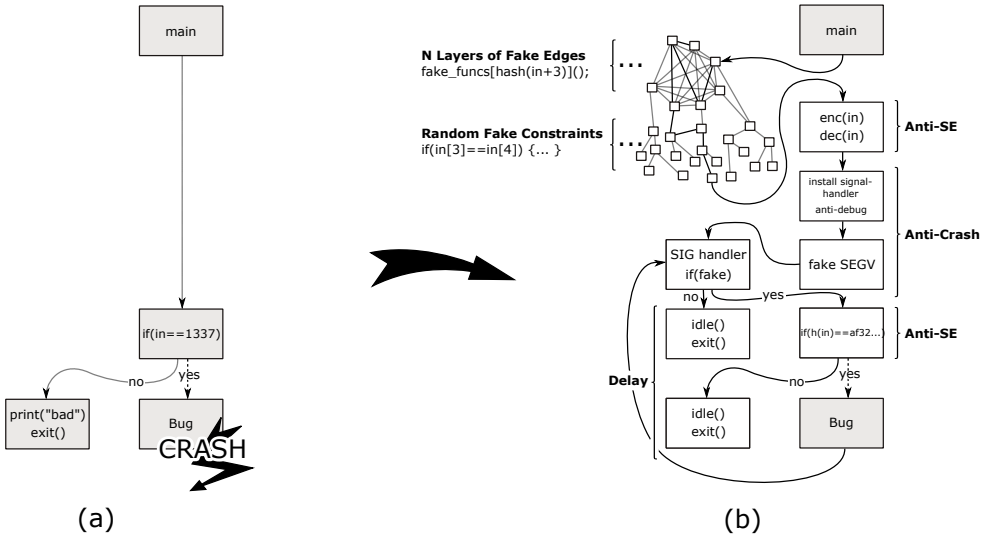


Figure 2: A simple program before (a) and after integration (b) with ANTIFUZZ.

To address **RQ 4.**, we evaluate ANTIFUZZ on binary executables from binutils to show the difference in test coverage in a protected and unprotected application. This experiment demonstrates that ANTIFUZZ does not simply hide bugs, but also drastically reduces the attack surface. It is worth mentioning that in all experiments mentioned above, the bug finding tools were able to find the bugs in a matter of minutes prior to enforcing ANTIFUZZ protection. After applying our techniques, there were *zero* bugs found by the tested tools within a period of 24 hours.

In the last step, we measure the overhead introduced by ANTIFUZZ using the SPEC CPU2006 benchmarking suite to answer **RQ 5.**.

Note that, due to the configurable nature of ANTIFUZZ, we use the following configuration for all experiments:

- **Attacking Coverage-guidance**: Generates 10,000 fake functions with constraints, and 10,000 basic blocks for random edge generation.
- **Delaying Execution**: The signal handler introduces a slowdown in case of a crash to timeout the application (in addition to slowdowns due to malformed inputs). The duration of the sleep is set to 750ms.
- **Preventing Crash Detection**: We enabled all techniques mentioned in Section 5.2.
- **Overloading Symbolic Execution Engines**: Important comparisons for equivalence were replaced with SHA-512 hash comparisons and the input data was encrypted and decrypted via AES-256 in ECB mode.

If the fuzzer supported both binary instrumentation and compile-time instrumentation, we used the compile-time instrumentation. While in reality, a fuzzer would have to use binary-only instrumentation mechanisms (given our attacker model), we chose to use compile-time instrumentation as it achieves better performance and is also more robust. Therefore, we erred on the side of caution by assuming that an attacker is more powerful than state-of-the-art tools.

6.1 ANTIFUZZ versus Software Obfuscation

One of the goals of software obfuscation is to prevent security researchers, who rely on traditional *manual* reverse engineering techniques, from finding bugs. In this section, we demonstrate that obfuscation on its own fails to thwart *automatic* bug finding tools.

Intuitively, blind fuzzers without feedback mechanisms are not hindered by obfuscation at all, because they neither have nor need any knowledge about the code. Feedback-driven fuzzers, however, *do* need access to edges and basic blocks to obtain coverage information they can use to guide the fuzzing process. Thus, obfuscating the control flow via common techniques such as control flow flattening or virtual machine based obfuscation [21] might impact coverage-guided fuzzers.

Experiment To demonstrate that obfuscation techniques alone do not protect an application from automatic bug finding tools, we obfuscated a dummy application (see Listing 2) with TIGRESS 2.2 [15] and let different fuzzers find the correct crashing input.

Listing 2: Dummy application that crashes if input is 'crsh'

```
int check(char* input, int size) {
  if(size == 4 && input[0] == 'c' && input[1] == 'r' &&
       input[2] == 's' && input[3] == 'h') {
    crash();
  }
}
```

For this experiment, we use AFL, HONGGFUZZ, KLEE and ZZUF which are representative of all three fuzzer categories. Note that VUZZER was excluded because (1) VUZZER is based on the IDA Pro disassembler, which is thwarted by obfuscation before the fuzzing process even begins, and (2) Tigress had trouble compiling non-64bit executables while VUZZER (at the time of the experiment) was not working on 64-bit binaries. Additionally, any fuzzer which is based on the aforementioned tools was excluded from the experiment. For example, QSYM and DRILLER use AFL with an additional symbolic execution engine. Therefore, if AFL is able to find the bug, we conclude that other tools that use AFL under the hood can also find the bug.

We configured TIGRESS by enabling as many of the obfuscation features as we could. The exact configuration is shown in Table 1 of Appendix A.

Result This experiment revealed that all fuzzers could find the crashing input despite all obfuscation techniques being enabled. This answers research question **RQ 1.**, current obfuscation techniques are not efficient against automated bug finding techniques. Even though changing the control-flow graph might have an impact on the feedback mechanism, the changes are static or random. In contrast, in ANTIFUZZ the additional information for the feedback mechanism is dependant on the input, which is a major difference between common obfuscation methods and our approach.

6.2 Finding Crashes in a Simple Dummy Application

To answer research question **RQ 2.**, we use the same dummy application from the previous experiment.

For this evaluation, we enable our anti-fuzzing techniques one at a time, rather than enabling all of them at once. This allows us to observe which fuzzer is vulnerable to each technique we introduced. We use this rather simple target for two reasons. (1) If a fuzzer is unable to find this very shallow bug, they will most likely also fail to find more complex crashes, and (2) the code is simple enough to be adjusted to different systems and fuzzers (e.g., DRILLER needs CGC binaries).

Any input that is not the crashing input is deemed to be malformed, i.e., ANTIFUZZ decides to slow down the application in that case. If countermeasures against program analysis techniques are activated, the data from the input file is first encrypted and then decrypted again. The comparisons against the individual bytes of "`crsh`" are done via hash comparisons (e.g., `hash("c") == hash(input[0])`). Signal tampering and anti-coverage techniques are all applied before the input file is opened. Since both PEACH and ZZUF are not able to overcome the four-byte constraints on their own, we provided ZZUF with the seed file where only the "c" character was missing. Similarly, PEACH was evaluated on an ELF64 parser (readelf). We modified the elf parser to include an additional one-byte check of a field in ELF64 that guards the crash.

Table 2: Evaluation against the dummy application. ✓ means ANTIFUZZ was successful in preventing bug finding (no crash was found) and ✗means that at least one crashing input was found. `None` means ANTIFUZZ was disabled, `All` means that all techniques against fuzzers (Coverage, Crash, Speed and Symbolic Execution) were turned on.

	None	Coverage	Crash	Speed	Symbolic Exec.	All
AFL	✗	✓	✓	✓	✗	✓
Honggfuzz	✗	✓	✓	✓	✗	✓
Vuzzer	✗	✓	✓	✓	✓	✓
Driller	✗	✓	-	-	✗	✓
Klee	✗	✓	✓	✓	✓	✓[a]
zzuff	✗	✗	✓	✗	✗	✓
Peach	✗	✗	✓	✓	✗	✓
QSYM	✗	✓	✓	✓	✗	✓

[a] Klee ran at least 24h and then crashed due to memory constraints.

Every possible combination of fuzzer and ANTIFUZZ configuration ran for a period of 24 hours. Moreover, in this experiment, the configuration with all fuzzing countermeasures enabled ("All") ran for a total of 100 hours.

Result The results of this experiment are shown in Table 2. Without ANTIFUZZ, it only took a couple of seconds up to a few minutes for all eight fuzzers to find the crashing input. However, when ANTIFUZZ was fully activated, *no* fuzzer was able to do so even after 100 hours. Comparing this table to Table 1 shows that our techniques clearly address the fundamental assumptions that fuzzers use to find bugs.

All coverage-guided fuzzers were impeded by our anti-coverage feature. As expected, all fuzzers were unable to find crashes when we used our anti-crash detection technique. It is worth mentioning that DRILLER was not tested with this configuration because the CGC environment does not allow custom signal handlers. Surprisingly, KLEE was also unable to find the crash because of its incomplete handling of custom signals. Since delaying execution technique (speed) also relies on custom signals, the experiment with DRILLER was omitted and KLEE failed to find the bug. ZZUF was able to crash the target because there were only 256 different inputs to try.

As expected, KLEE was not able to find the correct input once countermeasures against symbolic execution were activated. Surprisingly, VUZZER is confused by this technique as well. A closer inspection suggests that this behavior was due to the fact that this technique is also highly effective at obfuscating taint information.

6.3 Finding Crashes in LAVA-M

The dummy application demonstrated our ability to thwart fuzzers for simple examples. To make sure that our techniques also hold up on more complex applications (and answer **RQ 3.**), we evaluate ANTIFUZZ with the LAVA-M dataset [18], which consists of four applications (`base64`, `who`, `uniq` and `md5sum`) where several bugs were artificially

Table 3: Statistical analysis of the code coverage on eight binaries from binutils. The effect size is given in percentage of the branches that could be covered after enabling ANTIFUZZ as compared to the coverage achieved on an unprotected program. Experiments where the two-tailed Mann-Whitney U test resulted in $p < 0.05$ are displayed in bold.

	addr2line	ar	nm-new	objdump	readelf	size	strings	strip-new
vuzzer	**12.12%, p: 0.04**	-	**1.81%, p: 0.04**	**2.65%, p: 0.03**	**1.10%, p: 0.04**	13.41%, p: 0.33	**6.25%, p: 0.04**	0.84%, p: 0.19
afl	**9.49%, p: 0.04**	-	**1.92%, p: 0.04**	**4.98%, p: 0.04**	**0.70%, p: 0.04**	**6.30%, p: 0.04**	**16.17%, p: 0.04**	**4.52%, p: 0.04**
hongg	**0.00%, p: 0.03**	0.00%, p: 0.25	**0.00%, p: 0.03**	**0.00%, p: 0.03**	**0.00%, p: 0.03**	**0.00%, p: 0.03**	**0.00%, p: 0.03**	**0.00%, p: 0.03**
qsym	**7.12%, p: 0.04**	**11.69%, p: 0.03**	**5.30%, p: 0.04**	**5.47%, p: 0.04**	**1.75%, p: 0.04**	**9.79%, p: 0.04**	**8.55%, p: 0.04**	**4.89%, p: 0.04**

Table 4: Evaluation against `base64`, `uniq`, `who`, `md5sum` from the LAVA-M data set. ✓ means ANTIFUZZ was successful in preventing bug finding (no crash was found) and ✗ means that at least one crashing input was found, the # sign denotes the number of unique crashes found. `None` means ANTIFUZZ was disabled, `All` means that all techniques against fuzzers (Coverage, Crash, Speed and Symbolic Execution) are turned on.

	None	Coverage	Crash	Speed	Symbolic Execution	All
base64						
AFL	✗(#28)	✓	✓	✓	✗(#24)	✓
Honggfuzz	✗(#48)	✓	✓	✓	✗(#48)	✓
QSYM	✗(#48)	✓	✓	✓	✓	✓
Vuzzer	✗(#47)	✓	✓	✓	✗(#33)	✓
zzuf	✗(#1)	✗(#1)	✓	✓	✗(#1)	✓
uniq						
AFL	✗(#14)	✓	✓	✓	✗(#13)	✓
Honggfuzz	✗(#29)	✓	✓	✓	✗(#29)	✓
QSYM	✗(#14)	✓	✓	✓	✓	✓
Vuzzer	✗(#26)	✓	✓	✓	✗(#15)	✓
zzuf	✗(#1)	✗(#1)	✓	✓	✗(#1)	✓
who						
AFL	✗(#194)	✓	✓	✓	✗(#95)	✓
Honggfuzz	✗(#72)	✓	✓	✓	✗(#72)	✓
QSYM	✗(#1926)	✓	✓	✓	✓	✓
Vuzzer	✗(#266)	✓	✓	✓	✗(#260)	✓
zzuf	✗(#1)	✗(#1)	✓	✓	✗(#1)	✓
md5sum						
AFL	-	-	-	-	-	-
Honggfuzz	✗(#57)	✓	✓	✓	✗(#55)	✓
QSYM	✗(#34)	✓	✓	✓	✓	✓
Vuzzer	✗(#25)	✓	✓	✓	✗(#22)	✓
zzuf	-	-	-	-	-	-

inserted. All fuzzer configurations were allowed to run for 24 hours each. Due to `DWORD` comparisons that AFL has difficulty to solve, the AFL modification LAF-INTEL was used, which breaks comparisons (including string operations) down to single byte comparisons to allow for more nuanced edge generation during compilation. For blind fuzzers like ZZUF, solving four bytes is too hard, thus one constraint was reduced to a single bit-flip for this fuzzer alone.

Results Table 4 shows our result. The # sign denotes the number of unique crashes found (according to distinct LAVA-M fault IDs). Again we can see the same consistent result for all binaries: once ANTIFUZZ is turned on, it effectively prevents fuzzers from detecting bugs. The exceptional cases are similar to the ones we discussed in the previous section. In summary, these results demonstrate that our anti-fuzzing features are applicable to real-world binaries to prevent bug finding.

6.4 Reducing Code Coverage

As a next step, we want to answer **RQ 4.** by demonstrating that applying ANTIFUZZ results in far less coverage in coverage-based fuzzers. More specifically, we evaluated AFL, HONGGFUZZ, VUZZER, and QSYM against eight real-world binaries from the binutils collection (namely `addr2line`, `ar`, `size`, `strings`, `objdump`, `readelf`, `nm-new`, `strip-new`). Every fuzzer and every application was executed three times for 24 hours in the setting "None" (ANTIFUZZ is disabled) and then again in the setting "All" (all ANTIFUZZ features are enabled).

Result The results of this experiment are shown in Figure 3. For each of the eight binutils programs, we compare the performance of the four tested fuzzers (measured in the number of branches covered) without and with protection via ANTIFUZZ. It is apparent that ANTIFUZZ does indeed severely hinder fuzzers from extending code coverage. Note that in all cases, when ANTIFUZZ was activated, even after 24 hours the fuzzers could only reach coverage that would have been reached in the first few minutes without ANTIFUZZ.

We performed a statistical analysis on the resulting data, the results are shown in Table 3. All but three out of thirty experiments were statistically significant with $p < 0.05$ according to a two-tailed Mann–Whitney U test. Two of the insignificant results are from VUZZER, which displayed rather low coverage scores even without ANTIFUZZ enabled. The other insignificant result is on `ar`, a target where most bug finding tools fail due to a multi-byte comparison. Additionally, we calculated the reduction of the amount of covered code that resulted from enabling ANTIFUZZ. Typically (in half of the experiments), less than 3% of the code that was tested on an unprotected target could be covered when ANTIFUZZ was enabled. The 95th percentile of coverage was less than 13% of the code that the fuzzers found when targeting an unprotected program. In the worst result, we achieved a reduction to 17%. Therefore, we conclude that ANTIFUZZ will typically reduce the test coverage achieved by 90% to 95%.

6.5 Performance Overhead

Lastly, to answer **RQ 5.** we measure the performance overhead caused by using ANTIFUZZ on complex programs. For this purpose, we use the SPEC CPU2006 version 1.1 INT benchmark. This experiment consists of all benchmarks that

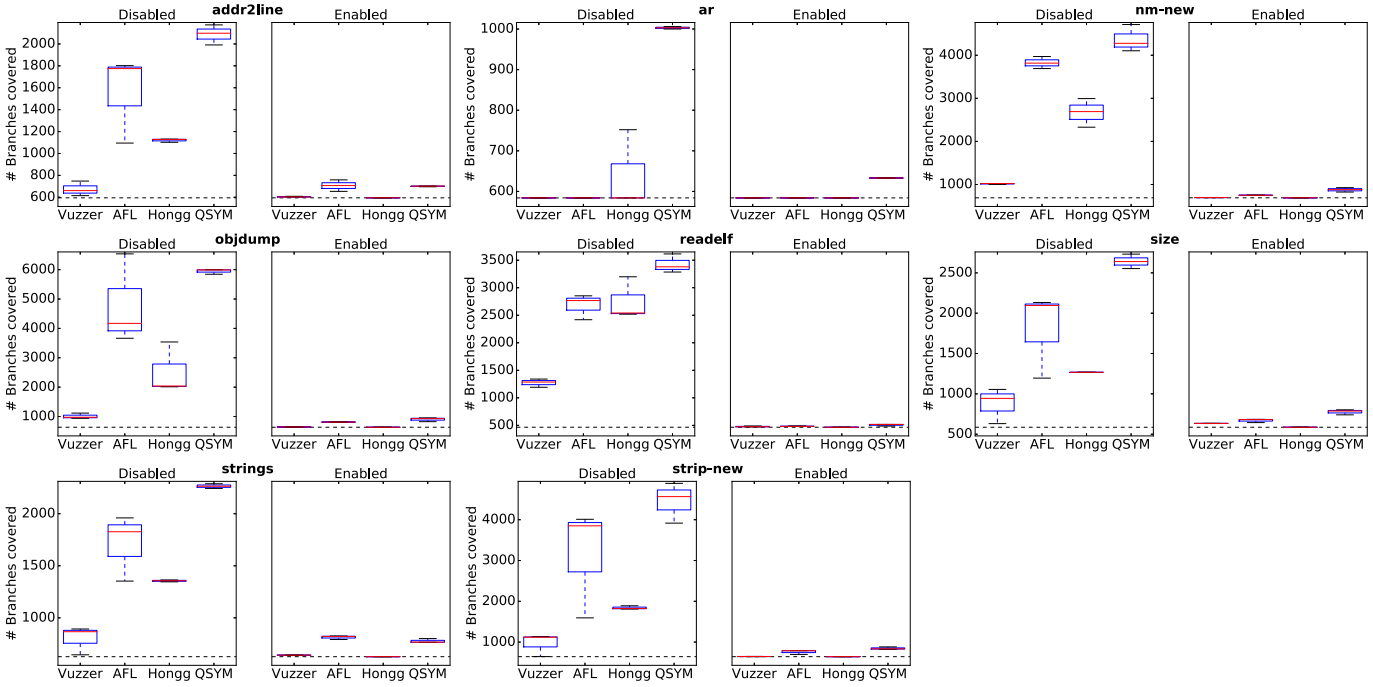


Figure 3: Evaluation of eight binutils binaries to show the branch coverage difference between unmodified binary ("Disabled") and binary with ANTIFUZZ ("Enabled"). The dashed line at the bottom is the baseline (i.e. the number of branches covered with the seed file).

take an input file (thus only 462.libquantum was excluded). The remaining benchmarks ran for three iterations each and were averaged over ten runs with the geometric mean.

Result The impact of ANTIFUZZ for each benchmark was insignificant enough to bear little to no observable overhead (see Table 5): most applications show small *negative* overheads (with the outlier being gcc with -3.80%), but the positive overheads also never reach 1%. The total average overhead is -0.42%. This is expected because `antifuzz_init()` is only called once when the input file is opened. Reading the file to memory and checking if the input data is well-formed usually happens only once in the beginning, thus it does not impact the computationally intensive main part of the benchmarks at all.

Table 5: SPEC CPU2006 INT benchmark.

Benchmark	Overhead
400.perlbench	0.13%
401.bzip2	0.11%
403.gcc	-3.80%
429.mcf	-0.36%
445.gobmk	0.89%
456.hmmer	0.32%
458.sjeng	0.43%
464.h264ref	-1.53%
471.omnetpp	-0.8%
473.astar	-1.06%
483.xalancbmk	0.17%
Total average	**-0.42%**

7 Limitations

In the following, we discuss limitations of both our proposed approach and implementation, and also consider threats to validity. For our current prototype implementation, a human analyst can likely remove the protection mechanisms added by ANTIFUZZ rather easily. However, according to our attacker model, we regard this threat out of scope in the context of this paper. Moreover, many other works have detailed techniques to prevent modification and human analysis using software obfuscation techniques [16, 17, 21, 24, 41, 43, 53]. For a more complete protection, we recommend to use a combination of both ANTIFUZZ as well as traditional anti-analysis/-patching techniques.

The delay-inducing technique should not be applied to any kind of public-facing server software, as this would drastically weaken the server against Denial-of-Service attacks. Instead of sleeping or busy waiting, one should implement a similar approach based on rate limitation.

The number of functions added as fake code results in a fixed file size increase of approximately 25MB. While this is less relevant for large software binaries, it might pose significant code size overhead for small binaries. However, for modern machines we deem this to be a minor obstacle.

Furthermore, it is worth mentioning that one can avoid this increase in file size by using self-modifying code. We explicitly decided not to use self-modifying code since such techniques have the tendency of making exploitation easier by using memory pages with read/write/execute privileges and potentially raising alerts in anti-virus products.

Furthermore, ANTIFUZZ in its current form requires developer involvement which is not optimal from a usability perspective. However, most of the manual work in ANTIFUZZ can be automated. In particular, we require the developer to perform the following tasks: (a) find error paths, (b) replace constant comparisons, and (c) annotate functions which read user input or data. It is relatively easy to automate items (b) and (c) via a compiler pass. The reason that finding error paths is more challenging is that there are many different ways for handling errors. On the other hand, the responsible developer is well aware of the error handling code. Adding a single function call in the error handler is straightforward and does not significantly increase the complexity of the code base.

Additionally, it is worth mentioning that the benchmarking suite which we used was focused on CPU intensive tasks rather than I/O bound tasks. We assume that using our prototype AES implementation to encrypt and decrypt every input significantly increases the overhead on I/O bound tasks. Therefore, we recommend to replace AES by a much weaker and faster encryption algorithm, as our goal is not to be cryptographically secure, but to confuse SMT solvers.

Finally, it has to be considered that automatic program transformations for obfuscation can always be thwarted [7]. Therefore, tools like ANTIFUZZ can never completely guarantee that they can defeat a motivated human analyst. Based on this observation, the situation for anti-fuzzing mechanisms like ANTIFUZZ is similar to obfuscation mechanisms: given sufficient interest from the attackers and defenders, a prolonged arms race is to be expected. This also means that as time passes, continuing this arms race will become more and more expensive for both sides involved. However, similar to obfuscation, we expect only the implementation of tools like ANTIFUZZ to become more complicated. Similarly to modern obfuscation tools, usage of anti-fuzzing defenses will most likely remain cheap.

As we cannot evaluate against techniques not yet invented, some of our techniques could be attacked by smarter fuzzers. The junk code that was inserted could be detected based on statistical patterns or the way it interacts with the rest of the execution. To counter this, more complex and individualized junk code fragments could be used. For example, junk code can change global variables that are also used in the original code (e.g., in opaque predicates).

8 Related Work

Obfuscating software against program understanding has been exhaustively researched. Common techniques include injecting junk code that is never executed [16, 53], often hidden behind conditional expressions that always evaluate to some fixed value [17]. The control flow can be further cloaked by creating many seemingly dissimilar paths that are picked randomly [43] to thwart dynamic analysis based approaches. Other common techniques include self-modifying code [41], which increases the difficulty of obtaining a useful disassembly and changes to the control-flow [21, 24]. Similarly, there has been some work that specifically target symbolic execution [51].

Recent research tried to address a very similar issue: To increase the cost of the attacker, Hu et al. [35] insert a large number of fake bugs into the target application. This approach has the advantage that it works against many different kinds of attack scenarios. However, they rely on the bugs being non-exploitable as otherwise the actual security of the application is reduced. For example, the authors state that they rely on the exact stack layout behavior of the chosen compiler. Any update to the compiler might render the previously "safe" bugs exploitable. Additionally, fuzzers generally tend to find many hundreds to thousands of crashes for each real bug uncovered. Adding some more crashes does not prevent the fuzzer from finding real bugs. The large number of crashes found might draw attention and common analysis techniques for bug triage (such as AFLs bug exploration mode) will greatly simplify weeding out the fake bugs.

In contrast, our approach is much more low key. Additionally, since in our approach no proper test coverage is achieved, no analysis of the produced fuzzing data will be able to uncover any bugs. An idea similar to our fake code insertions was also presented in a talk by Kang et al. [38]. However, they explicitly tried to prevent AFL in QEMU mode from finding a specific crashing path. In our scenario, the defenders do not know the specific crashing path, as otherwise, they would rather fix the bug. Additionally, as we demonstrated in our evaluation, our approach is effective across different fuzzers and does not attack a specific implementation.

Finally, a master thesis by Göransson and Edholm has introduced the idea of masking crashes and actively detecting if the program is being fuzzed, e.g., by detecting specific AFL environment variables [30]. Similarly to the work by Kang et al., the methods they devised are highly specific to the implementation of the only two fuzzers they considered: AFL and HONGGFUZZ. Additionally, to reduce the execution speed of fuzzers, they proposed to artificially decrease the overall performance of the program under test, whereas ANTIFUZZ only decreases the performance if the input is malformed.

9 Conclusion

In this paper, we categorized the general assumptions common to all current bug-finding tools. Based on this analysis, we developed techniques to systematically attack and break these assumptions (and thus a representative sample of contempo-

rary fuzzers). The evaluation demonstrated that obfuscation on its own fails to prevent fuzzing satisfyingly. In contrast, our techniques effectively prevent fuzzers from finding crashing inputs in simple programs, even if the crash was found in seconds in an unprotected application. Furthermore, we demonstrated that we get the same result for real-world applications, i.e., fuzzers are unable to detect any crashes or even achieve a significant amount of new code coverage. Our techniques also show no significant overhead when evaluated with the SPEC benchmark suite and can, therefore, be easily and efficiently integrated into projects with negligible impact to the performance.

In summary, we conclude that the techniques presented in this paper are well applicable to deter automated, dragnet-style hunting for bugs. In combination with common program obfuscation techniques, they will also hinder a targeted attack, as manual work is needed to reverse engineer and remove the anti-fuzzing measures before a more cost-efficient, automated fuzzing campaign can be started.

Acknowledgments

We would like to thank our shepherd Mathias Payer and the anonymous reviewers for their valuable comments and suggestions. This work was supported by the German Federal Ministry of Education and Research (BMBF Grant 16KIS0592K HWSec) and the German Research Foundation (DFG) within the framework of the Excellence Strategy of the Federal Government and the States - EXC 2092 CASA. In addition, this project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 786669 (ReAct). This paper reflects only the authors' view. The Research Executive Agency is not responsible for any use that may be made of the information it contains.

References

[1] Announcing oss-fuzz: Continuous fuzzing for open source software. https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html. Accessed: 2019-02-18.

[2] Circumventing fuzzing roadblocks with compiler transformations. https://lafintel.wordpress.com/. Accessed: 2019-02-18.

[3] Peach. http://www.peachfuzzer.com/. Accessed: 2019-02-18.

[4] Security oriented fuzzer with powerful analysis options. https://github.com/google/honggfuzz. Accessed: 2019-02-18.

[5] zzuf. https://github.com/samhocevar/zzuf. Accessed: 2019-02-18.

[6] Ali Abbasi, Thorsten Holz, Emmanuele Zambon, and Sandro Etalle. ECFI: Asynchronous Control Flow Integrity for Programmable Logic Controllers. In *Annual Computer Security Applications Conference (ACSAC)*, 2017.

[7] Andrew W. Appel. Deobfuscation is in NP, 2002.

[8] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[9] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[10] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[11] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[12] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[13] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *IEEE Symposium on Security and Privacy*, 2015.

[14] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy*, 2018.

[15] Christian Collberg. The Tigress C Diversifier/Obfuscator. http://tigress.cs.arizona.edu/. Accessed: 2019-02-18.

[16] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[17] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1998.

[18] Brendan Dolan, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, William Robertson, Frederick Ulrich, and Ryan Whelan. LAVA: large-scale automated vulnerability addition. In *IEEE Symposium on Security and Privacy*, 2016.

[19] Christopher Domas. Movfuscator: Turning 'mov' into a soul-crushing RE nightmare. https://recon.cx/2015/slides/recon2015-14-christopher-domas-The-movfuscator.pdf. Accessed: 2019-02-18.

[20] Joe W. Duran and Simeon Ntafos. A report on random testing. In *International Conference on Software Engineering (ICSE)*, 1981.

[21] Hui Fang, Yongdong Wu, Shuhong Wang, and Yin Huang. Multi-stage binary code obfuscation using improved virtual machine. In *International Conference on Information Security (ISC)*, 2011.

[22] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *IEEE Symposium on Security and Privacy*, 2018.

[23] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *International Conference on Software Engineering (ICSE)*, 2009.

[24] Jun Ge, Soma Chaudhuri, and Akhilesh Tyagi. Control flow based obfuscation. In *ACM Workshop on Digital Rights Management (DRM)*, 2005.

[25] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[26] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[27] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *Symposium on Network and Distributed System Security (NDSS)*, 2008.

[28] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *ACM International Conference on Automated Software Engineering (ASE)*, 2017.

[29] Peter Goodman. Shin GRR: Make Fuzzing Fast Again. https://blog.trailofbits.com/2016/11/02/shin-grr-make-fuzzing-fast-again/. Accessed: 2019-02-18.

[30] David Göransson and Emil Edholm. Escaping the Fuzz. Master's thesis, Chalmers University of Technology, Gothenburg, Sweden, 2016.

[31] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *USENIX Security Symposium*, 2013.

[32] HyungSeok Han and Sang Kil Cha. Imf: Inferred model-based fuzzer. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[33] Aki Helin. A general-purpose fuzzer. https://gitlab.com/akihe/radamsa. Accessed: 2019-02-18.

[34] Thorsten Holz and Frédéric Raynal. Detecting honeypots and other suspicious environments. *IEEE Information Assurance Workshop*, 2005.

[35] Zhenghao Hu, Yu Hu, and Brendan Dolan-Gavitt. Chaff bugs: Deterring attackers by making software buggier.

[36] Richard Johnson. Go speed tracer. https://talos-intelligence-site.s3.amazonaws.com/production/document_files/files/000/000/048/original/Go_Speed_Tracer.pdf. Accessed: 2019-02-18.

[37] Anastasis Keliris and Michail Maniatakos. Icsref: A framework for automated reverse engineering of industrial control systems binaries. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[38] Kang Li, Yue Yin, and Guodong Zhu. Afl's blindspot and how to resist afl fuzzing for arbitrary elf binaries. https://www.blackhat.com/us-18/briefings/schedule/index.html#afls-blindspot-and-how-to-resist-afl-fuzzing-for-arbitrary-elf-binaries-11048. Accessed: 2019-02-18.

[39] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: Program-state Based Binary Fuzzing. In *Joint Meeting on Foundations of Software Engineering*, 2017.

[40] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[41] Matias Madou, Bertrand Anckaert, Patrick Moseley, Saumya Debray, Bjorn De Sutter, and Koen De Bosschere. Software protection through dynamic code mutation. In *International Workshop on Information Security Applications (WISA)*, 2005.

[42] David Molnar, Xue Cong Li, and David Wagner. Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs. In *USENIX Security Symposium*, 2009.

[43] Andre Pawlowski, Moritz Contag, and Thorsten Holz. Probfuscation: an obfuscation approach using probabilistic control flows. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2016.

[44] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *IEEE Symposium on Security and Privacy*, 2018.

[45] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[46] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan M Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *USENIX Security Symposium*, 2014.

[47] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl: Hardware-assisted feedback fuzzing for os kernels. In *USENIX Security Symposium*, 2017.

[48] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[49] Vincent Ulitzsch, Bhargava Shastry, and Dominik Maier. Follow the white rabbit simplifying fuzz testing using fuzzexmachina. https://i.blackhat.com/us-18/Thu-August-9/us-18-Ulitzsch-Follow-The-White-Rabbit-Simplifying-Fuzz-Testing-Using-FuzzExMachina.pdf/. Accessed: 2019-02-18.

[50] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy*, 2010.

[51] Zhi Wang, Jiang Ming, Chunfu Jia, and Debin Gao. Linear obfuscation to combat symbolic execution. In *European Symposium on Research in Computer Security (ESORICS)*, 2011.

[52] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[53] Gregory Wroblewski. *General method of program code obfuscation*. PhD thesis, Wroclaw University of Technology, 2002.

[54] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security Symposium*, 2018.

[55] Michael Zalewski. "technical whitepaper" for afl-fuzz. http://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed: 2019-02-18.

[56] Michał Zalewski. american fuzzy lop. http://lcamtuf.coredump.cx/afl/. Accessed: 2019-02-18.

A TIGRESS Configuration

Table 1: Tigress configuration for ANTIFUZZ evaluation. Asterisk means: "apply to all functions".

Transform	Functions
Virtualize	check
Flatten	*
Split	check
InitOpaque	main
EncodeLiterals	*
EncodeArithmetic	*
AddOpaque	*
AntiTaintAnalysis	*
UpdateOpaque	*
Ident	*
InitEntropy	main
AntiAliasAnalysis	*
InitBranchFuns	check
RandomFuns	*
InitImplicitFlow	check

MOPT: Optimized Mutation Scheduling for Fuzzers

Chenyang Lyu[†], Shouling Ji[†,+,(✉)], Chao Zhang[¶,(✉)], Yuwei Li[†], Wei-Han Lee[§], Yu Song[†], and Raheem Beyah[‡]

[†]Zhejiang University, [+]Alibaba-Zhejiang University Joint Research Institute of Frontier Technologies,
[¶]BNRist & INSC, Tsinghua University, [§]IBM Research, [‡]Georgia Institute of Technology
E-mails: puppet@zju.edu.cn, sji@zju.edu.cn, chaoz@tsinghua.edu.cn, liyuwei@zju.edu.cn,
wei-han.lee1@ibm.com, zjujakesong@gmail.com, rbeyah@ece.gatech.edu.

Abstract

Mutation-based fuzzing is one of the most popular vulnerability discovery solutions. Its performance of generating interesting test cases highly depends on the mutation scheduling strategies. However, existing fuzzers usually follow a specific distribution to select mutation operators, which is inefficient in finding vulnerabilities on general programs. Thus, in this paper, we present a novel mutation scheduling scheme MOPT, which enables mutation-based fuzzers to discover vulnerabilities more efficiently. MOPT utilizes a customized Particle Swarm Optimization (PSO) algorithm to find the optimal selection probability distribution of operators with respect to fuzzing effectiveness, and provides a pacemaker fuzzing mode to accelerate the convergence speed of PSO. We applied MOPT to the state-of-the-art fuzzers AFL, AFLFast and VUzzer, and implemented MOPT-AFL, -AFLFast and -VUzzer respectively, and then evaluated them on 13 real world open-source programs. The results showed that, MOPT-AFL could find 170% more security vulnerabilities and 350% more crashes than AFL. MOPT-AFLFast and MOPT-VUzzer also outperform their counterparts. Furthermore, the extensive evaluation also showed that MOPT provides a good rationality, compatibility and steadiness, while introducing negligible costs.

1 Introduction

Mutation-based fuzzing is one of the most prevalent vulnerability discovery solutions. In general, it takes seed test cases and selects them in certain order, then mutates them in various ways, and tests target programs with the newly generated test cases. Many new solutions have been proposed in the past years, including the ones that improve the seed generation solution [1, 2, 3], the ones that improve the seed selection strategy [4, 5, 6, 7, 8], the ones that improve the testing speed and code coverage [9, 10, 11, 12], and the ones that integrate other techniques with fuzzing [13, 14, 15].

However, less attention has been paid to how to mutate test cases to generate new effective ones. A large number of well-recognized fuzzers, e.g., AFL [16] and its descendants, libFuzzer [17], honggfuzz [18] and VUzzer [6], usually predefine a set of *mutation operators* to characterize where to mutate (e.g., which bytes) and how to mutate (e.g., add, delete or replace bytes). During fuzzing, they use certain *mutation schedulers* to select operators from this predefined set, in order to mutate test cases and generate new ones for fuzzing. Rather than directly yielding a mutation operator, the mutation scheduler yields a probability distribution of predefined operators, and the fuzzer will select operators following this distribution. For example, AFL uniformly selects mutation operators.

There are limited solutions focusing on improving the mutation scheduler. Previous works [7, 8] utilize reinforcement learning to dynamically select mutation operators in each round. However, they do not show significant performance improvements in vulnerability discovery [7, 8]. Thus, a better mutation scheduler is demanded. We figure out that, most previous works cannot achieve the optimal performance because they fail to take the following issues into consideration.

Different operators' efficiency varies. Different mutation operators have different efficiency in finding crashes and paths (as shown in Fig. 3). Thus, fuzzers that select mutation operators with the uniform distribution are likely to spend unnecessary computing power on inefficient operators and decrease the overall fuzzing efficiency.

One operator's efficiency varies with target programs. Each operator's efficiency is program-dependent, and it is unlikely or at least difficult to statically infer this dependency. Thus the optimal mutation scheduler has to make decisions per program, relying on each operator's runtime efficiency on the target program.

One operator's efficiency varies over time. A mutation operator that performs well on the current test cases may per-

Chenyang Lyu and Shouling Ji are the co-first authors. Shouling Ji and Chao Zhang are the co-corresponding authors.

form poorly on the following test cases in extreme cases. As aforementioned, the optimal mutation scheduler rely on operators' history efficiency to calculate the optimal probability distribution to select operators. Due to the dynamic characteristic of operator efficiency, this probability calculation process should converge fast.

The scheduler incurs performance overhead. Mutation schedulers have impacts on the execution speed of fuzzers. Since the execution speed is one of the key factors affecting fuzzers' efficiency, a better mutation scheduler should have fewer computations, to avoid slowing down fuzzers.

Unbalanced data for machine learning. During fuzzing, the numbers of positive and negative samples are not balanced, e.g., a mutation operator could only generate interesting test cases with a small probability, which may affect the effectiveness of gradient descent algorithms and other machine learning algorithms [7, 8].

In this paper, we consider mutation scheduling as an optimization problem and propose a novel mutation scheduling scheme MOPT, aiming at solving the aforementioned issues and improving the fuzzing performance. Inspired by the well-known optimization algorithm *Particle Swarm Optimization (PSO)* [19], MOPT dynamically evaluates the efficiency of candidate mutation operators, and adjusts their selection probability towards the optimum distribution.

MOPT models each mutation operator as a particle moving along the probability space $[x_{min}, x_{max}]$, where x_{min} and x_{max} are the pre-defined minimal and maximal probability, respectively. Guided by the local best probability and global best probability, each particle (i.e., operator) moves towards its optimal selection probability, which could yield more good-quality test cases. Accordingly, the target of MOPT is to find an optimal selection probability distribution of operators by aggregating the probabilities found by the particles, such that the aggregation yields more good-quality test cases. Similar to PSO, MOPT iteratively updates each particle's probability according to its local best probability and the global best probability. Then, it integrates the updated probabilities of all particles to obtain a new probability distribution. MOPT can quickly converge to the best solution of the probability distribution for selecting mutation operators and thus improves the fuzzing performance significantly.

MOPT is a generic scheme that can be applied to a wide range of mutation-based fuzzers. We have applied it to several state-of-the-art fuzzers, including AFL [16], AFLFast [5] and VUzzer [6], and implement MOPT-AFL, -AFLFast and -VUzzer, respectively. In AFL and its descendants, we further design a special pacemaker fuzzing mode, which could further accelerate the convergence speed of MOPT.

We evaluated these prototypes on 13 real world programs. In total, MOPT-AFL discovered 112 security vulnerabilities, including 97 previously unknown vulnerabilities (among which 66 are confirmed by CVE) and 15 known CVE vulnerabilities. Compared to AFL, MOPT-AFL found 170% more vulnerabilities, 350% more crashes and 100% more program paths. MOPT-*AFLFast* and MOPT-*VUzzer* also outperformed their counterparts on our dataset. We further demonstrated the rationality, steadiness and low costs of MOPT.

In summary, we have made the following contributions:

- We investigated the drawbacks of existing mutation schedulers, from which we conclude that mutation operators should be scheduled based on their history performance.
- We proposed a novel mutation scheduling scheme MOPT, which is able to choose better mutation operators and achieve better fuzzing efficiency. It can be generally applied to a broad range of existing mutation-based fuzzers.
- We applied MOPT to several state-of-the-art fuzzers, including AFL, AFLFast and VUzzer, and evaluated them on 13 real world programs. The results showed that MOPT could find much more vulnerabilities, crashes and program paths, with good steadiness, compatibility and low cost.
- MOPT-AFL discovers 97 previously unknown security vulnerabilities, and helps the vendors improve their products' security. It also finds 15 previously known vulnerabilities in these programs (of latest versions), indicating that security patching takes a long time in practice. We open source MOPT-AFL along with the employed data, seed sets, and results at *https://github.com/puppet-meteor/MOpt-AFL* to facilitate the research in this area. A technical report with more details can also be found there [20].

2 Background

2.1 Mutation-based Fuzzing

Mutation-based fuzzing [5, 6, 13, 14, 15, 16, 17, 18] is good at discovering vulnerabilities, without utilizing prior knowledge (e.g., test case specification) of target programs. Instead, it generates new test cases by mutating some well-formed seed test cases in certain ways.

The general workflow of mutation-based fuzzing is as follows. The fuzzer (1) maintains a queue of seed test cases, which can be updated at runtime; (2) selects some seeds from the queue in certain order; (3) mutates the seeds in various ways; (4) tests target programs with the newly generated test cases, and reports vulnerabilities or updates the seed queue if necessary; then (5) goes back to step (2).

In order to efficiently guide the mutation and fuzzing, some fuzzers will also instrument target programs to collect runtime information during testing, and use it to guide seeds updating and decide which seeds to select and how to mutate them. In this paper, we mainly focus on the mutation phase (i.e., step (3)).

2.2 Mutation Operators

Mutation-based fuzzers could mutate seeds in infinite number of ways. Considering the performance and usability,

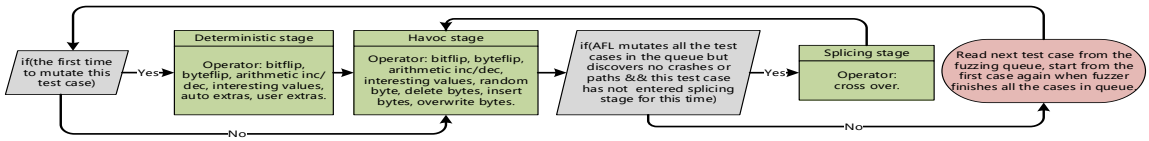


Figure 1: Three mutation scheduling schemes used in the three stages of AFL [16].

Table 1: Mutation operators defined by AFL [16].

Type	Meaning	Operators
bitflip	Invert one or several consecutive bits in a test case, where the stepover is 1 bit.	`bitflip 1/1`, `bitflip 2/1`, `bitflip 4/1`
byteflip	Invert one or several consecutive bytes in a test case, where the stepover is 8 bits.	`bitflip 8/8`, `bitflip 16/8`, `bitflip 32/8`
arithmetic inc/dec	Perform addition and subtraction operations on one byte or several consecutive bytes.	`arith 8/8`, `arith 16/8`, `arith 32/8`
interesting values	Replace bytes in the test cases with hard-coded interesting values.	`interest 8/8`, `interest 16/8`, `interest 32/8`
user extras	Overwrite or insert bytes in the test cases with user-provided tokens.	`user (over)`, `user (insert)`
auto extras	Overwrite bytes in the test cases with tokens recognized by AFL during `bitflip 1/1`.	`auto extras (over)`
random bytes	Randomly select one byte of the test case and set the byte to a random value.	`random byte`
delete bytes	Randomly select several consecutive bytes and delete them.	`delete bytes`
insert bytes	Randomly copy some bytes from a test case and insert them to another location in this test case.	`insert bytes`
overwrite bytes	Randomly overwrite several consecutive bytes in a test case.	`overwrite bytes`
cross over	Splice two parts from two different test cases to form a new test case.	`cross over`

in practice these fuzzers, including AFL [16] and its descendants, libFuzzer [17], honggfuzz [18] and VUzzer [6], usually predefine a set of mutation operators, and choose some of them to mutate seeds at runtime. These mutation operators characterize where to mutate (e.g., which bytes) and how to mutate (e.g., add, delete or replace bytes).

For example, the well-recognized fuzzer AFL predefines 11 types of mutation operators, as shown in Table 1. In each type, there could be several concrete mutation operators. For instance, the `bitflip 2/1` operator flips 2 consecutive bits, where the stepover is 1 bit. Note that, different fuzzers could define different mutation operators.

2.3 Mutation Scheduling Schemes

At runtime, mutation-based fuzzers continuously select some predefined mutation operators to mutate seed test cases. *Different fuzzers have different schemes to select operators.* For example, AFL employs three different scheduling schemes used in three stages, as shown in Fig. 1.

1. **Deterministic stage scheduler.** AFL applies a deterministic scheduling scheme for seed test cases that are picked to mutate for the first time. This scheduler employs 6 deterministic types of mutation operators in order, and applies them on the seed test cases one by one. For instance, it will apply `bitflip 8/8` to flip each byte of the seed test cases.

2. **Havoc stage scheduler.** The major mutation scheduling scheme of AFL is used in the havoc stage. As shown in Fig. 2, AFL first decides the number, denoted as R_t, of new test cases to generate in this stage. Each time, AFL selects a series of R_o mutation operators following the uniform distribution, and applies them on the seed to generate one test case. The havoc stage ends after R_t new test cases have been generated.

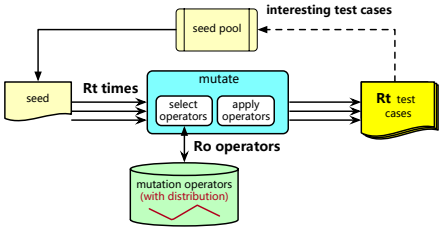


Figure 2: The general workflow of mutation-based fuzzing and mutation scheduling.

3. **Splicing stage scheduler.** In some rare cases, AFL works through the aforementioned two stages for all seeds, but fails to discover any unique crash or path in one round. Then AFL will enter a special splicing stage. In this stage, AFL only employs one operator `cross over` to generate new test cases. These new test cases will be fed to the havoc stage scheduler, rather than the program being tested, to generate new test cases.

The mutation scheduler in the first stage is deterministic and slow, while the one in the last stage is rarely used. The scheduler in the havoc stage, as shown in Fig. 2, is more generic and has been widely adopted by many fuzzers. Therefore, in this paper we mainly focus on improving the scheduler used in the havoc stage, which thus can be implemented in most mutation-based fuzzers. More specifically, we aim at finding an optimal probability distribution, following which the scheduler could select better mutation operators and improve the fuzzing efficiency.

2.4 Mutation Efficiency

Different mutation operators work quite differently. An intuitive assumption is that, they have different efficiency on different target programs. Some are better than others at generating the test cases, denoted as *interesting test cases*, that can trigger new paths or crashes.

To verify our hypothesis, we conducted an experiment on AFL to evaluate each operator's efficiency. To make the evaluation result deterministic, we only measured the interesting test cases produced by 12 mutation operators in the deterministic stage. The result is demonstrated in Fig. 3.

In the deterministic stage, the order of mutation operators and the times they are selected are fixed. Fig. 4 shows the

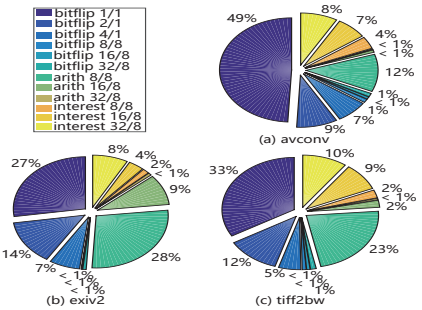


Figure 3: Percentages of interesting test cases produced by different operators in the deterministic stage of AFL.

order and the times that operators are selected by AFL during fuzzing `avconv`, indicating the time the fuzzer spent on.

• *Different mutation operators' efficiencies on one target program are different.* For most programs, the operators `bitflip 1/1`, `bitflip 2/1` and `arith 8/8` could yield more interesting test cases than other operators. On the other hand, several other mutation operators, such as `bitflip 16/8`, `bitflip 32/8` and `arith 32/8`, could only produce less than 2% of interesting test cases.

• *Each operator's efficiency varies with target programs.* An operator could yield good outputs on one program, but fail on another one. For example, `arith 8/8` performs well on `exiv2` and `tiff2bw`, but only finds 12% of the interesting test cases on `avconv`.

• *AFL spends most time on the deterministic stage.* We record the time each stage spends and the number of interesting test cases found by each stage in 24 hours, as shown in Fig. 5. We first analyze a special case. For `tiff2bw`, since AFL cannot find more interesting test cases, it finishes the deterministic stage of all the inputs in the fuzzing queue and skips the deterministic stage for a long time. Then, AFL spends most time on the havoc stage while finding nothing. For the other three cases, AFL spends more than 70% of the time on the deterministic stage. When fuzzing `avconv`, AFL even does not finish the deterministic stage of the first input in 24 hours. Another important observation is that the havoc stage is more efficient in finding interesting test cases compared to the deterministic stage. Moreover, since AFL spends too much time on the deterministic stage of one input, it cannot generate test cases from the later inputs in the fuzzing queue when fuzzing `avconv` and `pdfimages` given 24 hours. Note that since the splicing stage only uses `cross over` to mutate the test cases, it spends too little time to be shown in Fig. 5 compared to the other stages that will test the target program as mentioned in Section 2.3.

• *AFL spends much time on the inefficient mutation operators.* Fig. 3 shows that, the mutation operators `bitflip 1/1` and `bitflip 2/1` have found the most interesting test cases. But according to Fig. 4, they are only selected for a small number of times. On the other hand, inefficient operators like the ones of `interesting values` are selected too frequently but produce few interesting test cases, which

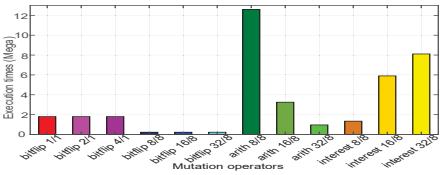


Figure 4: The times that mutation operators are selected when AFL fuzzes a target program `avconv`.

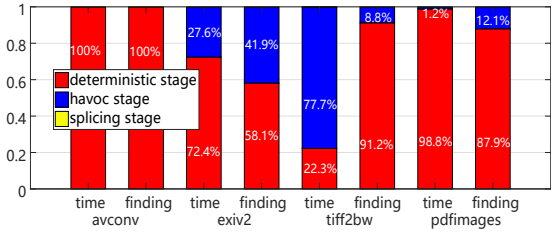


Figure 5: Percentages of time and interesting test cases used and found by the three stages in AFL, respectively.

decreases the fuzzing efficiency.

Motivation. Based on the analysis above, we observe that different mutation operators have different efficiencies. Hence, the mutation schedulers in existing fuzzers, which follow some pre-defined distributions, are not efficient. Ideally, more time should be spent on mutation operators that perform better at generating interesting test cases. Therefore, a better mutation scheduler is demanded.

3 Overview of MOPT

3.1 Design Philosophy

The mutation scheduler aims at choosing the next optimal mutation operator, which could find more interesting test cases, for a given runtime context. We simplify this problem as finding an optimal probability distribution of mutation operators, following which the scheduler chooses next operators when testing a target program.

Finding an optimal probability distribution for all mutation operators is challenging. Instead, we could first let each operator explore its own optimal probability. Then, based on those optimal probabilities, we could obtain a global optimal probability distribution of mutation operators.

The *Particle Swarm Optimization (PSO)* algorithm can be leveraged to find the optimal distribution of the operators and we detail the modification of PSO in our setting as follows.

3.2 Particle Swarm Optimization (PSO)

The PSO [19] algorithm is proposed by Eberhart and Kennedy, aiming at finding the optimal solution for a problem. It employs multiple particles to search the solution space iteratively, in which a position is a candidate solution.

As shown in Fig. 6, in each iteration, each particle is moved to a new position x_{now}, based on (1) its inertia (i.e.,

previous movement v_{now}), (2) displacement to its local best position L_{best} that *this particle* has found so far, and (3) displacement to the global best position G_{best} that *all particles* have found so far. Specifically, the movement of a particle P is calculated as follows:

$$v_{now}(P) \leftarrow w \times v_{now}(P) + r \times (L_{best}(P) - x_{now}(P)) + r \times (G_{best} - x_{now}(P)). \quad (1)$$

$$x_{now}(P) \leftarrow x_{now}(P) + v_{now}(P). \quad (2)$$

where w is the inertia weight and $r \in (0,1)$ is a random displacement weight.

Hence, each particle moves towards L_{best} and G_{best}, and is likely to keep moving to better positions. By moving towards G_{best}, multiple particles could work synchronously and avoid plunging into the local optimum. As a result, the swarm will be led to the optimal solution. Moreover, PSO is easy to implement with low computational cost, making it a good fit for optimizing mutation scheduling.

3.3 Design Details

MOPT aims to find an optimal probability distribution. Rather than employing particles to explore candidate distributions directly, we propose a customized PSO algorithm to explore each operator's optimal probability first, and then construct the optimal probability distribution.

3.3.1 Particles

MOPT employs a particle per operator, and tries to explore an optimal position for each operator in a predefined probability space $[x_{min}, x_{max}]$, where $0 < x_{min} < x_{max} \leq 1$.

The current position of a particle (i.e., operator) in the probability space, i.e., x_{now}, represents the probability that this operator will be selected by the scheduler. Due to the nature of probabilities, the sum of all the particles' probabilities in one iteration should be normalized to 1.

3.3.2 Local Best Position L_{best}

Similar to PSO, MOPT also appoints the best position that a particle has ever found as its local best position.

For a given particle, a position x_1 is better than x_2, if and only if, its corresponding operator yields more interesting test cases (with a same amount of invocations) in the former position than the latter. Thus, L_{best} is the position of the particle where the corresponding operator yields the most interesting test cases (given the same amount of invocations).

To enable this comparison, for each particle (i.e., operator), we measure its **local efficiency** eff_{now}, i.e., the number of interesting test cases contributed by this operator divided by the number of invocations of this operator *during one iteration*. We denote the largest eff_{now} as eff_{best}. Thus, L_{best} is the position where the operator obtains eff_{best} in history.

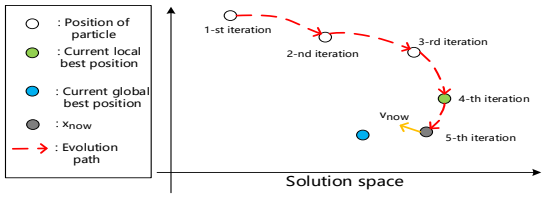


Figure 6: An example of illustrating the evolution of one particle at the 5-th iteration according to the PSO.

3.3.3 Global Best Position G_{best}

PSO appoints the best position that all particles have ever found as the global best position. Note that, unlike the original PSO which moves particles in a unified solution space, MOPT moves particles in different probability spaces (with same shape and size). Hence, there is no sole global best position fit for all particles. Instead, different particles have different global best positions (in different spaces) here.

In PSO, global best positions depend on the relationship between different particles. Hereby we also evaluate each particle's efficiency from a global perspective, denoted as **global efficiency** $global_{eff}$, by evaluating multiple swarms of particles at a time.

More specifically, we measure the number of interesting test cases contributed by each operator till now in all swarms, and use it as the particle's global efficiency $global_{eff}$. Then we compute the distribution of all particles' global efficiency. For each operator (i.e., particle), its global best position G_{best} is defined as the proportion of its $global_{eff}$ in this distribution. With this distribution, particles (i.e., operators) with higher efficiency can get higher probability to be selected.

3.3.4 Multiple Swarms

Given the definitions of particles, local best positions and global best positions, we could follow the PSO algorithm to approach to an optimal solution (i.e., a specific probability distribution of mutation operators).

However, unlike the original PSO swarm that has multiple particles exploring the solution space, the swarm defined by MOPT actually only explores one candidate solution (i.e., probability distribution) in the solution space, and thus is likely to fall into local optimum. Thus, MOPT employs multiple swarms and applies the customized PSO algorithm to each swarm, as shown in Fig. 7, to avoid local optimum.

Synchronization is required between these swarms. MOPT simply takes the most efficient swarm as the best and uses its distribution to schedule mutation during fuzzing. Here, we define the **swarm's efficiency** (denoted as $swarm_{eff}$) as the number of interesting test cases contributed by this swarm divided by the number of new test cases *during one iteration*.

Overview: In summary, MOPT employs multiple swarms and applies the customized PSO algorithm to each swarm. During fuzzing, the following three extra tasks are performed

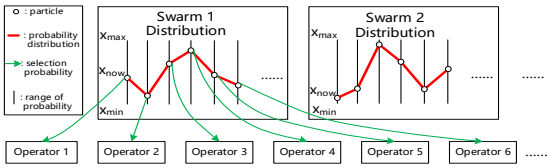


Figure 7: Illustration of the particle swarms of MOPT.

in each iteration of PSO.

• **T1:** *Locate local best positions for all particles in each swarm.* Within each swarm, each particle's local efficiency eff_{now} in one iteration is evaluated during fuzzing. For each particle, the position with the highest efficiency eff_{best} in history is marked as its local best position L_{best}.

• **T2:** *Locate global best positions for all particles across swarms.* Each particle's global efficiency $global_{eff}$ is evaluated across swarms. The distribution of the particles' global efficiency is then evaluated. The proportion of each particle's $global_{eff}$ in this distribution is used as its global best position G_{best}.

• **T3:** *Select the best swarm to guide fuzzing.* Each swarm's efficiency $swarm_{eff}$ in one iteration is evaluated. The swarm with the highest $swarm_{eff}$ is chosen, and its probability distribution in the current iteration is applied for further fuzzing.

Then, at the end of each iteration, MOPT moves the particles in each swarm in a similar way as PSO. More specifically, for a particle P_j in a swarm S_i, we update its position as follows.

$$\begin{aligned} v_{now}[S_i][P_j] \leftarrow & w \times v_{now}[S_i][P_j] \\ & + r \times (L_{best}[S_i][P_j] - x_{now}[S_i][P_j]) \\ & + r \times (G_{best}[P_j] \quad - x_{now}[S_i][P_j]). \end{aligned} \tag{3}$$

$$x_{now}[S_i][P_j] \leftarrow x_{now}[S_i][P_j] + v_{now}[S_i][P_j]. \tag{4}$$

where w is the inertia weight and $r \in (0,1)$ is a random displacement weight.

Further, we normalize these positions to meet some constraints. First, each particle's position is adjusted to fit in the probability space $[x_{min}, x_{max}]$. Then for each swarm, all its particles' positions (i.e., probabilities) will be normalized, such that the sum of these probabilities equals to 1.

After updating the positions of all particles in all swarms, the fuzzer could keep moving those particles into new positions, and enter a new iteration of PSO.

4 Implementation of MOPT

4.1 MOPT Main Framework

As shown in Fig. 8, MOPT consists of four core modules, i.e., the PSO initialization and updating modules, as well as the pilot fuzzing and core fuzzing modules.

The PSO initialization module is executed once and used for setting the initial parameters of the PSO algorithm. The

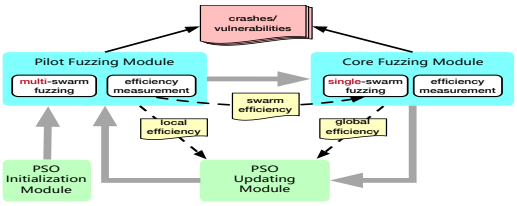


Figure 8: The workflow of MOPT.

other three modules form an iteration loop and work together to continuously fuzz target programs.

In each iteration of the loop, the PSO particles are updated once. In order to update particles' positions with the PSO algorithm, we need to find each particle's local best position and global best position in each iteration.

• The pilot fuzzing module employs multiple swarms, i.e., multiple probability distributions, to select mutation operators and fuzz. During fuzzing, the local efficiency of each particle in each swarm is measured. Hence, we could find the *local best position* of each particle in each swarm.

• Moreover, during the pilot fuzzing, each swarm's efficiency is also evaluated. Then, the *most efficient swarm* is chosen, and the core fuzzing module will use the probability distribution explored by it to schedule mutation operators.

• After the core fuzzing module finishes, the total number of interesting test cases contributed by each operator till now can be evaluated. Hence, each particle's global efficiency (i.e., *global best position*) could be evaluated.

With this iteration loop, the fuzzer could utilize the PSO to find an optimal probability distribution to select mutation operators, and gradually improve the fuzzing efficiency.

Note that, MOPT's workflow is independent from the target fuzzer, as long as the fuzzer's mutation scheduler uses a probability distribution to select operators. We do not need to change the behavior of the target fuzzer, except that evaluating the efficiency of the fuzzer in order to move PSO particles. The instrumentation to the target fuzzer is minimum and costs few performance overhead.

Hence, MOPT is a generic and practical mutation scheduling scheme, and can be applied to a variety of fuzzers.

4.1.1 PSO Initialization Module

This module initializes parameters for the PSO algorithm. More specifically, MOPT (1) sets the initial location x_{now} of each particle in each swarm with a random value, and normalizes the sum of x_{now} of all the particles in one swarm to 1; (2) sets the displacement of particle movement v_{now} of each particle in each swarm to 0.1; (3) sets the initial local efficiency eff_{now} of each particle in each swarm to 0; (4) sets the initial local best position L_{best} of each particle in each swarm to 0.5; and (5) sets the initial global best position G_{best} of each particle across swarms to 0.5. Note that, the initialization module only executes once when the fuzzer starts running.

4.1.2 Pilot Fuzzing Module

This module employs multiple swarms to perform fuzzing, where each swarm explores a different probability distribution. This module evaluates each swarm in order, and stops testing a swarm after it has generated a configurable number (denoted as $period_{pilot}$) of new test cases. The process of fuzzing with a specific swarm is as follows.

For each swarm, its probability distribution is used to schedule the selection of mutation operators and fuzz the target program. During fuzzing, the module will measure three measurements: (1) the number of interesting test cases contributed by a specific particle (i.e., operator), (2) the number of invocations of a specific particle, (3) the number of interesting test cases found by this swarm, by instrumenting target programs.

The local efficiency of each particle (in current swarm) is the first measurement divided by the second measurement. Hence, we could locate the *local best position* of each particle. The current *swarm's efficiency* is the third measurement divided by the test case count $period_{pilot}$. Therefore, we could find the most efficient swarm.

4.1.3 Core Fuzzing Module

This module will take the best swarm selected by the pilot fuzzing module, and use its probability distribution to perform fuzzing. It will stop after generating a configurable number (denoted as $period_{core}$) of new test cases.

Once it stops, we could measure the number of interesting test cases contributed by each particle, regardless which swarm it belongs to, from the start of PSO initialization till now. Then we could calculate the distribution between particles, and locate each particle's *global best position*.

Note that, if we only use one swarm in the pilot module, then the core module could be merged with the pilot module.

4.1.4 PSO Updating Module

With the information provided by the pilot and core fuzzing modules, this module updates the particles in each swarm, following Equations 3 and 4.

After updating each particle, we will enter the next iteration of PSO updates. Hence, we could approach to an optimal swarm (i.e., probability distribution for operators), use it to guide the core fuzzing module, and help improve the fuzzing efficiency.

4.2 Pacemaker Fuzzing Mode

Although applying MOPT to mutation-based fuzzers is generic, we realize the performance of MOPT can be further optimized when applied to specific fuzzers such as AFL.

Table 2: Objective programs evaluated in our experiments.

Target	Source file	Input format	Test instruction
mp42aac	Bento4-1-5-1	mp4	mp42aac @@ /dev/null
exiv2	exiv2-0.26-trunk	jpg	exiv2 @@ /dev/null
mp3gain	mp3gain-1_5_2	mp3	mp3gain @@ /dev/null
tiff2bw	libtiff-4.0.9	tiff	tiff2bw @@ /dev/null
pdfimages	xpdf-4.00	PDF	pdfimages @@ /dev/null
sam2p	sam2p-0.49.4	bmp	sam2p @@ EPS: /dev/null
avconv	libav-12.3	mp4	avconv -y -i @@ -f null -
w3m	w3m-0.5.3	text	w3m @@
objdump	binutils-2.30	binary	objdump –dwarf-check -C -g -f -dwarf -x @@
jhead	jhead-3.00	jpg	jhead @@
mpg321	mpg321_0.3.2	mp3	mpg321 -t @@ /dev/null
infotocap	ncurses-6.1	text	infotocap @@
podofopdfinfo	podofo-0.9.6	PDF	podofopdfinfo @@

Based on extensive empirical analysis, we realize that AFL and its descendants spend much more time on the deterministic stage, than on the havoc and splicing stages that can discover many more unique crashes and paths. MOPT therefore provides an optimization to AFL-based fuzzers, denoted as *pacemaker fuzzing mode*, which selectively avoids the time-consuming deterministic stage.

Specifically, when MOPT finishes mutating one seed test case, if it has not discovered any new unique crash or path for a long time, i.e., T that is set by users, it will selectively disable the deterministic stage for the following test cases. The pacemaker fuzzing mode has the following advantages.

• The deterministic stage spends too much time and would slow down the overall efficiency. On the other hand, MOPT only updates the probability distribution in the havoc stage, independent from the deterministic stage. Therefore, disabling the deterministic stage with *the pacemaker fuzzing mode* could accelerate the convergence speed of MOPT.

• In this mode, the fuzzer can skip the deterministic stage, without spending too much time on a sole test case. Instead, it will pick more seeds from the fuzzing queue for mutation, and thus has a better chance to find vulnerabilities faster.

• The deterministic stage may have good performance at the beginning of fuzzing, but becomes inefficient after a while. This mode selectively disables this stage only after the efficiency slows down, and thus benefits from this stage while avoiding wasting much time on it.

More specifically, MOPT provides two types of pacemaker fuzzing modes for AFL, based on whether the deterministic stage will be re-enabled or not: (1) MOPT-*AFL-tmp*, which will re-enable the deterministic stage again when the number of new interesting test cases exceeds a predefined threshold; (2) MOPT-*AFL-ever*, which will never re-enable the deterministic stage in the following fuzzing process.

5 Evaluation

5.1 Real World Datasets

We have evaluated MOPT on 13 open-source linux programs as shown in Table 2, each of which comes from dif-

Table 3: The unique crashes and paths found by AFL, MOPT-AFL-tmp and MOPT-AFL-ever on the 13 real world programs.

Program	AFL		MOPT-AFL-tmp				MOPT-AFL-ever			
	Unique crashes	Unique paths	Unique crashes	Increase	Unique paths	Increase	Unique crashes	Increase	Unique paths	Increase
mp42aac	135	815	**209**	+54.8%	1,660	+103.7%	199	+47.4%	**1,730**	+112.3%
exiv2	34	2,195	54	+58.8%	2,980	+35.8%	**66**	+94.1%	**4,642**	+111.5%
mp3gain	178	1,430	**262**	+47.2%	**2,211**	+54.6%	**262**	+47.2%	2,206	+54.3%
tiff2bw	4	4,738	**85**	+2,025.0%	**7,354**	+55.2%	43	+975.0%	7,295	+54.0%
pdfimages	23	12,915	357	+1,452.2%	22,661	+75.5%	**471**	+1,947.8%	**26,669**	+106.5%
sam2p	36	531	105	+191.7%	1,967	+270.4%	**329**	+813.9%	**3,418**	+543.7%
avconv	0	2,478	**4**	+4	**17,359**	+600.5%	1	+1	16,812	+578.5%
w3m	0	3,243	**506**	+506	5,313	+63.8%	182	+182	**5,326**	+64.2%
objdump	0	11,565	**470**	+470	19,309	+67.0%	287	+287	**22,648**	+95.8%
jhead	19	478	55	+189.5%	**489**	+2.3%	**69**	+263.2%	483	+1.0%
mpg321	10	123	**236**	+2,260.0%	1,054	+756.9%	229	+2,190.0%	**1,162**	+844.7%
infotocap	92	3,710	340	+269.6%	6,157	+66.0%	**692**	+652.2%	**7,048**	+90.0%
podofopdfinfo	79	3,397	**122**	+54.4%	**4,704**	+38.5%	114	+44.3%	4,694	+38.2%
total	610	47,618	2,805	+359.8%	93,218	+95.8%	**2,944**	+382.6%	**104,133**	+118.7%

ferent source files and has different functionality representing a broad range of programs. We choose these 13 programs mainly for the following reasons. First, many of the employed programs are also widely used in state-of-the-art fuzzing research [4, 5, 9, 10, 21]. Second, most programs employed in our experiments are real world programs from different vendors and have diverse functionalities and various code logic. Therefore, our datasets are representative and can all-sidedly measure the fuzzing performance of fuzzers to make our analysis more comprehensive. Third, all the employed programs are popular and useful open-source programs. Hence, evaluating the security of these programs are meaningful for the vendors and users of them.

5.2 Experiment Settings

The version of AFL used in our paper is 2.52b. We apply MOPT in the havoc stage of AFL and implement the prototypes of MOPT-AFL-tmp and MOPT-AFL-ever, where -tmp and -ever indicate the corresponding pacemaker fuzzing modes discussed in the previous section. The core functions of MOPT is implemented in C.

Platform. All the experiments run on a virtual machine configured with 1 CPU core of 2.40GHz E5-2640 V4, 4.5GB RAM and the OS of 64-bit Ubuntu 16.04 LTS.

Initial seed sets. Following the same seed collection and selection procedure as in previous works [3, 22, 23], we use randomly-selected files as the initial seed sets. In particular, for each objective program, we obtain 100 files with the corresponding input format as the initial seed set, e.g., we collect 100 mp3 files for `mp3gain`. The input format of each program is shown in Table 2. In particular, we first download the files with the corresponding input formats for each objective program from the music download websites, picture download websites, and so on (except for text files, where we obtain text files by randomly generating letters to fill them). Then, for the large files such as mp3 and PDF, we split them to make their sizes reasonable as seeds. Through this way, we have a large corpus of files with the corresponding input formats for each objective program. Finally, we randomly select 100 files from the corpus. These 100 files will be the initial seed set of all the fuzzers when fuzzing one objective program.

Evaluation metrics. The main evaluation metric is the number of the unique crashes discovered by each fuzzer. Since coverage-based fuzzers such as *AFLFast* [5] and *VUzzer* [6] consider that exploring more unique paths leads to more unique crashes, the second evaluation metric is the number of unique paths discovered by each fuzzer.

5.3 Unique Crashes and Paths Discovery

We evaluate AFL, MOPT-AFL-tmp and MOPT-AFL-ever on the 13 programs in Table 2, with each experiment runs for 240 hours. The results are shown in Table 3, from which we can deduce the following conclusions.

• For exploring unique crashes, MOPT-AFL-tmp and MOPT-AFL-ever are significantly more efficient than AFL on all the programs. In total, MOPT-AFL-tmp and MOPT-AFL-ever discover 2,195 and 2,334 more unique crashes than AFL on the 13 programs. Thus, MOPT-AFL has much better performance than AFL in exploring unique crashes.

• For triggering unique paths, MOPT-AFL-tmp and MOPT-AFL-ever also significantly outperform AFL. In total, MOPT-AFL-tmp and MOPT-AFL-ever found 45,600 and 56,515 more unique paths than AFL on the 13 programs. As a result, the proposed MOPT can improve the coverage of AFL remarkably.

• When considering the pacemaker fuzzing mode, MOPT-AFL-tmp and MOPT-AFL-ever discover the most unique crashes on 8 and 6 programs, respectively, while MOPT-AFL-ever discovers more crashes in total. Since the main difference between the two fuzzers is whether using the deterministic stage later, it may be an interesting future work to figure out how to employ the deterministic stage properly.

Table 4: Vulnerabilities found by AFL, MOPT-AFL-tmp and MOPT-AFL-ever.

Program	AFL				MOPT-AFL-tmp				MOPT-AFL-ever			
	Unknown vulnerabilities		Known vulnerabilities	Sum	Unknown vulnerabilities		Known vulnerabilities	Sum	Unknown vulnerabilities		Known vulnerabilities	Sum
	Not CVE	CVE	CVE		Not CVE	CVE	CVE		Not CVE	CVE	CVE	
mp42aac	/	1	1	2	/	2	1	3	/	5	1	6
exiv2	/	5	3	8	/	5	4	9	/	4	4	8
mp3gain	/	4	2	6	/	9	3	12	/	5	2	7
pdfimages	/	1	0	1	/	12	3	15	/	9	2	11
avconv	/	0	0	0	/	2	0	2	/	1	0	1
w3m	/	0	0	0	/	14	0	14	/	5	0	5
objdump	/	0	0	0	/	1	2	3	/	0	2	2
jhead	/	1	0	1	/	4	0	4	/	5	0	5
mpg321	/	0	1	1	/	0	1	1	/	0	1	1
infotocap	/	3	0	3	/	3	0	3	/	3	0	3
podofopdfinfo	/	5	0	5	/	6	0	6	/	6	0	6
tiff2bw	1	/	/	1	2	/	/	2	2	/	/	2
sam2p	5	/	/	5	14	/	/	14	28	/	/	28
Total	6	20	7	33	16	58	14	88	30	43	12	85

5.4 Vulnerability Discovery

To figure out the corresponding vulnerabilities of the crashes found in Section 5.3, we recompile the evaluated programs with *AddressSanitizer* [24] and reevaluate them with the discovered crash inputs. If the top three source code locations of the stack trace provided by *AddressSanitizer* are unique, we consider the corresponding crash input triggers a unique vulnerability of the objective program. This is a common way to find unique vulnerabilities in practice and has been used to calculate the stack hashing in [25]. Then, we check the vulnerability reports of the target program on the CVE website to see whether they correspond to some already existed CVEs. If not, we submit the vulnerability reports and the Proof of Concepts (PoCs) to the vendors and the CVE assignment team. The vulnerabilities discovered by AFL, MOPT-AFL-tmp and MOPT-AFL-ever are shown in Table 4, from which we have the following conclusions.

• Both MOPT-AFL-tmp and MOPT-AFL-ever discover more vulnerabilities than AFL by a wide margin. For instance, MOPT-AFL-tmp finds 45 more security CVEs than AFL; MOPT-AFL-ever finds 23 more unreported CVEs than AFL; Our fuzzers find 81 security CVEs with 66 new CVE IDs assigned on 11 programs. The results demonstrate that MOPT-AFL is very effective on exploring CVEs.

• Our fuzzers discover 15 previously known vulnerabilities published by CVE on the latest version of the objective programs. For instance, when fuzzing `pdfimages`, MOPT-AFL-tmp and MOPT-AFL-ever discover 3 and 2 existed vulnerabilities, respectively. The results demonstrate that security patching takes a long time in practice.

• AFL, MOPT-AFL-tmp and MOPT-AFL-ever discover 1, 2 and 2 unique vulnerabilities on `tiff2bw`, respectively. As for `sam2p`, MOPT-AFL-tmp and MOPT-AFL-ever discover 14 and 28 unique vulnerabilities, respectively. In comparison, AFL only finds 5 vulnerabilities. Since the vulnerabilities happened in the `tiff2bw` command-line program and the CVE assignment team thinks that `sam2p` is a UNIX command line program rather than a library, they cannot assign CVE IDs for the vulnerabilities on `tiff2bw` and `sam2p`. On all the 13 programs, MOPT-AFL-tmp and MOPT-AFL-ever discover 112 unique vulnerabilities in total, and AFL discovers 33 vulnerabilities.

5.5 CVE Analysis

In this subsection, we analyze the CVEs discovered in Section 5.4 in detail and discuss the performance of different fuzzers. We also measure the severity of each CVE for each program by leveraging the Common Vulnerability Scoring System (CVSS) [26] and show the highest score in Table 5. We can learn the following conclusions.

• Both MOPT-AFL-tmp and MOPT-AFL-ever find more kinds of vulnerabilities than AFL, which means MOPT-AFL does not limit on discovering specific kinds of vulnerabilities. In other words, the MOPT scheme can guide the fuzzing tools to discover various vulnerabilities.

• We realize that MOPT-AFL-tmp discovers significantly more unique vulnerabilities than MOPT-AFL-ever on `pdfimages` and `w3m`. We analyze the reasons as follows. First of all, we would like to clarify the functionalities of these two objective programs. `pdfimages` is used to save images from the PDF files as the image files locally. `w3m` is a pager and/or text-based browser, which can handle tables, cookies, authentication, and almost everything except for JavaScript. We notice that PDF files have complex structures and so do the web data handled by `w3m`. Thus, there are many magic byte checks in `pdfimages` and `w3m` to handle the complex structures. Because it is hard to randomly generate a particular value, the operators in the deterministic stage, such as flipping the bits one by one (**bitflip**) and replacing the bytes with interesting values (**interesting values**), are better than the ones in the havoc stage to pass the magic byte checks and to test deeper execution paths. MOPT-AFL-tmp performs better than MOPT-AFL-ever on `pdfimages` and `w3m` since MOPT-AFL-tmp enables the deterministic stage later while MOPT-AFL-ever does not. However, since the deterministic stage performs multiple kinds of operators on each

Table 5: The types and IDs of CVE discovered by AFL, MOPT-AFL-tmp and MOPT-AFL-ever.

Target	Types	AFL	MOPT-AFL-tmp	MOPT-AFL-ever	Severity
mp42aac	buffer overflow	CVE-2018-10785	CVE-2018-10785; CVE-2018-18037	CVE-2018-10785; CVE-2018-18037; CVE-2018-17814	4.3
	memory leaks	CVE-2018-17813	CVE-2018-17813	CVE-2018-17813; CVE-2018-18050; CVE-2018-18051	4.3
exiv2	heap overflow	CVE-2017-11339; CVE-2017-17723; CVE-2018-18036	CVE-2017-11339; CVE-2017-17723; CVE-2018-10780	CVE-2017-11339; CVE-2017-17723; CVE-2018-18036	5.8
	stack overflow	CVE-2017-14861	CVE-2017-14861	CVE-2017-14861	4.3
	buffer overflow	CVE-2018-18047	CVE-2018-17808; CVE-2018-18047	CVE-2018-18047	4.3
	segmentation violation	CVE-2018-18046	CVE-2018-18046	CVE-2018-18046	4.3
	memory access violation	CVE-2018-17809; CVE-2018-17807	CVE-2018-17809; CVE-2018-17823	CVE-2017-11337; CVE-2018-17809	4.3
mp3gain	stack buffer overflow	CVE-2017-14407	CVE-2017-14407; CVE-2018-17801; CVE-2018-17799	CVE-2017-14407	4.3
	global buffer overflow	CVE-2018-17800; CVE-2018-17802; CVE-2018-18045; CVE-2018-18043	CVE-2017-14409; CVE-2018-17800; CVE-2018-17803; CVE-2018-17802; CVE-2018-18045; CVE-2018-18043; CVE-2018-18044	CVE-2018-17800; CVE-2018-17803; CVE-2018-17802; CVE-2018-18045; CVE-2018-18043	6.8
	segmentation violation	CVE-2017-14406	CVE-2017-14412	CVE-2017-14412	6.8
	memcpy param overlap		CVE-2018-17824		5.8
pdfimages	heap buffer overflow		CVE-2018-8103; CVE-2018-18054		4.3
	stack overflow	CVE-2018-17114	CVE-2018-16369; CVE-2018-17114; CVE-2018-17115; CVE-2018-17116; CVE-2018-17117; CVE-2018-17119; CVE-2018-17120; CVE-2018-17121; CVE-2018-17122; CVE-2018-18053; CVE-2018-18055	CVE-2018-16369; CVE-2018-17115; CVE-2018-17116; CVE-2018-17119; CVE-2018-17121; CVE-2018-17122; CVE-2018-18053	6.1
	global buffer overflow		CVE-2018-8102	CVE-2018-8102	4.3
	alloc dealloc mismatch		CVE-2018-17118	CVE-2018-17118	4.3
	segmentation violation			CVE-2018-17123; CVE-2018-17124	4.3
avconv	segmentation violation		CVE-2018-17804	CVE-2018-17804	4.3
	memory leaks		CVE-2018-17805		4.3
w3m	segmentation violation		CVE-2018-17815; CVE-2018-17816; CVE-2018-17817; CVE-2018-17818; CVE-2018-17819; CVE-2018-17821; CVE-2018-17822; CVE-2018-18038; CVE-2018-18039; CVE-2018-18040; CVE-2018-18041; CVE-2018-18042; CVE-2018-18052	CVE-2018-17816; CVE-2018-18040; CVE-2018-18041; CVE-2018-18042	5.3
	memory leaks		CVE-2018-17820	CVE-2018-17820	4.3
objdump	stack exhaustion		CVE-2018-12700	CVE-2018-12641	5.0
	stack overflow		CVE-2018-9138; CVE-2018-16617	CVE-2018-9138	4.3
jhead	heap buffer overflow	CVE-2018-17810	CVE-2018-17810; CVE-2018-17811; CVE-2018-18048; CVE-2018-18049	CVE-2018-17810; CVE-2018-17811; CVE-2018-17812; CVE-2018-18048; CVE-2018-18049	4.3
mpg321	heap buffer overflow	CVE-2017-12063	CVE-2017-12063	CVE-2017-12063	4.3
infotocap	memory leaks	CVE-2018-16614	CVE-2018-16614	CVE-2018-16614	4.3
	segmentation violation	CVE-2018-16615; CVE-2018-16616	CVE-2018-16615; CVE-2018-16616	CVE-2018-16615; CVE-2018-16616	4.3
podofopdfinfo	stack overflow	CVE-2018-18216; CVE-2018-18221; CVE-2018-18222	CVE-2018-18216; CVE-2018-18217; CVE-2018-18221; CVE-2018-18222	CVE-2018-18216; CVE-2018-18217; CVE-2018-18218; CVE-2018-18221	4.7
	heap buffer overflow	CVE-2018-18219	CVE-2018-18219	CVE-2018-18219	4.3
	segmentation violation	CVE-2018-18220	CVE-2018-18220	CVE-2018-18220	4.3

bit/byte of the test cases, it takes a lot of time to finish all the operations on each test case in the fuzzing queue, leading to the low efficiency. On the other hand, MOPT-AFL-tmp temporarily uses the deterministic stage on different test cases in the fuzzing queue to avoid this disadvantage.

• Interestingly, we can also see that although we fuzz the objective programs with the latest version, MOPT-AFL still discovers already existed CVEs. For instance, we reproduce the Proof of Concepts (PoCs) of CVE-2017-17723 of `exiv2`, which can cause the overflow and has 5.8 CVSS Score according to *CVE Details* [27]. It may because the vendors do not patch the vulnerabilities before the release or they patch the vulnerabilities while MOPT-AFL still discovers other PoCs. Therefore, the servers using these programs may be attacked because of these vulnerabilities. In addition, most of the discovered vulnerabilities can crash the programs and allow remote attackers to launch denial of service attacks via a crafted file. Thus, a powerful fuzzer is needed to improve the security patching.

Case study: CVE-2018-18054 in `pdfimages`. An interesting vulnerability we found is a heap buffer overflow in `pdfimages`. Although the PDF files in the seed set do not contain pictures that use the CCITTFax encoding, a test case generated by MOPT-AFL-tmp still triggers the CCITTFax decoding process of `pdfimages`. Furthermore, even the PDF syntax of this test case is partially damaged, `pdfimages` continues to extract the pictures from it. Then, the test case triggers the function `GBool CCITTFaxStream::readRow()` in `Stream.cc` for multiple times and finally accesses the data that exceed the index of the array `refLine`, which leads to a heap buffer overflow. This vulnerability shows the powerful mutation capability of MOPT-AFL-tmp, which not only generates a structure similar to an encoding algorithm but also triggers an array out of bounds.

5.6 More Analysis on Discovered Crashes

In this subsection, we give a close look on the growth of the number of unique crashes discovered by MOPT-AFL-ever, MOPT-AFL-tmp and AFL. The results are shown in Fig. 9, from which we have the following conclusions.

• Both MOPT-AFL-ever and MOPT-AFL-tmp are effective at finding unique crashes. On most programs, they take fewer than 100 hours to find more unique crashes than AFL does in 240 hours.

• We can learn from Fig. 9 (c) and (f) that within a relatively short time, AFL may discover more crashes than MOPT-AFL-ever and MOPT-AFL-tmp. The reasons are as follows. First, mutation-based fuzzing has randomness, and naturally such randomness may cause performance shaking within a short time. However, relatively stable performance will exhibit in a long time-scale as shown in the experiments. Second, the selection probability distribution of mutation operators in MOPT-AFL-ever and MOPT-AFL-tmp adopts random initialization, which may cause fuzzing randomness in the early fuzzing time. Thus, to reduce the performance instability of fuzzing, a relatively long time experiment is necessary, e.g., we run our experiments for 240 hours.

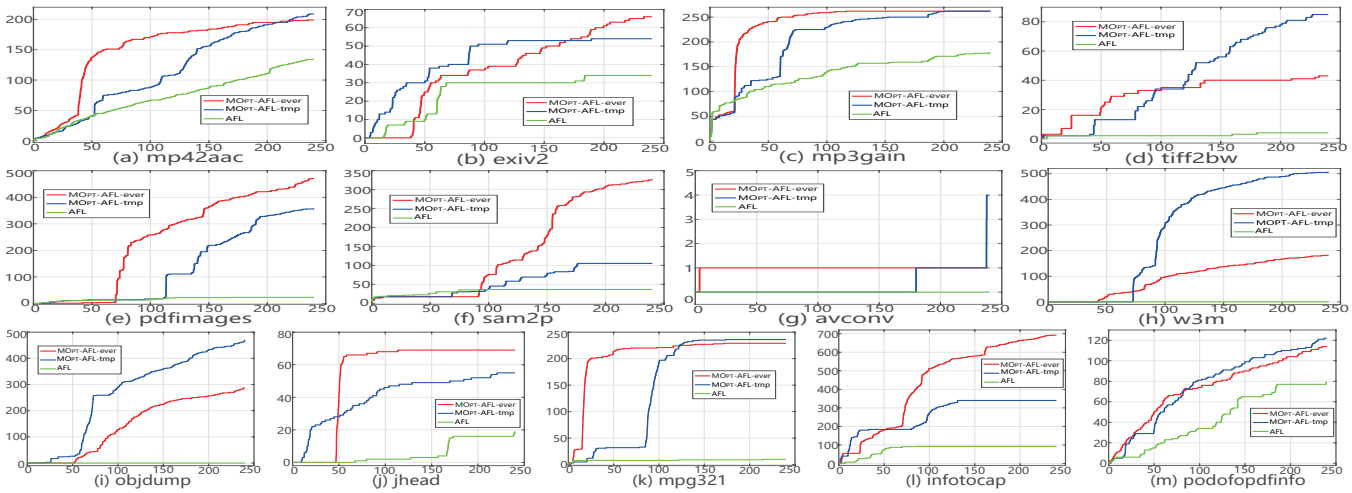


Figure 9: The number of unique crashes discovered by MOPT-AFL-ever, MOPT-AFL-tmp and AFL over 240 hours. X-axis: time (over 240 hours). Y-axis: the number of unique crashes.

5.7 Compatibility Analysis

In addition to AFL, we also generalize our analysis to several state-of-the-art mutation-based fuzzers, e.g., AFLFast [5] and VUzzer [6], and study the compatibility of MOPT.

AFLFast [5] is one of the coverage-based fuzzers. By using a power schedule to guide the fuzzer towards low frequency paths, AFLFast can detect more unique paths and explore the vulnerabilities much faster than AFL. To examine the compatibility of MOPT, we implement MOPT-AFLFast-tmp and MOPT-AFLFast-ever based on AFLFast.

VUzzer [6] is a fuzzer that focuses on exploring deeper paths. VUzzer can evaluate a test case with the triggered path and select the test cases with higher fitness scores to generate subsequent test cases. The mutation strategy of VUzzer is different from AFL. In each period, VUzzer generates a fixed number of mutated test cases, evaluates their fitness and only keeps *POPSIZE* test cases with the highest fitness scores to generate test cases in the next period, where *POPSIZE* is the population number of the parent test cases set by users. We regard the mutation operators as the high-efficiency operators that can generate the test cases with top-(*POPSIZE/3*) fitness scores. Then, we combine MOPT with VUzzer and implement MOPT-VUzzer. Since VUzzer does not have a deterministic stage like AFL, we do not consider the pacemaker fuzzing mode here.

Now, we evaluate MOPT-AFLFast-tmp, MOPT-AFLFast-ever, and MOPT-VUzzer on `mp42aac`, `exiv2`, `mp3gain`, `tiff2bw`, `pdfimages`, `sam2p` and `mpg321`. Each experiment is lasted for 240 hours with the same settings as in Section 5.2. Specifically, we change the OS as the 32-bit Ubuntu 14.04 LTS for VUzzer and MOPT-VUzzer because of VUzzer's implementation restriction. The results are shown in Table 6. We have the following conclusions.

- MOPT-AFLFast-tmp and MOPT-AFLFast-ever have much better performance than AFLFast in discovering unique crashes on all the programs. For instance, MOPT-AFLFast-ever finds 327 more crashes than AFLFast on `pdfimages`. When combining MOPT with VUzzer, MOPT-VUzzer discovers more unique crashes than VUzzer on `mp42aac`, `mp3gain`, `sam2p` and `mpg321`. As a result, MOPT cannot only be combined with state-of-the-art fuzzers like AFLFast, but also be compatible with the different fuzzers like VUzzer to improve the fuzzing performance.
- MOPT-based fuzzers can explore more unique paths than their counterparts. For instance, MOPT-AFLFast-tmp discovers 2,156 more paths than AFLFast on `mp42aac`; MOPT-AFLFast-ever finds 14,777 more than AFLFast on `pdfimages`. MOPT-VUzzer has a better coverage performance than VUzzer on `mp3gain`. Overall, MOPT can help the mutation-based fuzzers discover more unique paths.
- MOPT-AFL has an outstanding performance in comparison to state-of-the-art fuzzers. MOPT-AFL outperforms AFLFast with a significant advantage on all the programs except `mp42aac`. For instance, MOPT-AFL-tmp and MOPT-AFL-tmp discover 85 and 43 more unique crashes than AFLFast on `tiff2bw`. Furthermore, MOPT-AFL-tmp and MOPT-AFL-ever find dozens of times more unique crashes than VUzzer on most programs.

5.8 Evaluation on LAVA-M

Recently, the LAVA-M dataset is proposed as one of the standard benchmarks to examine the performance of fuzzers [28]. It has 4 target programs, each of which contains the listed and unlisted bugs. The authors provide the test cases that can trigger the listed bugs. However, no test cases were provided for the unlisted bugs, making them more difficult to be found. For completeness, we test AFL, MOPT-AFL-ever, AFLFast, MOPT-AFLFast-ever, VUzzer and MOPT-VUzzer on LAVA-M with the same initial seed set and the same settings as in Section 5.7, for 5 hours. Furthermore, we run MOPT-AFL-ever with Angora [9] and QSYM [29] parallelly to construct MOPT-Angora and MOPT-QSYM,

Table 6: The compatibility of the MOPT scheme.

		mp42aac	exiv2	mp3gain	tiff2bw	pdfimages	sam2p	mpg321
AFL	Unique crashes	135	34	178	4	23	36	10
	Unique paths	815	2,195	1,430	4,738	12,915	531	123
MOPT-AFL-tmp	Unique crashes	**209**	54	**262**	**85**	357	105	**236**
	Unique paths	1,660	2,980	**2,211**	**7,354**	22,661	1,967	1,054
MOPT-AFL-ever	Unique crashes	199	**66**	**262**	43	**471**	**329**	229
	Unique paths	**1,730**	**4,642**	2,206	7,295	**26,669**	**3,418**	**1,162**
AFLFast	Unique crashes	210	0	171	0	18	37	8
	Unique paths	1,233	159	1,383	5,114	12,022	603	122
MOPT-AFLFast-tmp	Unique crashes	**393**	51	**264**	5	292	**196**	**230**
	Unique paths	**3,389**	2,675	2,017	7,012	24,164	2,587	**1,208**
MOPT-AFLFast-ever	Unique crashes	384	**58**	259	**18**	**345**	114	30
	Unique paths	2,951	**2,887**	**2,102**	**7,642**	**26,799**	**2,623**	160
VUzzer	Unique crashes	12	0	54,500	0	0	13	3,598
	Unique paths	12%	9%	50%	13%	25%	18%	18%
MOPT-VUzzer	Unique crashes	**16**	0	**56,109**	0	0	**16**	**3,615**
	Unique paths	12%	9%	**51%**	13%	25%	18%	18%

Table 7: Evaluation on LAVA-M. The incremental number is the number of the discovered unlisted bugs.

Program	Listed bugs	Unlisted bugs	AFL Bugs	MOPT-AFL-ever Bugs	AFLFast Bugs	MOPT-AFLFast-ever Bugs	VUzzer Bugs	MOPT-VUzzer Bugs	AFL-Angora Bugs	MOPT-Angora Bugs	AFL-QSYM Bugs	MOPT-QSYM Bugs
base64	44	4	4	39	7	36	14	17	44(+2)	44(+3)	24	44(+4)
md5sum	57	4	2	23	1	18	38	41	57(+4)	57(+4)	57(+1)	57(+1)
uniq	28	1	5	27	7	15	22	24	26	28(+1)	1	18
who	2,136	381	1	5	2	6	15	23	1,622(+65)	2,069(+145)	312(+46)	774(+70)

run AFL with them parallelly to construct AFL-Angora and AFL-QSYM, and evaluate them on the LAVA-M dataset under the same experiment settings. The results are shown in Table 7, from which we have the following conclusions.

• MOPT-based fuzzers significantly outperform their counterparts on LAVA-M. For instance, MOPT-AFL-ever finds 35 more listed bugs than AFL on `base64`. MOPT-VUzzer finds more listed bugs than VUzzer on all the four programs. Both MOPT-Angora and MOPT-QSYM find significantly more unique bugs on `who` compared to their counterparts. Thus, MOPT is effective in improving the performance of mutation-based fuzzers.

• *The fuzzers, which use symbolic execution or similar techniques, perform significantly better than others on LAVA-M.* MOPT again exhibits good compatibility and can be integrated with general mutation-based fuzzers. For instance, MOPT-Angora finds significantly more unique bugs than AFL-Angora, and MOPT improves the performance of AFL-QSYM in 3 cases. From Table 7, in addition to the compatibility, MOPT can find the unique bugs and paths which the symbolic execution fails to find.

6 Further Analysis

6.1 Steadiness Analysis

Following the guidance of [25] and to make our evaluation more comprehensive, we conduct three extra groups of evaluations in this subsection. In the following, we detail the seed selection process, the evaluation methodology, and the analysis of the results.

Evaluation methodology and setup. To provide statistical evidences of our improvements, we measure the performance of MOPT-AFL-ever, AFL, Angora [9] and VUzzer [6] on five programs including `mp3gain`, `pdfimages`, `objdump`, `jhead` and `infotocap` (the detail of each program is shown in Table 2). Each program is tested by each fuzzer for 24 hours, on a virtual machine configured with one CPU core of 2.40Ghz E5-2640 V4, 4.5GB RAM and the OS of 64-bit Ubuntu 16.04 LTS. To eliminate the effect of randomness, we run each testing for 30 times.

To investigate the influence of the initial seed set on the performance of MOPT, we consider using various initial seed sets in our experiments such as an empty seed, or the seeds with different coverage, which are widely used in previous works [1, 5, 9, 21].

In the first group of experiments, each program is fed with an empty seed, which is a text file containing a letter 'a'. In the second and third groups of experiments, each program is fed with 20 and 200 well-formed seed inputs, respectively. In the third group, Angora is skipped since it reports errors to fuzz `pdfimages` when given 200 seed PDF files.

To obtain the seed inputs, we first download more than necessary (e.g., 1,700) input files with correct formats from the Internet. For example, we download mp3 files from the music download websites. Then, we split the input files (of format PDF and mp3) into a reasonable size if they are too large. Further, we utilize AFL-cmin [16] to evaluate each input file's coverage, and remove the inputs that have redundant coverage. In the remaining input files, we randomly select 20 (i.e., for the second group) or 200 (i.e., for the third group) seeds for the corresponding objective program.

Evaluation metrics. We measure the widely adopted metrics, i.e., *number of unique crashes* and *number of unique*

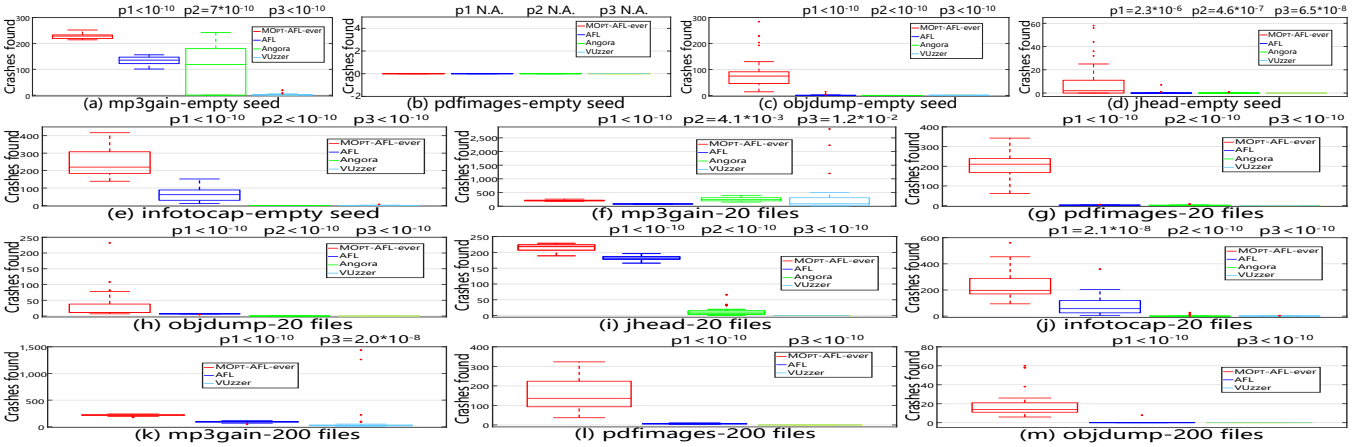


Figure 10: The boxplot generated by the number of unique crashes from 30 trials, which are found by AFL, MOPT-AFL-ever, Angora and VUzzer on five programs when fed with an empty seed, with 20 well-formed seed inputs and with 200 well-formed seed inputs. Y-axis: the number of unique crashes discovered in 24 hours.

bugs, to compare the performance of each fuzzer. To obtain the unique bugs, we recompile objective programs with the *AddressSanitizer* [24] instrumentation, and reevaluate the programs with the discovered crash inputs. If the top three source code locations of the stack trace provided by *AddressSanitizer* are unique, we consider the corresponding crash input triggers a unique bug of the objective program. This is a common way to find unique bugs in practice and has been used to calculate the stack hashing in [25].

Note that we do the statistical tests and use the p value [30] to measure the performance of the three fuzzers (suggested by [25]). In particular, $p1$ is the p value yielded from the difference between the performance of MOPT-AFL-ever and AFL, $p2$ is the p value yielded from the difference between the performance of MOPT-AFL-ever and Angora, and $p3$ is the p value generated from the difference between the performance of MOPT-AFL-ever and VUzzer.

We further validate the reliability of our p value analysis leveraging the Benjamini-Hochberg (BH) procedure [31]. For the details, please refer to the technical report [20].

Results and analysis. The number of unique crashes and unique bugs are shown in Fig. 10 and Fig. 11, respectively. From the results, we can learn the following facts.

• As shown in Fig. 10, among all the 13 evaluation settings, MOPT-AFL-ever discovers more unique crashes than the other fuzzers in 11 evaluations. In these 11 evaluations, $p1$, $p2$ and $p3$ are smaller than 10^{-5}, meaning that the distribution of the number of unique crashes discovered by MOPT-AFL-ever and the other fuzzers is widely different, which demonstrates a significant statistical evidence for MOPT's improvement. Therefore, according to the statistical results of 30 trials, MOPT-AFL-ever performs better than AFL, Angora and VUzzer in most cases.

• As for the number of discovered unique bugs, MOPT-AFL-ever still performs significantly better than AFL, Angora and VUzzer in most cases. For instance, the minimum number of unique bugs discovered by MOPT-AFL-ever among the 30 runs is more than the maximum number of that discovered by other fuzzers when fuzzing `objdump` and `jead` with 20 files as the initial seed set. Further, we find that both Angora and VUzzer discover more unique crashes but fewer unique bugs than MOPT-AFL-ever when fuzzing `mp3gain` with the 20 files. This indicates that their deduplication strategies do not work well in this evaluation.

• When using an empty seed as the initial seed set to fuzz `pdfimages`, all the fuzzers cannot discover any unique crash. The reason is that PDF files have complex structures. The test cases mutated from an empty seed are hard to generate such complex structures, which leads to the poor fuzzing performance. This reminds us the motivation of generation-based fuzzers and shows that: *although fuzzers like AFL may perform better with an empty seed, they cannot discover more crashes on the programs that require complex input formats when using an empty seed.*

6.2 Stepwise Analysis of MOPT Main Framework and Pacemaker Fuzzing Mode

To validate the effectiveness of MOPT main framework and the pacemaker fuzzing mode, we implement MOPT-AFL-off (that is based on MOPT-AFL-ever while disabling the pacemaker fuzzing mode) and AFL-ever (that is based on AFL and only implements the pacemaker fuzzing mode). We re-evaluate AFL, MOPT-AFL-off, AFL-ever and MOPT-AFL-ever on `pdfimages`, `w3m`, `objdump` and `infotocap` for 240 hours. The results are shown in Table 8.

MOPT *Main Framework (without Pacemaker Fuzzing Mode).* We can learn from Table 8 that MOPT-AFL-off discovers more crashes than AFL. For instance, on `w3m`, AFL cannot discover any crash in 240 hours, while MOPT-AFL-off discovers 74 unique crashes. Note that if without the pacemaker fuzzing mode, MOPT-AFL-off uses the havoc stage less frequently and iterates the selection distribution

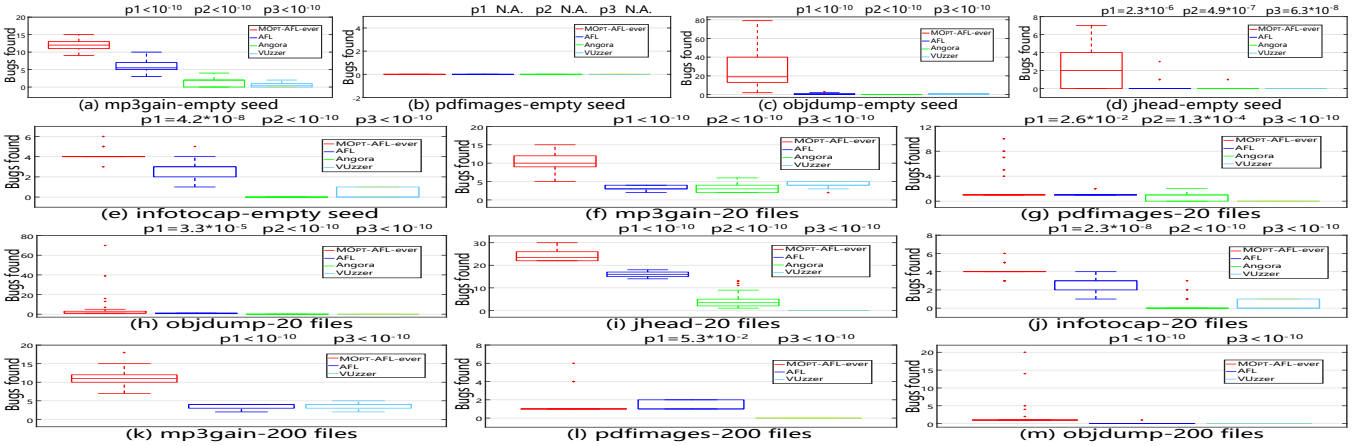


Figure 11: The boxplot generated by the number of unique bugs from 30 trials, which are found by AFL, MOPT-AFL-ever, Angora and VUzzer on five programs when fed with an empty seed, with 20 well-formed seed inputs and with 200 well-formed seed inputs. Y-axis: the number of unique bugs discovered in 24 hours.

Table 8: The results of AFL, MOPT-AFL-off, AFL-ever and MOPT-AFL-ever on 4 target programs.

Program	AFL		MOPT-AFL-off		AFL-ever		MOPT-AFL-ever	
	Unique crashes	Unique paths	Unique crashes	Unique paths	Unique crashes	Unique paths	Unique crashes	Unique paths
pdfimages	16	10,027	18	12,129	43	9,906	322	24,306
w3m	0	3,250	74	3,835	44	5,007	138	5,227
objdump	5	11,163	77	15,032	170	23,392	239	24,918
infotocap	86	3,179	97	4,112	436	6,808	687	7,109
total	107	27,619	266	35,108	693	45,113	1,386	61,560

more slowly, which limits the performance of the MOPT main framework. Comparing MOPT-AFL-ever with AFL-ever, we can learn that MOPT-AFL-ever has a better capability to explore unique crashes than AFL-ever. As for the coverage, MOPT-AFL-off discovers more unique paths than AFL, and the same situation applies for MOPT-AFL-ever and AFL-ever. As a conclusion, both two comparison groups demonstrate that the MOPT scheme without the pacemaker fuzzing mode can also improve the performance of AFL on exploring unique crashes and paths, but a better performance can be achieved if integrating the pacemaker fuzzing mode.

Pacemaker Fuzzing Mode. For discovering unique crashes, AFL-ever discovers 165 more unique crashes than AFL on `objdump`. Additionally, MOPT-AFL-ever finds 1,120 more unique crashes than MOPT-AFL-off on the 4 programs in total. As for the coverage, AFL-ever is better than AFL on `w3m`, `objdump` and `infotocap`. MOPT-AFL-ever finds nearly twice as many unique paths as MOPT-AFL-off on all the programs except `w3m`. As a conclusion, the experiments demonstrate that the pacemaker fuzzing mode can help fuzzers find much more unique crashes and paths.

In summary, *both the* MOPT *main framework and pacemaker fuzzing mode can improve the fuzzing performance significantly, while the combination of both parts would result in an even better performance (corresponding to* MOPT-*AFL-ever).* To further clarify this point, we use the number of unique crashes discovered by MOPT-AFL-ever as the baseline and observe the approximate fuzzing performance

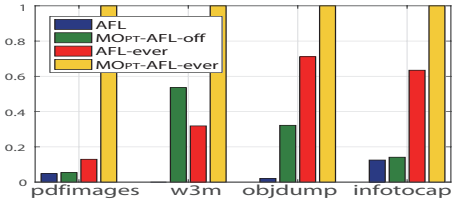


Figure 12: The ratio of the unique crashes discovered by 4 fuzzers, with MOPT-AFL-ever as the baseline.

of each part. The results are shown in Fig. 12.

From the results, the improvement of the pacemaker fuzzing mode is relatively limited for fuzzing; however, without the pacemaker fuzzing mode, MOPT cannot converge fast to the proper selection probability distribution, which on the other hand limits the fuzzing performance either. Nevertheless, *the performance can be significantly improved if we combine AFL with the complete* MOPT *scheme.*

6.3 Iteration Analysis of Selection Probability

To demonstrate the effectiveness of MOPT in obtaining the proper selection probability for the mutation operators, we record the probability of `bitflip 1/1`, `arith 8/8` and `interest 16/8` obtained by the particles in one swarm when using MOPT-AFL-ever to fuzz `w3m` and `pdfimages`. The results are shown in Fig. 13, from which we have the following observations.

• Different mutation operators have different proper selection probabilities on each program. Moreover, the proper

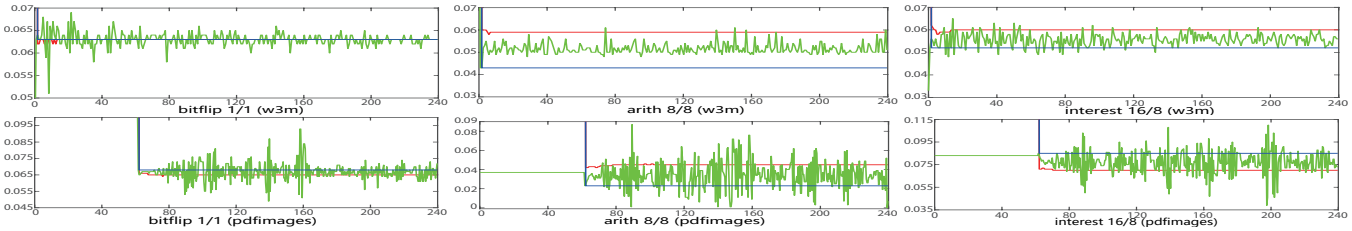


Figure 13: The probability when using MOPT-AFL-ever to fuzz `w3m` and `pdfimages`. X-axis: time (over 240 hours). Y-axis: the selection probability of the corresponding mutation operator. Green line: x_{now}. Red line: G_{best}. Blue line: L_{best}.

selection probability of one mutation operator varies with the objective programs. The results are consistent with our motivation that it is desired to dynamically determine the selection probability of operators during the fuzzing process.

• G_{best} and L_{best} quickly converge to the proper values. For instance, it only takes an hour for G_{best} and L_{best} to converge to the proper values when fuzzing `w3m`. When the proper values of G_{best} and L_{best} are the same, x_{now} will converge to this value and oscillate around. Otherwise, x_{now} will oscillate between G_{best} and L_{best} to explore whether there is a better selection probability.

• We can learn from Fig. 13 that MOPT iterates slowly at first and iterates fast later on `pdfimages`. The reasons are that (1) the deterministic stage is effective at finding interesting test cases in the early fuzzing time; (2) the fuzzer spends a long time on the deterministic stage of one test case when fuzzing `pdfimages`. When the efficiency of the deterministic stage decreases, i.e., it cannot discover any new crash or path for a long time, MOPT-AFL-ever enters the pacemaker fuzzing mode and will not use the deterministic stage again. Then the selection probability converges quickly and MOPT iterates fast. The results demonstrate that the design of the pacemaker fuzzing mode is reasonable and meaningful, which exploits the deterministic stage at first and avoids repeating its high computation when it is inefficient.

6.4 Overhead Analysis

In order to compare the execution efficiency of each fuzzer in Section 5.3, we collect the total execution times of each fuzzer, which are the times a fuzzer uses the generated test cases to test an objective program, within 240 hours. The results are shown in Table 9, from which we learn the following conclusions.

Although MOPT fuzzers take partial computing power to improve the mutation scheduler, the execution efficiency of MOPT-AFL-tmp and MOPT-AFL-ever is still comparable with AFL on most programs. In many cases, although the MOPT-AFL fuzzers test the objective programs for fewer times, they find much more crashes and paths than AFL.

Interestingly, MOPT-AFL can execute the tests faster than AFL on several programs. Moreover, the MOPT-AFL yields a better average execution efficiency on the 13 programs. We analyze the reasons as follows. The execution speed of each test case is different, and thus the test cases with slow execution speed will take more time consumption. When the fuzzing queue of AFL contains slow test cases, it will generate a number of test cases mutated from the slow test cases in the deterministic stage, which may also be executed slowly with a high probability and decrease AFL's execution efficiency. As for MOPT-AFL fuzzers, they will generate much fewer mutated cases from the slow test cases since they tend to disable the deterministic stage when it is not efficient. Therefore, MOPT-AFL fuzzers will spend much less time on the slow test cases, followed by yielding a high execution efficiency.

6.5 Long Term Parallel Experiments

We run the long term parallel experiments in order to verify the performance of MOPT-AFL in parallel for a long time. In each experiment, AFL, MOPT-AFL-tmp and MOPT-AFL-ever, are employed to fuzz `pdfimages` in parallel. Each experiment has three instances denoted by Fuzzer1, Fuzzer2 and Fuzzer3, with 20 carefully selected PDF files filtered from AFL-cmin [16] as the initial seed set. According to the parallel design of AFL and MOPT-AFL, the Fuzzer1 of AFL, MOPT-AFL-tmp and MOPT-AFL-ever will still perform the deterministic stage, while their Fuzzer2 and Fuzzer3 will disable it in the parallel experiments. Each experiment runs on a virtual machine configured with four CPU cores of 2.40Ghz E5-2640 V4, 4.5 GB RAM and the OS of 64-bit Ubuntu 16.04 LTS. The total CPU time of each experiment exceeds 70 days till the writing of this report and AFL, MOPT-AFL-tmp and MOPT-AFL-ever discover 1,778, 2,907 and 2,702 unique crashes, respectively.

The results are shown in Table 10, from which we can see that AFL's performance of discovering unique crashes is obviously inferior to MOPT-AFL's. Fuzzer1 of AFL enables the deterministic stage all the time and only discovers 11 unique crashes in more than 23 days, demonstrating the inefficiency of the deterministic stage. What's more, the performance of Fuzzer1 of MOPT-AFL-tmp is much better than that of MOPT-AFL-ever and AFL. We conjecture the reasons as follows. Since PDF files require the strict file format, there are many unique execution paths in `pdfimages` that contain strict magic byte checks. The operators, e.g., `bitflip 1/1`, in the deterministic stage are better at gen-

Table 9: The total execution times and executions per second of AFL, MOPT-AFL-tmp and MOPT-AFL-ever.

Program	AFL		MOPT-AFL-tmp			MOPT-AFL-ever		
	Total execution times	Executions per second	Total execution times	Executions per second	Increase	Total execution times	Executions per second	Increase
mp42aac	**127.1M**	**147.12**	126.8M	146.71	-0.28%	124.6M	144.26	-1.94%
exiv2	35.1M	40.58	27.6M	31.89	-21.41%	**46.5M**	**53.83**	+32.65%
mp3gain	**182.2M**	**210.90**	117.2M	135.60	-35.70%	121.4M	140.53	-33.38%
tiff2bw	**906.7M**	**1,049.43**	613.2M	709.74	-32.37%	623.4M	721.55	-31.24%
pdfimages	91.7M	106.17	88.8M	102.80	-3.17%	**108.5M**	**125.59**	+18.29%
sam2p	42.6M	49.34	**52.3M**	**60.58**	+22.78%	28.9M	33.47	-32.16%
avconv	**48.6M**	**56.27**	43.3M	50.08	-11.00%	42.0M	48.61	-13.61%
w3m	104.4M	120.78	123.2M	142.64	+18.10%	**204.6M**	**236.75**	+96.02%
objdump	383.7M	444.13	436.7M	505.42	+13.80%	**843.8M**	**976.58**	+119.89%
jhead	418.5M	484.41	1,372.6M	1,588.63	+227.95%	**1,476.1M**	**1,708.40**	+252.68%
mpg321	119.7M	138.52	158.1M	182.94	+32.07%	**165.2M**	**191.17**	+38.01%
infotocap	**218.1M**	**252.41**	157.1M	181.88	-27.94%	199.9M	231.36	-8.34%
podofopdfinfo	379.6M	439.37	**411.3M**	**476.05**	+8.35%	340.2M	393.80	-10.37%
average	254.8M	294.95	310.7M	359.59	+15.93%	**360.43M**	**417.16**	+35.54%

erating the correct magic bytes since fuzzers will flip every bit in the current test case to generate new test cases. In the later time, MOPT-AFL-tmp will enable the deterministic stage again while MOPT-AFL-ever will not. Thus MOPT-AFL-tmp is better at discovering unique paths containing magic byte checks than MOPT-AFL-ever. As for AFL, since it will go through the deterministic stage for all the test cases, it spends most time on this stage and discovers few unique crashes and paths. While MOPT-AFL-tmp will disable the deterministic stage when it cannot discover any interesting test case for a long time, after some time, it will re-enable the deterministic stage again and will perform the deterministic stage with the widely different test cases in the fuzzing queue. Therefore, MOPT-AFL-tmp can keep efficient fuzzing performance and can perform the deterministic stage on widely different test cases.

We can also observe from Table 10 that AFL's Fuzzer2 and Fuzzer3 find much more unique crashes than its Fuzzer1 without of the deterministic stage. The Fuzzer1 of MOPT-AFL-tmp and MOPT-AFL-ever finds much more crashes than AFL's Fuzzer1 and suppresses the performance of Fuzzer2 and Fuzzer3 in some way. Meantime, the Fuzzer2 and Fuzzer3 of both MOPT-AFL-tmp and MOPT-AFL-ever perform better than those of AFL. All these results demonstrate the improvement of the customized PSO algorithm.

7 Limitation and Discussion

In order to further analyze the compatibility of MOPT, we are eager to combine it with state-of-the-art fuzzers such as CollAFL [4] and Steelix [10] after they open-source their system code. By leveraging MOPT as an optimal strategy for selecting mutation operators, we believe the performance of these systems can be further enhanced.

In our evaluation, we consider 13 real world programs and several seed selection strategies, which are still a limited number of scenarios. In our evaluation, overall, MOPT-AFL discovers 31 vulnerabilities on `tiff2bw` and `sam2p` and 66 unreported CVEs on the other 11 programs. Furthermore, both MOPT-Angora and MOPT-QSYM perform better than previous methods on the benchmark dataset LAVA-M. Therefore, the proposed MOPT is promising to explore vulnerabilities for real world programs. Nevertheless, the performance advantage exhibited in our evaluation may not be applicable to all the possible programs and seeds. Our evaluation can be enhanced by further conducting more in-depth evaluation in large-scale. To make our evaluation more comprehensive, we are planning to perform a large-scale evaluation of MOPT using more real world programs and benchmarks in the future.

As a future work, it is interesting to investigate better mutation operators to further enhance the effectiveness of MOPT. Constructing a more comprehensive and representative benchmark dataset to systematically evaluate the performance of fuzzers is another interesting future work.

Table 10: The performance of three fuzzers in the long term parallel experiments when fuzzing `pdfimages`.

		Fuzzer1	Fuzzer2	Fuzzer3	Total
AFL	Unique crashes	11	871	896	1,778
	Unique paths	24,763	29,329	29,329	83,421
MOPT-AFL-tmp	Unique crashes	834	1,031	1,042	2,907
	Unique paths	30,098	31,600	31,520	93,218
MOPT-AFL-ever	Unique crashes	723	974	1,005	2,702
	Unique paths	28,047	30,910	30,966	89,923

8 Related Work

In this section, we summarize the existing fuzzing mechanisms and the related seed selection strategies.

Mutation-based fuzzing. AFL is one of the most well-recognized fuzzers because of its high-efficiency and ease of use [16]. Multiple efficient fuzzers were developed based on AFL [4, 5]. To improve fuzzing performance, some combined the mutation-based fuzzing with other bug detection technologies [13, 14, 15, 32]. Another method to improve mutation-based fuzzers is coverage-based fuzzing [6, 10, 11]. Li et al. proposed a vulnerability-oriented fuzzer named *V-Fuzz* that pays more attention to potentially vulnerable components [33]. Yun et al. presented a fast concolic

execution engine named *QSYM* to help fuzzers explore more bugs and paths [29]. By solving the path constraints without symbolic execution, *Angora* presented by Chen et al. can significantly increase the branch coverage of programs [9].

MOPT presented in our paper is a scheme of improving the test case mutation process and generating high-quality mutated test cases. Taking the advantage of its compatibility, it can be combined with most of the aforementioned fuzzers.

Although in this paper we focus on using MOPT to improve mutation-based fuzzers, it can also be implemented in other kinds of fuzzers, such as generation-based fuzzers and kernel fuzzers, if they have the issues to select proper operators to generate test cases. MOPT can also be combined with most existing seed selection strategies since they can provide better initial seed sets for fuzzers. We briefly introduce the state-of-the-art related works in these area as follows.

Generation-based fuzzing. Generation-based fuzzers focus on the programs that require the test cases with specific input formats [34, 35]. Recently, Wang et al. presented a novel data-driven seed generation approach named *Skyfire* to generate interesting test cases for XML and XSL [1]. Godefroid et al. presented a RNN-based machine learning technique to automatically generate a grammar for the test cases with complex input formats [36].

Other fuzzing strategies. Several works presented effective kernel fuzzers [37, 38]. Xu et al. [12] implemented three new operating primitives to benefit large-scale fuzzing and cloud-based fuzzing services. You et al. presented *SemFuzz* to learn from vulnerability-related texts and automatically generate Proof-of-Concept (PoC) exploits [39]. Petsios et al. proposed *SlowFuzz* to trigger algorithmic complexity vulnerabilities [22]. Klees et al. performed extensive experiments and proposed several guidelines to improve the experimental evaluations for fuzzing [25]. Some works proposed state-of-the-art directed greybox fuzzers to rapidly reach the target program locations [21, 40]. Recently, several works [7], [8] employ the reinforcement learning algorithms as the mutation schedulers and propose their fuzzing frameworks, respectively. However, the performance improvement of these methods is limited based on their experimental results.

Seed selection strategies. Several works focused on how to select a better seed set [2, 3]. Nichols et al. showed that using the generated files of GAN to reinitialize AFL can find more unique paths of `ethkey` [41]. Lyu et al. presented *SmartSeed* to leverage machine learning algorithms to generate high-quality seed files for different input formats [42].

9 Conclusion

We first studied the issues of existing mutation-based fuzzers which employ the uniform distribution for selecting mutation operators. To overcome these issues, we presented a mutation scheduling scheme, named MOPT, based on Particle Swarm Optimization (PSO). By using MOPT to search the optimal selection distribution for mutation operators and leveraging the pacemaker fuzzing mode to further accelerate the convergence speed of searching, MOPT can efficiently and effectively determine the proper distribution for selecting mutation operators. Our evaluation on 13 real-world applications demonstrated that MOPT-based fuzzers can significantly outperform the state-of-the-art fuzzers such as AFL, AFLFast and VUzzer in most cases. We also conducted systematic analysis to demonstrate the rationality, compatibility, low cost characteristic and steadiness of MOPT. Our fuzzers found 81 security CVEs on 11 real world programs, of which 66 are the newly reported CVEs. Overall, MOPT can serve as a key enabler for mutation-based fuzzers in discovering software vulnerabilities, crashes and program paths.

Acknowledgments

We sincerely appreciate the shepherding from Adam Doupé. We would also like to thank the anonymous reviewers for their valuable comments and input to improve our paper. This work was partly supported by NSFC under No. 61772466, the Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars under No. LR19F020003, the Provincial Key Research and Development Program of Zhejiang, China under No. 2017C01055, and the Alibaba-ZJU Joint Research Institute of Frontier Technologies. Chao Zhang's work was partly supported by the NSFC under No. 61772308 and U1736209. Wei-Han Lee's work is partly sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence under Agreement Number W911NF-16-3-0001.

References

[1] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *S&P*, 2017.

[2] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," in *CCS*, 2013.

[3] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing." in *USENIX*, 2014.

[4] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *S&P*, 2018.

[5] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *CCS*, 2016.

[6] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *NDSS*, 2017.

[7] K. Böttinger, P. Godefroid, and R. Singh, "Deep reinforcement fuzzing," *arXiv preprint arXiv:1801.04589*, 2018.

[8] W. Drozd and M. D. Wagner, "Fuzzergym: A competitive framework for fuzzing and learning," *arXiv preprint arXiv:1807.07490*, 2018.

[9] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *S&P*, 2018.

[10] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: program-state based binary fuzzing," in *FSE*, 2017.

[11] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," in *S&P*, 2018.

[12] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing new operating primitives to improve fuzzing performance," in *CCS*, 2017.

[13] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: a guided fuzzer to find buffer boundary violations." in *USENIX*, 2013.

[14] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *S&P*, 2015.

[15] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, 2016.

[16] "American Fuzzy Lop," http://lcamtuf.coredump.cx/afl/.

[17] K. Serebryany, "Continuous fuzzing with libfuzzer and addresssanitizer," in *SecDev*, 2016.

[18] R. Swiecki, "Honggfuzz," *Available online at: http://code. google. com/p/honggfuzz*, 2016.

[19] R. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," in *MHS*, 1995.

[20] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song and R. Beyah, "MOPT: Optimized Mutation Scheduling for Fuzzers, Technical Report," https://github.com/puppet-meteor/MOpt-AFL.

[21] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *CCS*, 2017.

[22] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *CCS*, 2017.

[23] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "Nezha: Efficient domain-independent differential testing," in *S&P*, 2017.

[24] "AddressSanitizer," http://clang.llvm.org/docs/AddressSanitizer.html.

[25] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *CCS*, 2018.

[26] "Common Vulnerability Scoring System (CVSS)," https://www.first.org/cvss.

[27] "Cve details," https://www.cvedetails.com/.

[28] B. Dolangavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in *S&P*, 2016.

[29] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "Qsym: A practical concolic execution engine tailored for hybrid fuzzing," in *USENIX*, 2018.

[30] "p value," https://en.wikipedia.org/wiki/P-value.

[31] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: a practical and powerful approach to multiple testing," *J R STAT SOC B*, 1995.

[32] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *S&P*, 2010.

[33] Y. Li, S. Ji, C. Lyu, Y. Chen, J. Chen, Q. Gu, and C. Wu, "V-fuzz: Vulnerability-oriented evolutionary fuzzing," *arXiv preprint arXiv:1901.01142*, 2019.

[34] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments." in *USENIX*, 2012.

[35] K. Dewey, J. Roesch, and B. Hardekopf, "Language fuzzing using constraint logic programming," in *ASE*, 2014.

[36] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *ASE*, 2017.

[37] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: interface aware fuzzing for kernel drivers," in *CCS*, 2017.

[38] H. Han and S. K. Cha, "Imf: Inferred model-based fuzzer," in *CCS*, 2017.

[39] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang, "Semfuzz: Semantics-based automatic generation of proof-of-concept exploits," in *CCS*, 2017.

[40] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *CCS*, 2018.

[41] N. Nichols, M. Raugas, R. Jasper, and N. Hilliard, "Faster fuzzing: Reinitialization with deep neural models," *arXiv preprint arXiv:1711.02807*, 2017.

[42] C. Lyu, S. Ji, Y. Li, J. Zhou, J. Chen, P. Zhou, and J. Chen, "Smartseed: Smart seed generation for efficient fuzzing," *arXiv preprint arXiv:1807.02606*, 2018.

EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers

Yuanliang Chen[1], Yu Jiang[1]*, Fuchen Ma[1], Jie Liang[1], Mingzhe Wang[1], Chijin Zhou[1], Xun Jiao[2], Zhuo Su[1]
[1]*School of Software, Tsinghua University, KLISS*
[2]*Department of Electrical and Computer Engineering, Villanova University*

Abstract

Fuzzing is widely used for vulnerability detection. There are various kinds of fuzzers with different fuzzing strategies, and most of them perform well on their targets. However, in industrial practice, it is found that the performance of those well-designed fuzzing strategies is challenged by the complexity and diversity of real-world applications. In this paper, we systematically study an ensemble fuzzing approach. First, we define the diversity of base fuzzers in three heuristics: diversity of coverage information granularity, diversity of input generation strategy and diversity of seed selection and mutation strategy. Based on those heuristics, we choose several of the most recent base fuzzers that are as diverse as possible, and propose a globally asynchronous and locally synchronous (GALS) based seed synchronization mechanism to seamlessly ensemble those base fuzzers and obtain better performance. For evaluation, we implement `EnFuzz` based on several widely used fuzzers such as QSYM and FairFuzz, and then we test them on LAVA-M and Google's fuzzing-test-suite, which consists of 24 widely used real-world applications. This experiment indicates that, under the same constraints for resources, these base fuzzers perform differently on different applications, while EnFuzz always outperforms other fuzzers in terms of path coverage, branch coverage and bug discovery. Furthermore, `EnFuzz` found 60 new vulnerabilities in several well-fuzzed projects such as libpng and libjpeg, and 44 new CVEs were assigned.

1 Introduction

Fuzzing is one of the most popular software testing techniques for bug and vulnerability detection. There are many fuzzers for academic and industrial usage. The key idea of fuzzing is to generate plenty of inputs to execute the target application and monitor for any anomalies. While each fuzzer develops its own specific fuzzing strategy to generate inputs, there are in general two main types of strategies. One is a generation-based strategy which uses the specification of input format, e.g. grammar, to generate complex inputs. For example, IFuzzer [33] takes a context-free grammar as specification to generate parse trees for code fragments. Radamsa [22] reads sample files of valid data and generates interesting different outputs from them. The other main strategy is a mutation-based strategy. This approach generates new inputs by mutating the existing seeds (good inputs contributing to improving the coverage). Recently, mutation-based fuzzers are proposed to use coverage information of target programs to further improve effectiveness for bug detection. For example, libFuzzer [10] mutates seeds by utilizing the SanitizerCoverage [11] instrumentation to track block coverage, while AFL [39] mutates seeds by using static instrumentation to track edge coverage.

Based on the above mentioned two fuzzers, researchers have performed many optimizations. For example, AFLFast [16] improves the fuzzing strategy of AFL by selecting seeds that exercise low-frequency paths for additional mutations, and FairFuzz [26] optimizes AFL's mutation algorithm to prioritize seeds that hit rare branches. AFLGo [15] assigns more mutation times to the seeds closer to target locations. QSYM [38] uses a practical concolic execution engine to solve complex branches of AFL. All of these optimized fuzzers outperform AFL on their target applications and have already detected a large number of software bugs and security vulnerabilities.

However, when we apply these optimized fuzzers to some real-world applications, these fuzzing strategies are inconsistent in their performance, their effectiveness on different applications varies accordingly. For example, in our evaluation on 24 real-world applications, AFLFast and FairFuzz perform better than AFL on 19 applications, while AFL performs better on the other 5 applications. Compared with AFL, libFuzzer performs better on 17 applications but worse on the other 7 applications. For the parallel mode of fuzzing which is widely-used in industry, AFLFast and FairFuzz only detected 73.5% and 88.2% of the unique bugs of AFL. These results show that the performance of existing fuzzers is challenged by the complexity and diversity of real-world applications. For a given real-world application, we cannot evaluate which fuzzer is better unless we spend significant time analyzing them or running each of these fuzzers one by one. This would waste a lot of human and computing resources [25]. This indicates that many of the current fuzzing strategies have a lack of robustness — the property of being strong and stable consistently in constitution. For industrial practice, more robust fuzzing strategies are desired when applied across a large number of different applications.

In this paper, we systematically study the performance of an ensemble fuzzing approach. First, we define the diversity of base fuzzers focusing on three heuristics: diversity of coverage information granularity, diversity of input generation strategy, as well as diversity of seed mutation and selection strategy. Then, we implement an ensemble architecture with a global asynchronous and local synchronous(GALS) based seed synchronization mechanism to integrate these base fuzzers effectively. To enhance cooperation among different base fuzzers, the mechanism synchronizes interesting seeds(i.e., test cases covering new paths or triggering new crashes) periodically to all fuzzers running on the same target application. At the same time, it maintains a global coverage map to help collect those interesting seeds asynchronously from each base fuzzer.

For evaluation, we implement a prototype of `EnFuzz`, based on several high-performance base fuzzers, including AFL, AFLFast, FairFuzz, QSYM, libFuzzer and Radamsa. All fuzzers are repeatedly tested on two widely used benchmarks — LAVA-M and Google's fuzzer-test-suite, following the kernel rules of evaluating fuzzing guideline [25]. The average number of paths executed, branches covered and unique crashes discovered are used as metrics. The results demonstrate that, with the same resource usage, the base fuzzers perform differently on different applications, while `EnFuzz` consistently and effectively improves the fuzzing performance. For example, there are many cases where the original AFL performs better on some real-world applications than the two optimized fuzzers FairFuzz and AFLFast. In all cases, the ensemble fuzzing always outperforms all other base fuzzers.

Specifically, on Google's fuzzer-test-suite consisting of real-world applications with a code base of 80K-220K LOCs, compared with AFL, AFLFast, FairFuzz, QSYM, libFuzzer and Radamsa, `EnFuzz` discovers 76.4%, 140%, 100%, 81.8%, 66.7% and 93.5% more unique bugs, executes 42.4%, 61.2%, 45.8%, 66.4%, 29.5% and 44.2% more paths and covers 15.5%, 17.8%, 12.9%, 26.1%, 19.9% and 14.8% more branches respectively. For the result on LAVA-M consisting of applications with a code base of 2K-4K LOCs, it outperforms each base fuzzer as well. For further evaluation on more widely used and several well-fuzzed open-source projects such as Libpng and jpeg, `EnFuzz` finds 60 new real vulnerabilities, 44 of which are security-critical vulnerabilities and registered as new CVEs. However, other base fuzzers only detect 35 new vulnerabilities at most.

This paper makes the following main contributions:

1. While many earlier works have mentioned the possibility of using ensemble fuzzing, we are among the first to systematically investigate the practical ensemble fuzzing strategies and the effectiveness of ensemble fuzzing of various fuzzers. We evaluate the performance of typical fuzzers through a detailed empirical study. We define the diversity of base fuzzers and study the effects of diversity on their performance.

2. We implement a concrete ensemble approach with seed synchronization to improve the performance of existing fuzzers. `EnFuzz` shows a more robust fuzzing practice across diverse real world applications. The prototype[1] is also scalable and open-source so as to integrate other fuzzers.

3. We apply `EnFuzz` to fuzz several well-fuzzed projects such as libpng and libjpeg from GitHub, and several commercial products such as `libiec61850` from Cisco. Within 24 hours, 60 new security vulnerabilities were found and 44 new CVEs were assigned, while other base fuzzers only detected 35 new vulnerabilities at most. `EnFuzz` has already been deployed in industrial practice, and more new CVEs are being reported[1].

The rest of this paper is organized as follows: Section 2 introduces related work. Section 3 illustrates ensemble fuzzing by a simple example. Section 4 elaborates ensemble fuzzing, including the base fuzzer selection and ensemble architecture design. Section 5 presents the implementation and evaluation of `EnFuzz`. Section 6 discusses the potential threats of `EnFuzz`, and we conclude in section 7. The appendix shows some empirical evaluations and observations.

2 Related Work

Here below, we introduce the work related to generation-based fuzzing, mutation-based fuzzing, fuzzing in practice and the main differences between these projects. After that we summarize the inspirations and introduce our work.

2.1 Generation-based Fuzzing

Generation-based fuzzing generates a massive number of test cases according to the specification of input format, e.g. a grammar. To fuzz the target applications that require inputs in complex format, the specifications used are crucial. There are many types of specifications. Input model and context-free grammar are the two most common types. Model-based fuzzers [1, 20, 34] follow a model of protocol. Hence, they are able to find more complex bugs by creating complex interactions with the target applications. Peach [20] is one of the most popular model-based fuzzers with both generation and mutation abilities. It develops two key models: the data model determines the format of complex inputs and the state model describes the concrete method for cooperating with fuzzing targets. By integrating fuzzing with models of data and state, Peach works effectively. Skyfire [34] first learns a context-sensitive grammar model, and then it generates massive inputs based on this model.

Some other popular fuzzers [21, 24, 31, 33, 37] generate inputs based on context free grammar. P Godefroid [21] enhances the whitebox fuzzing of complex structured-input applications by using symbolic execution, which directly generates grammar-based constraints whose satisfiability is examined using a custom grammar-based constraint solver. Csmith [37] is designed for fuzzing C-compilers. It generates plenty of random C programs in the C99 standard as the inputs. This tool can be used to generate C programs exploring a typical combination of C-language features while

[1] https://github.com/enfuzz/enfuzz

being free of undefined and unspecified behaviors. LAVA [31] generates effective test suites for the Java virtual machine by specifying production grammars. IFuzzer [33] first constructs parse trees based on a language's context-free grammar, then it generates new code fragments according to these parse trees. Radamsa [22] is a widely used generation-based fuzzer. It works by reading sample files of valid data and generating interestingly different outputs from them. Radamsa is an extreme "black-box" fuzzer, it needs no information about the program nor the format of the data. One can pair it with coverage analysis during testing to improve the quality of the sample set during a continuous fuzzing test.

2.2 Mutation-based Fuzzing

Mutation-based fuzzers [2, 17, 23] mutate existing test cases to generate new test cases without any input grammar or input model specification. Traditional mutation-based fuzzers such as zzuf [23] mutate the test cases by flipping random bits with a predefined ratio. In contrast, the mutation ratio of SYMFUZZ [17] is assigned dynamically. To detect bit dependencies of the input, it leverages white-box symbolic analysis on an execution trace, then it dynamically computes an optimal mutation ratio according to these dependencies. Furthermore, BFF [2] integrates machine learning with evolutionary computation techniques to reassign the mutation ratio dynamically.

Other popular AFL family tools [15, 16, 26, 39] apply various strategies to boost the fuzzing process. AFLFast [16] regards the process of target application as a Markov chain. A path-frequency based power schedule is responsible for computing the times of random mutation for each seed. As with AFLFast, AFLGo [15] also proposes a simulated annealing-based power schedule, which helps fuzz the target code. FairFuzz [26] mainly focuses on the mutation algorithm. It only mutates seeds that hit rare branches and it strives to ensure that the mutant seeds hit the rarest one. (Wen Xu et.al.) [36] propose several new primitives , speeding up AFL by 6.1 to 28.9 times. Unlike AFL family tools which track the hit count of each edge, libFuzzer [10] and honggfuzz [5] utilize the SanitizerCoverage instrumentation method provided by the Clang compiler. To track block coverage, they track the hit count of each block as a guide to mutate the seeds during fuzzing. SlowFuzz [30] prioritizes seeds that use more computer resources (e.g., CPU, memory and energy), increasing the probability of triggering algorithmic complexity vulnerabilities. Furthermore, some fuzzers use concolic executors for hybrid fuzzing. Both Driller [32] and QSYM use mutation-based fuzzers to avoid path exploration of symbolic execution, while concolic execution is selectively used to drive execution across the paths that are guarded by narrow-ranged constraints.

2.3 Cluster and Parallel Fuzzing in Industry

Fuzzing has become a popular vulnerability discovery solution in industry [28] and has already found a large number of dangerous bugs and security vulnerabilities across a wide range of systems so far. For example, Google's OSS-Fuzz [4] platform has found more than 1000 bugs in 5 months with thousands of virtual machines [9]. ClusterFuzz is the distributed fuzzing infrastructure behind OSS-Fuzz, and automatically executes libFuzzer powered fuzzer tests on scale [12, 13]. Initially built for fuzzing Chrome at scale, ClusterFuzz integrates multiple distributed libFuzzer processes, and performs effectively with corpus synchronization. ClusterFuzz mainly runs multiple identical instances of libFuzzer on distributed system for one target application. There is no diversity between these fuzzing instances.

In industrial practice, many existing fuzzers also provide a parallel mode, and they work well with some synchronization mechisms. For example, each instance of AFL in parallel mode will periodically re-scan the top-level sync directory for any test cases found by other fuzzers [3, 7]. libFuzzer in parallel will also use multiple fuzzing engines to exchange the corpora [6]. These parallel mode can effectively improve the performance of fuzzer. In fact, the parallel mode can be seen as a special example of ensemble fuzzing which uses multiple same base fuzzers. However, all these base fuzzers have a lack of diversity when using the same fuzzing strategy.

2.3.1 Main Differences

Unlike the previous works, we are not proposing a new concrete generation-based or mutation-based fuzzing strategy. Nor do we run multiple identical fuzzers with multiple cores or machines. Instead, inspired by the seed synchronization of ClusterFuzz and AFL in parallel mode, we systematically study the possibility of the ensemble fuzzing of diverse fuzzers mentioned in the earlier works. Referred to the kernel descriptions of the evaluating fuzzing guidelines [25], we empirically evaluate most state-of-the-art fuzzers, and identify some valuable results, especially for their performance variation across different real applications. To generate a stronger ensemble fuzzer, we choose multiple base fuzzers that are as diverse as possible based on three heuristics. We then implement an ensemble approach with global asynchronous and local synchronous based seed synchronization.

3 Motivating Example

To investigate the effectiveness of ensemble fuzzing, we use a simple example in Figure 1 which takes two strings as input, and crashes when one of the two strings is "Magic Str" and the other string is "Magic Num".

Many existing fuzzing strategies tend to be designed with certain preferences. Suppose that we have two different fuzzers $fuzzer_1$ and $fuzzer_2$: $fuzzer_1$ is good at solving the "Magic Str" problem, so it is better for reaching targets T1 and T3, but fails to reach targets T2 and T4. $fuzzer_2$ is good at solving the "Magic Num" problem so it is better for reaching targets T2 and T6, but fails to reach targets T1 and T5. If we use these two fuzzers separately, we can only cover one path and two branches. At the same time, if we use them simultaneously and ensemble their final fuzzing results without seed synchronization, we can cover two paths and four branches. However, if we ensemble these two fuzzers with some synchronization mechanisms throughout the fuzzing process, then, once $fuzzer_1$ reaches T1, it synchronizes the

```
void crash(char* A, char* B){
    if (A == "Magic Str"){              => T1
        if (B == "Magic Num") {
            bug();                      => T4
        }else{
            normal();                   => T3
        }
    }else if (A == "Magic Num"){        => T2
        if (B == "Magic Str"){
            bug();                      => T5
        }else{
            normal();                   => T6
        }
    }
}
```

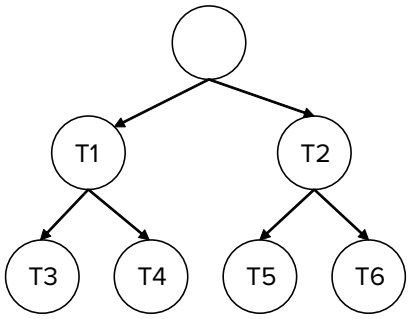


Figure 1: Motivating example of ensemble fuzzing with seed synchronization.

seed that can cover T1 to $fuzzer_2$. As a result, then, with the help of this synchronized seed, $fuzzer_2$ can also reach T1, and because of its ability to solve the "Magic Num" problem, $fuzzer_2$ can further reach T4. Similarly, with the help of the seed input synchronized by $fuzzer_2$, $fuzzer_1$ can also further reach T2 and T5. Accordingly, all four paths and all six branches can be covered through this ensemble approach.

Table 1: covered paths of each fuzzing option

Tool	T1-T3	T1-T4	T2-T5	T2-T6
fuzzer1	✓			
fuzzer2				✓
ensemble fuzzer1 and fuzzer2 without seed synchronization	✓			✓
ensemble fuzzer1 and fuzzer2 with seed synchronization	✓	✓	✓	✓

The ensemble approach in this motivating example works based on the following two hypotheses: (1) $fuzzer_1$ and $fuzzer_2$ expert in different domains; (2) the interesting seeds can be synchronized to all base fuzzers in a timely way. To satisfy the above hypotheses as much as possible, successful ensemble fuzzers rely on two key points: (1) the first is to select base fuzzers with great diversity (as yet to be well-defined); (2) the second is a concrete synchronization mechanism to enhance effective cooperation among those base fuzzers.

4 Ensemble Fuzzing

For an ensemble fuzzing, we need to construct a set of base fuzzers and seamlessly combine them to test the same target application together. The overview of this approach is presented in Figure 2. When a target application is prepared for fuzzing, we first choose several existing fuzzers as base fuzzers. The existing fuzzing strategies of any single fuzzer are usually designed with preferences. In real practice, these preferences vary greatly across different applications. They can be helpful in some applications, but may be less effective on other applications. Therefore, choosing base fuzzers with more diversity can lead to better ensemble performance. After the base fuzzer selection, we integrate fuzzers with the globally asynchronous and locally synchronous based seed synchronization mechanism so as to monitor the fuzzing status of these base fuzzers and share interesting seeds among them. Finally, we collect crash and coverage information and feed this information into the fuzzing report.

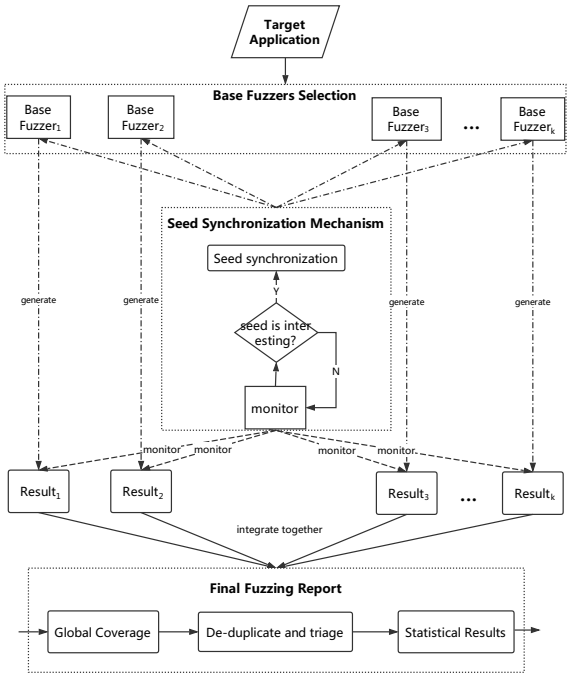


Figure 2: The overview of ensemble fuzzing consists of base fuzzer selection and ensemble architecture design. The base fuzzer selection contains the diversity heuristic definition, and the architecture design includes the seed synchronization mechanism as well as final fuzzing report.

4.1 Base Fuzzer Selection

The first step in ensemble fuzzing is to select a set of base fuzzers. These fuzzers can be generation-based fuzzers, e.g. Peach and Radamsa, or mutation-based fuzzers, e.g. libFuzzer and AFL. We can randomly choose some base fuzzers, but selecting base fuzzers with well-defined diversity improves the performance of an ensemble fuzzer.

We classify the diversity of base fuzzers according to three heuristics: seed mutation and selection strategy diversity, coverage information granularity diversity, inputs generation strategy diversity. The diversity heuristics are as follows:

1. Seed mutation and selection strategy based heuristic: the diversity of base fuzzers can be determined by the variability of seed mutation strategies and seed selection strategies. For example, AFLFast selects seeds that exercise low-frequency paths and mutates them more times, FairFuzz strives to ensure that the mutant seeds hit the rarest branches.

2. Coverage information granularity based heuristic: many base fuzzers determine interesting inputs by tracking different coverage information. Hence, the coverage information is critical, and different kinds of coverage granularity tracked by fuzzers enhances diversity. For example, libFuzzer guides seed mutation by tracking block coverage while AFL tracks edge coverage.

3. Input generation strategy based heuristic: fuzzers with different input generation strategies are suitable for different tasks. For example, generation-based fuzzers use the specification of input format to generate test cases, while the mutation-based fuzzers mutate initial seeds by tracking code coverage. So the generation-based fuzzers such as Radamsa perform better on complex format inputs and the mutation-based fuzzers such as AFL prefer complex logic processing.

Based on these three basic heuristics, we should be able to select a diverse set of base fuzzers with large diversity. It is our intuition that the diversity between the fuzzers following in two different heuristics is usually larger than the fuzzers that follows in the same heuristic. So, the diversity among the AFL family tools should be the least, while the diversity between Radamsa and AFL, between Libfuzzer and AFL, and between QSYM and AFL is should be greater. In this paper, we select base fuzzers manually based on the above heuristics. the base fuzzers will be dynamically selected according to the real-time coverage information.

4.2 Ensemble Architecture Design

After choosing base fuzzers, we need to implement a suitable architecture to integrate them together. As presented in Figure 2, inspired by the seed synchronization of AFL in parallel mode, one core mechanism is designed — the globally asynchronous and locally synchronous (GALS) based seed synchronization mechanism. The main idea is to identify the interesting seeds (seeds that can cover new paths or new branches or can detect new unique crashes) from different base fuzzers asynchronously and share those interesting seeds synchronously among all fuzzing processes.

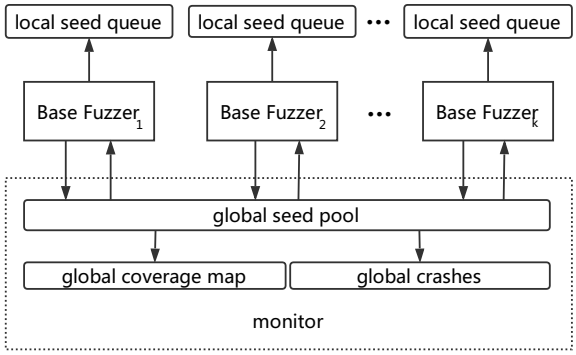


Figure 3: The data structure of global asynchronous and local synchronous based seed synchronization mechanism.

ALGORITHM 1: Action of local base fuzzer

```
   Input : Local seed pool of base fuzzer queue
1  repeat
2      foreach seed s of the queue do
3          s' = Mutate(s);
4          Cover = Run(s');
5          // if seeds s' causes new crash or have new
           // coverage, then store it in own seed pool and
           // push it to the global seed pool asynchronously;
6          if Cover.causeCrash() then
7              crashes.push(s');
8              queue.push(s');
9              GlobalSeedPool.push(s');
10         else if Cover.haveNewCoverage() then
11             queue.push(s');
12             GlobalSeedPool.push(s');
13         end
14     end
15 until timeout or abort-signal;
   Output : Global crashing seeds crashes
```

This seed synchronization mechanism employs a global-local style data structure as shown in Figure 3. The local seed queue is maintained by each base fuzzer, while the global pool is maintained by the monitor for sharing interesting seeds among all base fuzzers. In ensemble fuzzing, the union of these base fuzzers' results is needed to identify interesting seeds during the whole fuzzing process. Accordingly, the global coverage map is designed, and any new paths or new branches covered by the interesting seeds will be added into this global map. This global map can not only help decide which seeds to be synchronized, but also help de-duplicate and triage the results. Furthermore, to output the final fuzzing report after completing all fuzzing jobs, any interesting seeds which contribute to triggering unique crashes will be stored in the global crashes list.

First, let us take a look at the seed synchronization solution of the base fuzzer, which mainly describes how base fuzzers

contribute the interesting seeds asynchronously to the global pool. As presented in lines 2-4 of algorithm 1, for each single base fuzzer, it works with a local input seed queue and runs a traditional continuous fuzzing loop. It has three main steps: (1) Select input seeds from the queue, (2) mutate the selected input seeds to generate new candidate seeds, (3) run the target program with the candidate seeds, track the coverage and report vulnerabilities. Once the candidate seeds have new coverage or cause unique crashes, they will be regarded as interesting seeds and be pushed asynchronously into the global seed pool, as presented in lines 6-12.

ALGORITHM 2: Action of global monitor *sync*

```
   Input  : Base fuzzers list BaseFuzzers[]
            Initial seeds S
            Synchronization period period
1  // set up each base fuzzers ;
2  foreach base fuzzer f of the BaseFuzzers[] do
3  |   fuzzer.setup();
4  end
5  // set up thread monitor for monitoring ;
6  monintor.setup();
7  GlobalCover.initial();
8  GlobalSeedPool.initial();
9  GlobalSeedPool.push(S);
10 repeat
11 |   foreach seed s of the GlobalSeedPool do
12 |   |   // Skip synchronized seeds ;
13 |   |   if s.isSync() == False then
14 |   |   |   foreach base fuzzer f of the BaseFuzzers[] do
15 |   |   |   |   Cover = f.run(s) ;
16 |   |   |   |   // update the global coverage ;
17 |   |   |   |   newCover =
   |   |   |   |    (Cover ∪ GlobalCover) − GlobalCover ;
18 |   |   |   |   GlobalCover = Cover ∪ GlobalCover;
19 |   |   |   |   // synchronize the seed s to base fuzzer f ;
20 |   |   |   |   if Cover.causeCrash() and
   |   |   |   |    !newCover.isEmpty() then
21 |   |   |   |   |   crashes.push(s);
22 |   |   |   |   |   f.queue.push(s);
23 |   |   |   |   else if !newCover.isEmpty() then
24 |   |   |   |   |   f.queue.push(s);
25 |   |   |   |   else
26 |   |   |   |   |   continue;
27 |   |   |   |   end
28 |   |   |   end
29 |   |   else
30 |   |   |   continue;
31 |   |   end
32 |   |   s.setSync(True);
33 |   end
34 |   // waiting until next seed synchronization ;
35 |   sleep(period);
36 until timeout or abort-signal;
   Output : Crashing seeds crashes
```

Second, let us see the seed synchronization solution of the monitor process, which mainly describes how the monitor process synchronously dispatches the interesting seeds in the global pool to the local queue of each base fuzzer. When all base fuzzers are established, a thread named `monitor` will be created for monitoring the execution status of these fuzzing jobs, as in lines 2-6 of algorithm 2. It initializes the global coverage information to record the global fuzzing status of target applications by all the base fuzzer instances and then creates the global seed pool with the initial seeds, as in lines 7-9 of algorithm 2. It then runs a continuous periodically synchronizing loop — each base fuzzer will be synchronously dispatched with the interesting seeds from the global seed pool. Each base fuzzer will incorporate the seeds into its own local seed queue, once the seeds are deemed to be interesting seeds (seeds contribute to the coverage or crash and has not been generated by the local fuzzer), as in line 15-24 . To lower the overhead of seed synchronization, a thread `monitor` is designed to work periodically. Due to this globally asynchronous and locally synchronous based seed synchronization mechanism, base fuzzers cooperate effectively with each other as in the motivating example in Figure 1.

5 Evaluation

To present the effectiveness of ensemble fuzzing, we first implement several prototypes of ensemble fuzzer based on the state-of-the-art fuzzers. Then, we refer to some kernel descriptions of evaluating fuzzing guideline [25]. We conduct thorough evaluations repeatedly on LAVA-M and Google's fuzzer-test-suite, several well-fuzzed open-source projects from GitHub, and several commercial products from companies. Finally, according to the results, we answer the following three questions: (1) Can ensemble fuzzer perform better? (2) How do different base fuzzers affect Enfuzz? (3) How does Enfuzz perform on real-world applications

5.1 Ensemble Fuzzer Implementation

We implement ensemble fuzzing based on six state-of-the-art fuzzers, including three edge-coverage guided mutation-based fuzzers – AFL, AFLFast and FairFuzz, one block-coverage guided mutation-based fuzzer – libFuzzer, one generation-based fuzzer – Radamsa and one most recently hybrid fuzzer – QSYM. These are chosen as the base fuzzers for the following reasons (Note that `EnFuzz` is not limited to these six and other fuzzers can also be easily integrated, such as honggfuzz, ClusterFuzzer etc.):

- Easy integration. All the fuzzers are open-source and have their core algorithms implemented precisely. It is easy to integrate those existing fuzzers into our ensemble architecture. We do not have to implement them on our own, which eliminates any implementation errors or deviations that might be introduced by us.

- Fair comparison. All the fuzzers perform very well and are the latest and widely used fuzzers, as is seen by their comparisons with each other in prior literature, for example, QSYM outperforms similar fuzzers such as Angora [18] and VUzzer. We can evaluate their performance on real-world applications without modification.

- Diversity demonstration. All these fuzzers have different fuzzing strategies and reflect the diversity among correspondence with the three base diversity heuristics

Table 2: Diversity among these base fuzzers

Tool	diversity compared with AFL
AFLFast	Seed mutation and selection strategy based rule: the times of random mutation for each seed is computed by a Markov chain model. The seed selection strategy is different.
FairFuzz	Seed mutation and selection strategy based rule: only mutates seeds which hit rare branches and strives to ensure the mutant seeds hit the rarest one. The seed mutation strategy is different.
libFuzzer	Coverage information granularity based rule: libFuzzer mutates seeds by utilizing the SanitizerCoverage instrumentation, which supports tracking block coverage; while AFL uses static instrumentation with a bitmap to track edge coverage. The coverage information granularity is different.
Radamsa	Input generation strategy based rule: Radamsa is a widely used generation-based fuzzer which generates different inputs sample files of valid data. The input generation strategy is different.
QSYM	QSYM is a practical fast concolic execution engine tailored for hybrid fuzzing. It makes hybrid fuzzing scalable enough to test complex, real-world applications.

mentioned in section 4.1: coverage information granularity diversity, input generation strategy diversity, seed mutation and selection strategy diversity. The concrete diversity among these base fuzzers is listed in Table 2.

To demonstrate the performance of ensemble fuzzing and the influence of diversity among base fuzzers, five prototypes are developed. (1) `EnFuzz-A`, an ensemble fuzzer only based on AFL, AFLFast and FairFuzz. (2) `EnFuzz-Q`, an ensemble fuzzer based on AFL, AFLFast, FairFuzz and QSYM, a practical concolic execution engine is included. (3) `EnFuzz-L`, an ensemble fuzzer based on AFL, AFLFast, FairFuzz and libFuzzer, a block-coverage guided fuzzer is included. (4) `EnFuzz`, an ensemble fuzzer based on AFL, AFLFast, libFuzzer and Radamsa, a generation-based fuzzer is further added .(5) `EnFuzz`$^-$, with the ensemble of same base fuzzers (AFL, AFLFast and FairFuzz), but without the seed synchronization, to demonstrate the effectiveness of the global asynchronous and local synchronous based seed synchronization mechanism. During implementation of the proposed ensemble mechanism, we address the following challenges:

1) Standard Interface Encapsulating The interfaces of these fuzzers are different. For example, AFL family tools use the function `main`, but libFuzzer use a function `LLVMFuzzerTestOneInput`. Therefore, it is hard to ensemble them together. We design a standard interface to encapsulate the complexity of different fuzzing tools. This standard interface takes seeds from the file system, and writes the results back to the file system. All base fuzzers receive inputs and produce results through this standard interface, through which different base fuzzers can be ensembled easily.

2) libFuzzer Continuously Fuzzing The fuzzing engine of libFuzzer will be shut down when it finds a crash, while other tools continue fuzzing until manually closed. It is unfair to compare libFuzzer with other tools when the fuzzing time is different. The persistent mode of AFL is a good solution to this problem. Once AFL sets up, the fuzzer parent will `fork` and `execve` a new process to fuzz the target. When the target process crashes, the parent will collect the crash and resume the target, then the process simply loops back to the start. Inspired by the AFL persistent mode, we set up a thread named `Parent` to monitor the state of libFuzzer. Once it shuts down, `Parent` will resume the libFuzzer.

3) Bugs De-duplicating and Triaging We develop a tool for crash analysis. We compile all the target applications with AddressSanitizer, and test them with the crash samples. When the target applications crash, the coredump file, which consists of the recorded state of the working memory will be saved. Our tool first loads coredump files, then gathers the frames of each crash; finally, it identifies two crashes as identical if and only if the top frame is identical to the other frame. The method above is prone to underestimating bugs. For example, two occurrences of heap overflow may crash at the cleanup function at exit. However, the target program is instrumented with AddressSanitizer. As the program terminates immediately when memory safety problems occur, the top frame is always relevant to the real bug. In practice, the original duplicate unique crashes have been drastically de-duplicated to a humanly check-able number of unique bugs, usually without duplication. Even though there are some extreme cases that different top frames for one bug, the result can be further refined by manual crash analysis.

4) Seeds effectively Synchronizing The implementation of the seed synchronization mechanism: all base fuzzers have implemented the communication logic following the standard interface. Each base fuzzer will put interesting seeds into its own local seed pool, and the monitor thread `sync` will periodically make each single base fuzzer pull synchronized seeds from the global seed pool through a communication channel. This communication channel is implemented based on file system. A shorter period consumes too many resources, which leads to a decrease in fuzzing performance. A longer period will make seed synchronizing untimely, which also affects the performance. After multiple attempts with different values, it is found that the synchronization interval affects the performance at the beginning of fuzzing, while little impact was observed in the long term. The interval of 120s is identified with the fastest convergence.

5.2 Data and Environment Setup

Firstly, we evaluate ensemble fuzzing on LAVA-M [19], which consistis of four buggy programs, file, base64, md5sum and who. LAVA-M is a test suite that injects hard-to-find bugs in Linux utilities to evaluate bug-finding techniques. Thus the test is adequate for demonstrating the effectiveness of ensemble fuzzing. Furthermore, to reveal the practical performance of ensemble fuzzing, we also evaluate our work based on fuzzer-test-suite [8], a widely used benchmark from Google.

The test suite consists of popular open-source real-world applications. This benchmark is chosen to avoid the potential bias of the cases presented in literature, and for its great diversity, which helps demonstrate the performance variation of existing base fuzzers.

We refer to the kernel criteria and settings of evaluation from the fuzzing guidelines [25], and integrate the three widely used metrics from previous literature studies to compare the results on these real-world applications more fairly, including the number of paths, branches and unique bugs. To get unique bugs, we use crash's stack backtraces to deduplicate unique crashes, as mentioned in the previous subsection. The initial seeds for all experiments are the same. We use the test cases originally included in their applications or empty seed if such initial seeds do not exist.

The experiment on fuzzer-test-suite is conducted ten times in a 64-bit machine with 36 cores (Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz), 128GB of main memory, and Ubuntu 16.04 as the host OS with SMT enabled. Each binary is hardened by AddressSanitizer [11] to detect latent bugs. First, we run each base fuzzer for 24 hours with one CPU core in single mode. Next, since `EnFuzz-L`, `EnFuzz` and `EnFuzz-Q` need at least four CPU cores to ensemble these four base fuzzers, we also run each base fuzzer in parallel mode for 24 hours with four CPU cores. In particular, `EnFuzz-A` and `EnFuzz`$^-$ only ensembles three types of base fuzzers (AFL, AFLFast and FairFuzz). To use the same resources, we set up two AFL instances, one AFLFast instance and one FairFuzz instance. This experimental setup ensures that the computing resources usage of each ensemble fuzzer is the same as any base fuzzers running in parallel mode. While most metrics converged to similar values during multithreaded fuzzing. The variation of those statistical test results is small (between -5% 5%), we just use the averages in this paper.

5.3 Preliminary Evaluation on LAVA-M

We first evaluate ensemble fuzzing on LAVA-M, which has been used for testing other systems such as Angora, T-Fuzz and QSYM, and QSYM shows the best performance. We run `EnFuzz-Q` (which ensembles AFL, AFLFast, FairFuzz and QSYM) on the LAVA-M dataset. To demonstrate its effectiveness, we also run each base fuzzer using the same resources — four instances of AFL in parallel mode, four instances of AFLFast in parallel mode, four instances of FairFuzz in parallel mode, QSYM with four CPU cores used in parallel mode (two instances of concolic execution engine and two instances of AFL). To identify unique bugs, we used built-in bug identifiers provided by the LAVA project. The results are presented in Table 3, 4 and 5, which show the number of paths executed, branches covered and unique bugs detected by AFL, AFLFast, FairFuzz, QSYM, `EnFuzz-Q`.

From Tables 3, 4 and 5, we found that AFL, AFLFast and FairFuzz perform worse due to the complexity of their branches. The practical concolic execution engine helps QSYM solve complex branches and find significantly more bugs. The base code of the four applications in LAVA-M are small (2K-4K LOCs) and concolic execution could work well on them. However, real projects have code bases that easily reach 10k LOCs. Concolic execution might perform worse or even get hanged, as presented in the latter subsections. Furthermore, when we ensemble AFL, AFLFast, FairFuzz and QSYM together with the GALS based seed synchronization mechanism – `EnFuzz-Q` always performs the best in both coverage and bug detection. In total, compared with AFL, AFLFast, FairFuzz and QYSM, `EnFuzz-Q` executes 44%, 45%, 43% and 7.7% more paths, covers 195%, 215%, 194% and 5.8% more branches, and detectes 8314%, 19533%, 12989% and 0.68% more unique bugs respectively. From these preliminary statistics, we can determine that the performance of fuzzers can be improved by our ensemble approach.

Table 3: Number of paths covered by AFL, AFLFast, FairFuzz, QSYM and `EnFuzz-Q` on LAVA-M.

Project	AFL	AFLFast	FairFuzz	QSYM	`EnFuzz-Q`
base64	1078	1065	1080	1643	**1794**
md5sum	589	589	601	1062	**1198**
who	4599	4585	4593	5621	**5986**
uniq	476	453	471	693	**731**
total	6742	6692	6745	9019	**9709**

Table 4: Number of branches covered by AFL, AFLFast, FairFuzz, QSYM and `EnFuzz-Q` on LAVA-M.

Project	AFL	AFLFast	FairFuzz	QSYM	`EnFuzz-Q`
base64	388	358	389	960	**993**
md5sum	230	208	241	2591	**2786**
who	813	791	811	1776	**1869**
uniq	1085	992	1079	1673	**1761**
total	2516	2349	2520	7000	**7409**

Table 5: Number of bugs found by AFL, AFLFast, FairFuzz, QSYM and `EnFuzz-Q` on LAVA-M.

Project	AFL	AFLFast	FairFuzz	QSYM	`EnFuzz-Q`
base64	1	1	0	41	**42**
md5sum	0	0	1	**57**	**57**
who	2	0	1	1047	**1053**
uniq	11	5	7	25	**26**
total	14	6	9	1170	**1178**

5.4 Evaluation on Google's fuzzer-test-suite

While LAVA-M is widely used, Google's fuzzer-test-suite is more practical with many more code lines and containing real-world bugs. To reveal the effectiveness of ensemble fuzzing, we run `EnFuzz` (which only ensembles AFL, AFLFast, LibFuzzer and Radamsa) on all of the 24 real-world applications of Google's fuzzer-test-suite for 24 hours 10 times. As a comparison, we also run each base fuzzer in parallel mode with four CPU cores used. To identify unique bugs, we used stack backtraces to deduplicate crashes. The results are presented

in Tables 6, 7 and 8, which shows the average number of paths executed, branches covered and unique bugs detected by AFL, AFLFast, FairFuzz, LibFuzzer, Radamsa, QSYM and EnFuzz respectively.

Table 6: Average number of paths covered by each tool on Google's fuzzer-test-suite for ten times.

Project	AFL	AFLFast	FairFuzz	LibFuzzer	Radamsa	QSYM	EnFuzz
boringssl	3286	2816	3393	5525	3430	2973	**7136**
c-ares	146	116	146	191	146	132	**253**
guetzli	3248	2550	1818	3844	3342	2981	**4508**
lcms	1682	1393	1491	1121	1416	1552	**2433**
libarchive	12842	10111	12594	22597	12953	11984	**31778**
libssh	110	102	110	362	110	149	**377**
libxml2	14888	13804	14498	28797	17360	13172	**35983**
openssl-1.0.1	3992	3501	3914	2298	3719	3880	**4552**
openssl-1.0.2	4090	3425	3956	2304	3328	3243	**4991**
openssl-1.1.0	4051	3992	4052	2638	3593	4012	**4801**
pcre2	79581	66894	71671	59616	78347	60348	**85386**
proj4	342	302	322	509	341	323	**709**
re2	12093	10863	12085	15682	12182	10492	**17155**
woff2	23	16	20	447	22	24	**1324**
freetype2	19086	18401	20655	25621	18609	17707	**27812**
harfbuzz	12398	11141	14381	16771	11021	12557	**16894**
json	1096	963	721	1081	1206	1184	**1298**
libjpeg	1805	1579	2482	1486	1632	1636	**2638**
libpng	582	568	587	586	547	606	**781**
llvm	8302	8640	9509	10169	8019	7040	**10935**
openthread	268	213	230	1429	266	365	**1506**
sqlite	298	322	294	580	413	300	**636**
vorbis	1484	1548	1593	1039	1381	1496	**1699**
wpantund	4914	5112	5691	4881	4891	4941	**5823**
Total	190607	168372	186213	209574	188274	163097	**271408**
Improvement	–	11% ↓	2% ↓	9% ↑	1% ↓	14%↓	**42%** ↑

The first six columns of Table 6 reveal the issue of the performance variation in those base fuzzers, as they perform variously on different applications. Comparing AFL family tools, AFL performs better than the other two optimized fuzzers on 14 applications. Compared with AFL, libFuzzer performs better on 15 applications, but worse on 9 applications. Radamsa performs better on 8 applications, but also worse on 16 applications. QSYM performs better on 9 applications, but also worse on 15 applications. Table 7 and Table 8 show similar results on branch coverage and bugs.

From Table 6, it is interesting to see that compared with those optimized fuzzers based on AFL (AFLFast, FairFuzz, Radamsa and QSYM), original AFL performs the best on 14 applications in parallel mode with 4 CPU cores. For the total number of paths executed, AFL performs the best and AFLFast performs the worst in parallel mode. While in single mode with one CPU core used, the situation is exactly the opposite, and the original AFL only performs the best on 5 applications, as presented in Table 14 of the appendix.

The reason for performance degradation of these optimizations in parallel mode is that their studies lack the consideration for synchronizing the additional guiding information. Take AFLFast for example, it models coverage-based fuzzing as Markov Chain, and the times of random mutation for each seed will be computed by a power scheduler. This strategy works well in single mode, but it would fail in parallel mode because the statistics of each fuzzer's scheduler are limited in current thread. Our evaluation demonstrates that many optimized fuzzing strategies could be useful in single mode, but fail in the parallel mode even if this is the mode widely used in industry practice. This experiment has been missing by many prior literature studies. A potential solution for this degradation is to synchronize the additional guiding information in their implementation, similar to the work presented in PAFL [27].

Table 7: Average number of branches covered by each tool on n Google's fuzzer-test-suite for ten times.

Project	AFL	AFLFast	FairFuzz	LibFuzzer	Radamsa	QSYM	EnFuzz
boringssl	3834	3635	3894	3863	3880	3680	**4108**
c-ares	**285**	276	**285**	202	**285**	**285**	**285**
guetzli	3022	2723	1514	**4016**	3177	3011	3644
lcms	3985	3681	3642	3015	2857	3731	**4169**
libarchive	10580	9267	8646	8635	11415	9416	**13949**
libssh	614	614	614	573	614	**636**	614
libxml2	15204	14845	14298	13346	19865	14747	**21899**
openssl-1.0.1	4011	3967	3996	3715	4117	4032	**4673**
openssl-1.0.2	4079	4004	4021	3923	4074	3892	**4216**
openssl-1.1.0	9125	9075	9123	8712	9017	9058	**9827**
pcre2	50558	48004	49430	36539	51881	36208	**53912**
proj4	267	267	267	798	267	261	**907**
re2	17918	17069	17360	16001	17312	16323	**19688**
woff2	120	120	120	2785	120	121	**3945**
freetype2	53339	52404	56653	57325	52715	48547	**58192**
harfbuzz	38163	36313	43077	39712	37959	38194	**44708**
json	7048	6622	5138	6583	7231	7169	**7339**
libjpeg	12345	11350	15688	10342	12009	11468	**17071**
libpng	4135	4393	4110	4003	3961	4085	**4696**
llvm	55003	56619	58306	57021	54312	48008	**62918**
openthread	3109	2959	2989	5421	3102	3634	**5579**
sqlite	2850	2847	2838	3123	3012	2853	**3216**
vorbis	12136	13524	13053	10032	11234	12849	**14318**
wpantund	40667	40867	41404	39816	40317	40556	**43217**
Total	352397	345445	360466	339501	354733	322764	**407090**
Improvement	–	1% ↓	2% ↓	3% ↑	0.6% ↓	8%↓	**16%** ↑

Table 8: Average number of unique bugs found by each tool on n Google's fuzzer-test-suite for ten times.

Project	AFL	AFLFast	FairFuzz	LibFuzzer	Radamsa	QSYM	EnFuzz
boringssl	0	0	0	**1**	0	0	**1**
c-ares	**3**	2	**3**	1	2	2	**3**
guetzli	0	0	0	**1**	0	0	**1**
lcms	1	1	1	**2**	1	1	**2**
libarchive	0	0	0	**1**	0	0	**1**
libssh	0	0	0	1	0	1	**2**
libxml2	1	1	1	**3**	2	1	**3**
openssl-1.0.1	3	2	3	2	2	3	**4**
openssl-1.0.2	5	4	4	1	5	5	**6**
openssl-1.1.0	5	5	5	3	4	5	**6**
pcre2	6	4	5	2	5	4	**8**
proj4	2	0	1	1	1	1	**3**
re2	1	0	1	1	0	1	**2**
woff2	1	0	0	**2**	1	1	1
freetype2	0	0	0	0	0	0	0
harfbuzz	0	0	**1**	**1**	0	0	**1**
json	2	1	0	1	**3**	2	**3**
libjpeg	0	0	0	0	0	0	0
libpng	0	0	0	0	0	0	0
llvm	1	1	**2**	**2**	1	1	**2**
openthread	0	0	0	**4**	0	0	**4**
sqlite	0	0	0	**3**	1	1	**3**
vorbis	3	**4**	3	3	3	**4**	**4**
wpantund	0	0	0	0	0	0	0
Total	34	25	30	37	31	33	**60**
Improvement	–	26% ↓	12% ↑	6% ↓	9% ↑	3%↓	**76%** ↑

From the fifth columns of Table 6 and Table 14, we find that compared with Radamsa in single mode, the improvement achieved by Radamsa is limited in parallel mode. There are two main reasons: (1) Too many useless inputs generated by Radamsa slow down the seed-sharing efficiency among all

instances of AFL. This seed-sharing mechanism does not exist in single mode. (2) Some interesting seeds can be created in parallel mode and shared among all instances of AFL. These seeds overlap with the inputs generated by Radamsa. So this improvement is limited in parallel mode.

For the EnFuzz which integrates AFL, AFLFast, libFuzzer and Radamsa as base fuzzers and, compared with AFL, AFLFast, FairFuzz, QSYM, LibFuzzer and Radamsa, it shows the strongest robustness and always performs the best. In total, it discovers 76.4%, 140%, 100%, 81.8%, 66.7% and 93.5% more unique bugs, executes 42.4%, 61.2%, 45.8%, 66.4%, 29.5% and 44.2% more paths and covers 15.5%, 17.8%, 12.9%, 26.1%, 19.9% and 14.8% more branches respectively. These statistics demonstrate that it helps mitigate performance variation and improves robustness and performance by the ensemble approach with globally asynchronous and locally synchronous seed synchronization mechanism.

5.5 Effects of Different Fuzzing Integration

To study the effects of the globally asynchronous and locally synchronous based seed synchronization mechanism, we conduct a comparative experiment on EnFuzz$^-$and EnFuzz-A, both ensemble the same base fuzzers (two AFL, one AFLFast, one FairFuzz) in parallel mode with four CPU cores. To study the effects of different base fuzzers on ensemble fuzzing, we also run EnFuzz-Q, EnFuzz-L and EnFuzz on Google's fuzzer-test-suite for 24 hours 10 times. To identify unique bugs, we used stack backtraces to deduplicate crashes. The results are presented in Tables 9, 10 and 11, which shows the average number of paths executed, branches covered and unique bugs detected by EnFuzz$^-$, EnFuzz-A, EnFuzz-Q, EnFuzz-L, and EnFuzz, respectively.

Compared with EnFuzz-A, EnFuzz$^-$ ensembles the same base fuzzers (AFL, AFLFast and FairFuzz), but does not implement the seed synchronization mechanism. EnFuzz$^-$ performs much worse on all applications. In total, it only covers 68.5% paths, 78.3% branches and detects 32.4% unique bugs of EnFuzz-A. These statistics demonstrate that the globally asynchronous and locally synchronous based seed synchronization mechanism is critical to the ensemble fuzzing.

Table 10: Average number of branches covered by each Enfuzz on Google's fuzzer-test-suite for ten times.

Project	EnFuzz$^-$	EnFuzz-A	EnFuzz-Q	EnFuzz-L	EnFuzz
boringssl	3210	3996	4013	4016	**4108**
c-ares	**285**	**285**	**285**	**285**	**285**
guetzli	2074	3316	3246	3531	**3644**
lcms	2872	4054	4152	4098	**4169**
libarchive	6092	12689	11793	13267	**13949**
libssh	613	614	**640**	614	614
libxml2	14428	17657	16932	21664	**21899**
openssl-1.0.1	3612	4194	4204	4538	**4673**
openssl-1.0.2	4037	4176	**4292**	4202	4216
openssl-1.1.0	8642	9371	9401	9680	**9827**
pcre2	32471	51801	52751	52267	**53912**
proj4	267	267	267	**907**	**907**
re2	16300	18070	18376	19323	**19688**
woff2	120	120	121	3939	**3945**
freetype2	49927	55952	54193	58018	**58192**
harfbuzz	33915	43301	43379	44419	**44708**
json	4918	7109	7146	7268	**7339**
libjpeg	9826	15997	15387	16984	**17071**
libpng	3816	4487	4502	4589	**4696**
llvm	49186	58681	58329	60104	**62918**
openthread	2739	3221	4015	5503	**5579**
sqlite	2318	2898	2971	3189	**3216**
vorbis	10328	13872	13993	14210	**14318**
wpantund	33749	41537	41663	43104	**43217**
Total	295745	377665	376051	399719	**407090**
Improvement	–	27% ↑	28% ↑	35% ↑	**38%** ↑

Table 9: Average number of paths covered by each Enfuzz on Google's fuzzer-test-suite for ten times.

Project	EnFuzz$^-$	EnFuzz-A	EnFuzz-Q	EnFuzz-L	EnFuzz
boringssl	2590	4058	3927	6782	**7136**
c-ares	149	167	159	251	**253**
guetzli	2066	3501	3472	4314	**4508**
lcms	1056	1846	1871	2253	**2433**
libarchive	4823	14563	14501	28531	**31778**
libssh	109	140	152	**377**	**377**
libxml2	11412	19928	18738	33940	**35983**
openssl-1.0.1	3496	4015	4095	4417	**4552**
openssl-1.0.2	3949	4976	**5012**	4983	4991
openssl-1.1.0	3850	4291	4383	4733	**4801**
pcre2	57721	81830	82642	84681	**85386**
proj4	362	393	399	708	**709**
re2	9053	13019	14453	17056	**17155**
woff2	19	25	24	1314	**1324**
freetype2	17692	22512	20134	26421	**27812**
harfbuzz	10438	14997	15019	16328	**16894**
json	648	1101	1183	1271	**1298**
libjpeg	1395	2501	2475	2588	**2638**
libpng	480	601	652	706	**781**
llvm	7953	9706	9668	10883	**10935**
openthread	197	281	743	1489	**1506**
sqlite	279	311	325	598	**636**
vorbis	928	1604	1639	1673	**1699**
wpantund	4521	5718	5731	5797	**5823**
Total	145186	212084	211397	262094	**271408**
Improvement	–	46% ↑	48% ↑	80% ↑	**87%** ↑

Table 11: Average number of bugs found by each Enfuzz on Google's fuzzer-test-suite for ten times.

Project	EnFuzz$^-$	EnFuzz-A	EnFuzz-Q	EnFuzz-L	EnFuzz
boringssl	0	0	0	**1**	**1**
c-ares	1	**3**	2	**3**	**3**
guetzli	0	0	**1**	**1**	**1**
lcms	0	1	1	**2**	**2**
libarchive	0	0	**1**	**1**	**1**
libssh	0	0	**2**	**2**	**2**
libxml2	1	1	1	2	**3**
openssl-1.0.1	0	3	3	**4**	**4**
openssl-1.0.2	3	5	5	5	**6**
openssl-1.1.0	2	5	5	**6**	**6**
pcre2	3	6	6	7	**8**
proj4	0	2	2	2	**3**
re2	0	1	1	2	**2**
woff2	0	**1**	**1**	**1**	**1**
freetype2	0	0	0	0	**0**
harfbuzz	0	**1**	**1**	**1**	**1**
json	1	2	2	2	**3**
libjpeg	0	0	0	0	**0**
libpng	0	0	0	0	**0**
llvm	0	1	1	**2**	**2**
openthread	0	0	1	3	**4**
sqlite	0	1	1	2	**3**
vorbis	1	**4**	**4**	**4**	**4**
wpantund	0	0	0	0	0
Total	12	37	41	53	**60**
Improvement	–	208% ↑	242% ↑	342% ↑	**400%** ↑

For EnFuzz-A, which ensembles AFL, AFLFast and FairFuzz as base fuzzers and implements the seed synchronization with global coverage map, compared with AFL, AFLFast and FairFuzz running in parallel mode with four CPU cores used (as shown in Table 6, Table 7 and Table 8), it always executes more paths and covers more branches on all applications. In total, it covers 11.3%, 25.9% and 13.9% more paths, achieves 7.2%, 9.3% and 4.8% more covered branches, and triggers 8.8%, 48% and 23% more unique bugs. It reveals that the robustness and performance can be improved even when the diversity of base fuzzers is small.

For the EnFuzz-Q which integrates AFL, AFLFast, FairFuzz and QYSM as base fuzzers, the results are shown in the fourth columns of Tables 9, 10 and 11. Compared with EnFuzz-A, EnFuzz-Q covers 1.1% more paths, executes 1.0% more branches and triggers 10.8% more unique bugs than EnFuzz-A. The improvement is significantly smaller on Google's fuzzer-test-suite than on LAVA-M.

The reason for performance degradation between experiments on LAVA-M and Google fuzzer-test-suite is that the base codes of the four applications (who, uniq, base64 and md5sum) in LAVA-M are small (2K-4K LOCs). The concolic execution engine works well on them, but usually performs the opposite or even hangs on real projects in fuzzer-test-suite whose code base easily reaches 100k LOCs.

For the EnFuzz-L which integrates AFL, AFLFast, FairFuzz and libFuzzer as base fuzzers, the results are presented in the seventh columns of Tables 9, 10 and 11. As mentioned in section A, the diversity among these base fuzzers is much larger than with EnFuzz-A. Compared with EnFuzz-A, EnFuzz-L always performs better on all target applications. In total, it covers 23.6% more paths, executes 5.8% more branches and triggers 42.4% more unique bugs than EnFuzz-A.

For the EnFuzz which integrates AFL, AFLFast, libFuzzer and Radamsa as base fuzzers, the diversity is the largest because they cover all three diversity heuristics. Compared with EnFuzz-L, it performs better and covers 3.6% more paths, executes 1.8% more branches and triggers 13.2% more unique bugs. Both EnFuzz and EnFuzz-L performs better than EnFuzz-Q. These statistics demonstrate that the more diversity among these base fuzzers, the better the ensemble fuzzer should perform. For real applications with a large code base, compared with hybrid conclic fuzzing or ensemble fuzzing with symbolic execution, the ensemble fuzzing without symbolic execution may perform better.

5.6 Fuzzing Real-World Applications

We apply EnFuzz to fuzz more real-world applications from GitHub and commercial products from Cisco, some of which are well-fuzzed projects such as the image processing library libpng and libjepg, the video processing library libwav, the IoT device communication protocol libiec61850 used in hundreds of thousands of cameras, etc. EnFuzz also performs well. Within 24 hours, besides the coverage improvements, EnFuzz finds 60 more unknown real bugs including 44 successfully registered as CVEs, as shown in Table 13. All of these new bugs and security vulnerabilities are detected in a 64-bit machine with 36 cores (Intel(R) Xeon(R) CPU E5-2630 v3@2.40GHz), 128GB of main memory, and Ubuntu 16.04 as the host OS.

Table 12: Unique previously unknown bugs detected by each tool within 24 hours on some real-world applications.

Project	AFL	AFLFast	FairFuzz	LibFuzzer	QSYM	EnFuzz
Bento4_mp4com	5	4	5	5	4	**6**
Bento4_mp4tag	5	4	4	5	4	**7**
bitmap	1	1	1	0	1	**2**
cmft	1	1	0	1	0	**2**
ffjpeg	1	1	1	0	1	**2**
flif	1	1	1	2	1	**3**
imageworsener	**1**	0	0	0	**1**	**1**
libjpeg-05-2018	3	3	3	4	3	**5**
libiec61850	3	2	2	1	2	**4**
libpng-1.6.34	2	1	1	1	2	**3**
libwav_wavgain	3	2	3	0	2	**5**
libwav_wavinfo	2	1	2	4	2	**5**
LuPng	1	1	1	3	1	**4**
pbc	5	5	6	7	6	**9**
pngwriter	1	1	1	1	**2**	**2**
total	35	28	31	34	32	**60**

As a comparison, we also run each tool on those real-world applications to detect unknown vulnerabilities. The results are presented in table 12. EnFuzz found all 60 unique bugs, while other tools only found a portion of these bugs. Compared with AFL, AFLFast, FairFuzz, LibFuzzer and QSYM, EnFuzz detected 71.4%, 114%, 93.5%, 76.4%, 87.5% more unique bugs respectively. The results demonstrate the effectiveness of EnFuzz in detecting real vulnerabilities in more general projects. For example, in the well-fuzzed projects libwav and libpng, we can still detect 13 more real bugs, 7 of which are assigned as CVEs. We give an analysis of the project libpng for a more detailed illustration. libpng is a widely used C library for reading and writing PNG image files. It has been fuzzed many times and is one of the projects in Google's OSS-Fuzz, which means it has been continually fuzzed by multiple fuzzers many times. But with EnFuzz, we detect three vulnerabilities, including one segmentation fault, one stack-buffer-overflow and one memory leak. The first two vulnerabilities were assigned as CVEs (CVE-2018-14047, CVE-2018-14550).

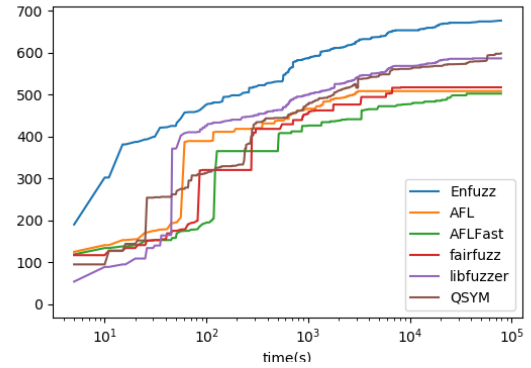
In particular, CVE-2018-14047 allows remote attackers to cause a segmentation fault via a crafted input. We analyze the vulnerability with AddressSanitizer and find it is a typical memory access violation. The problem is that in function png_free_data in line 564 of png.c, the info_ptr attempts to access an invalid area of memory. The error occurs in png_free_data during the free of text-related data with specifically crafted files, and causes reading of invalid or unknown memory, as show in Listing 1. The new vulnerabilities and CVEs in the IoT device communication protocol libiec6185 can also crash the service and have already been confirmed and repaired.

We also apply each base fuzzer (AFL, AFLFast, FairFuzz, libFuzzer and QSYM) to fuzz libpng separately, the above vulnerability is not detected. To trigger this bug, 6 function calls and 11 compares (2 for integer, 1 for boolean and 8 for

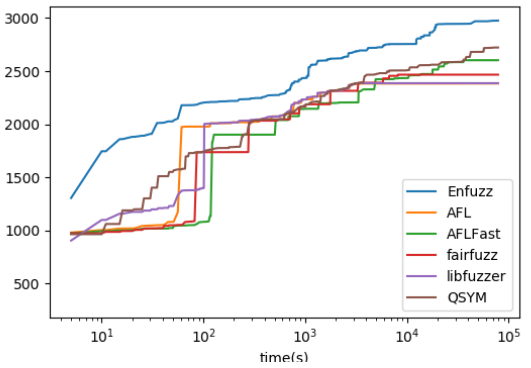
```
#ifdef PNG_TEXT_SUPPORTED
/* Free text item num or (if num == -1)
    all text items */
    if (info_ptr->text != NULL &&
        ((mask & PNG_FREE_TEXT) &
    info_ptr->free_me) != 0)
```

Listing 1: The error code of libpng for CVE-2018-14047

pointer) are required. It is difficult for other fuzzers to detect bugs in such deep paths without the seeds synchronization of EnFuzz. The performances of these fuzzers over time in libpng are presented in Figure 4. The results demonstrate that generalization and scalability limitations exist in these base fuzzers – the two optimized fuzzers AFLFast and FairFuzz perform worse than the original AFL for libpng, while EnFuzz performs the best. Furthermore, except for those evaluations on benchmarks and real projects, EnFuzz had already been deployed in industry practice, and more new CVEs were being continuously reported.



(a) Number of paths over time



(b) Number of branches over time

Figure 4: Performance of each fuzzer over time in libpng. Each fuzzer runs in four CPU cores for 24 hours.

Table 13: The 44 CVEs detected by EnFuzz in 24 hours.

Project	Count	CVE-2018-Number
Bento4_mp4com	6	14584, 14585, 14586, 14587, 14588, 14589
Bento4_mp4tag	6	13846, 13847, 13848, 14590, 14531, 14532
bitmap	1	17073
cmft	1	13833
ffjpeg	1	16781
flif	1	12109
imageworsener	1	16782
libjpeg-05-2018	4	11212, 11213, 11214, 11813
libiec61850	3	18834, 18937, 19093
libpng-1.6.34	2	14048, 14550
libwav_wavgain	2	14052, 14549
libwav_wavinfo	3	14049, 14050, 14051
LuPng	3	18581, 18582, 18583
pbc	9	14736, 14737, 14738, 14739, 14740, 14741, 14742, 14743, 14744
pngwriter	1	14047

6 Discussion

Based on benchmarks such as LAVA-M and Google's fuzzer-test-suite, and several real projects, we demonstrate that this ensemble fuzzing approach outperforms any base fuzzers. However, some limitations still threaten the performance of ensemble fuzzing. The representative limitations and the workarounds are discussed below.

The first potential threat is the insufficient and imprecise diversity of base fuzzers. Section 4.1 describes our base fuzzer selection, we propose three different heuristics to indicate diversity of base fuzzers, including diversity of coverage information granularity, diversity of input genera-tion strategy, and diversity of seed mutation selection strategy. According to these three heuristics, we select AFL, AFLFast, FairFuzz, libFuzzer, Radamsa and QSYM as the base fuzzers. Furthermore, we implement four prototypes of ensemble fuzzing and demonstrate that the greater the diversity of base fuzzers, the better the ensemble fuzzer performs. However, these three different heuristics of diversity may be insufficient. More diversity measures need to be proposed in future work. For example, initial seeds determine the initial direction of fuzzing and, thus, are significantly important for fuzzing, especially for mutation-based fuzzers. Some fuzzers utilize initial seeds generated by symbolic execution [29, 35] while some other fuzzers utilize initial seeds constructed by domain experts or grammar specifications. However, we select base fuzzers manually according to the initial diversity heuristic, which is also not accurate enough.

A possible solution to this threat is to quantify the initial diversity value among different fuzzers for more accurate selection. As defined in [14], the variance or diversity is a measure of the distance of the data in relation to the average. The average standard deviation of a data set is a percentage that indicates how much, on average, each measurement differs from the other. To evaluate the diversity of different base fuzzers, we can choose the most widely used AFL and its path

coverage as a baseline and then calculate standard deviation of each tool from this baseline on the Google fuzzing-test-suite. Then we can calculate the standard deviation of these values as the initial measure of diversity for each base fuzzer, as presented in formula (2) and (1), where n means the number of applications fuzzed by these base fuzzers, p_i means the number of paths covered by the current fuzzer of the target application i and p_{A_i} means the number of paths covered by AFL of the application i.

$$mean = \frac{1}{n}\sum_{i=1}^{n}\frac{p_i - p_{A_i}}{p_{A_i}} \tag{1}$$

$$diversity = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{p_i - p_{A_i}}{p_{A_i}} - mean\right)^2 \tag{2}$$

Take the diversity of AFLFast, FairFuzz, Radamsa, QSYM, and libFuzzer for example, as shown in the statistics presented in Table 14 of the appendix, compared with AFL on different applications, the diversity of AFLFast is 0.040; the diversity of FairFuzz is 0.062; the diversity of Radamsa is 0.197; the diversity of QSYM is 0.271; the diversity of libFuzzer is 11.929. In the same way, the deviation on branches covered and the bugs detected can be calculated. We can add these three values together with different weight for the final diversity quantification. For example, the bug deviation should be assigned with more weights, because from prior research, coverage metrics (the number of paths or branches) are not necessarily correlated well with bugs found. A more advanced way to evaluate the amount of diversity would be to count how many paths/branches/bugs were found by one fuzzer and not by any of the others.

The second potential threat is the mechanism scalability of the ensemble architecture. Section 4.2 describes the ensemble architecture design, and proposes the globally asynchronous and locally synchronous based seed synchronization mechanism. The seed synchronization mechanism focuses on enhancing cooperation among these base fuzzers during their fuzzing processes. With the help of seeds sharing, the performance of ensemble fuzzing is much improved and is better than any of the constituent base fuzzers with the same computing resources usage. However, this mechanism can still be improved for better scalability on different applications and fuzzing tasks. `EnFuzz` only synchronizes the coarse-grained information – interesting seeds, rather than the fine-grained information. For example, we could synchronize the execution trace and array index values of each base fuzzer to improve their effectiveness in cooperation. Furthermore, we currently select and mix base fuzzers manually according to three heuristics. When scaled to arbitrary number of cores, it should be carefully investigated with huge number of empirical evaluations. A possible solution is that the base fuzzers will be dynamically selected and initiated with different number of cores according to the real-time number of paths/branches/bugs found individually by each fuzzer. In the beginning, we have a set of different base fuzzers; then Enfuzz selects n (this number can be configured) base fuzzers randomly. If one fuzzer cannot contribute to coverage for a long time, then it will be terminated, and one new base fuzzer from the sets will be setup for fuzzing or the existing live base fuzzer with better coverage will be allocated with more cores.

We can also apply some effective ensemble mechanisms in ensemble learning such as Boosting to ensemble fuzzing to improve the scalability. Boosting is a widely used ensemble mechanism which will reweigh the base learner dynamically to improve the performance of the ensemble learner: examples that are misclassified gain weight and examples that are classified correctly lose weight. To implement this idea in ensemble fuzzing, we could start up a *master* thread to monitor the execution statuses of all base fuzzers and record more precise information of each base fuzzer, then reassign each base fuzzer some interesting seeds accordingly.

For the number of base fuzzers and parameters in ensemble fuzzing implementation, it is scalable for integration of most fuzzers. Theoretically, the more base fuzzers with diversity, the better ensemble fuzzing performs. We only use four base fuzzers in our evaluation with four CPU cores. The more computing resources we get, higher performance the fuzzing practice acquires. Furthermore, in our implementation, we have tried different values of period time, and the results are very sensitive to the specific setting of this value. It only affects the performance in the beginning, but affects little in the end. Furthermore, refering to the GALS system design, we can also allocate a different synchronization frequency for each local fuzzer dynamically.

7 Conclusion

In this paper, we systematically investigate the practical ensemble fuzzing strategies and the effectiveness of ensemble fuzzing of various fuzzers. Applying the idea of ensemble fuzzing, we bridge two gaps. First, we come up with a method for defining the diversity of base fuzzers and propose a way of selecting a diverse set of base fuzzers. Then, inspired by AFL in parallel mode, we implement a concrete ensemble architecture with one effective ensemble mechanism, a seed synchronization mechanism. `EnFuzz` always outperforms other popular base fuzzers in terms of unique bugs, path and branch coverage with the same resource usage. `EnFuzz` has found 60 new bugs in several well-fuzzed projects and 44 new CVEs were assigned. Our ensemble architecture can be easily utilized to integrate other base fuzzers for industrial practice.

Our future work will focus on three directions: the first is to try some other heuristics and more accurate accumulated quantification of diversity in base fuzzers; the second is to improve the ensemble architecture with more advanced en- semble mechanism and synchronize more fine-grained information; the last is to improve the ensemble architecture with intelligent resource allocation such as dynamically adjusting the synchronization period for each base fuzzer, and allocating more CPU cores to the base fuzzer that shares more interesting seeds.

Acknowledgments

We thank the anonymous reviewers, and our shepherd Thorsten Holz, for their helpful feedback and the support from Huawei. Yu Jiang is the correspondence author.

References

[1] Fuzzer automation with spike. http://resources.infosecinstitute.com/fuzzer-automation-with-spike/. [Online; accessed 12-February-2018].

[2] Cert bff - basic fuzzing framework. https://vuls.cert.org/confluence/display/tools/CERT+BFF+-+Basic+Fuzzing+Framework, 2012. [Online; accessed 10-April-2018].

[3] Afl in parallel mode. https://github.com/mcarpenter/afl/blob/master/docs/parallel_fuzzing.txt, 2016. [Online; accessed 10-April-2019].

[4] Continuous fuzzing for open source software. https://opensource.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html, 2016. [Online; accessed 10-April-2018].

[5] Google. honggfuzz. https://google.github.io/honggfuzz/, 2016. [Online; accessed 10-April-2018].

[6] libfuzzer in parallel mode. https://github.com/google/fuzzer-test-suite/blob/master/tutorial/libFuzzerTutorial.md, 2016. [Online; accessed 10-April-2019].

[7] Technical details for afl. http://lcamtuf.coredump.cx/afl/technical_details.txt, 2016. [Online; accessed 10-April-2019].

[8] fuzzer-test-suite. https://github.com/google/fuzzer-test-suite, 2017. [Online; accessed 10-April-2018].

[9] Google security blog. https://security.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html, 2017. [Online; accessed 10-April-2018].

[10] libfuzzer. https://llvm.org/docs/LibFuzzer.html, 2017. [Online; accessed 10-April-2018].

[11] Sanitizercoverage in llvm. https://clang.llvm.org/docs/SanitizerCoverage.html, 2017. [Online; accessed 10-April-2018].

[12] Clusterfuzz document. https://github.com/google/oss-fuzz/blob/master/docs/clusterfuzz.md, 2018. [Online; accessed 2-November-2018].

[13] Clusterfuzz integration document. https://chromium.googlesource.com/chromium/src/testing/libfuzzer/+/HEAD/clusterfuzz.md, 2018. [Online; accessed 2-November-2018].

[14] BENJAMIN, J. R., AND CORNELL, C. A. *Probability, statistics, and decision for civil engineers*. Courier Corporation, 2014.

[15] BÖHME, M., PHAM, V.-T., NGUYEN, M.-D., AND ROYCHOUDHURY, A. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS17)* (2017).

[16] BÖHME, M., PHAM, V.-T., AND ROYCHOUDHURY, A. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 1032–1043.

[17] CHA, S. K., WOO, M., AND BRUMLEY, D. Program-adaptive mutational fuzzing. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015), IEEE, pp. 725–741.

[18] CHEN, P., AND CHEN, H. Angora: Efficient fuzzing by principled search. *arXiv preprint arXiv:1803.01307* (2018).

[19] DOLAN-GAVITT, B., HULIN, P., KIRDA, E., LEEK, T., MAMBRETTI, A., ROBERTSON, W., ULRICH, F., AND WHELAN, R. Lava: Large-scale automated vulnerability addition. In *Security and Privacy (SP), 2016 IEEE Symposium on* (2016), IEEE, pp. 110–121.

[20] EDDINGTON, M. Peach fuzzing platform. *Peach Fuzzer* (2011), 34.

[21] GODEFROID, P., KIEZUN, A., AND LEVIN, M. Y. Grammar-based whitebox fuzzing. In *ACM Sigplan Notices* (2008), vol. 43, ACM, pp. 206–215.

[22] HELIN, A. Radamsa. https://gitlab.com/akihe/radamsa, 2016.

[23] HOCEVAR, S. zzuf - multi-purpose fuzzer. http://caca.zoy.org/wiki/zzuf, 2007. [Online; accessed 10-April-2018].

[24] HOLLER, C., HERZIG, K., AND ZELLER, A. Fuzzing with code fragments. In *USENIX Security Symposium* (2012), pp. 445–458.

[25] KLEES, G., RUEF, A., COOPER, B., WEI, S., AND HICKS, M. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), ACM, pp. 2123–2138.

[26] LEMIEUX, C., AND SEN, K. Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage. *arXiv preprint arXiv:1709.07101* (2017).

[27] LIANG, J., JIANG, Y., CHEN, Y., WANG, M., ZHOU, C., AND SUN, J. Pafl: extend fuzzing optimizations of single mode to industrial parallel mode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2018), ACM, pp. 809–814.

[28] LIANG, J., WANG, M., CHEN, Y., JIANG, Y., AND ZHANG, R. Fuzz testing in practice: Obstacles and solutions. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2018), IEEE, pp. 562–566.

[29] OGNAWALA, S., HUTZELMANN, T., PSALLIDA, E., AND PRETSCHNER, A. Improving function coverage with munch: a hybrid fuzzing and directed symbolic execution approach. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing* (2018), ACM, pp. 1475–1482.

[30] PETSIOS, T., ZHAO, J., KEROMYTIS, A. D., AND JANA, S. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), ACM, pp. 2155–2168.

[31] SIRER, E. G., AND BERSHAD, B. N. Using production grammars in software testing. In *ACM SIGPLAN Notices* (1999), vol. 35, ACM, pp. 1–13.

[32] STEPHENS, N., GROSEN, J., SALLS, C., DUTCHER, A., WANG, R., CORBETTA, J., SHOSHITAISHVILI, Y., KRUEGEL, C., AND VIGNA, G. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS* (2016), vol. 16, pp. 1–16.

[33] VEGGALAM, S., RAWAT, S., HALLER, I., AND BOS, H. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security* (2016), Springer, pp. 581–601.

[34] WANG, J., CHEN, B., WEI, L., AND LIU, Y. Skyfire: Data-driven seed generation for fuzzing, 2017.

[35] WANG, M., LIANG, J., CHEN, Y., JIANG, Y., JIAO, X., LIU, H., ZHAO, X., AND SUN, J. Safl: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (2018), ACM, pp. 61–64.

[36] XU, W., KASHYAP, S., MIN, C., AND KIM, T. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), ACM, pp. 2313–2328.

[37] YANG, X., CHEN, Y., EIDE, E., AND REGEHR, J. Finding and understanding bugs in c compilers. In *ACM SIGPLAN Notices* (2011), vol. 46, ACM, pp. 283–294.

[38] YUN, I., LEE, S., XU, M., JANG, Y., AND KIM, T. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th {USENIX} Security Symposium ({USENIX} Security 18)* (2018), pp. 745–761.

[39] ZALEWSKI, M. American fuzzy lop. https://github.com/mcarpenter/afl, 2015.

A Preliminary demonstration of diversity among base fuzzers

To help select base fuzzers with larger diversity, we need to estimate the diversity between each base fuzzer. In general, the more differently they perform on different applications, the more diversity among these base fuzzers. Accordingly, we first run each base fuzzer in single mode, with one CPU core on Google's fuzzer-test-suite for 24 hours. Table 14 and Table 15 show the number of paths and branches covered by AFL, AFLFast, FairFuzz, libFuzzer, Radamsa and QSYM. Table 16 shows the corresponding number of unique bugs. Below we present the performance effects of the three diversity heuristics proposed in Section 4.1 in detail.

1) Effects of seed mutation and seed selection strategy – what kind of mutation and selection strategy you use, what kind of path and branch you would cover The first three columns of Table 14 show the performance of the AFL family tools. Their differences are the seed mutation and seed selection strategies. The original AFL performs the best on 5 applications, but performs the worst on other 10 applications. AFLFast performs the best on 13 applications, and only performs the worst on 4 applications. FairFuzz also performs the best on 8 applications, but the worst on the other 9 applications. Although the total number of paths covered improves slightly, the performance variation on each application is huge, ranging from -57% to 38% in single cases.

From the first three columns in Table 15 and Table 16, we get the same observation that the performance of these optimized fuzzers varies significantly on different applications. Although the total number of covered branches and unique crashes improves slightly, the deviation of each application is huge. AFLFast selects seeds that exercise low-frequency paths to mutate more times. Take project lcms for example, this seed selection strategy exercises more new paths by avoiding covering "hot paths" too many times, but on project libarchive, its "hot path" may be the key to further paths. FairFuzz mutates seeds to hit rare branches. Take project libxml2 for example, the rare branch fuzzing strategy guides FairFuzz into deeper areas and covers more branches. However, on libarchive, this strategy fails. FairFuzz spends much time in deep paths and branches, ignoring breadth search. Unlike libxml2, the breadth first search strategy of other fuzzers is more effective on libarchive. In general, the mutation and selection strategy decides the depth and breath of the covered branch and path.

2) Effects of coverage information granularity–what kind of guided information you use, what kind of coverage metric you improve. The diversity between AFL and libFuzzer is their coverage information granularity. According to the fourth column of Table 14, we find that compared with AFL, libFuzzer performs better on 17 applications, and covers 30.3% more paths in total. However, according to the fourth column of the Table 15, compared with AFL, libFuzzer only performs better on 11 applications, which means on 6 applications, libFuzzer covers more paths but less branches. For total branch count, AFL covers 7.3% more than libFuzzer. The reason is that AFL mutates seed by tracking edge hit counts while libFuzzer utilizes the SanitizerCoverage instrumentation to track block hit counts. AFL prefers to cover more branches

while libFuzzer is better at executing more paths. In general, edge-guided means more branches covered, and block-guided means more paths covered.

Table 14: Average number of paths for single mode.

Project	AFL	AFLFast	FairFuzz	libFuzzer	Radamsa	QSYM
boringssl	1334	1674	1760	**3528**	1682	1207
c-ares	80	84	88	**123**	78	72
guetzli	1382	1090	1030	**1773**	1562	1268
lcms	656	**864**	434	338	550	605
libarchive	3756	2834	1630	**10124**	4570	3505
libssh	64	68	62	**201**	63	87
libxml2	5762	7956	8028	**19663**	9392	5098
openssl-1.0.1	**2397**	2103	2285	1709	2303	2330
openssl-1.0.2	2456	**2482**	2040	1881	2108	1947
openssl-1.1.0	2439	2380	**2501**	1897	2311	2416
pcre2	32310	35288	36176	20981	**37850**	24501
proj4	220	218	218	**334**	182	208
re2	5860	6014	5016	**6327**	5418	5084
woff2	14	10	12	**224**	10	15
freetype2	7748	10939	10714	**16360**	9825	7188
harfbuzz	6793	8068	8668	**10800**	5688	6881
json	466	412	408	499	**564**	504
libjpeg	704	**979**	722	448	634	638
libpng	170	159	76	263	493	**577**
llvm	4830	**5760**	5360	5646	4593	4096
openthread	104	123	127	**976**	144	141
sqlite	179	193	172	**431**	256	180
vorbis	891	**1122**	821	848	875	898
wpantund	2959	3048	**3513**	3510	3146	2975
Total	83575	93867	91862	**108884**	94296	72422

Table 15: Average number of branches for single mode.

Project	AFL	AFLFast	FairFuzz	libFuzzer	Radamsa	QSYM
boringssl	2645	3054	3115	3608	**3641**	2539
c-ares	**126**	122	**126**	100	**126**	**126**
guetzli	1913	1491	1428	**2774**	2118	1906
lcms	2216	**2755**	935	2661	1661	2075
libarchive	4906	3961	2387	3561	**5263**	4366
libssh	604	604	604	518	604	**626**
libxml2	10082	12407	12655	13037	**14287**	9779
openssl-1.0.1	3809	3879	**3901**	2591	2993	3829
openssl-1.0.2	3978	4015	3883	2308	**4068**	3796
openssl-1.1.0	8091	8132	8212	7810	**8292**	8032
pcre2	27308	29324	28404	13463	**30615**	19557
proj4	264	260	260	**683**	264	258
re2	15892	15970	15073	11369	**16485**	14477
woff2	114	112	114	**1003**	114	115
freetype2	36798	44028	45319	45541	**49468**	33492
harfbuzz	16872	16051	**19045**	18659	16782	16886
json	4462	3626	**4846**	4547	4821	4538
libjpeg	6865	8495	4028	**8828**	6982	6377
libpng	1917	1878	1135	1651	2126	**2294**
llvm	54107	55697	**57356**	51548	53427	47226
openthread	2062	2473	2646	**5295**	2231	2410
sqlite	2706	**2784**	2771	2178	2190	2709
vorbis	11836	**13561**	12605	5902	11217	12531
wpantund	36059	36620	**37269**	28694	37075	35960
Total	255631	271299	268116	238329	**276850**	235903

3) Effects of Input generation strategy–what kind of generation strategy you use, what kind of corresponding application you fuzz better. The diversity between AFL and Radamsa is the input generation strategy. From the fifth columns of Table 14 and Table 15, compared with AFL, the plenty of inputs generated by Radamsa have some side effects on most target applications (14 applications). Too many extra inputs will slow down the execution speed of the fuzzer. However, for some applications, the inputs generated by Radamsa will improve the performance effectively. Take libxml2 for example, Radamsa has some domain knowledge that prefers to generate some structured data and specific complex format data. These domain knowledge are not available in most mutation-based fuzzers, and this is a critical disadvantage of AFL. But with the help of generation-based fuzzers, the performance of AFL can be improved greatly.

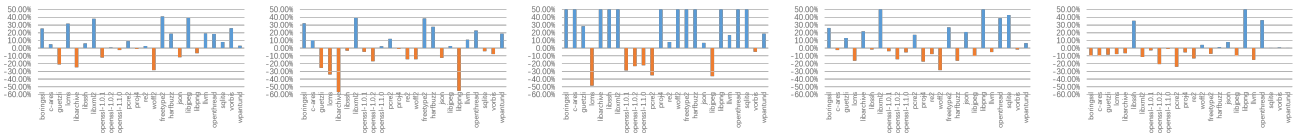
Table 16: Average number of bugs for single mode.

Project	AFL	AFLFast	FairFuzz	libFuzzer	Radamsa	QSYM
boringssl	0	0	0	**1**	0	0
c-ares	1	**2**	**2**	1	**2**	1
guetzli	0	0	0	0	0	0
lcms	0	0	0	0	0	0
libarchive	0	0	0	0	0	0
libssh	0	0	0	**1**	0	0
libxml2	0	**1**	0	**1**	**1**	0
openssl-1.0.1	0	0	0	0	0	0
openssl-1.0.2	**2**	1	0	1	1	**2**
openssl-1.1.0	0	0	0	0	0	0
pcre2	**2**	1	1	1	**2**	1
proj4	0	0	0	**1**	0	0
re2	0	0	0	**1**	0	0
woff2	0	0	0	**1**	0	0
freetype2	0	0	0	0	0	0
harfbuzz	0	0	0	**1**	0	0
json	**1**	**1**	0	0	**1**	0
libjpeg	0	0	0	0	0	0
libpng	0	**1**	**1**	**1**	**1**	**1**
llvm	0	0	**1**	**1**	0	**1**
openthread	0	0	0	**1**	0	0
sqlite	0	0	0	**1**	**1**	**1**
vorbis	1	1	**2**	1	1	**2**
wpantund	0	0	0	0	0	0
Total	7	8	7	**15**	10	9

In conclusion: Different base fuzzers perform variously on distinct target applications, showing the diversity for the base fuzzers. The more diversity of these base fuzzers, the more differently they perform on different applications. Furthermore, the above three types of effects should be considered and could be incorporated into the fuzzing evaluation guideline [25] to avoid biased test cases or metrics selection when evaluating different types of fuzzing optimization.

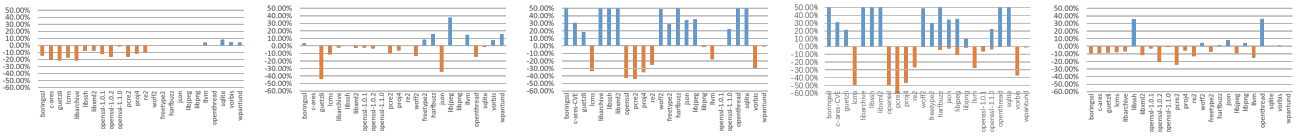
B Does performance vary in different modes?

We choose AFL as the baseline, and compare other tools with AFL on path coverage to demonstrate the performance variation. Figure 5 shows the average number of paths executed on Google's fuzzer-test-suite by each base fuzzer compared with AFL in single mode. We also collect the result of each base fuzzer running in parallel mode with four threads, and the result is presented in Figure 6. Figure 7 shows the average



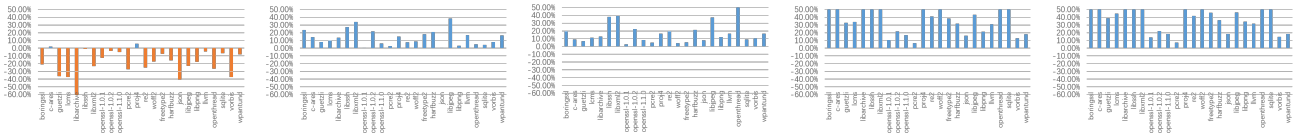
(a) performance of AFLFast in single thread (b) performance of FairFuzz in single thread (c) performance of libFuzzer in single thread (d) performance of Radamsa in single thread (d) performance of QSYM in single thread

Figure 5: Paths covered by base fuzzers compared with AFL in single mode on a single core.



(a) performance of AFLFast in four threads (b) performance of FairFuzz in four threads (c) performance of libFuzzer in four threads (d) performance of Radamsa in four threads (d) performance of QSYM in four threads

Figure 6: Paths covered by base fuzzers compared with AFL in parallel mode with four threads on four cores.



(a) performance of `EnFuzz`$^-$ in four threads (a) performance of `EnFuzz-A` in four threads (a) performance of `EnFuzz-Q` in four threads (b) performance of `EnFuzz-L` in four threads (c) performance of `EnFuzz` in four threads

Figure 7: Paths covered by `EnFuzz` with four threads on four cores compared with AFL in parallel mode with four threads on four cores. `EnFuzz`$^-$ without the proposed seed synchronization performs the worst, and `EnFuzz` performs the best.

number of paths executed by `EnFuzz` compared with AFL in parallel mode with four CPU cores. From these results, we get the following conclusions:

- From the results of Figure 5 and Figure 6, we find that compared with AFL, the two optimized fuzzers AFLFast and FairFuzz, block coverage guided fuzzer libFuzzer, generation-based fuzzer Radamsa and hybrid fuzzer QSYM perform variously on different applications both in single mode and in parallel mode. It demonstrates that the performance of these base fuzzers is challenged by the diversity of the diverse real applications. The performance of their fuzzing strategies cannot constantly perform better than AFL. The performance variation exists in these state-of-the-art fuzzers.

- Comparing the result of Figure 5 and Figure 6, we find that the performance of these base fuzzers in parallel mode are quite different from those in single mode, especially for AFLFast and FairFuzz. In single mode, the other two optimized base fuzzers perform better than AFL in many applications. But in parallel mode, the result is completely opposite that the original AFL performs better on almost all applications.

- From the result of Figure 7, it reveals that `EnFuzz-A`, `EnFuzz-L` and `EnFuzz` always perform better than AFL on the target applications. For the same computing resources usage where AFL running in parallel mode with four CPU cores, `EnFuzz-A` covers 11.26% more paths than AFL, ranging from 4% to 38% in single cases, `EnFuzz-Q` covers 12.48% more paths than AFL, ranging from 5% to 177% in single cases, `EnFuzz-L` covers 37.50% more paths than AFL, ranging from 13% to 455% in single cases. `EnFuzz` covers 42.39% more paths than AFL, ranging from 14% to 462% in single cases. Through ensemble fuzzing, the performance variation can be reduced.

- From the result of Figure 7, it reveals that `EnFuzz`$^-$ without seed synchronization performs worse than AFL parallel mode under the same resource constraint. Compared with `EnFuzz-A`, `EnFuzz-Q` covers 1.09% more paths, `EnFuzz-L` covers 23.58% more paths. For `EnFuzz`, it covers 27.97% more paths than `EnFuzz-A`, 26.59% more paths than `EnFuzz-Q`, 3.6% more paths than `EnFuzz-L`, and always performs the best on all applications. The more diversity among those integrated base fuzzers, the better performance of ensemble fuzzing, and the seed synchronization contributes more to the improvements.

In conclusion: the performance of the state-of-the-art fuzzers is greatly challenged by the diversity of those real-world applications, and it can be improved through the ensemble fuzzing approach. Furthermore, those optimized strategies work in single mode can not be directly scaled to parallel mode which is widely used in industrial practice. The ensemble fuzzing approach is a critical enhancement to the single and parallel mode of those optimized strategies.

GRIMOIRE: Synthesizing Structure while Fuzzing

Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi,
Sergej Schumilo, Simon Wörner and Thorsten Holz

Ruhr-Universität Bochum, Germany

Abstract

In the past few years, fuzzing has received significant attention from the research community. However, most of this attention was directed towards programs without a dedicated parsing stage. In such cases, fuzzers which leverage the input structure of a program can achieve a significantly higher code coverage compared to traditional fuzzing approaches. This advancement in coverage is achieved by applying large-scale mutations in the application's input space. However, this improvement comes at the cost of requiring expert domain knowledge, as these fuzzers depend on structure input specifications (e. g., grammars). Grammar inference, a technique which can automatically generate such grammars for a given program, can be used to address this shortcoming. Such techniques usually infer a program's grammar in a pre-processing step and can miss important structures that are uncovered only later during normal fuzzing.

In this paper, we present the design and implementation of GRIMOIRE, a fully automated coverage-guided fuzzer which works without any form of human interaction or preconfiguration; yet, it is still able to efficiently test programs that expect highly structured inputs. We achieve this by performing large-scale mutations in the program input space using grammar-like combinations to synthesize new highly structured inputs without any pre-processing step. Our evaluation shows that GRIMOIRE outperforms other coverage-guided fuzzers when fuzzing programs with highly structured inputs. Furthermore, it improves upon existing grammar-based coverage-guided fuzzers. Using GRIMOIRE, we identified 19 distinct memory corruption bugs in real-world programs and obtained 11 new CVEs.

1 Introduction

As the amount of software impacting the (digital) life of nearly every citizen grows, effective and efficient testing mechanisms for software become increasingly important. The publication of the fuzzing framework AFL [65] and its success at uncovering a huge number of bugs in highly relevant software has spawned a large body of research on effective feedback-based fuzzing. AFL and its derivatives have largely conquered automated, dynamic software testing and are used to uncover new security issues and bugs every day. However, while great progress has been achieved in the field of fuzzing, many hard cases still require manual user interaction to generate satisfying test coverage. To make fuzzing available to more programmers and thus scale it to more and more target programs, the amount of expert knowledge that is required to effectively fuzz should be reduced to a minimum. Therefore, it is an important goal for fuzzing research to develop fuzzing techniques that require less user interaction and, in particular, less domain knowledge to enable more automated software testing.

Structured Input Languages. One common challenge for current fuzzing techniques are programs which process highly structured input languages such as interpreters, compilers, text-based network protocols or markup languages. Typically, such inputs are consumed by the program in two stages: parsing and semantic analysis. If parsing of the input fails, deeper parts of the target program—containing the actual application logic—fail to execute; hence, bugs hidden "deep" in the code cannot be reached. Even advanced feedback fuzzers—such as AFL—are typically unable to produce diverse sets of syntactically valid inputs. This leads to an imbalance, as these programs are part of the most relevant attack surface in practice, yet are currently unable to be fuzzed effectively. A prominent example are browsers, as they parse a multitude of highly-structured inputs, ranging from XML or CSS to JavaScript and SQL queries.

Previous approaches to address this problem are typically based on manually provided grammars or seed corpora [2, 14, 45, 52]. On the downside, such methods require human experts to (often manually) specify the grammar or suitable seed corpora, which becomes next to impossible for applications with undocumented or proprietary input specifications. An orthogonal line of work tries to utilize advanced program analysis techniques to automatically infer grammars

[4, 5, 25]. Typically performed as a pre-processing step, such methods are used for generating a grammar that guides the fuzzing process. However, since this grammar is treated as immutable, no additional learning takes place during the actual fuzzing run.

Our Approach. In this paper, we present a novel, fully automated method to fuzz programs with a highly structured input language, without the need for any human expert or domain knowledge. Our approach is based on two key observations: First, we can use code coverage feedback to automatically infer structural properties of the input language. Second, the precise and "correct" grammars generated by previous approaches are actually unnecessary in practice: since fuzzers have the virtue of high test case throughput, they can deal with a significant amount of noise and imprecision. In fact, in some programs (such as `Boolector`) with a rather diverse set of input languages, the additional noise even benefits the fuzz testing. In a similar vein, there are often program paths which can only be accessed by inputs *outside* of the formal specifications, e. g., due to incomplete or imprecise implementations or error handling code.

Instead of using a pre-processing step, our technique is directly integrated in the fuzzing process itself. We propose a set of generalizations and mutations that resemble the inner workings of a grammar-based fuzzer, without the need for an explicit grammar. Our generalization algorithm analyzes each newly found input and tries to identify substrings of the input which can be replaced or reused in other positions. Based on this information, the mutation operators recombine fragments from existing inputs. Overall, this results in synthesizing new, structured inputs without prior knowledge of the underlying specification.

We have implemented a prototype of the proposed approach in a tool called GRIMOIRE[1]. GRIMOIRE does not need any specification of the input language and operates in an automated manner without requiring human assistance; in particular, without the need for a format specification or seed corpus. Since our techniques make no assumption about the program or its environment behavior, GRIMOIRE can be easily applied to closed-source targets as well.

To demonstrate the practical feasibility of our approach, we perform a series of experiments. In a first step, we select a diverse set of programs for a comparative evaluation: we evaluate GRIMOIRE against other fuzzers on four scripting language interpreters (`mruby`, `PHP`, `Lua` and `JavaScriptCore`), a compiler (`TCC`), an assembler (`NASM`), a database (`SQLite`), a parser (`libxml`) and an SMT solver (`Boolector`). Demonstrating that our approach can be applied in many different scenarios without requiring any kind of expert knowledge, such as an input specification. The evaluation results show that our approach outperforms all existing coverage-guided fuzzers; in the case of `Boolector`, GRIMOIRE finds up to 87% more coverage than the baseline (REDQUEEN). Second, we evaluate GRIMOIRE against state-of-the-art grammar-based fuzzers. We observe that in situations where an input specification is available, it is advisable to use GRIMOIRE in addition to a grammar fuzzer to further increase the test coverage found by grammar fuzzers. Third, we evaluate GRIMOIRE against current state-of-the-art approaches that use automatically inferred grammars for fuzzing and found that we can significantly outperform such approaches. Overall, GRIMOIRE found 19 distinct memory corruption bugs that we manually verified. We responsibly disclosed all of them to the vendors and obtained 11 CVEs. During our evaluation, the next best fuzzer only found 5 of these bugs. In fact, GRIMOIRE found more bugs than all five other fuzzers combined.

Contributions. In summary, we make the following contributions:

- We present the design, implementation and evaluation of GRIMOIRE, an approach to fully automatically fuzz highly structured formats with no human interaction.
- We show that even though GRIMOIRE is a binary-only fuzzer that needs no seeds or grammar as input, it still outperforms many fuzzers that make significantly stronger assumptions (e. g., access to seeds, grammar specifications and source code).
- We found and reported multiple bugs in various common projects such as `PHP`, `gnuplot` and `NASM`.

2 Challenges in Fuzzing Structured Languages

In this section, we briefly summarize essential information paramount to the understanding of our approach. To this end, we provide an overview of different fuzzing approaches, while focusing on their shortcomings and open challenges. In particular, we describe those details of AFL (e. g., code coverage) that are necessary to understand our approach. Additionally, we explain how fuzzers explore the state space of a program and how grammars aid the fuzzing process.

Generally speaking, fuzzing is a popular and efficient software testing technique used to uncover bugs in applications. Fuzzers typically operate by producing a large number of test cases, some of which may trigger bugs. By closely monitoring the runtime execution of these test cases, fuzzers are able to locate inputs causing faulty behavior. In an abstract view, one can consider fuzzing as randomly exploring the state space of the application. Typically, most totally random inputs are rejected early by the target application and

[1] A *grimoire* is a magical book that recombines magical elements to formulas. Furthermore, it has the same word stem as the Old French word for grammar—namely, *gramaire*.

do not visit interesting parts of the state space. Thus, in our abstract view, the state space has interesting and uninteresting regions. Efficient fuzzers somehow have to ensure that they avoid uninteresting regions most of the time. Based on this observation, we can divide fuzzers into three broad categories, namely: (a) blind, (b) coverage-guided and (c) hybrid fuzzers, as explained next.

2.1 Blind Fuzzing

The most simple form of a fuzzer is a program which generates a stream of random inputs and feeds it to the target application. If the fuzzer generates inputs without considering the internal behavior of the target application, it is typically referred to as a *blind fuzzer*. Examples of blind fuzzers are RADAMSA [29], PEACH [14], Sulley [45] and ZZUF [32]. To obtain new inputs, fuzzers traditionally can build on two strategies: *generation* and *mutation*.

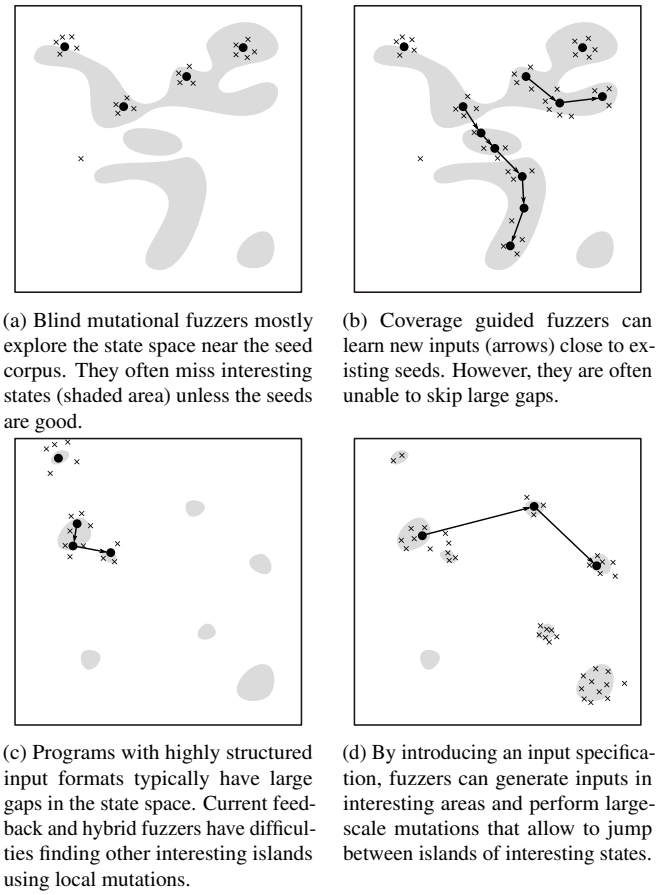
Fuzzers employing the former approach have to acquire a specification, typically a grammar or model, of an application's expected input format. Then, a fuzzer can use the format specification to be able to generate novel inputs in a somewhat efficient way. Additionally, in some cases, a set of valid inputs (a so-called *corpus*) might be required to aid the generation process [46, 58].

On the other hand, fuzzers which employ a mutation-based strategy require only an initial corpus of inputs, typically referred to as *seeds*. Further test cases are generated by randomly applying various mutations on initial seeds or novel test cases found during fuzzing runs. Examples for common mutators include bit flipping, splicing (i. e., recombining two inputs) and repetitions [14, 29, 32]. We call these mutations *small-scale mutations*, as they typically change small parts of the program input.

Blind fuzzers suffer from one major drawback. They either require an extensive corpus or a well-designed specification of the input language to provide meaningful results. If a program feature is not represented by either a seed or the input language specification, a blind fuzzer is unlikely to exercise it. In our abstract, state space-based view, this can be understood as blindly searching the state space near the seed inputs, while failing to explore interesting neighborhoods, as illustrated in Figure 1(a). To address this limitation, the concept of coverage-guided fuzzing was introduced.

2.2 Coverage-guided Fuzzing

Coverage-guided fuzzers employ lightweight program coverage measurements to trace how the execution path of the application changes based on the provided input (e. g., by tracking which basic blocks have been visited). These fuzzers use this information to decide which input should be stored or discarded to extend the corpus. Therefore, they are able to evolve inputs that differ significantly from the original seed corpus



(a) Blind mutational fuzzers mostly explore the state space near the seed corpus. They often miss interesting states (shaded area) unless the seeds are good.

(b) Coverage guided fuzzers can learn new inputs (arrows) close to existing seeds. However, they are often unable to skip large gaps.

(c) Programs with highly structured input formats typically have large gaps in the state space. Current feedback and hybrid fuzzers have difficulties finding other interesting islands using local mutations.

(d) By introducing an input specification, fuzzers can generate inputs in interesting areas and perform large-scale mutations that allow to jump between islands of interesting states.

Figure 1: Different fuzzers exploring distinct areas in state space.

while at the same time exercising new program features. This strategy allows to gradually explore the state of the program as it uncovers new paths. This behavior is illustrated in Figure 1(b). The most prominent example of a coverage-guided fuzzer is AFL [65]. Following the overwhelming success of AFL, various more efficient coverage-guided fuzzers such as ANGORA [12], QSYM [64], T-FUZZ [47] or REDQUEEN [3] were proposed.

From a high-level point of view, all these AFL-style fuzzers can be broken down into three different components: (i) the input queue stores and schedules all inputs found so far, (ii) the mutation operations produce new variants of scheduled inputs and (iii) the global coverage map is used to determine whether a new variant produced novel coverage (and thus should be stored in the queue).

From a technical point of view, this maps to AFL as follows: Initially, AFL fills the input queue with the seed inputs. Then, it runs in a continuous fuzzing loop, composed of the following steps: (1) Pick an input from the input queue, then (2) apply multiple mutation operations on it. After each mutation, (3) execute the target application with the selected input. If new coverage was triggered by the input, (4) save it back to the queue. To determine whether new coverage was triggered,

AFL compares the results of the execution with the values in the global coverage map.

This global coverage map is filled as follows: AFL shares a memory area of the same size as the global coverage map with the fuzzing target. During execution, each transition between two basic blocks is assigned a position inside this shared memory. Every time the transition is triggered, the corresponding entry (one byte) in the shared memory map is incremented. To reduce overhead incurred by large program traces, the shared coverage map has a fixed size (typically 2^{16} bytes). While this might introduce collisions, empirical evaluation has shown that the performance gains make up for the loss in the precision [66].

After the target program terminates, AFL compares the values in the shared map to all previous runs stored in the global coverage map. To check if a new edge was executed, AFL applies the so-called *bucketing*. During bucketing, each entry in the shared map is rounded to a power of 2 (i. e., at most a single bit is set in each entry). Then, a simple binary operation is used to check if any new bits are present in the shared map (but not the global map). If any new bit is present, the input is stored in the queue. Furthermore, all new bits are also set to 1 in the global coverage map. We distinguish between *new bits* and *new bytes*. If a new bit is set to 1 in a byte that was previously zero, we refer to it as a *new byte*. Intuitively, a new byte corresponds to new coverage while a new bit only illustrates that a known edge was triggered more often (e. g., more loop iterations were observed).

Example 1. *For example, consider some execution a while after starting the fuzzer run for a program represented by its Control-Flow Graph (CFG) in Figure 2ⓐ. Assume that the fictive execution of an input causes a loop between B and C to be executed 10 times. Hence, the shared map is updated as shown in ⓑ, reflecting the fact that edges A → B and C → D were executed only once, while the edges B → C and C → B were encountered 10 (*`0b1010`*) times. In ⓒ, we illustrate the final bucketing step. Note how* `0b1010` *is put into the bucket* `0b1000`*, while* `0b0001` *is moved into the one identified by* `0b0001`*. Finally, AFL checks whether the values encountered in this run triggered unseen edges in ⓓ. To this end, we compare the shared map to the global coverage map and update it accordingly (see ⓔ), setting bits set in the shared but not global coverage map. As visualized in ⓕ, a new bit was set for two entries, while a new byte was found for one. This means that the edge between C → D was previously unseen, thus the input used for this example triggered new coverage.*

While coverage-guided fuzzers significantly improve upon blind fuzzers, they can only learn from new coverage if they are able to guess an input that triggers the new path in the program. In certain cases, such as multi-byte magic values, the probability of guessing an input necessary to trigger a different path is highly unlikely. These kind of situations occur if there is a significant gap between interesting areas in the state space and existing mutations are unlikely to cross the uninteresting gap. The program displayed in the Figure 1(b) illustrates a case with only one large gap in the program space. Thus, this program is well-suited for coverage-guided fuzzing. However, current mutation-based coverage-guided fuzzers struggle to explore the whole state space because the island in the lower right is never reached. To overcome this limitation, hybrid fuzzer were introduced; these combine coverage-guided fuzzing with more in-depth program analysis techniques.

2.3 Hybrid Fuzzing

Hybrid fuzzers typically combine coverage-guided fuzzing with program analysis techniques such as symbolic execution, concolic execution or taint tracking. As noted above, fast and cheap fuzzing techniques can uncover the bulk of the easy-to-reach code. However, they struggle to trigger program paths that are highly unlikely. On the other hand, symbolic or concolic execution does not move through the state space randomly. Instead, these techniques use an SMT solver to find inputs that trigger the desired behavior. Therefore, they can cover hard-to-reach program locations. Still, as a consequence of the precise search technique, they struggle to explore large code regions due to significant overhead.

By combining fuzzing and reasoning-based techniques, one can benefit from the strength of each individual technique, while avoiding the drawbacks. Purely symbolic approaches have proven difficult to scale. Therefore, most current tools such as SAGE [21], DRILLER [54] or QSYM [64] use concolic execution instead. This mostly avoids the state explosion problem by limiting the symbolic execution to a single path. To further reduce the computation cost, some fuzzers such as VUZZER [50] and ANGORA [12] only use taint tracking. Both approaches still allow to overcome the common multi-byte magic value problem. However, they lose the ability to explore behavior more globally.

While hybrid fuzzers can solve constraints over individual values of the input, they are typically not efficient at solving constraints on the *overall* structure of the input. Consider target programs such as a script interpreter. To uncover a new valid code path, the symbolic executor usually has to consider a completely different path through the parsing stage. This leads to a large number of very large gaps in the state space as illustrated in Figure 1(c). Therefore, concolic execution or taint tracking-based tools are unable to solve these constraints. In purely symbolic execution-based approaches, this leads to a massive state explosion.

2.4 Coverage-guided Grammar Fuzzing

Beside the problem of multi-byte magic values, there is another issue which leads to large gaps between interesting

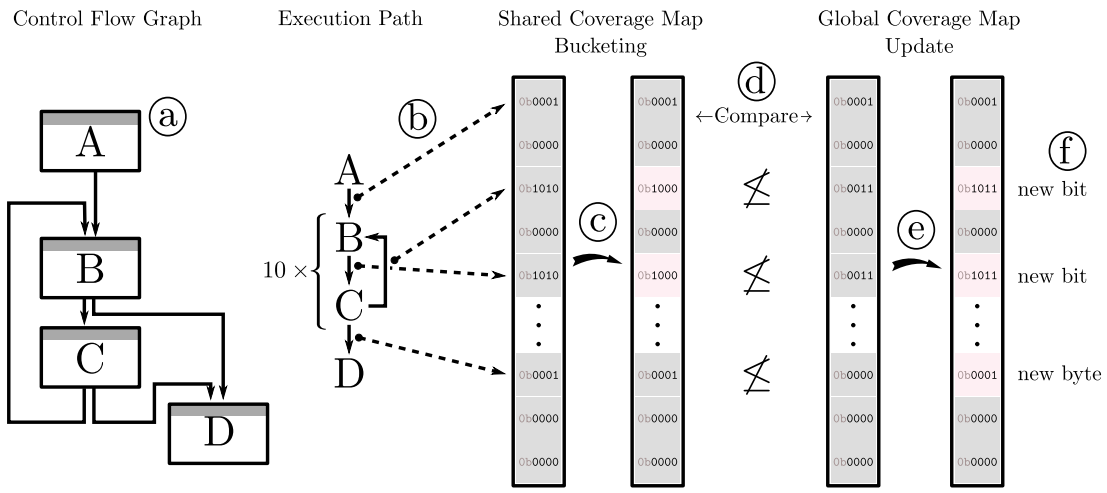


Figure 2: The process of tracing a path in a program and introducing new bits and bytes in the global coverage map.

parts of the state space: programs with structured input languages. Examples for such programs are interpreters, compilers, databases and text-based Internet protocols. As mentioned earlier, current mutational blind and coverage-guided as well as hybrid fuzzers cannot efficiently fuzz programs with structured input languages. To overcome this issue, *generational fuzzers* (whether blind, coverage-guided or hybrid) use a specification of the input language (often referred to as a *grammar*) to generate valid inputs. Thereby, they reduce the space of possible inputs to a subset that is much more likely to trigger interesting states. Additionally, coverage-guided grammar fuzzers can mutate inputs in this reduced subset by using the provided grammar. We call these mutations *large-scale mutations* since they modify large part of the input. This behavior is illustrated in Figure 1(d).

Therefore, the performance of fuzzers can be increased drastically by providing format specifications to the fuzzer, as implemented in NAUTILUS [2] and AFLSMART [48]. These specifications let the fuzzer spend more time exercising code paths deep in the target application. Particularly, the fuzzer is able to sensibly recombine inputs that trigger interesting features in a way that has a good chance of triggering more interesting behaviors.

Grammar fuzzers suffer from two major drawbacks. First, they require human effort to provide precise format specification. Second, if the specification is incomplete or inaccurate, the fuzzer lacks the capability to address these shortcomings. One can overcome these two drawbacks by automatically inferring the specification (grammar).

2.5 Grammar Inference

Due to the impact of grammars on software testing, various approaches have been developed that automatically can generate input grammars for target programs. Bastani et al. [5] introduced GLADE, which uses a modified version of the target as a black-box oracle that tests if a given input is syntactically valid. GLADE turns valid inputs into regular expressions that generate (mostly) valid inputs. Then, these regular expressions are turned into full grammars by trying to introduce recursive replacement rules. In each step, the validity of the resulting grammar is tested using multiple oracle queries. This approach has three significant drawbacks: First, the inference process takes multiple hours for complex targets such as scripting languages. Second, the user needs to provide an automated testing oracle, which might not be trivial to produce. Third, in the context of fuzzing, the resulting grammars are not well suited for fuzzing as our evaluation shows (see Section 5.4 for details). Additionally, this approach requires a pre-processing step before fuzzing starts in order to infer a grammar from the input corpus.

Other approaches use the target application directly and thus avoid the need to create an oracle. AUTOGRAM [34], for instance, uses the original program and taint tracking to infer grammars. It assumes that the functions that are called during parsing reflect the non-terminals of the intended grammar. Therefore, it does not work for recursive descent parsers. PYGMALION [25] is based on simplified symbolic execution of Python code to avoid the dependency on a set of good inputs. Similar to AUTOGRAM, PYGMALION assumes that the function call stack contains relevant information to identify recursive rules in the grammar. This approach works well for hand-written, recursive descent parsers; however, it will have severe difficulties with parsers generated by parser generators. These parsers are typically implemented as table-driven automatons and do not use function calls at all. Additionally, robust symbolic execution and taint tracking are still challenging for binary-only targets.

2.6 Shortcomings of Existing Approaches

To summarize, current automated software testing approaches have the following disadvantages when used for fuzzing of programs that accept structured input languages:

- **Needs Human Assistance.** Some techniques require human assistance to function properly. Either in terms of providing information or in terms of modifying the target program.
- **Requires Source Code.** Some fuzzing techniques require access to source code. This puts them at a disadvantage as they cannot be applied to proprietary software in binary format.
- **Requires a Precise Environment Model.** Techniques based on formal reasoning such as symbolic/concolic execution as well as taint tracking require precise semantics of the underlying platform as well as semantics of all used Operating System (OS) features (e. g., syscalls).
- **Requires a Good Corpus.** Many techniques only work if the seed corpus already contains most features of the input language.
- **Requires a Format Specification.** Similarly, many techniques described in this section require precise format specifications for structured input languages.
- **Limited To Certain Types of Parsers.** Some approaches make strong assumptions about the underlying implementation of the parser. Notably, some approaches are unable to deal with parses generated by common parser generators such as `GNU Bison` [15] or `Yacc` [37].
- **Provides Only Small-scale Mutations.** As discussed in this section, various approaches cannot provide mutations that cross large gaps in the program space.

Table 1: Requirements and limitations of different fuzzers and inference tools when used for fuzzing structured input languages. If a shortcoming applies to a tool, it is denoted with ✗, otherwise with ✓.

	PEACH	AFL	REDQUEEN	QSYM	ANGORA	NAUTILUS	AFLSMART	GLADE	AUTOGRAM	PYGMALION	GRIMOIRE
human assistance	✗	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓
source code	✓	✓	✓	✓	✗	✗	✓	✓	✗	✗	✓
environment model	✓	✓	✓	✗	✗	✓	✓	✓	✗	✗	✓
good corpus	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓	✓
format specifications	✗	✓	✓	✓	✓	✗	✗	✓	✓	✓	✓
certain parsers	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓
small-scale mutations	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓

We analyzed existing fuzzing methods, the results of this survey are shown in Table 1. We found that all current approaches have at least one shortcoming for fuzzing programs with highly structured inputs. In the next section, we propose a design that avoids all the mentioned drawbacks.

3 Design

Based on the challenges identified above, we now introduce the design of GRIMOIRE, a fully automated approach that synthesizes the target's structured input language during fuzzing. Furthermore, we present large-scale mutations that cross significant gaps in the program space. Note that none of the limitations listed in Table 1 applies to our approach. To emphasize, our design does not require any previous information about the input structure. Instead, we learn an ad-hoc specification based on the program semantics and use it for coverage-guided fuzzing.

We first provide a high-level overview of GRIMOIRE, followed by a detailed description. GRIMOIRE is based on identifying and recombining fragments in inputs that trigger new code coverage during a normal fuzzing session. It is implemented as an additional fuzzing stage on top of a coverage-guided fuzzer. In this stage, we strip every new input (that is found by the fuzzer and produced new coverage) by replacing those parts of the input that can be modified or replaced without affecting the input's new coverage by the symbol □. This can be understood as a generalization, in which we reduce inputs to the fragments that trigger new coverage, while maintaining information about *gaps* or *candidate positions* (denoted by □). These gaps are later used to splice in fragments from other inputs.

Example 2. *Consider the input "*`if(x>1) then x=3 end`*" and assume it was the first input to trigger the coverage for a syntactically correct if-statement as well as for "*`x>1`*". We can delete the substring "*`x=3`*" without affecting the interesting new coverage since the if-statement remains syntactically correct. Additionally, the space between the condition and the "*`then`*" is not mandatory. Therefore, we obtain the generalized input "*`if(x>1)□then □end`*".*

After a set of inputs was successfully generalized, fragments from the generalized inputs are recombined to produce new candidate inputs. We incorporate various different strategies to combine existing fragments, learned tokens (a special form of substrings) and strings from the binary in an automated manner.

Example 3. *Assume we obtained the following generalized inputs: "*`if(x>1)□then □end`*" and "*`□x=□y+□`*". We can use this information in many ways to generate plausible recombinations. For example, starting with the input "*`if(x>1)□then □end`*", we can replace the second gap with the second input, obtaining "*`if(x>1)□then □x=□y+□end`*". Afterwards, we choose the slice "*`□y+□`*" from the second input and splice it into the fourth gap and obtain "*`if(x>1)□then □x=□y+□y+□end`*". In a last step,*

we replace all remaining gaps by an empty string. Thus, the final input is "`if(x>1)then x=y+y+end`".

One could think of our approach as a context-free grammar with a single non-terminal input □ and all fragments of generalized inputs as production rules. Using these loose, grammar-like recombination methods in combination with feedback-driven fuzzing, we are able to automatically learn interesting structures.

3.1 Input Generalization

We try to generalize inputs that produced new coverage (e. g., inputs that introduced new bytes to the bitmap, cf. Section 2.2). The generalization process (Algorithm 1) tries to identify parts of the input that are irrelevant and fragments that caused new coverage. In a first step, we use a set of rules to obtain fragment boundaries (Line 3). Consecutively, we remove individual fragments (Line 4). After each step, we check if the reduced input still triggers the same new coverage bytes as the original input (Line 5). If this is the case, we replace the fragment that was removed by a □ and keep the reduced input (Line 6).

Algorithm 1: Generalizing an input through fragment identification.

```
   Data: input is the input to generalize, new_bytes are the new
         bytes of the input, splitting_rule defines how to split an
         input
   Result: A generalized version of input
1  start ← 0
2  while start < input.length() do
3      end ← find_next_boundary(input, splitting_rule)
4      candidate ← remove_substring(input, start, end)
5      if get_new_bytes(candidate) == new_bytes then
6          input ← replace_by_gap(input, start, end)
7      start ← end
8  input ← merge_adjacent_gaps(input)
```

Example 4. *Consider input* "`pprint 'aaaa'`" *triggers the new bytes* 20 *and* 33 *because of the pprint statement. Furthermore, assume that we use a rule that splits inputs into non-overlapping chunks of length two. Then, we obtain the chunks* "`pp`", "`ri`", "`nt`", "` '`", "`aa`", "`aa`" *and* "`'`". *If we remove any of the first four chunks, the modified input will not trigger the same new bytes since we corrupted the pprint statement. However, if we remove the fifth or sixth chunk, we still trigger the bytes* 20 *and* 33 *since the pprint statement remains valid. Therefore, we reduce the input to* "`pprint '□□'`". *As we have two adjacent* □*, we merge them into one. The generalized input is* "`pprint '□'`".

To generalize an input as much as possible, we use several fragmentation strategies for which we apply Algorithm 1 repeatedly. First, we split the input into overlapping chunks of size 256, 128, 64, 32, 2 and 1 to remove large uninteresting parts as early as possible. Afterwards, we dissect at different separators such as '`.`', '`;`', '`,`', '`\n`', '`\r`', '`\t`', '`#`' and '` `'. As a consequence, we can remove one or more statements in code, comments and other parts that did not cause the input's new coverage. Finally, we split at different kinds of brackets and quotation marks. These fragments can help to generalize constructs such as function parameters or nested expressions. In detail, we split in between of '`()`', '`[]`', '`{}`', '`<>`' as well as single and double quotes. To guess different nesting levels in between these pairs of opening/closing characters, we extend Algorithm 1 as follows: If the current index `start` matches an opening character, we search the furthermost matching closing character, create a `candidate` by removing the substring in between and check if it triggers the same new coverage. We iteratively do this by choosing the next furthermost closing character—effectively shrinking the fragment size—until we find a substring that can be removed without changing the `new_bytes` or until we reach the index `start`. In doing so, we are able to remove the largest matching fragments from the input that are irrelevant for the input's new coverage.

Since we want to recombine (generalized) inputs to find new coverage—as we describe in the following section—we store the original input as well as its generalization. Furthermore, we split the generalized input at every □ and store the substrings *(tokens)* in a set; these tokens often are syntactically interesting fragments of the structured input language.

Example 5. *We map the input* "`if(x>1) then x=3 end`" *to its generalization* "`if(x>1)`□`then` □`end`". *In addition, we extract the tokens* "`if(x>1)`", "`then `" *and* "`end`". *For the generalized input* "□`x=`□`y+`□", *we remember the tokens* "`x=`" *and* "`y+`".

3.2 Input Mutation

GRIMOIRE builds upon knowledge obtained from the generalization stage to generate inputs that have good chances of finding new coverage. For this, it recombines (fragments of) generalized inputs, tokens and strings (stored in a dictionary) that are automatically obtained from the data section of the target's binary. On a high level, we can divide our mutations into three standalone operations: input extension, recursive replacement and string replacement.

Given the current input from the fuzzing queue, we add these mutations to the so-called *havoc phase* [3] as described in Algorithm 2. First, we use Redqueen's `havoc_amount` to determine—based on the input's performance—how often we should apply the following mutations (in general, between 512 and 1024 times). First, if the input triggered new bytes in the bitmap, we take its generalized form and apply the large-scale mutations `input_extension` and `recursive_replacement`. Afterwards, we take the original input string (accessed by `input.content()`) and apply the

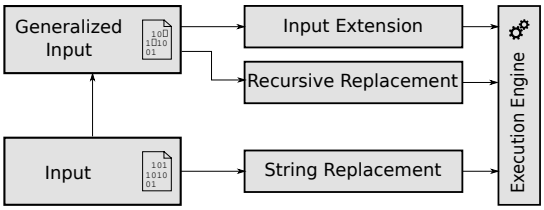


Figure 3: A high-level overview of our mutations. Given an input, we apply various mutations on its generalized and original form. Each mutation then feeds mutated variants of the input to the fuzzer's execution engine.

`string_replacement` mutation. This process is illustrated in Figure 3.

Algorithm 2: High-level overview of the mutations introduced in GRIMOIRE.

```
Data: input is the current input in the queue, generalized is the
      set of all previously generalized inputs, tokens and strings
      from the dictionary, strings is the provided dictionary
      obtained from the binary
content ← input.content()
n ← havoc_amount(input.performance())
for i ← 0 to n do
    if input.is_generalized() then
        input_extension(input, generalized)
        recursive_replacement(input, generalized)
    string_replacement(content, strings)
```

Before we describe our mutations in detail, we explain two functions that all mutations have in common—`random_generalized` and `send_to_fuzzer`. The function `random_generalized` takes as input a set of all previously generalized inputs, tokens and strings from the dictionary and returns—based on random coin flips—a random (slice of a) generalized input, token or string. In case we pick an input slice, we select a substring between two arbitrary □ in a generalized input. This is illustrated in Algorithm 3. The other function, `send_to_fuzzer`, implies that the fuzzer executes the target application with the mutated input. It expects concrete inputs. Thus, mutations working on generalized inputs first replace all remaining □ by an empty string.

Algorithm 3: Random selection of a generalized input, slice, token or string.

```
Data: generalized is the set of all previously generalized inputs,
      tokens and strings from the dictionary
Result: rand is a random generalized input, slice token or string
if random_coin() then
    if random_coin() then
        rand ← random_slice(generalized)
    else
        rand ← random_token_or_string(generalized)
else
    rand ← random_generalized_input(generalized)
```

3.2.1 Input Extension

The input extension mutation is inspired by the observation that—in highly structured input languages—often inputs are chains of syntactically well-formed statements. Therefore, we extend an generalized input by placing another randomly chosen generalized input, slice, token or string before and after the given one. This is described in Algorithm 4.

Algorithm 4: Overview of the input extension mutation.

```
Data: input is the current generalized input, generalized is the
      set of all previously generalized inputs, tokens and strings
      from the dictionary
rand ← random_generalized(generalized_inputs)
send_to_fuzzer(concat(input.content(),
rand.content()))
send_to_fuzzer(concat(rand.content(),
input.content()))
```

Example 6. *Assume that the current input is "*`pprint 'aaaa'`*" and its generalization is "*`pprint '□'`*". Furthermore, assume that we randomly choose a previous generalization "*`□x=□y+□`*". Then, we concretize their generalizations to "*`pprint '$$'`*" and "*`x=y+`*" by replacing remaining gaps with an empty string. Afterwards, we concatenate them and obtain "*`pprint '$$'x=y+`*" and "*`x=y+pprint '$$'`*".*

3.2.2 Recursive Replacement

The recursive replacement mutation recombines knowledge about the structured input language—that was obtained earlier in the fuzzing run—in a grammar-like manner. As illustrated in Algorithm 5, given a generalized input, we extend its beginning and end by □—if not yet present—such that we always can place other data before or behind the input. Afterwards, we randomly select $n \in \{2,4,8,16,32,64\}$ and perform the following operations n times: First, we randomly select another generalized input, input slice, token or string. Then, we call `replace_random_gap` which replaces an arbitrary □ in the first generalized input by the chosen element. Furthermore, we enforce □ before and after the replacement such that these □ can be subject to further replacements. Finally, we concretize the mutated input and send it to the fuzzer. The recursive replacement mutator has a (comparatively) high likelihood of producing new structurally interesting inputs compared to more small-scale mutations used by current coverage-guided fuzzers.

Example 7. *Assume that the current input is "*`pprint 'aaaa'`*". We take its generalization "*`pprint '□'`*" and extend it to "*`□pprint '□'□`*". Furthermore, assume that we already generalized the inputs "*`if(x>1)□then □end`*" and "*`□x=□y+□`*". In a first mutation, we choose to replace the first □ with the slice "*`if(x>1)□`*". We extend the slice to "*`□if(x>1)□`*" and obtain "*`□if(x>1)□pprint`

Algorithm 5: Overview of the recursive replacement mutation.

```
Data: input is the current generalized input, generalized is the
      set of all previously generalized inputs, tokens and strings
      from the dictionary
1 input ← pad_with_gaps(input)
2 for i ← 0 to random_power_of_two() do
3 |   rand ← random_generalized(generalized_inputs)
4 |   input ← replace_random_gap(input, rand)
5 send_to_fuzzer(input.content())
```

Algorithm 6: Overview of the string replacement mutation.

```
Data: input is the input string, strings is the provided dictionary
      obtained from the binary
1 sub ← find_random_substring(input, strings)
2 if sub then
3 |   rand ← random_string(strings)
4 |   data ← replace_random_instance(input, sub, rand)
5 |   send_to_fuzzer(data)
6 |   data ← replace_all_instances(input, sub, and)
7 |   send_to_fuzzer(data)
```

'□'□". *Afterwards, we choose to replace the third* □ *with "*□x=□y+□*" and obtain "*□if(x>1)□pprint '□x=□y+□'□*". In a final step, we replace the remaining* □ *with an empty string and obtain "*if(x>1)pprint 'x=y+'*".*

3.2.3 String Replacement

Keywords are important elements of structured input languages; changing a single keyword in an input can lead to completely different behavior. GRIMOIRE's string replacement mutation performs different forms of replacements, as described in Algorithm 6. Given an input, it locates all substrings in the input that match strings from the obtained dictionary and chooses one randomly. GRIMOIRE first selects a random occurrence of the matching substring and replaces it with a random string. In a second step, it replaces all occurrences of the substring with the same random string. Finally, the mutation sends both mutated inputs to the fuzzer. As an example, this mutation can be helpful to discover different methods of the same object by replacing a valid method call with different alternatives. Also, changing all occurrences of a substring allows us to perform more syntactically correct mutations, such as renaming of variables in the input.

Example 8. *Assume the "*if(x>1)pprint 'x=y+'*" and that the strings "if", "while", "key", "pprint", "eval", "+", "=" and "–" are in the dictionary. Thus, the string replacement mutation can generate inputs such as "*while(x>1)pprint 'x=y+'*", "*if(x>1)eval 'x+y+'*" or "*if(x>1)pprint 'x=y-'*". Furthermore, assume that the string "x" is also in the dictionary. Then, the string replacement mutation can replace all occurrences of the variable "*x*" in "*if(x>1)pprint 'x=y+'*" and obtain "*if(key>1)pprint 'key=y+'*".*

4 Implementation

To evaluate the algorithms introduced in this paper, we built a prototype implementation of our design. Our implementation, called GRIMOIRE, is based on REDQUEEN's [3] source code. This allows us to implement our techniques within a state-of-the-art fuzzing framework. REDQUEEN is applicable to both open and closed source targets running in user or kernel space, thus enabling us to target a wide variety of programs. While REDQUEEN is entirely focused on solving magic bytes and similar constructs which are local in nature (i. e., require only few bytes to change), GRIMOIRE assumes that this kind of constraints can be solved by the underlying fuzzer. It uses global mutations (that change large parts of the input) based on the examples that the underlying fuzzer finds. Since our technique is merely based on common techniques implemented in coverage-guided fuzzers—for instance, access to the execution bitmap—it would be a feasible engineering task to adapt our approach to other current fuzzers, such as AFL.

More precisely, GRIMOIRE is implemented as a set of patches to REDQUEEN. After finding new inputs, we apply the generalization instead of the minimization algorithm that was used by AFL and REDQUEEN. Additionally, we extended the havoc stage by large-scale mutations as explained in Section 3. To prevent GRIMOIRE from spending too much time in the generalization phase, we set a user-configurable upper bound; inputs whose length exceeds this bound are not be generalized. Per default, it is set to 16384 bytes. Overall, about 500 lines were written to implement the proposed algorithms.

To support reproducibility of our approach, we open source the fuzzing logic, especially the implementation of GRIMOIRE as well as its interaction with REDQUEEN at `https://github.com/RUB-SysSec/grimoire`.

5 Evaluation

We evaluate our prototype implementation GRIMOIRE to answer the following research questions.

RQ 1 How does GRIMOIRE compare to other state-of-the-art bug finding tools?

RQ 2 Is our approach useful even when proper grammars are available?

RQ 3 How does our approach improve the performance on targets that require highly structured inputs?

RQ 4 How does our approach perform compared to other grammar inference techniques for the purpose of fuzzing?

RQ 5 How do our mutators impact fuzzing performance?

RQ 6 Can GRIMOIRE identify new bugs in real-world applications?

To answer these questions, we perform three individual experiments. First, we evaluate the coverage produced by various fuzzers on a set of real-world target programs. In the second experiment, we analyze how our techniques can be combined with grammar-based fuzzers for mutual improvements. Finally, we use GRIMOIRE to uncover a set of vulnerabilities in real-world target applications.

5.1 Measurement Setup

All experiments are performed on an Ubuntu Server 16.04.2 LTS with an Intel i7-6700 processor with 4 cores and 24 GiB of RAM. Each tool is evaluated over 12 runs for 48 hours to obtain statistically meaningful results. In addition to other statistics, we also measure the effect size by calculating the difference in the median of the number of basic blocks found in each run. Additionally, we perform a Mann Whitney U test (using `scipy` 1.0 [38]) and report the resulting p-values. All experiments are performed with the tool being pinned to a dedicated CPU in single-threaded mode. Tools other than GRIMOIRE and REDQUEEN require source-code access; we use the fast *clang*-based instrumentation in these cases. Additionally, to ensure a fair evaluation, we provide each fuzzer with a dictionary containing the strings found inside of the target binary. In all cases, except NAUTILUS (which crashed on larger bitmaps), we increase the bitmap size from 2^{16} to 2^{19}. This is necessary since we observe more collisions in the global coverage map for large targets which causes the fuzzer to discard new coverage. For example, in `SQLite` (1.9 MiB), 14% of the global coverage map entries collide [66]. Since we deal with even larger binaries such as `PHP` which is nearly 19 MiB, the bitmap fills up quickly. Based on our empirical evaluation, we observed that 2^{19} is the smallest sufficient size that works for all of our target binaries.

Furthermore, we disable the so-called *deterministic stage* [66]. This is motivated by the observation that these deterministic mutations are not suited to find new coverage considering the nature of highly structured inputs. Finally—if not stated otherwise—we use the same uninformed seed that the authors of REDQUEEN used for their experiments: `"ABC...XYZabc...xyz012...789!"$...~+*"`.

As noted by Aschermann et al. [3], there are various definitions of a basic block. Fuzzers such as AFL change the number of basic blocks in a program. Thus, to enable a fair comparison in our experiments, we measure the coverage produced by each fuzzer on the same uninstrumented binary. Therefore, the numbers of basic blocks found and reported in our paper might differ from other papers. However, they are consistent within all of our experiments.

For our experiments, we select a diverse set of target programs. We use four scripting language interpreters (`mruby-1.4.1` [41], `php-7.3.0` [57], `lua-5.3.5` [36] and `JavaScriptCore, commit "f1312"` [1]) a compiler (`tcc-0.9.27` [6]), an assembler (`nasm-2.14.02` [56]), a database (`sqlite-3.25` [31]), a parser (`libxml-2.9.8` [59]) and an SMT solver (`boolector-3.0.1` [44]). We select these four scripting language interpreters so that we can directly compare the results to NAUTILUS. Note that our choice of targets is additionally governed by architectural limitations of REDQUEEN which GRIMOIRE is based on. REDQUEEN uses Virtual Machine Introspection (VMI) to transfer the target binary—including all of its dependencies—into the Virtual Machine (VM). The maximum transfer size using VMI in REDQUEEN is set to 64 MiB. Programs such as `Python` [49], `GCC` [18], `Clang` [40], `V8` [24] and `SpiderMonkey` [43] exceed our VMI limitation; thus, we can not evaluate them. We select an alternative set of target binaries that are large enough but at the same time do not exceed our 64 MiB transfer size limit. Hence, we choose `JavaScriptCore` over `V8` and `SpiderMonkey`, `mruby` over `ruby` and `TCC` over `GCC` or `Clang`. Finally, we tried to evaluate GRIMOIRE with `ChakraCore` [42]. However, `ChakraCore` fails to start inside of the REDQUEEN Virtual Machine for unknown reasons. Still, GRIMOIRE performs well on large targets such as `JavaScriptCore` and `PHP`.

5.2 State-of-the-Art Bug Finding Tools

To answer **RQ 1**, we perform 12 runs on eight targets using GRIMOIRE and four state-of-the-art bug finding tools. We choose AFL (version 2.52b) because it is a well-known fuzzer and a good baseline for our evaluation. We select QSYM (commit "6f00c3d") and ANGORA (commit "6ff81c6"), two state-of-the-art hybrid fuzzers which employ different program analysis techniques, namely symbolic execution and taint tracking. Finally, we choose REDQUEEN as a state-of-the-art coverage-guided fuzzer, which is also the baseline of GRIMOIRE. As a consequence, we are able to directly observe the improvements of our method. Note that we could not compile `libxml` for ANGORA instrumentation. Therefore, ANGORA is missing in the `libxml` plot.

The results of our coverage measurements are shown in Figure 4. As we can see, in all cases GRIMOIRE provides a significant advantage over the baseline (unmodified REDQUEEN). Surprisingly, in most cases, neither ANGORA, REDQUEEN, nor QSYM seem to have a significant edge over plain AFL. This can be explained by the fact that REDQUEEN and ANGORA mostly aim to overcome certain "magic byte" fuzzing roadblocks. Similarly, QSYM is also effective to solve these roadblocks. Since we provide a dictionary with strings from the target binary to each fuzzer, these roadblocks become much less common. Thus, the techniques introduced in ANGORA, REDQUEEN and QSYM are less relevant given the seeds provided to the fuzzers. However, in the case of `TCC`, we can observe that providing the strings dictionary does not help AFL. Therefore, we believe that ANGORA and REDQUEEN

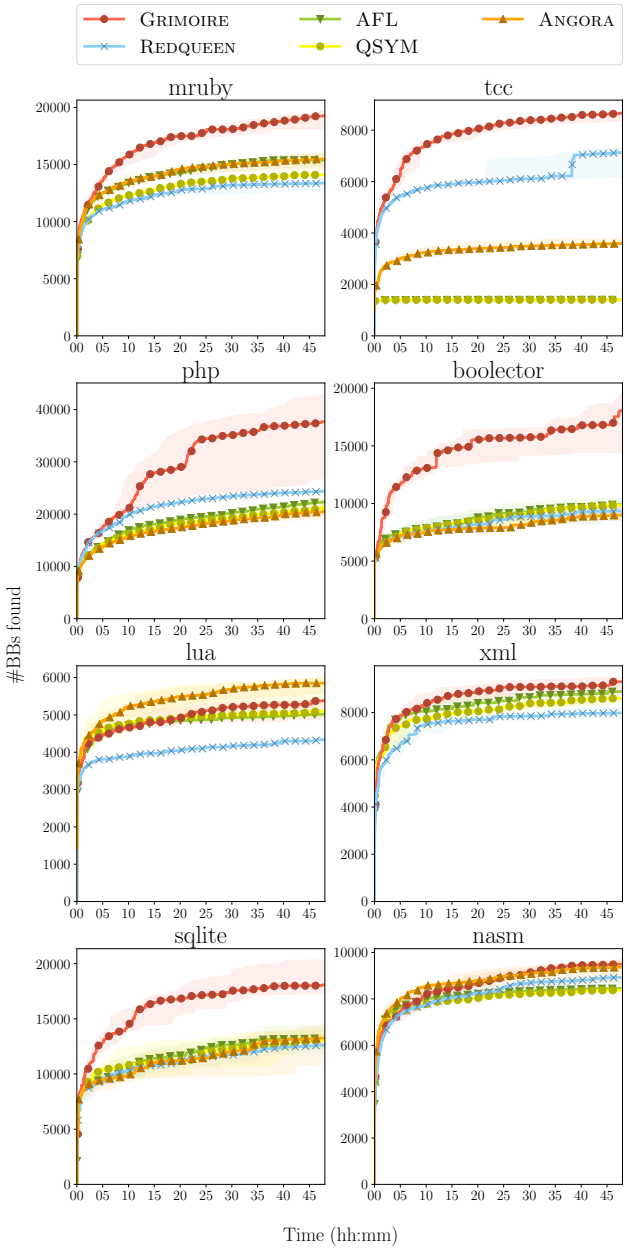


Figure 4: The coverage (in basic blocks) produced by various tools over 12 runs for 48h on various targets. Displayed are the median and the 66.7% intervals.

find strings that are not part of the dictionary and thus outperform AFL.

A complete statistical description of the results is given in the appendix (Table 7). We perform a confirmatory statistical analysis on the results, as shown in Table 2. The results show that in all but two cases (Lua and NASM), GRIMOIRE offers relevant and significant improvements over all state-of-the-art alternatives. On average, it finds nearly 20% more coverage than the second best alternative.

Table 2: Confirmatory data analysis of our experiments. We compare the coverage produced by GRIMOIRE against the best alternative. The effect size is the difference of the medians in basic blocks. In most experiments, the effect size is relevant and the changes are highly significant: it is typically multiple orders of magnitude smaller than the usual bound of $p < 5.0E\text{-}02$ (bold).

Target	Best Alternative	Effect Size ($\Delta = \tilde{A} - \tilde{B}$)	Effect Size in % of Best	p-value
mruby	ANGORA	3685	19.3%	**1.8E-05**
TCC	REDQUEEN	1952	22.6%	**7.8E-05**
PHP	REDQUEEN	11238	31.6%	**1.8E-05**
Boolector	AFL	7671	43.9%	**1.8E-05**
Lua	ANGORA	-478	-8.2%	**4.5E-04**
libxml	AFL	308	3.4%	**1.8E-02**
SQLite	ANGORA	4846	26.8%	**1.8E-05**
NASM	ANGORA	272	2.9%	9.7E-02

Lua accepts both source files (text) as well as byte code. GRIMOIRE can only make effective mutations in the domain of language features and not the bytecode. However, other fuzzers can perform on both; this is why ANGORA outperforms GRIMOIRE on this target. It is worth mentioning that GRIMOIRE outperforms REDQUEEN, the baseline on top of which our approach is implemented.

To partially answer **RQ 1**, we showed that in terms of *code coverage*, GRIMOIRE outperforms other state-of-the-art bug finding tools (in most cases). Second, to answer **RQ 3**, we demonstrated that GRIMOIRE significantly improves the performance on targets with highly structured inputs when compared to our baseline (REDQUEEN).

5.3 Grammar-based Fuzzers

Generally, we expect grammar-based fuzzers to have an edge over grammar inference fuzzers like GRIMOIRE since they have access to a manually crafted grammar. To quantify this advantage, we evaluate GRIMOIRE against current grammar-based fuzzers. To this end, we choose NAUTILUS (commit "dd3554a"), a state-of-the-art coverage-guided fuzzer, since it can fuzz a wide variety of targets if provided with a hand-written grammar. We evaluate on the targets used in NAUTILUS' experiments, mruby, PHP and Lua, as their grammars are available. Unfortunately, GRIMOIRE is not capable of running ChakraCore, the fourth target NAUTILUS was evaluated on; thus, we replace it by JavaScriptCore and use NAUTILUS' JavaScript grammar. We observed that the original version of NAUTILUS had some timeout problems during fuzzing where the timeout detection did not work properly. We fixed this for our evaluation.

For each of the four targets, we perform an experiment with the same setup as the first experiment (again, 12 runs for 48 hours). The results are shown in Figure 5. As expected, our completely automated method is defeated in most cases by NAUTILUS since it uses manually fine-tuned grammars.

Surprisingly, in the case of `mruby`, we find that GRIMOIRE is able to outperform even NAUTILUS.

To evaluate whether GRIMOIRE is still useful in scenarios where a grammar is available, we perform another experiment. We extract the corpus produced by NAUTILUS after half of the time (i. e., 24 hours) and continue to use GRIMOIRE for another 24 hours using this seed corpus. For these *incremental runs*, we reduce GRIMOIRE's upper bound for input generalization to 2,048 bytes; otherwise, our fuzzer would mainly spend time in the generalization phase since NAUTILUS produces very large inputs. The results are displayed in Figure 5 (incremental). This experiment demonstrates that even despite manual fine-tuning, the grammar often contains blind spots, where an automated approach such as ours can infer the implicit structure which the program expects. This structure may be quite different from the specified grammar. As Figure 5 shows, by using the corpus created by NAUTILUS, GRIMOIRE surpasses NAUTILUS individually in all cases (**RQ 2**). A confirmatory statistical analysis of the results is presented in Table 3. In three cases, GRIMOIRE is able to improve upon hand written grammars by nearly 10%.

Table 3: Confirmatory data analysis of our experiment. We compare the coverage produced by GRIMOIRE against NAUTILUS with hand written grammars. The effect size is the difference of the medians in basic blocks in the incremental experiment. In three experiments, the effect size is relevant and the changes are highly significant (marked bold, $p < 5.0\text{E-}02$). Note that we abbreviate `JavaScriptCore` with `JSC`.

Target	Best Alternative	Effect Size ($\Delta = \bar{A} - \bar{B}$)	Effect Size in % of Best	p-value
`mruby`	NAUTILUS	2025	10.0%	**1.8E-05**
`Lua`	NAUTILUS	553	5.2%	5.0E-02
`PHP`	NAUTILUS	5465	9.3%	**3.6E-03**
`JSC`	NAUTILUS	15445	11.0%	**1.8E-05**

Additionally, we intended to compare GRIMOIRE against CODEALCHEMIST and JSFUNFUZZ, two other state-of-the art grammar-based fuzzers which specialize on JavaScript engines. Although these two fuzzers are not coverage-guided—making a fair evaluation challenging—we consider the comparison of specialized JavaScript grammar-based fuzzers to general-purpose grammar-based fuzzers as interesting. Unfortunately, JSFUNFUZZ was not working with `JavaScriptCore` out of the box as it is specifically tailored to `SpiderMonkey`. Since it requires significant modifications to run on `JavaScriptCore`, we considered the required engineering effort to be out of scope for this paper. On the other hand, CODEALCHEMIST requires an extensive seed corpus of up to 60,000 valid JavaScript files—which were not released together with the source files. We tried to replicate the seed corpus as described by the authors of CODEALCHEMIST. However, despite the authors' kind help, we were unable to run CODEALCHEMIST with our corpus.

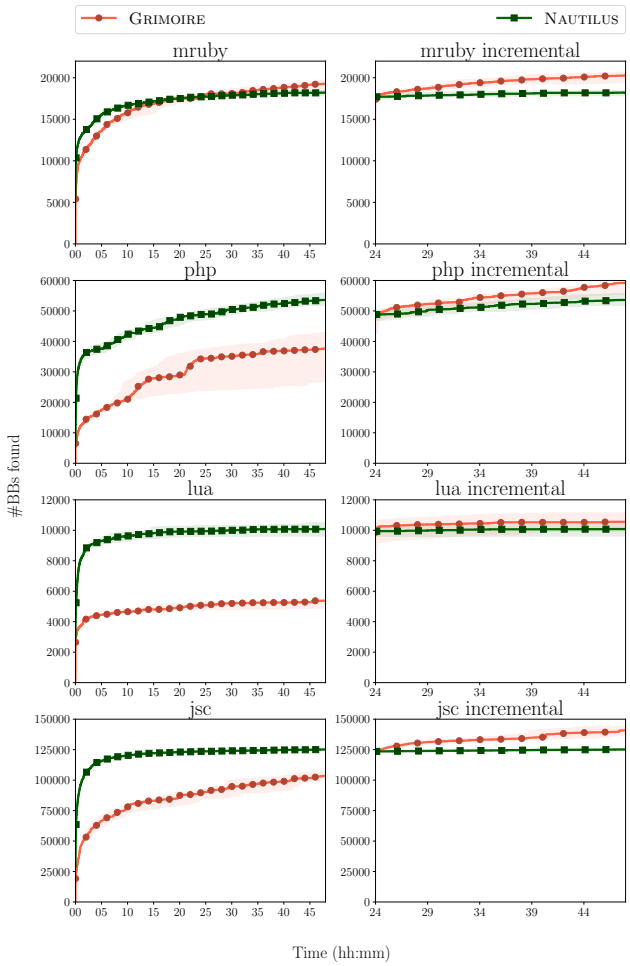


Figure 5: The coverage (in basic blocks) produced by GRIMOIRE and NAUTILUS (using the hand written grammars of the authors of NAUTILUS) over 12 runs at 48 h on various targets. The incremental plots show how running NAUTILUS for 48h compares to running NAUTILUS for the first 24h and then continue fuzzing for 24h with GRIMOIRE. Displayed are the median and the 66.7% confidence interval.

Overall, these experiments confirm our assumption that grammar-based fuzzers such as NAUTILUS have an edge over grammar inference fuzzers like GRIMOIRE. However, deploying our approach on top of a grammar-based fuzzer (incremental runs) increases code coverage. Therefore, we partially respond to **RQ 1** and provide an answer to **RQ 2** by stating that GRIMOIRE is a valuable addition to current fuzzing techniques.

5.4 Grammar Inference Techniques

To answer **RQ 4**, we compare our approach to other grammar inference techniques in the context of fuzzing. Existing work in this field includes GLADE, AUTOGRAM and PYGMALION. However, since PYGMALION targets only Python and AUTOGRAM only Java programs, we cannot evaluate

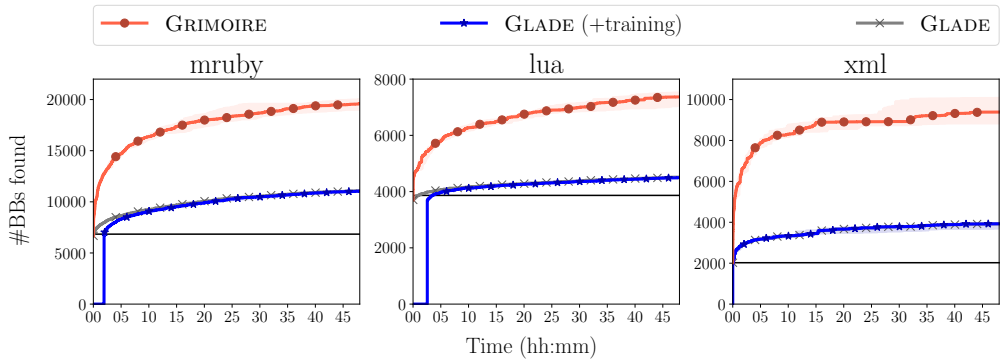


Figure 6: Comparing GRIMOIRE against GLADE (median and 66.7% interval). In the plot for GLADE +Training, we include the training time that glade used. For comparison, we also include plots where we omit the training time. The horizontal bar displays the coverage produced by the seed corpus that GLADE used during training.

them as GRIMOIRE only supports targets that can be traced with Intel-PT (since REDQUEEN heavily depends on it).

Therefore, for this evaluation, we use GLADE (commit "b9ef32e"), a state-of-the-art grammar inference tool. It operates in two stages. Given a program as black-box oracle as well as a corpus of valid input samples, it learns a grammar in the first stage. In the second stage, GLADE uses this grammar to produce inputs that can be used for fuzzing. GLADE does not generate a continuous stream of inputs, hence we modified it to provide such capability. We then use these inputs to measure the coverage achieved by GLADE in comparison to GRIMOIRE. Note that due to the excessive amount of inputs produced by GLADE, we use a corpus minimization tool—`afl-cmin`—to identify and remove redundant inputs before measuring the coverage [66].

Note, we have to extend GLADE for each target that is not natively supported and must manually create a valid seed corpus. For this reason, we restrict ourselves to the three targets `libxml`, `mruby` and `Lua`. From these, `libxml` is the only one that was also used in GLADE's evaluation. Therefore, we are able to re-use their provided corpus for this target. We choose the other two since we want to achieve comparability with regards to previous experiments.

To allow for a fair comparison, we provide the same corpus to GRIMOIRE. Again, we repeat all experiments 12 times for 48 hours each. The results of this comparison are depicted in Figure 6. Note that this figure includes two different experiments of GLADE. In the first experiment, we include the time GLADE spent on training into the measurement while for the second measurement, GLADE is provided the advantage of concluding the training stage before measurement is started for the fuzzing process. As can be seen in Figure 6, GRIMOIRE significantly outperforms GLADE on all targets for both experiments. Similar to earlier experiments, we perform a confirmatory statistical analysis. The results are displayed in Table 4; they are in all cases relevant and statistically significant. If we consider only the new coverage found (beyond what is already contained in the training set), we are able to outperform GLADE by factors from two to five. We therefore conclude in response to **RQ 4** that we significantly exceed comparative grammar inference approaches in the context of fuzzing.

We designed another experiment to evaluate whether GLADE's automatically inferred grammar can be used for NAUTILUS and how it performs compared to hand written grammars. However, GLADE does not use the grammar directly but remembers how the grammar was produced from the provided test cases and uses the grammar only to apply local mutations to the input. Unfortunately, as a consequence, their grammar contains multiple unproductive rules, thus preventing their usage in NAUTILUS.

Table 4: Confirmatory data analysis of our experiments. We compare the coverage produced by GRIMOIRE against GLADE. The effect size is the difference of the medians in basic blocks. In all experiments, the effect size is relevant and the changes are highly significant: it is multiple orders of magnitude smaller than the usual bound of p < 5.0E-02 (bold).

Target	Best Alternative	Effect Size ($\Delta = \bar{A} - \bar{B}$)	Effect Size in % of Best	p-value
`mruby`	GLADE	8546	43.6%	**9.1E-05**
`Lua`	GLADE	2775	38.1%	**9.1E-05**
`libxml`	GLADE	5213	57.2%	**9.1E-05**

5.5 Mutations Statistic

During the aforementioned experiments, we also collected various statistics on how effective different mutators are. We measured how much time was spent using GRIMOIRE's different mutation strategies as well as how many of the inputs were found by each strategy. This allows us to rank mutation strategies based on the number of new paths found per time used. The strategies include a havoc stage, REDQUEEN's Input-to-State-based mutation stage and our structural mutation stage. The times for our structural mutators include the

generalization process (including the necessary minimization that also benefits the other mutators).

As Table 5 shows, our structural mutators are competitive with other mutators, which answers **RQ 5**. As the coverage results in Figure 4 show, the mutators are also able to uncover paths that would not have been found otherwise.

Table 5: Statistics for each of GRIMOIRE's mutation strategies (i. e., our structured mutations, REDQUEEN's Input-to-State-based mutations and havoc). For every target evaluated we list the total number of inputs found by a mutation, the time spent on this strategy and the ratio of inputs found per minute.

Mutation	Target	#Inputs	Time Spent (min)	#Inputs/Min
Structured	mruby	9040	1531.18	5.90
	PHP	27063	2467.17	**10.97**
	Lua	2849	2064.49	1.38
	SQLite	5933	1325.26	**4.48**
	TCC	6618	2271.03	2.91
	Boolector	3438	2399.85	1.43
	libxml	4883	2001.38	2.44
	NASM	12696	1955.42	**6.49**
	JavaScriptCore	38465	2460.95	**15.63**
Input-to-State	mruby	814	268.23	3.03
	PHP	902	111.46	8.09
	Lua	530	307.12	1.73
	SQLite	603	768.72	0.78
	TCC	1020	118.23	**8.63**
	Boolector	325	102.87	**3.16**
	libxml	967	359.03	2.69
	NASM	1329	213.84	6.22
	JavaScriptCore	400	82.76	4.83
Havoc	mruby	2010	339.03	**5.93**
	PHP	2546	278.21	9.15
	Lua	1684	492.99	**3.42**
	SQLite	1827	742.13	2.46
	TCC	2514	484.73	5.19
	Boolector	956	373.85	2.56
	libxml	2173	504.86	**4.30**
	NASM	2876	678.59	4.24
	JavaScriptCore	3800	279.62	13.59

5.6 Real-World Bugs

We use GRIMOIRE on a set of different targets to observe whether it is able to uncover previously unknown bugs (**RQ 6**). To this end, we manually triaged bugs found during our evaluation. As illustrated in Table 6, GRIMOIRE found more bugs than all other tools in the evaluation combined. We responsibly disclosed all of them to the vendors. For these, 11 CVEs were assigned. Note that we found a large number of bugs that did not lead to assigned CVEs. This is partially because projects such as PHP do not consider invalid inputs as security relevant, even when custom scripts can trigger memory corruption. We conclude **RQ 6** by finding that GRIMOIRE is indeed able to uncover novel bugs in real-world applications.

Table 6: Overview of submitted bugs and CVEs. Fuzzers which did not find the bug during our evaluation are denoted by ✗, while those who did are marked by ✓. We indicate targets not evaluated by a specific fuzzer with '-'. We abbreviate Use-After-Free (UAF), Out-of-Bounds (OOB) and Buffer Overflow (BO).

Target	CVE	Type	GRIMOIRE	REDQUEEN	AFL	QSYM	ANGORA	NAUTILUS
PHP		OOB-write	✓	✗	✗	✗	✗	✓
PHP		OOB-read	✓	✗	✗	✓	✓	✗
PHP		OOB-read	✓	✗	✗	✗	✗	✓
PHP		OOB-read	✓	✗	✗	✗	✗	✗
TCC	2018-20374	OOB-write	✓	✗	✗	✗	✗	-
TCC	2018-20375	OOB-write	✓	✓	✗	✗	✗	-
TCC	2018-20376	OOB-write	✓	✓	✗	✗	✗	-
TCC	2019-12495	OOB-write	✓	✗	✗	✗	✗	-
TCC	2019-9754	OOB-write	✓	✓	✗	✗	✗	-
TCC		OOB-write	✗	✓	✗	✗	✗	-
Boolector	2019-7559	OOB-write	✓	✗	✗	✗	✗	-
Boolector	2019-7560	UAF-write	✓	✗	✗	✗	✗	-
NASM	2019-8343	UAF-write	✓	✓	✗	✗	✗	-
NASM		OOB-write	✓	✗	✓	✗	✗	-
NASM		OOB-write	✓	✗	✗	✗	✗	-
NASM		OOB-write	✓	✗	✗	✗	✗	-
NASM		OOB-write	✓	✗	✓	✗	✗	-
NASM		OOB-write	✗	✗	✓	✗	✗	-
gnuplot	2018-19490	BO	✓	-	-	-	-	-
gnuplot	2018-19491	BO	✓	-	-	-	-	-
gnuplot	2018-19492	BO	✓	-	-	-	-	-

6 Discussion

The methods introduced in this paper produce significant performance gains on targets that expect highly structured inputs without requiring any expert knowledge or manual work. As we have shown, GRIMOIRE can also be used to support grammar-based fuzzers with well-tuned grammars but cannot outperform them on their own. In contrast to similar methods, our approach does not rely on complex primitives such as symbolic execution or taint tracking. Therefore, it can easily be integrated into existing fuzzers. Additionally, since GRIMOIRE is based on REDQUEEN, it can be used on a wide variety of binary-only targets, ranging from userland programs to operating system kernels.

Despite all advantages, our approach has significant difficulties with more syntactically complex constructs, such as matching the ID of opening and closing tags in XML or identifying variable constructs in scripting languages. For instance, while GRIMOIRE is able to produce nested inputs such as "<a><a><a>F00", it struggles to generalize "<a>□" to the more unified representation "<$\boxed{A}$>□</$\boxed{B}$>" with the constraint $A = B$. A solution for such complex constructs could be the following generalization heuristic: (i) First, we record the new coverage for the current input. (ii) We then change only a single occurrence of a substring in our input and record its new coverage. For instance, consider that we replace a single occurrence of "a" by "b" in "<a><a><a>F00" and obtain "<b><a><a>F00". This change results in an invalid XML tag which leads to different coverage compared to the one observed in (i). (iii) Finally, we change multiple instances of the same substring and compare the new coverage of the modified input with the one obtained in (i). If we

achieved the same new coverage in (iii) and (i), we can assume that the modified instances of the same substring are related to each other. For example, we replace multiple occurrences of "a" with "b" and obtain "<a><a>F00". In this example, the coverage is the same as for the original input since the XML remains syntactically correct.

Similarly, our generalization approach might be too coarse in many places. Obtaining more precise rules would help uncovering deeper parts of the target application in cases where multiple valid statements have to be produced. Consider, for instance, a scripting language interpreter such as the ones used in our evaluation. Certain operations might require a number of constructors to be successfully called. For example, it might be necessary to get a valid path object to obtain a file object that can finally be used to perform a read operation. A more precise representation would be highly useful in such cases. One could try to infer whether a combination is "valid" by checking if the combination of two inputs exercises the combination of the new coverage introduced by both inputs. For instance, assume that input "a□b" triggers the coverage bytes 7 and 10 and that input "□=□" triggers coverage byte 20. Then, a combination of these two inputs such as "□a□=□b" could trigger the coverage bytes 7, 10 and 20. Using this information, it might be possible to infer more precise grammar descriptions and thus generate inputs that are closer to the target's semantics than it is currently possible in GRIMOIRE. While this approach would most likely further reduce the gap between hand-written grammars and inferred grammars, well-designed hand-written grammars will always have an edge over fuzzers with no prior knowledge: any kind of inference algorithm first needs to uncover structures before the obtained knowledge can be used. A grammar-based fuzzer has no such disadvantage. If available, human input can improve the results of grammar inference or steer its direction. An analyst can provide a partial grammar to make the grammar-fuzzer focus on a specific interesting area and avoid exploring paths that are unlikely to contain bugs. Therefore, GRIMOIRE is useful if the grammar is unknown or under-specified but cannot be considered a full replacement for grammar-based fuzzers.

7 Related Work

A significant number of approaches to improve the performance of different fuzzing strategies has been proposed over time. Early on, fuzzers typically did not observe the inner workings of the target application, yet different approaches were proposed to improve various aspects of fuzzers: different mutation strategies were evaluated [14, 29], the process of selecting and scheduling of seed inputs was analyzed [11,51,61] and, in some cases, even learned language models were used to improve the effectiveness of fuzzing [22, 27]. After the publication of AFL [65], the research focus shifted towards coverage-guided fuzzing techniques. Similarly to the previous work on blind fuzzing, each individual component of AFL was put under scrutiny. For example, AFLFAST [8] and AFLGo [7] proposed scheduling mechanisms that are better suited to some circumstances. Both, COLLAFL [16] and `InsTrim` [35], enhanced the way in which coverage is generated and stored to reduce the amount of memory needed. Other publications improved the ways in which coverage feedback is collected [23, 53, 55, 62]. To advance the ability of fuzzers to overcome constraints that are hard to guess, a wide array of techniques were proposed. Commonly, different forms of symbolic execution are used to solve these challenging instances [9, 10]. In most of these cases, a restricted version of symbolic execution (concolic execution) is used [19–21, 26, 54, 60]. To further improve upon these techniques, DigFuzz [67] provides a better scheduling for inputs to the symbolic executor. Sometimes, instead of using these heavy-weight primitives, more lightweight techniques such as taint tracking [12, 17, 26, 50], patches [3, 13, 47, 60] or instrumentation [3, 39] are used to overcome the same hurdles.

While these improvements generally work very well for binary file formats, many modern target programs work with highly structured data. To target these programs, generational fuzzing is typically used. In such scenarios, the user can often provide a grammar. In most cases, fuzzers based on this technique are blind fuzzers [14, 33, 45, 52, 63].

Recent projects such as AFLSMART [48], NAUTILUS [2] and ZEST [46] combined the ideas of generational fuzzing with coverage guidance. CODEALCHEMIST [28] even ventures beyond syntactical correctness. To find novel bugs in mature JavaScript interpreters, it tries to automatically craft syntactically and semantically valid inputs by recombining input fragments based on inferred types of variables. All of these approaches require a good format specification and—in some cases—good seed corpora. CODEALCHEMIST even needs access to a specialized interpreter for the target language to trace and infer type annotations. In contrast, our approach has no such preconditions and is thus easily integrable into most fuzzers.

Finally, to alleviate some of the disadvantages that the mentioned grammar-based strategies have, multiple approaches were developed to automatically infer grammars for given programs. GLADE [5] can systematically learn an approximation to the context-free grammars parsed by a program. To learn the grammar, it needs an oracle that can answer whether a given input is valid or not as well as a small set of valid inputs. Similar techniques are used by PYGMALION [25] and AUTOGRAM [34]. However, both techniques directly learn from the target application without requiring a modified version of the target. AUTOGRAM still needs a large set of inputs to trace, while PYGMALION can infer grammars based solely on the target application. Additionally, both approaches require complex analysis passes and even symbolic execution to produce grammars. These techniques cannot easily be scaled

to large binary applications. Finally, all three approaches are computationally expensive.

8 Conclusion

We developed and demonstrated the first fully automatic algorithm that integrates large-scale structural mutations into the fuzzing process. In contrast to other approaches, we need no additional modifications or assumptions about the target application. We demonstrated the capabilities of our approach by evaluating our implementation called GRIMOIRE against various state-of-the-art coverage-guided fuzzers. Our evaluation shows that we outperform other coverage-guided fuzzers both in terms of coverage and the number of bugs found. From this observation, we conclude that it is possible to significantly improve the fuzzing process in the absence of program input specifications. Furthermore, we conclude that even when a program input specification is available, our approach is still useful when it is combined with a generational fuzzer.

Acknowledgements

We would like to thank our shepherd Deian Stefan and the anonymous reviewers for their valuable comments and suggestions. Furthermore, we would like to thank Moritz Contag, Thorsten Eisenhofer, Joel Frank, Philipp Görz and Maximilian Golla for their valuable feedback. This work was supported by the German Research Foundation (DFG) within the framework of the Excellence Strategy of the Federal Government and the States - EXC 2092 CASA. In addition, this project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 786669 (ReAct). This paper reflects only the authors' view. The Research Executive Agency is not responsible for any use that may be made of the information it contains.

References

[1] APPLE INC. JavaScriptCore. https://github.com/WebKit/webkit/tree/master/Source/JavaScriptCore.

[2] ASCHERMANN, C., FRASSETTO, T., HOLZ, T., JAUERNIG, P., SADEGHI, A.-R., AND TEUCHERT, D. Nautilus: Fishing for deep bugs with grammars. In *Symposium on Network and Distributed System Security (NDSS)* (2019).

[3] ASCHERMANN, C., SCHUMILO, S., BLAZYTKO, T., GAWLIK, R., AND HOLZ, T. REDQUEEN: Fuzzing with input-to-state correspondence. In *Symposium on Network and Distributed System Security (NDSS)* (2019).

[4] BASTANI, O., SHARMA, R., AIKEN, A., AND LIANG, P. Synthesizing program input grammars. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2017).

[5] BASTANI, O., SHARMA, R., AIKEN, A., AND LIANG, P. Synthesizing program input grammars. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2017).

[6] BELLARD, F. TCC: Tiny C compiler. https://bellard.org/tcc/.

[7] BÖHME, M., PHAM, V.-T., NGUYEN, M.-D., AND ROYCHOUDHURY, A. Directed greybox fuzzing. In *ACM Conference on Computer and Communications Security (CCS)* (2017).

[8] BÖHME, M., PHAM, V.-T., AND ROYCHOUDHURY, A. Coverage-based greybox fuzzing as Markov chain. In *ACM Conference on Computer and Communications Security (CCS)* (2016).

[9] CADAR, C., DUNBAR, D., AND ENGLER, D. R. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symposium on Operating Systems Design and Implementation (OSDI)* (2008).

[10] CHA, S. K., AVGERINOS, T., REBERT, A., AND BRUMLEY, D. Unleashing Mayhem on binary code. In *IEEE Symposium on Security and Privacy* (2012).

[11] CHA, S. K., WOO, M., AND BRUMLEY, D. Program-adaptive mutational fuzzing. In *IEEE Symposium on Security and Privacy* (2015).

[12] CHEN, P., AND CHEN, H. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy* (2018).

[13] DREWRY, W., AND ORMANDY, T. Flayer: Exposing application internals. In *Proceedings of the first USENIX workshop on Offensive Technologies* (2007), USENIX Association.

[14] EDDINGTON, M. Peach fuzzer: Discover unknown vulnerabilities. https://www.peach.tech/.

[15] FREE SOFTWARE FOUNDATION. GNU Bison. https://www.gnu.org/software/bison/.

[16] GAN, S., ZHANG, C., QIN, X., TU, X., LI, K., PEI, Z., AND CHEN, Z. CollAFL: Path sensitive fuzzing. In *IEEE Symposium on Security and Privacy* (2018).

[17] GANESH, V., LEEK, T., AND RINARD, M. Taint-based directed whitebox fuzzing. In *International Conference on Software Engineering (ICSE)* (2009).

[18] GNU PROJECT. GCC, the GNU compiler collection. https://gcc.gnu.org/.

[19] GODEFROID, P., KIEZUN, A., AND LEVIN, M. Y. Grammar-based whitebox fuzzing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2008).

[20] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: Directed automated random testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2005).

[21] GODEFROID, P., LEVIN, M. Y., MOLNAR, D. A., ET AL. Automated whitebox fuzz testing. In *Symposium on Network and Distributed System Security (NDSS)* (2008).

[22] GODEFROID, P., PELEG, H., AND SINGH, R. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (2017), pp. 50–59.

[23] GOODMAN, P. Shin GRR: Make fuzzing fast again. https://blog.trailofbits.com/2016/11/02/shin-grr-make-fuzzing-fast-again/.

[24] GOOGLE LLC. V8. https://v8.dev/.

[25] GOPINATH, R., MATHIS, B., HÖSCHELE, M., KAMPMANN, A., AND ZELLER, A. Sample-free learning of input grammars for comprehensive software fuzzing. *arXiv preprint arXiv:1810.08289* (2018).

[26] HALLER, I., SLOWINSKA, A., NEUGSCHWANDTNER, M., AND BOS, H. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *USENIX Security Symposium* (2013).

[27] HAN, H., AND CHA, S. K. IMF: Inferred model-based fuzzer. In *ACM Conference on Computer and Communications Security (CCS)* (2017).

[28] HAN, H., OH, D., AND CHA, S. K. CodeAlchemist: Semantics-aware code generation to find vulnerabilities in JavaScript engines. In *Symposium on Network and Distributed System Security (NDSS)* (2019).

[29] HELIN, A. A general-purpose fuzzer. https://github.com/aoh/radamsa.

[30] HEX-RAYS. IDA pro. https://www.hex-rays.com/products/ida/.

[31] HIPP, D. R. SQLite. https://www.sqlite.org/index.html.

[32] HOCEVAR, S. zzuf. https://github.com/samhocevar/zzuf.

[33] HOLLER, C., HERZIG, K., AND ZELLER, A. Fuzzing with code fragments. In *USENIX Security Symposium* (2012).

[34] HÖSCHELE, M., AND ZELLER, A. Mining input grammars from dynamic taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (2016).

[35] HSU, C.-C., WU, C.-Y., HSIAO, H.-C., AND HUANG, S.-K. INSTRIM: Lightweight instrumentation for coverage-guided fuzzing. In *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research* (2018).

[36] IERUSALIMSCHY, R., CELES, W., AND DE FIGUEIREDO, L. H. Lua. https://www.lua.org/.

[37] JOHNSON, S. Yacc: Yet another compiler-compiler. http://dinosaur.compilertools.net/yacc/.

[38] JONES, E., OLIPHANT, T., AND PETERSON, P. Scipy: Open source scientific tools for Python. http://www.scipy.org/, 2001–.

[39] LI, Y., CHEN, B., CHANDRAMOHAN, M., LIN, S.-W., LIU, Y., AND TIU, A. Steelix: Program-state based binary fuzzing. In *Joint Meeting on Foundations of Software Engineering* (2017).

[40] LLVM PROJECT. Clang: a C language family frontend for LLVM. https://clang.llvm.org/.

[41] MATSUMOTO, Y. mruby. http://mruby.org/.

[42] MICROSOFT. ChakraCore. https://github.com/Microsoft/ChakraCore.

[43] MOZILLA FOUNDATION / MOZILLA CORPORATION. SpiderMonkey. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey.

[44] NIEMETZ, A., PREINER, M., AND BIERE, A. Boolector 2.0 system description. *Journal on Satisfiability, Boolean Modeling and Computation 9* (2015), 53–58.

[45] OPENRCE. Sulley: A pure-python fully automated and unattended fuzzing framework. https://github.com/OpenRCE/sulley.

[46] PADHYE, R., LEMIEUX, C., SEN, K., PAPADAKIS, M., AND TRAON, Y. L. Zest: Validity fuzzing and parametric generators for effective random testing. *arXiv preprint arXiv:1812.00078* (2018).

[47] PENG, H., SHOSHITAISHVILI, Y., AND PAYER, M. T-Fuzz: fuzzing by program transformation. In *IEEE Symposium on Security and Privacy* (2018).

[48] PHAM, V.-T., BÖHME, M., SANTOSA, A. E., CĂCIULESCU, A. R., AND ROYCHOUDHURY, A. Smart greybox fuzzing, 2018.

[49] PYTHON SOFTWARE FOUNDATION. Python. https://www.python.org/.

[50] RAWAT, S., JAIN, V., KUMAR, A., COJOCAR, L., GIUFFRIDA, C., AND BOS, H. VUzzer: Application-aware evolutionary fuzzing. In *Symposium on Network and Distributed System Security (NDSS)* (Feb. 2017).

[51] REBERT, A., CHA, S. K., AVGERINOS, T., FOOTE, J. M., WARREN, D., GRIECO, G., AND BRUMLEY, D. Optimizing seed selection for fuzzing. In *USENIX Security Symposium* (2014).

[52] RUDERMAN, J. Introducing jsfunfuzz. http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz (2007).

[53] SCHUMILO, S., ASCHERMANN, C., GAWLIK, R., SCHINZEL, S., AND HOLZ, T. kAFL: Hardware-assisted feedback fuzzing for OS kernels. In *USENIX Security Symposium* (2017).

[54] STEPHENS, N., GROSEN, J., SALLS, C., DUTCHER, A., WANG, R., CORBETTA, J., SHOSHITAISHVILI, Y., KRUEGEL, C., AND VIGNA, G. Driller: Augmenting fuzzing through selective symbolic execution. In *Symposium on Network and Distributed System Security (NDSS)* (2016).

[55] SWIECKI, R. Security oriented fuzzer with powerful analysis options. https://github.com/google/honggfuzz.

[56] THE NASM DEVELOPMENT TEAM. NASM. https://www.nasm.us/.

[57] THE PHP GROUP. PHP. http://php.net/.

[58] VEGGALAM, S., RAWAT, S., HALLER, I., AND BOS, H. IFuzzer: An evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security (ESORICS)* (2016), pp. 581–601.

[59] VEILLARD, DANIEL. The XML C parser and toolkit of Gnome. http://xmlsoft.org/.

[60] WANG, T., WEI, T., GU, G., AND ZOU, W. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy* (2010).

[61] WOO, M., CHA, S. K., GOTTLIEB, S., AND BRUMLEY, D. Scheduling black-box mutational fuzzing. In *ACM Conference on Computer and Communications Security (CCS)* (2013).

[62] XU, W., KASHYAP, S., MIN, C., AND KIM, T. Designing new operating primitives to improve fuzzing performance. In *ACM Conference on Computer and Communications Security (CCS)* (2017).

[63] YANG, X., CHEN, Y., EIDE, E., AND REGEHR, J. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Notices* (6 2011), vol. 46, ACM, pp. 283–294.

[64] YUN, I., LEE, S., XU, M., JANG, Y., AND KIM, T. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *USENIX Security Symposium* (2018), pp. 745–761.

[65] ZALEWSKI, M. american fuzzy lop. http://lcamtuf.coredump.cx/afl/.

[66] ZALEWSKI, M. Technical "whitepaper" for afl-fuzz. http://lcamtuf.coredump.cx/afl/technical_details.txt.

[67] ZHAO, L., DUAN, Y., YIN, H., AND XUAN, J. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *Symposium on Network and Distributed System Security (NDSS)* (2019).

A Statistics on Basic Block Coverage

Table 7: Statistics on basic block coverage for tested fuzzers. In the column "Best Coverage", we provide the highest number of basic blocks a run found and the percentage relative to the number of basic blocks obtained from `IDA Pro [30]`.

Target	Best Coverage (#BBS / %)	Fuzzer	Mean (%)	Median (%)	**Median (#BBs)**	Std Deviation	Skewness	Kurtosis
`mruby`	20258 / 70.5%	GRIMOIRE	66.1%	66.6%	19137	4.55	−0.54	−0.76
		AFL	53.7%	53.4%	15355	4.28	0.14	−0.27
		ANGORA	53.3%	53.8%	15452	4.87	0.17	−0.96
		QSYM	49.2%	49.0%	14084	2.20	0.33	0.95
		REDQUEEN	45.9%	46.4%	13339	4.64	−0.98	0.05
`TCC`	9211 / 77.6%	GRIMOIRE	71.8%	72.9%	8647	5.71	−1.89	3.68
		AFL	11.8%	11.8%	1397	3.80	1.27	1.14
		ANGORA	31.0%	30.3%	3600	6.51	1.01	0.06
		QSYM	11.9%	11.8%	1403	3.26	1.52	2.59
		REDQUEEN	56.7%	56.4%	6695	8.13	0.03	−1.93
`PHP`	46805 / 27.9%	GRIMOIRE	20.8%	21.2%	35606	20.26	0.12	−1.38
		AFL	13.2%	13.3%	22323	3.64	−0.09	−0.96
		ANGORA	12.1%	12.2%	20501	6.39	−0.37	−0.58
		QSYM	12.7%	12.7%	21276	2.60	0.22	−1.11
		REDQUEEN	14.5%	14.5%	24367	1.87	0.37	−0.83
`Boolector`	23207 / 33.1%	GRIMOIRE	25.2%	24.9%	17461	16.77	0.51	−0.65
		AFL	14.0%	14.0%	9790	7.46	0.30	−0.57
		ANGORA	13.2%	12.8%	8986	9.20	0.79	−0.17
		QSYM	13.7%	14.0%	9782	6.94	−0.39	−1.24
		REDQUEEN	13.3%	13.3%	9305	9.63	0.21	−1.23
`Lua`	6205 / 64.1%	GRIMOIRE	54.4%	55.2%	5339	6.47	0.20	−0.73
		AFL	51.9%	51.9%	5016	1.61	0.84	−0.15
		ANGORA	59.9%	60.1%	5817	2.96	0.05	−1.39
		QSYM	54.8%	52.6%	5091	9.52	1.07	−0.65
		REDQUEEN	44.5%	44.4%	4299	2.30	−0.30	−1.19
`libxml`	10437 / 13.2%	GRIMOIRE	11.7%	11.6%	9190	5.52	0.98	0.02
		AFL	11.1%	11.2%	8881	3.40	−0.39	−0.92
		ANGORA	0.0%	0.0%	0	nan	0.00	−3.00
		QSYM	10.8%	10.8%	8598	2.36	0.95	1.45
		REDQUEEN	10.1%	10.1%	7979	3.72	0.72	−0.25
`SQLite`	22031 / 57.1%	GRIMOIRE	48.6%	46.8%	18064	9.25	0.80	−0.72
		AFL	34.6%	33.9%	13072	10.02	0.60	−0.34
		ANGORA	33.1%	34.2%	13218	12.12	−0.30	−1.05
		QSYM	33.4%	33.6%	12988	10.91	−0.33	−0.18
		REDQUEEN	32.3%	32.6%	12599	4.77	0.18	−0.21
`NASM`	10015 / 51.1%	GRIMOIRE	47.7%	48.4%	9483	7.58	−2.58	5.67
		AFL	43.2%	43.0%	8442	1.68	1.07	1.09
		ANGORA	46.9%	47.0%	9211	5.27	0.06	−1.19
		QSYM	42.1%	42.6%	8357	4.72	−1.49	2.40
		REDQUEEN	44.9%	45.5%	8928	4.21	−0.20	−0.89